Points-to analysis in almost linear time

Bjarne Steensgaard

March, 1995

Technical Report
MSR-TR-95-08

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Points-to analysis in almost linear time

Bjarne Steensgaard

March, 1995

Abstract

We present an interprocedural points-to analysis based on type inference that has an almost linear time cost complexity. To our knowledge, this is the fastest points-to analysis algorithm yet described. The type inferred for a pointer variable represents a set of abstract locations conservatively describing the possible locations pointed to by the pointer variable. The type inferred for a function variable represents a set of functions it may point to and a type signature for these functions. The results are equivalent to those of a flow-insensitive alias analysis (and control flow analysis) that assumes alias relations are transitive.

Even if more accurate points-to analysis results are desired, it may be advantageous to perform the almost linear time points-to analysis anyway. The analysis results are useful for making program representations sparser, and can thus be used to reduce running time and space requirements for subsequent program analyses in a compiler, including a more precise points-to analysis.

The focus of this work is on the practical aspects of performing a points-to analysis by type inference.

1 Introduction

Dependence and interference analyses are fundamental to optimizing compilers. Almost universally used analyses include alias analyses computing pairs of expressions that may be aliased (e.g., [LR92, LRZ93, CBC93]) and points-to analyses computing a store model of abstract locations (e.g., [CWZ90, EGH94]). The results of these analyses are used both for optimizing programs and for making program representations sparser in order to speed up other analyses.

Existing interprocedural alias and points-to analyses are often slow. We present a very fast algorithm that performs an interprocedural points-to analysis by type inference. The types used in the algorithm represent sets of abstract locations. The types include relations between sets of abstract locations, in effect representing a model of the store (called a storage shape graph in [CWZ90]). The algorithm has time cost complexity of $O(N\alpha(N,N))$, where α is a (very slowly increasing) inverse Ackermann's function [Tar83], and N is the size of the input program.

The analysis is performed using the same unification techniques as Henglein's efficient type inference analysis [Hen91]. It is realistic to assume similar analysis performance in terms of speed, which has been reported to be between 1000 and 10000 lines/second.

The precision of the generated results are the same as that of any other flow-insensitive alias algorithm that assumes that alias relations are transitive (e.g., if `*y` is aliased to `x` and `*y` also is aliased to `z`, then `x` and `z` are assumed to be aliased). Most current alias and points-to analyses generate better results. The results of the fast analysis algorithm are still useful for many optimizations and for creating sparse program representations. The fast analysis results may also be used to reduce the size of the abstract values propagated by more precise analyses.

$$
\begin{array}{rcl}
S & ::= & \mathbf{x} = \mathbf{y} \\
 & | & \mathbf{x} = \&\mathbf{y} \\
 & | & \mathbf{x} = *\mathbf{y} \\
 & | & \mathbf{x} = \mathbf{op}(\mathbf{y}_1 \ldots \mathbf{y}_n) \\
 & | & \mathbf{x} = \mathbf{allocate}(\mathbf{y}) \\
 & | & *\mathbf{x} = \mathbf{y} \\
 & | & \mathbf{x} = \mathbf{fun}(\mathbf{f}_1 \ldots \mathbf{f}_n) \rightarrow (\mathbf{r}_1 \ldots \mathbf{r}_m)\ S^* \\
 & | & \mathbf{x}_1 \ldots \mathbf{x}_m = \mathbf{p}(\mathbf{y}_1 \ldots \mathbf{y}_n)
\end{array}
$$

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

The major technical contributions of this paper include identifying a a domain of types for describing store models that are linear in the size of the source program and showing how type inference over this domain of types can be performed in almost linear time using techniques similar to those of [Hen91].

We have used type inference to solve a very practical problem, namely that of performing a points-to analysis. The focus of this paper is therefore more on the practical aspects than on the type theoretical aspects of the algorithm.

In Section 2, we present our source language. In Section 3, we present the set of types used for the fast points-to analysis and a set of rules describing when programs are well-typed. In Section 4, we describe how to use type inference to compute a set of types for a program that obeys the given type rules. In Section 5, we explain how to use the analysis results to create sparse program representations and to reduce the size of abstract value domains for other analyses. In Section 6, we list related work, and we provide a summary in Section 7.

