Program Analysis and Specialization for the C Programming Language

Ph.D. Thesis

Lars Ole Andersen

DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
Denmark
email: `lars@diku.dk`

May 1994

Chapter 4

Pointer Analysis

We develop an efficient, inter-procedural pointer analysis for the C programming language. The analysis approximates for every variable of pointer type the set of objects it may point to during program execution. This information can be used to improve the accuracy of other analyses.

The C language is considerably harder to analyze than for example Fortran and Pascal. Pointers are allowed to point to both stack and heap allocated objects; the address operator can be employed to compute the address of an object with an lvalue; type casts enable pointers to change type; pointers can point to members of structs; and pointers to functions can be defined.

Traditional pointer analysis is equivalent to alias analysis. For example, after an assignment '`p = &x`', '`*p`' is aliased with '`x`', as denoted by the alias pair $\langle *p, x \rangle$. In this chapter we take another approach. For an object of pointer type, the set of objects the pointer *may point to* is approximated. For example, if in the case of the assignments '`p = &x; p = &y`', the result of the analysis will be a map $[p \mapsto \{x, y\}]$. This is a more economical representation that requires less storage, and is suitable for many analyses.

We specify the analysis by the means of a non-standard *type inference system*, which is related to the standard semantics. From the specification, a *constraint-based* formulation is derived and an efficient inference algorithm developed. The use of non-standard type inference provides a clean separation between specification and implementation, and gives a considerably simpler analysis than previously reported in the literature.

This chapter also presents a technique for *inter-procedural* constraint-based program analysis. Often, context-sensitive analysis of functions is achieved by copying of constraints. This increases the number of constraints exponentially, and slows down the solving. We present a method where constraints over *vectors* of pointer types are solved. This way, only a few more constraint are generated than in the intra-procedural case.

Pointer analysis is employed in the *C-Mix* system to determine side-effects, which is then used by binding-time analysis.

4.1 Introduction

When the lvalue of two objects coincides the objects are said to be *aliased*. An alias is for instance introduced when a pointer to a global variable is created by the means of the address operator. The aim of alias analysis is to approximate the set of aliases at runtime. In this chapter we present a related but somewhat different pointer analysis for the C programming language. For every pointer variable it computes the set of abstract locations the pointer may point to.

In languages with pointers and/or call-by-reference parameters, alias analysis is the core part of most other data flow analyses. For example, live-variable analysis of an expression '`*p = 13`' must make worst-case assumptions without pointer information: '`p`' may reference all (visible) objects, which then subsequently must be marked "live". Clearly, this renders live-variable analysis nearly useless. On the other hand, if it is known that only the aliases $\{\langle *p, x\rangle, \langle *p, y\rangle\}$ are possible, only '`x`' and '`y`' need to be marked "live".

Traditionally, aliases are represented as an equivalence relation over abstract locations [Aho *et al.* 1986]. For example, the alias introduced due to the expression '`p = &x`' is represented by the *alias set* $\{\langle *p, x\rangle\}$. Suppose that the expressions '`q = &p; *q = &y`' are added to the program. The alias set then becomes $\{\langle *p, x\rangle, \langle *q, p\rangle, \langle **q, x\rangle, \langle **q, y\rangle\}$, where the latter aliases are *induced* aliases. Apparently, the size of an alias set may evolve rather quickly in a language with multi-level pointers such as C. Some experimental evidence: Landi's alias analysis reports more than 2,000,000 program-point specific aliases in a 3,000 line program [Landi 1992a].

Moreover, alias sets seem excessively general for many applications. What needed is an answer to "which objects may this pointer point to"? The analysis of this chapter answer this question.

4.1.1 What makes C harder to analyze?

The literature contains a substantial amount of work on alias analysis of Fortran-like languages, see Section 4.11. However, the C programming language is considerably more difficult to analyze; some reasons for this include: multi-level pointers and the address operator '&', structs and unions, runtime memory allocations, type casts, function pointers, and separate compilation.

As an example, consider an assignment '`*q = &y`' which adds a point-to relation to p (assuming '`q`' points to '`p`') even though '`p`' is not syntactically present in the expression. With only single-level pointers, the variable to be updated is syntactically present in the expression.[1] Further, in C it is possible to have pointers to both heap and stack allocated objects, as opposed to Pascal that abandon the latter. We shall mainly be concerned with analysis of pointers to stack allocated objects, due to our specific application.