2 The source language

We will describe the points-to analysis for a small imperative language with pointers to locations, pointers to functions, dynamic allocation, and the ability to compute the address of a variable. Since the analysis is flow insensitive, the control structures of the language are irrelevant. The abstract syntax of the relevant statements of the language are shown in Figure 1.

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

Functions are constant values described by the $\mathbf{fun}(\ldots)\rightarrow(\ldots)$ expression form. The **f** variables are formal parameters (sometimes called *in parameters*), and the **r** variables are return values (sometimes called *out parameters*). Function calls have call-by-value semantics [ASU86]. Both formal and return parameter variables may be assigned to by the statements in the function body.

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

We assume that programs are fairly well-behaved. The analysis algorithm is not able to deal with bitwise duplication of pointer values (a covert value flow channel operating by bit-controlled control flow; not by data flow), or programmer knowledge about how the compiler allocates variables relative to each other. The

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

result = fact(10)
```

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

analysis algorithm as presented below will however deal with pointers type cast into integers and back to pointers, exclusive-or operations on pointer values, etc., where there is a real flow of values.

3 Types for points-to analysis

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

We use types to model how storage is used in a program at runtime (a storage model). Locations of variables and locations created by dynamic allocation are all described by types. The type describes the location as well as the contents of the location. Two locations must have the same type if the points-to analysis indicates that a pointer may point to either location.

We also use types to describe both user-defined functions. The type of a function is a signature consisting of a list of types of the argument variables and a list of types of the result variables. Note that the types only indirectly describe the argument and result values by describing the locations in which they must be stored.

Locations may represent and values may be aggregate objects with multiple pointers. An object may simultaneously contain both pointers to locations and pointers to functions. To describe this, the type of a location must be defined in terms of a location type describing the locations and a function type describing the functions that the location may contain pointers to. In this paper, we will not try to distinguish between individual elements of aggregate objects.

The types can be viewed as abstract locations. The set of types inferred for a program represents a store model [CWZ90] which is valid for all program points. The store model conservatively models all the points-to relations that may hold at runtime. Alias relations may be extracted from the store model [EGH94].

Our goal is a points-to analysis with an almost linear time cost complexity. The size of the store model for a program represented by the types must therefore be linear in the size of the input program. To ensure this limitation, a location type may only have one location type component, which is equivalent to only allowing one outgoing edge from each abstract location in the store model. Only allowing one outgoing edge from each abstract location in the store model implies that aliases are transitive in an alias or points-to analysis.

The types used by our points-to analysis are described by the following productions:

$$\begin{array}{lcl}\tau & ::= & \bot \mid \mathbf{ref}(V \to \alpha) \\ V & ::= & \mathcal{P}(\text{absloc}) \\ \alpha & ::= & (\tau \times \lambda) \\ \lambda & ::= & \bot \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})\end{array}$$

The τ types describe locations, the α types describe values, and the λ types describe functions. The V part of a location type is a set of abstract locations, which are described by the type. A value may be an aggregate object containing pointers to both locations and functions, so types of values is a cross-product of types describing locations and types describing functions.

Types may be recursive, and it may therefore be impossible to write out the types. The types are instead represented by type variables.

Points-to analysis by type inference is the problem of assigning types to all locations in a program, such that the program is well-typed given a set of typing rules. More precisely, the problem is to assign types to all locations in a program, so that two locations are described by different types, unless they have to be described by the same type in order for the program to be well-typed.

Figure 3 contains the typing rules for the source language in natural deduction style [Kah87]. The program is well-typed if, given a typing of all variables and constants, all the statements of the program are well-typed.

Implicit in the rules is the assumption that one abstract location will represent all instances of variables with the same name (multiple instances are only possible for local variables). We also assume that one abstract location is used to describe all dynamically allocated locations allocated by execution of the same statement. All references to abstract locations are however removed from the rules in Figure 3 for clarity reasons.

The typing rule for assignment of the **op**$(\ldots)$ expression form is designed to make the analysis as safe as possible. The typing rule assumes that any pointer operand may result in a pointer (in)to the same object, or result in some value from which such a pointer can be reconstructed. This is a perfectly reasonable assumption for, e.g., the binary exclusive-or operation, if the analysis is to be very safe. Another typing rule may be appropriate for the equality comparison operator and many other operators, but for the purposes of exposition, the typing rule as written is sufficient. In practice, different typing rules should be used for different operators.

The typing rule for dynamic allocation implies that a location type is required to describe the value stored in the variable assigned to. The type used to describe the allocated location may not be the type of any variable in the program. The type of the allocated location is only indirectly available from the type of the variable assigned to. Given the set of typing rules in Figure 3, all locations allocated in the same statement will have the same type. Locations allocated by different allocation statements may have different types if we use abstract locations that distinguish between them.

The type rules for definition of functions and calling of functions assume a call-by-value semantics. Call-by-reference semantics is easily simulated in call-by-value semantics by rewriting the program to pass a pointer to the locations as a value parameter. Call-by-name and copy-restore (also called *copy-in copy-out* and *value-result*) semantics is for the purposes of the fast points-to analysis identical to call-by-reference semantics. The fast analysis algorithm can therefore easily be applied to programs with call-by-reference, call-by-name, and copy-restore semantics as well as programs with only call-by-value semantics.

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

$$\frac{\mathbf{x} : \mathbf{ref}(\_ \to \alpha) \quad \mathbf{y} : \mathbf{ref}(\_ \to \alpha)}{\mathit{welltyped}(\mathbf{x} = \mathbf{y})}$$

$$\frac{\mathbf{x} : \mathbf{ref}(\_ \to (\tau \times \_)) \quad \mathbf{y} : \tau}{\mathit{welltyped}(\mathbf{x} = \&\mathbf{y})}$$

$$\frac{\mathbf{x} : \mathbf{ref}(\_ \to \alpha) \quad \mathbf{y} : \mathbf{ref}(\_ \to (\mathbf{ref}(\_ \to \alpha) \times \_))}{\mathit{welltyped}(\mathbf{x} = *\mathbf{y})}$$

$$\frac{\begin{array}{c}\mathbf{x} : \mathbf{ref}(\_ \to \alpha) \\ \mathbf{y}_1 : \mathbf{ref}(\_ \to \alpha) \ \ldots \ \mathbf{y}_n : \mathbf{ref}(\_ \to \alpha)\end{array}}{\mathit{welltyped}(\mathbf{x} = \mathbf{op}(\mathbf{y}_1 \ \ldots \ \mathbf{y}_n))}$$

$$\frac{\mathbf{x} : \mathbf{ref}(\_ \to (\mathbf{ref}(\_ \to \alpha) \times \_)) \quad \mathbf{y} : \tau}{\mathit{welltyped}(\mathbf{x} = \mathbf{allocate}(\mathbf{y}))}$$

$$\frac{\mathbf{x} : \mathbf{ref}(\_ \to (\mathbf{ref}(\_ \to \alpha) \times \_)) \quad \mathbf{y} : \mathbf{ref}(\_ \to \alpha)}{\mathit{welltyped}(*\mathbf{x} = \mathbf{y})}$$

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

$$\frac{\begin{array}{c}\mathbf{x}_1 : \mathbf{ref}(\_ \to \alpha_{n+1}) \ \ldots \ \mathbf{x}_m : \mathbf{ref}(\_ \to \alpha_{n+m}) \\ \mathbf{p} : \mathbf{ref}(\_ \to (\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))) \\ \mathbf{y}_1 : \mathbf{ref}(\_ \to \alpha_1) \ \ldots \ \mathbf{y}_n : \mathbf{ref}(\_ \to \alpha_n)\end{array}}{\mathit{welltyped}(\mathbf{x}_1, \ldots, \mathbf{x}_m = \mathbf{p}(\mathbf{y}_1 \ldots \mathbf{y}_n))}$$

Figure 3: Type rules for the relevant statement types of the source language. All variables are assumed to have been assigned a type, but the type environment is omitted in the equations for purposes of exposition. "_" is a wild-card value in the rules, imposing no restrictions on the type component it represents. For clarity, we have omitted in the rules the requirements that the abstract location corresponding to a given variable is in the V part of the type of the variable (and the same for dynamically allocated locations).

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

a: $\tau_1 = \mathbf{ref}(S_1 \rightarrow (\tau_4 \times \lambda_1))$
b: $\tau_2 = \mathbf{ref}(S_2 \rightarrow (\tau_5 \times \lambda_2))$
c: $\tau_3 = \mathbf{ref}(S_3 \rightarrow (\tau_5 \times \lambda_3))$
x: $\tau_4 = \mathbf{ref}(S_4 \rightarrow \alpha_1)$
y: $\tau_5 = \mathbf{ref}(S_5 \rightarrow (\tau_4 \times \lambda_4))$
z: τ_4
p: $\tau_6 = \mathbf{ref}(S_6 \rightarrow \alpha_2)$

Figure 4: Example program and a typing of same that obeys the typing rules. Different program variables are described by different type variables unless they must be described by the same type variable for the program to be well-typed.

4 Efficient type inference

Given the set of typing rules, we want to assign types to all locations in the source program, such that the program is well-typed, and that two locations are described by different type variables, unless they have to be described by the same type variable for the program to be well-typed. Computing the type to assign to a variable is called type inference. In this section we describe how type inference given the set of typing rules in Figure 3 can be performed in almost linear time.