A special characteristic of the C language is that implementation-defined features are supported by the Standard. An example of this is cast of integral values to pointers.[2]

[1] It can easily be shown that call-by-reference and single-level pointers can simulate multi-level pointers.

[2] Recall that programs relying on implementation-defined features are non-strictly conforming.

Suppose that '`long table[]`' is an array of addresses. A cast '`q = (int **)table[1]`' renders '`q`' to be implementation-defined, and accordingly worst-case assumptions must be made in the case of '`*q = 2`'.

4.1.2 Points-to analysis

For every object of pointer type we determine a *safe* approximation to the set of locations the pointer *may* contain during program execution, for all possible input. A special case is function pointers. The result of the analysis is the set of functions the pointer may invoke.

Example 4.1 We represent point-to information as a map from program variables to sets of object "names". Consider the following program.

```
int main(void)
{
   int x, y, *p, **q, (*fp)(char *, char *);
   p = &x;
   q = &p;
   *q = &y;
   fp = &strcmp;
}
```

A safe point-to map is

$$[p \mapsto \{x, y\}, q \mapsto \{p\}, fp \mapsto \{strcmp\}]$$

and it is also a *minimal* map. **End of Example**

A point-to relation can be classified *static* or *dynamic* depending on its creation. In the case of an array '`int a[10]`', the name '`a`' statically points to the object '`a[]`' representing the content of the array.[3] Moreover, a pointer to a struct points, when suitable converted, to the initial member [ISO 1990]. Accurate static point-to information can be collected during a single pass of the program.

Point-to relations created during program execution are called *dynamic*. Examples include '`p = &x`', that creates a point-to relation between '`p`' and '`x`'; an '`alloc()`' call that returns a pointer to an object, and '`strdup()`' that returns a pointer to a string. More general, *value setting* functions may create a dynamic point-to relation.

Example 4.2 A point-to analysis of the following program

```
char *compare(int first, char *s, char c)
{
   char (*fp)(char *, char);
   fp = first? &strchr : &strrchr;
   return (*fp)(s,c);
}
```

will reveal $[fp \mapsto \{strchr, strrchr\}]$. **End of Example**

It is easy to see that a point-to map carries the same information as an alias set, but it is a more compact representation.

[3] We treat arrays as aggregates.

4.1.3 Set-based pointer analysis

In this chapter we develop a flow-insensitive *set-based point-to analysis* implemented via *constraint solving*. A set-based analysis consists of two parts: a specification and an inference algorithm.

The specification describes the *safety* of a pointer approximation. We present a set of inference rules such that a pointer abstraction map fulfills the rules only if the map is safe. This gives an algorithm-independent characterization of the problem.

Next, we present a *constraint-based* characterization of the specification, and give a constraint-solving algorithm. The constraint-based analysis works in two phases. First, a *constraint system* is generated, capturing dependencies between pointers and abstract locations. Next, a *solution* to the constraints is found via an iterative solving procedure.

Example 4.3 Consider again the program fragment in Example 4.1. Writing T_p for the abstraction of '`p`', the following constraint system could be generated:

$$\{T_p \supseteq \{x\}, T_q \supseteq \{p\}, *T_q \supseteq \{y\}, T_{fp} \supseteq \{strcmp\}\}$$

with the interpretation of the constraint $*T_q \supseteq \{y\}$: "the objects '`q`' may point to contain y". **End of Example**

Constraint-based analysis resembles classical data-flow analysis, but has a stronger semantical foundation. We shall borrow techniques for iterative data-flow analysis to solve constraint systems with finite solutions [Kildall 1973].

4.1.4 Overview of the chapter

This chapter develops a flow-insensitive, context-sensitive constraint-based point-to analysis for the C programming language, and is structured as follows.

In Section 4.2 we discuss various degrees of accuracy a value-flow analysis can implement: intra- and inter-procedural analysis, and flow-sensitive versus flow-insensitive analysis. Section 4.3 considers some aspects of pointer analysis of C.

Section 4.4 specifies a sticky, flow-insensitive pointer analysis for C, and defines the notion of *safety*. In Section 4.5 we give a constraint-based characterization of the problem, and prove its correctness.