In short, we start by assigning different type variables to all variables and constants and different type variables for the sets of locations and sets of functions that the variables may point to. Then we process all the statements of the program exactly once. When processing a statement, if different type variables occur in places where the same type variable should occur according to the typing rules, the found type variables are joined. Joining two type variables means replacing both of them throughout the program with a single type variable. Joining two type variables is made a local operation by using fast union/find data structures for type variables. When a type assignment consistent with the typing rules has been computed, all that is necessary to compute the actual types is the set of abstract locations represented by each location type.

We assume that given a variable, the first occurrence in the program of the variable can be found in constant time. For programs with variable declarations, this corresponds to requiring that variable names have been resolved. This enables us to find (the current approximation to) the type of a variable in constant time.

We use equivalence class representatives (ECRs) for type variables. Each ECR is associated with a type, where component types are represented by equivalence class representatives for type variables. In the first part of the algorithm, the V components of types are are irrelevant. Therefore, we only need type variables for the τ and λ components of the types.

As the first step of the type inference algorithm, we assign unique ECRs to all variables and (non-function) constants. The types associated with constants are bottom types. The types associated with the variables are of the form $\mathbf{ref}(\_ \rightarrow (E_1 \times E_2))$, where E_1 and E_2 are unique ECRs associated with the bottom type. The initial number of ECRs is therefore $C + 3V$, where C is the number of non-function constants in the program, and V is the number of variables in the program.

Then we process all the statements of the program. When processing a statement, the typing rules of Figure 3 are used to determine which types must be the same. If a statement is not well-typed because two types are represented by different ECRs, the two ECRs are combined into a single ECR, and the component types associated with the ECRs are unified. Figure 5 defines how each statement type in the source language is processed. Figure 6 defines how to join two types represented by ECRs.

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

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

$\mathbf{x} = *\mathbf{y}$:
  let $\mathbf{ref}(\_ \rightarrow (\tau_1 \times \lambda_1)) = \mathbf{type}(\mathbf{ecr}(\mathbf{x}))$
      $\mathbf{ref}(\_ \rightarrow (\tau_2 \times \lambda_2)) = \mathbf{type}(\mathbf{ecr}(\mathbf{y}))$ in
    if $\mathbf{type}(\tau_2) = \bot$ then
      $\mathbf{type}(\tau_2) \leftarrow \mathbf{ref}(\_ \rightarrow (\tau_1 \times \lambda_1))$
    else
      let $\mathbf{ref}(\_ \rightarrow (\tau_3 \times \lambda_3)) = \mathbf{type}(\tau_2)$ in
        if $\tau_1 \neq \tau_3$ then $\mathbf{join}(\tau_1, \tau_3)$
        if $\lambda_1 \neq \lambda_3$ then $\mathbf{join}(\lambda_1, \lambda_3)$

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

$\mathbf{x} = \mathbf{allocate}(\mathbf{y})$:
  let $\mathbf{ref}(\_ \rightarrow (\tau \times \lambda)) = \mathbf{type}(\mathbf{ecr}(\mathbf{x}))$ in
    if $\mathbf{type}(\tau) = \bot$ then
      let $e_1 = \mathbf{MakeECR}()$
          $e_2 = \mathbf{MakeECR}()$
          $e_3 = \mathbf{MakeECR}()$ in
        $\mathbf{type}(e_1) \leftarrow \mathbf{ref}(\_ \rightarrow (e_2 \times e_3))$
        $\mathbf{type}(\mathbf{ecr}(\mathbf{x})) \leftarrow \mathbf{ref}(\_ \rightarrow (e_1 \times \lambda))$

$*\mathbf{x} = \mathbf{y}$:
  let $\mathbf{ref}(\_ \rightarrow (\tau_1 \times \lambda_1)) = \mathbf{type}(\mathbf{ecr}(\mathbf{x}))$
      $\mathbf{ref}(\_ \rightarrow (\tau_2 \times \lambda_2)) = \mathbf{type}(\mathbf{ecr}(\mathbf{y}))$
    if $\mathbf{type}(\tau_1) = \bot$ then
      $\mathbf{type}(\tau_1) \leftarrow \mathbf{ref}(\_ \rightarrow (\tau_2 \times \lambda_2))$
    else
      let $\mathbf{ref}(\_ \rightarrow (\tau_3 \times \lambda_3)) = \mathbf{type}(\tau_1)$ in
        if $\tau_2 \neq \tau_3$ then $\mathbf{join}(\tau_2, \tau_3)$
        if $\lambda_2 \neq \lambda_3$ then $\mathbf{join}(\lambda_2, \lambda_3)$

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

$\mathbf{x}_1, \ldots, \mathbf{x}_m = \mathbf{p}(\mathbf{y}_1 \ldots \mathbf{y}_n)$:
  let $\mathbf{ref}(\_ \rightarrow (\tau \times \lambda)) = \mathbf{type}(\mathbf{ecr}(\mathbf{p}))$
    if $\mathbf{type}(\lambda) = \bot$ then
      $\mathbf{type}(\lambda) \leftarrow \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha n+m)$
      where
        $\mathbf{ref}(\_ \rightarrow \alpha_i) = \mathbf{type}(\mathbf{ecr}(\mathbf{y}_i))$, for $i \leq n$
        $\mathbf{ref}(\_ \rightarrow \alpha_i) = \mathbf{type}(\mathbf{ecr}(\mathbf{x}_{i-n}))$, for $i > n$
    else
      let $\mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathbf{type}(\lambda)$ in
        for $i \in [1 \ldots n]$ do
          let $(\tau_1 \times \lambda_2) = \alpha_i$
              $\mathbf{ref}(\_ \rightarrow (\tau_2 \times \lambda_2)) = \mathbf{type}(\mathbf{ecr}(\mathbf{y}_i))$ in
            if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$
            if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_1, \lambda_2)$
        for $i \in [1 \ldots m]$ do
          let $(\tau_1 \times \lambda_2) = \alpha_{n+i}$
              $\mathbf{ref}(\_ \rightarrow (\tau_2 \times \lambda_2)) = \mathbf{type}(\mathbf{ecr}(\mathbf{x}_i))$ in
            if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$
            if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_1, \lambda_2)$
```

Figure 5: Inference rules corresponding to the typing rules given in Figure 3. **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 join of the types represented by the ECRs x and y, and is defined in Figure 6. **MakeECR**() constructs a new ECR, which is associated with the bottom type, $\bot$.

```
$\mathbf{join}(e_1, e_2)$:
  let $t_1 = \mathbf{type}(e_1)$
      $t_2 = \mathbf{type}(e_2)$
      $e = \text{ecr-union}(e_1, e_2)$ in
     $\mathbf{type}(e) \leftarrow \mathbf{unify}(t_1, t_2)$

$\mathbf{unify}(\mathbf{ref}(\_ \to (\tau_1 \times \lambda_1)), \mathbf{ref}(\_ \to (\tau_2 \times \lambda_2)))$:
  if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$
  if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_1, \lambda_2)$

$\mathbf{unify}(\mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}),$
        $\mathbf{lam}(\alpha'_1 \ldots \alpha'_n)(\alpha'_{n+1} \ldots \alpha'_{n+m}))$
  let $(\tau_1 \times \lambda_1) = \alpha_i$
      $(\tau_2 \times \lambda_2) = \alpha'_i$ in
     if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$
     if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_1, \lambda_2)$
```

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

The processing of a program statement changes the typing of the program to ensure that the statement is well-typed. Any statement that was well-typed before the change will also be well-typed after the change. Processing all the statements in a program ensures that all statements are well-typed and consequently that the program is well-typed.