Section 4.6 extends the analysis into a context-sensitive inter-procedural analysis. A sticky analysis merges all calls to a function, resulting in loss of precision. We present a technique for context-sensitive constraint-based analysis based on static-call graphs.

Section 4.7 presents a constraint-solving algorithm. In Section 4.8 we discuss algorithmic aspects with emphasis on efficiency, and Section 4.9 documents the usefulness of the analysis by providing some benchmarks from an existing implementation.

Flow-sensitive analyses are more precise than flow-insensitive analyses. In Section 4.10 we investigate program-point, constraint-based pointer analysis of C. We show why multi-level pointers render this kind of analysis difficult.

Finally, Section 4.11 describe related work, and Section 4.12 presents topics for future work and concludes.

4.2 Pointer analysis: accuracy and efficiency

The precision of a value-flow analysis can roughly be characterized by two properties: flow-sensitivity and whether it is inter-procedural vs. intra-procedural. Improved accuracy normally implies less efficiency and more storage usage. In this section we discuss the various degrees of accuracy and their relevance with respect to C programs.

4.2.1 Flow-insensitive versus flow-sensitive analysis

A data-flow analysis that takes control-flow into account is called *flow-sensitive*. Otherwise it is *flow-insensitive*. The difference between the two is most conspicuous by the treatment of `if` statements. Consider the following lines of code.

```
int x, y, *p;
if ( test ) p = &x; else p = &y;
```

A flow-sensitive analysis records that in the branches, '`p`' is assigned the address of '`x`' and '`y`', respectively. After the branch, the information is merged and '`p`' is mapped to both '`x`' and '`y`'. The discrimination between the branches is important if they for instance contain function calls '`foo(p)`' and '`bar(p)`', respectively.

A flow-insensitive analysis summarizes the pointer usage and states that '`p`' may point to '`x`' and '`y`' in both branches. In this case, spurious point-to information would be propagated to '`foo()`' and '`bar()`'.

The notion of flow-insensitive and flow-sensitive analysis is intimately related with the notion of *program-point specific* versus *summary* analysis. An analysis is *program-point specific* is if it computes point-to information for each program point.[4] An analysis that maintains a summary for each variable, valid for *all* program points of the function (or a program, in the case of a global variable), is termed a *summary* analysis. Flow-sensitive analyses must inevitably be program-point specific.

Flow-sensitive versus in-sensitive analysis is a trade off between accuracy and efficiency: a flow-sensitive analysis is more precise, but uses more space and is slower.

Example 4.4 Flow-insensitive and flow-sensitive analysis.

```
/* Flow-insensitive */
int main(void)
{
   int x, y, *p;
   p = &x;
   /* p ↦ { x,y } * /
   foo(p);
   p = &y;
   /* p ↦ { x,y } */
}
```

```
/* Flow-sensitive */
int main(void)
{
   int x, y, *p;
   p = &x;
   /* p ↦ {x} */
   foo(p);
   p = &y;
   /* p ↦ {y} */
}
```

[4] The analysis does not necessarily have to compute the complete set of pointer variable bindings; only at "interesting" program points.

Notice that in the flow-insensitive case, the spurious point-to information $p \mapsto \{y\}$ is propagated into the function '`foo()`'. **End of Example**

We focus on flow-insensitive (summary) pointer analysis for the following reasons. First, in our experience, most C programs consist of many small functions.[5] Thus, the extra approximation introduced by summarizing all program points appears to be of minor importance. Secondly, program-point specific analyses may use an unacceptable amount of storage. This, pragmatic argument matters when large programs are analyzed. Thirdly, our application of the analysis does not accommodate program-point specific information, *e.g.* the binding-time analysis is program-point insensitive. Thus, flow-sensitive pointer analysis will not improve upon binding-time separation (modulo the propagation of spurious information — which we believe to be negligible).

We investigate program-point specific pointer analysis in Section 4.10.

4.2.2 Poor man's program-point analysis

By a simple transformation it is possible to recover some of the accuracy of a program-point specific analysis, without actually collecting information at each program point.

Let an assignment $e_1 = e_2$, where e_1 is a variable and e_2 is independent from pointer variables, be called an *initialization assignment*. The idea is to rename pointer variables when they are initialized.

Example 4.5 Poor man's flow-sensitive analysis of Example 4.4. The variable '`p`' has been "copied" to '`p1`' and '`p2`'.

```
int main(void)
{
   int x, y, *p1, *p2;
   p1 = &x;
   /* p1 ↦ {x} */
   foo(p1);
   p2 = &y;
   /* p2 ↦ {y} */
}
```

Renaming of variables can clearly be done automatically. **End of Example**

The transformation fails on indirect initializations, *e.g.* an assignment '`*q = &x;`', where '`q`' points to a pointer variable.[6]

4.2.3 Intra- and inter-procedural analysis

Intra-procedural analysis is concerned with the data flow in function bodies, and makes worst-call assumption about function calls. In this chapter we shall use 'intra-procedural' in a more strict meaning: functions are analysed context-independently. *Inter-procedural*

[5] As opposed to Fortran, that tends to use "long" functions.

[6] All flow-sensitive analyses will gain from this transformation, including binding-time analysis.

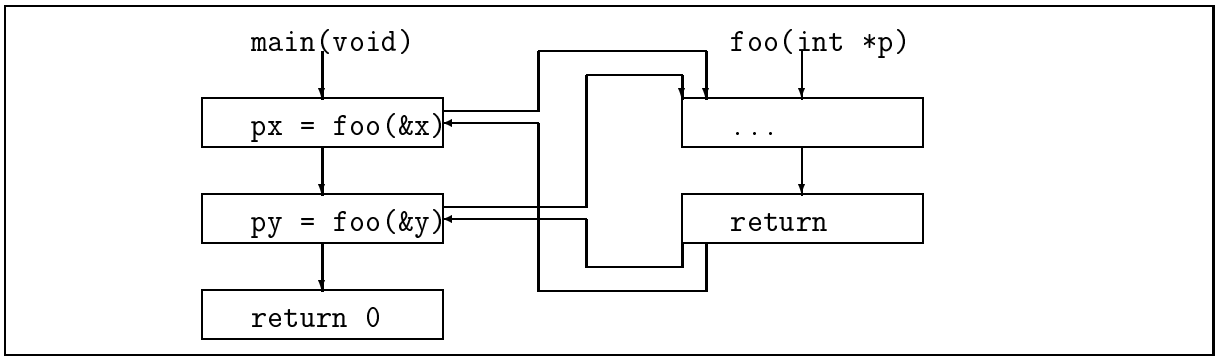

Figure 34: Inter-procedural call graph for program in Example 4.6

analysis infers information under consideration of call contexts. Intra-procedural analysis is also called *monovariant* or *sticky*, and inter-procedural analysis is also known as *polyvariant*.

Example 4.6 Consider the following program.

```
int main(void)              int *foo(int *p)
{                           {
   int x,y,*px,*py;             ...
   px = foo(&x);                return p;
   py = foo(&y);            }
   return 0;
}
```

An intra-procedural analysis merges the contexts of the two calls and computes the points-to information $[px, py \mapsto \{x, y\}]$. An inter-procedural analysis differentiates between to two calls. Figure 34 illustrates the inter-procedural call graph. **End of Example**

Inter-procedural analysis improves the precision of intra-procedural analysis by preventing calls to interfere. Consider Figure 34 that depicts the inter-procedural call graphs of the program in Example 4.6. The goal is that the value returned by the first call is not erroneous propagated to the second call, and vice versa. Information must only be propagated through *valid* or *realizable* program paths [Sharir and Pnueli 1981]. A control-path is realizable when the inter-procedural exit-path corresponds to the entry path.

4.2.4 Use of inter-procedural information

Inter-procedural analysis is mainly concerned with the *propagation* of value-flow information through functions. Another aspect is the *use* of the inferred information, *e.g.* for optimization, or to drive other analyses. Classical inter-procedural analyses produce a summary for each function, that is, all calls are merged. Clearly, this degrades the number of possible optimizations.

Example 4.7 Suppose we apply inter-procedural constant propagation to a program containing the calls '`bar(0)`' and '`bar(1)`'. Classical analysis will merge the two calls and henceforth classify the parameter for 'non-const', ruling out *e.g.* compile-time execution of an `if` statement [Callahan *et al.* 1986]. **End of Example**

An aggressive approach would be either to *inline* functions into the caller or to *copy* functions according to their use. The latter is also known as *procedure cloning* [Cooper *et al.* 1993,Hall 1991].