The initialization and processing phases of the algorithm can be done simultaneously, requiring only a single pass over the program. The analysis can also be performed while constructing the internal representation of the program.

During the processing of the statements of the program, $3A$ new ECRs may be created, where A is the number of allocation statements in the program. The total number of ECRs created is therefore linear in the size of the program.

Since types are recursive, joining two type variables (ECRs) may require recursively joining other type variables. The cost of joining two type variables and their component types can therefore be more than a constant. However, the maximal number of join operations possible is limited by the number of type variables, which is proportional to the size of the program.

The use of fast union/find data structures means that checking for equality of two ECRs has a cost that on average is $\alpha(N, N)$, where α is a (very slowly increasing) inverse of Ackermann's function [Tar83], and N is the size of the source program.

Adding the various costs, the worst case time cost complexity of the type inference phase of the algorithm is $O(N\alpha(N, N))$.

Replacing the ECRs with ordinary type variables after the type inference phase is straightforward. Computing the V components of the types is done by visiting all variables exactly once, adding the abstract locations associated with the variable to the type describing the variable. The abstract locations describing dynamically allocated locations are added while visiting each allocation statement exactly once.

5 Creating sparse program representations

Even if more accurate points-to analysis results than provided by the analysis algorithm described above are desired, it may be advantageous to perform this analysis anyway. The analysis results can be used for making program representations sparser, and possibly also to reduce the domain of abstract values propagated by data flow analyses (including a more precise alias or points-to analysis).

One sparse program representation is Static Single Assignment (SSA) form [CFR$^+$91] and its extensions, Factored SSA form (FSSA) form [CCF94] and Location Factoried SSA form [Ste95]. All three representations have dependence edges between use and def points (with ϕ nodes at control flow merge points). The use and def points are defined in terms of variables (or in terms of abstract locations for non-Fortran programs).

The types computed by our analysis represent a set of abstract locations. If a variable of a given type is updated, the assignment may be considered a def point for all of the abstract locations represented by that type. If we use the results of the almost linear points-to analysis to generate a FSSA form of the program, there will be many parallel dependence edges in the graph.

We could instead use an alternate definition of use and def points in terms of the types of variables (types as computed by the fast points-to analysis described in the present paper). A type def point for a given type is a statement where a variable of that type is assigned a value. A type use point for a given type is an expression using a variable of that type. We can now construct a type FSSA form of the program using type use and type def points instead of the conventional use and def points.

Consider an assignment to a single variable of a given type representing a set of abstract locations. In the conventional FSSA form, there would be parallel dependence edges to and from this assignment, each edge representing one abstract location possibly modified by the assignment. The maximal number of dependence edges is quadratic in the size of the source program. In the type FSSA form, the parallel edges are replaced by a single edge. The maximal number of dependence edges in the type FSSA form is linear in the size of the source program. This factoring of dependence edges is incidentally a special case of Assignment Factored SSA form.

Using type FSSA form, data flow analyses may reduce the size of the domains of abstract values propagated by data flow analyses. An example of such an analysis is Emami, Ghiya, and Hendren's points-to analysis [EGH94]. Their algorithm uses a square bit matrix whose dimensions are given by the number of variables in the source program. A set bit in the matrix means that the location/variable corresponding to one coordinate may point to the location/variable corresponding to the other coordinate. Given the results of the fast points-to analysis, we know that a given bit will never be set unless one of the variables is described by a location type whose location type component is the type of the other variable.

Using the results of the fast points-to analysis, Emami, Ghiya and Hendren's points-to analysis need only represent the parts of the matrix that have bits that may be set. Instead of one bit matrix, a matrix for each points-to type can be used instead. Given a points-to type, the dimensions of the matrix are given by the number of abstract locations represented by the type and the number of abstract locations represented by the location type component of that type. This may lead to very large space savings for large programs. In addition, instead of propagating the large matrix from statement to statement, the smaller matrices can be propagated along the dependence edges of the type FSSA form, thereby skipping irrelevant statements.

6 Related work

Henglein used type inference to perform a binding time analysis in almost linear time [Hen91]. He takes a slightly more formal approach than we have done for the points-to analysis. His presentation of the algorithm in terms of constraint systems may be more familiar to people working with type theory.