We develop a flexible approach where each function is annotated with both context specific information and a summary. At a later stage the function can then be cloned, if so desired. We return to this issue in Chapter 6, and postpone the decision whether to clone a function or not.[7]

We will assume that a program's static-call graph is available. Recall that the static-call graph approximates the invocation of functions, and assigns a *variant* number to functions according to the call contexts. For example, if a function is called in n contexts, the function has n variants. Even though function not are textually copied according to contexts, it is useful to imagine that n variants of the function's parameters and local variables exist. We denote by v^i the variable corresponding to the i'th variant.

4.2.5 May or must?

The *certainty* of a pointer abstraction can be characterized by *may* or *must*. A *may point-to* analysis computes for every pointer set of abstract locations that the pointer *may* point to at runtime. A *must point-to* analysis computes for every pointer a set of abstract locations that the pointer *must* point to.

May and must analysis is also known as *existential* and *universal* analysis. In the former case, there must exists a path where the point-to relation is valid, in the latter case the point-to relation must be valid on all paths.

Example 4.8 Consider live-variable analysis of the expression '`x= *p`'. Given must point-to information $[p \mapsto \{y\}]$, '`y`' can be marked "live". On the basis of may point to information $[p \mapsto \{y, z\}]$, both '`y` and '`z`' must be marked "live". **End of Example**

We shall only consider may point-to analysis in this chapter.

4.3 Pointer analysis of C

In this section we briefly consider pointer analysis of some of the more intricate features of C such separate compilation, external variables and non-strictly complying expressions, *e.g.* type casts, and their interaction with pointer analysis.

4.3.1 Structures and unions

C supports user-defined structures and unions. Recall from Section 2.3.3 that struct variables sharing a common type definition are separated (are given different names) during parsing. After parsing, a value-flow analysis unions (the types of) objects that (may) flow together.

[7] Relevant information includes number of calls, the size of the function, number of calls in the functions.

Example 4.9 Given definitions '`struct S { int *p;} s,t,u;`', variants of the struct type will be assigned to the variables, *e.g.* '`s`' will be assigned the type '`struct S1`'. Suppose that the program contains the assignment '`t = s`'. The value-flow analysis will then merge the type definitions such that '`s`' and '`t`' are given the same type ('`struct S1`', say), whereas '`u`' is given the type '`struct S3`', say. **End of Example**

Observe: struct variables of different type *cannot flow together*. Struct variables of the same type *may flow together*. We exploit this fact the following way.

Point-to information for field members of a struct variable is associated with the *definition* of a struct; not the struct objects. For example, the point to information for member '`s.p`' (assuming the definitions from the Example above) is represented by '`S1.p`', where '`S1`' is the "definition" of '`struct S1`'. The definition is common for all objects of that type. An important consequence: in the case of an assignment '`t = s`', the fields of '`t`' do not need to be updated with respect to '`s`' — the value-flow analysis have taken care of this.

Hence, the pointer analysis is factorized into the two sub-analyses

1. a (struct) value-flow analysis, and
2. a point-to propagation analysis

where this chapter describes the propagation analysis. We will (continue to) use the term pointer analysis for the propagation analysis.

Recall from Chapter 2 that some initial members of unions are truly shared. This is of importance for pointer analysis if the member is of pointer type. For simplicity we we will not take this aspect into account. The extension is straightforward, but tedious to describe.

4.3.2 Implementation-defined features

A C program can comply to the Standard in two ways. A *strictly conforming* program shall not depend on implementation-defined behavior but a *conforming* program is allowed to do so. In this section we consider type casts that (in most cases) are non-strictly conforming.

Example 4.10 Cast of an integral value to a pointer or conversely is an implementation-defined behaviour. Cast of a pointer to a pointer with less alignment requirement and back again, is strictly conforming [ISO 1990]. **End of Example**

Implementation-defined features cannot be described accurately by an architecture-independent analysis. We will approximate pointers that may point to any object by the unique abstract location 'Unknown'.

Definition 4.1 *Let '*`p`*' be a pointer. If a pointer abstraction maps '*`p`*' to* Unknown, $[p \mapsto \text{Unknown}]$*, when '*`p`*' may point to* all *accessible objects at runtime.* □

The abstract location 'Unknown' corresponds to "know nothing", or "worst-case".

Example 4.11 The goal parameters of a program must be described by 'Unknown', *e.g.* the '`main`' function

```
int main(int argc, char **argv)
{ ... }
```

is approximated by [`argv` $\mapsto$ {Unknown}]. **End of Example**

In this chapter we will *not* consider the `setjmp`' and '`longjmp`' macros.

4.3.3 Dereferencing unknown pointers

Suppose that a program contains an assignment through an Unknown pointer, *e.g.* '`*p = 2`', where [$p \mapsto$ {Unknown}]. In the case of live-variable analysis, this implies that worst-case assumptions must be made. However, the problem also affects the pointer analysis.

Consider an assignment '`*q = &x`', where '`q`' is unknown. This implies after the assignment, *all* pointers may point to '`x`'. Even worse, an assignment '`*q = p`' where '`p`' is unknown renders *all* pointers unknown.

We shall proceed as follows. If the analysis reveals that an Unknown pointer may be dereferenced in the left hand side of an assignment, the analysis stops with "worst-case" message. This corresponds to the most inaccurate pointer approximation possible. Analyses depending on pointer information must make worst-case assumptions about the pointer usage.

For now we will assume that Unknown pointers are *not* dereferenced in the left hand side of an assignment. Section 4.8 describes handling of the worst-case behaviour.

4.3.4 Separate translation units

A C program usually consists of a collection of translation units that are compiled separately and linked to an executable. Each file may refer to variables defined in other units by the means of '`extern`' declarations. Suppose that a pointer analysis is applied to a single module.

This has two consequences. Potentially, global variables may be modified by assignments in other modules. To be safe, worst-case assumptions, *i.e.* Unknown, about global variables must be made. Secondly, functions may be called from other modules with unknown parameters. Thus, to be safe, all functions must be approximated by Unknown.

To obtain results other than trivial we shall avoid separate analysis, and assume that "relevant" translation units are merged; *i.e.* we consider solely monolithic programs. The subject of Chapter 7 is separate program analysis, and it outlines a separate pointer analysis based on the development in this chapter.

Constraint 4.1 *i*) No global variables of pointer type may be modified by other units. *ii*) Functions are assumed to be static to the translation unit being analyzed.

It is, however, convenient to sustain the notion of an object being "external". For example, we will describe the function '`strdup()`' as returning a pointer to an 'Unknown' object.

4.4 Safe pointer abstractions

A *pointer abstraction* is a map from abstract program objects (variables) to sets of abstract locations. An abstraction is *safe* if for every object of pointer type, the set of concrete addresses it may contain at runtime is safely described by the set of abstract locations. For example, if a pointer '`p`' may contain the locations l_x (location of '`x`') and l_g (location of '`g`') at runtime, a safe abstraction is $p \mapsto \{x, g\}$.

In this section we define *abstract locations* and make precise the notion of *safety*. We present a specification that can be employed to check the safety of an abstraction. The specification serves as foundation for the development of a constraint-based pointer analysis.

4.4.1 Abstract locations

A pointer is a variable containing the distinguished constant '`NULL`' or an address. Due to casts, a pointer can (in principle) point to an arbitrary address. An *object* is a set of logically related locations, *e.g.* four bytes representing an integer value, or n bytes representing a struct value. Since pointers may point to functions, we will also consider functions as objects.

An object can either be allocated on the program stack (local variables), at a fixed location (strings and global variables), in the code space (functions), or on the heap (runtime allocated objects). We shall only be concerned with the run time allocated objects brought into existence via '`alloc()`' calls. Assume that all calls are labeled uniquely.[8] The label l of an '`alloc`^{l}`()`' is used to denote the set of (anonymous) objects allocated by the '`alloc`^{l}`()`' call-site. The label l may be thought of as a pointer of a relevant type.