The points-to analysis that closest resembles our analysis is Weihl's [Wei80]. His analysis is also flow insensitive and deals with pointers to functions. His algorithm does not assume that alias relations are transitive and will therefore in many cases produce better results than our algorithm will. On the other hand, his algorithm does not distinguish between one or several levels of pointer indirection. His algorithm has a time cost complexity that is cubic in the size of the source program, where our algorithm has an almost linear time cost complexity.

More precise points-to analysis exist, e.g., [CWZ90] and [EGH94]. [CWZ90] is primarily concerned with heap allocated structures, whereas [EGH94] is concerned with stack allocated structures. Both algorithms are flow sensitive data flow analyses. [CWZ90] has polynomial time cost complexity. [EGH94] has polynomial complexity in the size of an unfolded call graph, but exponential complexity in the size of the program.

Another type of dependence analysis is alias analysis. Where a points-to analysis build and maintains a model of the store during analysis, an alias analysis builds and maintains a list of access path expressions that may evaluate to the same location (in other words: they are aliased). The most relevant alias analysis algorithms are described in [CCF94] and [LR92, LRZ93]. The length of access-paths are k-limited, using a relatively simple truncation mechanism to eliminate extra path elements.

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

A higher order points-to analysis by type inference has been developed by Tofte and Talpin for the purposes of creating an ML interpreter without a garbage collector [TT94]. The analysis is based on type inference over a polymorphic domain of types. Variables are allocated from (written to) regions that are passed as extra parameters. Their algorithm does not deal with objects that may be updated after being assigned an initial value (as is normal for imperative programs). Whether their work can be generalized to work for general imperative programs is an open question.

7 Conclusion and future work

We have described a very fast points-to analysis operating by type inference over a set of types representing sets of abstract locations. Each type represents a location object in a store model which is valid for all program points. We have outlined how the analysis results can be used to make program representations sparser and to reduce the size of abstract domains used for data flow analyses.

The presented algorithm has been implemented in the context of the VDG compiler environment at Microsoft Research [WCES94a, WCES94b]. Preliminary results show that each type variable on average only describes a small number of abstract locations[1]. Preliminary measurements indicate that the average number of abstract locations described by a single type variable is between 3 and 4.

The algorithm can be improved in a number of ways. Distinct elements of aggregate objects may be represented by separate type variables (unless the object is viewed as a union or overlay). A restricted (k-limited) degree of polymorphism may also be achieved without sacrificing the time cost complexity of the algorithm. This may prove advantageous, as many functions in real-life programs only perform limited manipulation of locations accessible through argument pointers (although [Ruf95] suggests that the advantage may not be very substantial).

In the presented type system, the types are monomorphic. The type system may be generalized to an ML-style polymorphic system. Whether the type system may be generalized to a polymorphic system while retaining its usefulness for making program representations sparser is still unknown.

[1]It was considered more important to add another component to our system than to perform measurements on this component. Now this other component is practically finished, so the measurements can be performed Real Soon Now.

Acknowledgments

I would like to thank Roger Crew and Daniel Weise for helpful discussions. I also want to express my gratitude to them and the other members of the program analysis group at Microsoft Research for commenting on earlier versions of this paper, for providing a stimulating environment, and for co-developing the VDG, without which this work would not have come into existence.

References

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

[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. ACM Press, January 1993.

[CCF94] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

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

[Deu94] Alian Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN'94: Conference on Programming Language Design and Implementation, (PLDI)*, pages 230–241, Orlando, FL, June 20-24 1994. Proceedings also appear as SIGPLAN Notices 29(6), 1994.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN'94: Conference on Programming Language Design and Implementation, (PLDI)*, pages 242–256, Orlando, FL, June 20-24 1994. Proceedings also appear as SIGPLAN Notices 29(6), 1994.

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

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247)*, pages 22–39. Springer-Verlag, 1987.

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

[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. ACM Press, 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. ACM Press, June 1993.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. To appear in PLDI'95, June 1995.

[Ste95] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 62–70, San Francisco, CA, January 22 1995. Proceedings to appear as an issue of SIGPLAN Notices.

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

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

[WCES94a] 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, Portland, OR, January 1994.

[WCES94b] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. Technical Report MSR-TR-94-03, Microsoft Research, Redmond, WA, April 13, 1994.

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