Example 4.12 Consider the program lines below.

```
int x, y, *p, **q, (*fp)(void);
struct S *ps;
p = &x;
q = &p;
*q = &y;
fp = &foo;
ps = alloc^1(S);
```

We have: $[p \mapsto \{x, y\}, q \mapsto \{p\}, fp \mapsto \{foo\}, ps \mapsto \{1\}]$. **End of Example**

Consider an application of the address operator &. Similar to an '`alloc()`' call, it "returns" a pointer to an object. To denote the set of objects the application "returns", we assume assume a unique labeling. Thus, in '`p = &`^{2}`x`' we have that '`p`' points to the same object as the "pointer" '2', that is, x.

Definition 4.2 *The set of* abstract locations ALoc *is defined inductively as follows:*

[8] Objects allocated by the means of 'malloc' are considered 'Unknown'.

- *If v is the name of a global variable:* $v \in \text{ALoc}$.
- *If v is a parameter of some function with n variants:* $v^i \in \text{ALoc}, i = 1, \ldots, n$.
- *If v is a local variable in some function with n variants:* $v^i \in \text{ALoc}, i = 1, \ldots, n$.
- *If s is a string constant:* $s \in \text{ALoc}$.
- *If f is the name of a function with n variants:* $f^i \in \text{ALoc}, i = 1, \ldots, n$.
- *If f is the name of a function with n variants:* $f_0^i \in \text{ALoc}, i = 1, \ldots, n$.
- *If l is the label of an alloc in a function with n variants:* $l^i \in \text{ALoc}, i = 1, \ldots, n$.
- *If l is the label of an address operator in a function with n variants:* $l^i \in \text{ALoc}$.
- *If $o \in$ ALoc denotes an object of type "array":* $o[] \in \text{ALoc}$.
- *If S is the type name of a struct or union type:* $S \in \text{ALoc}$.
- *If $S \in$ ALoc is of type "struct" or "union":* $S.i \in \text{ALoc}$ *for all fields i of S.*
- Unknown $\in$ ALoc.

Names are assumed to be unique. □

Clearly, the set ALoc is finite for all programs. The analysis maps a pointer into an element of the set $\wp(\text{ALoc})$. The element Unknown denotes an arbitrary (unknown) address. This means that the analysis abstracts as follows.

Function invocations are collapsed according to the program's static-call graph (see Chapter 2). This means for a function f with n variants, only n instances of parameters and local variables are taken into account. For instance, due to the 1-limit imposed on recursive functions, all instances of a parameters in a recursive function invocation chain are identified. The location f_0 associated with function f denotes an *abstract return location*, *i.e.* a unique location where f "delivers" its result value.

Arrays are treated as aggregates, that is, all entries are merged. Fields of struct objects of the same name are merged, *e.g.* given definition '`struct S { int x;} s,t`', fields '`s.x`' and '`t.x`' are collapsed.

Example 4.13 The merging of struct fields may seen excessively conservatively. However, recall that we assume programs are type-separated during parsing, and that a value-flow analysis is applied that identifier the type of struct objects that (may) flow together, see Section 2.3.3. **End of Example**

The unique abstract location Unknown denotes an arbitrary, unknown address, which both be valid or illegal.

Even though the definition of abstract locations actually is with respect to a particular program, we will continue to use ALoc independently of programs. Furthermore, we will assume that the type of the object, an abstract location denotes, is available. For example, we write "if $S \in$ ALoc is of struct type", for "if the object $S \in$ ALoc denotes is of struct type". Finally, we implicitly assume a binding from a function designator to the parameters. If f is a function identifier, we write $f : x_i$ for the parameter x_i of f.

4.4.2 Pointer abstraction

A *pointer abstraction* $\tilde{\mathcal{S}} : \text{ALoc} \to \wp(\text{ALoc})$ is a map from abstract locations to sets of abstract locations.

Example 4.14 Consider the following assignments.

```
int *p, *q;
extern int *ep;
p = (int *)0xabcd;
q = (int *)malloc(100*sizeof(int));
r = ep;
```

The pointer '`p`' is assigned a value via a non-portable cast. We will approximate this by Unknown. Pointer '`q`' is assigned the result of '`malloc()`'. In general, pointers returned by external functions are approximated by Unknown. Finally, the pointer '`r`' is assigned the value of an external variable. This is also approximated by Unknown.

A refinement would be to approximate the content of external pointers by a unique value Extern. Since we have no use for this, besides giving more accurate warning messages, we will not pursue this. **End of Example**

A pointer abstraction $\tilde{\mathcal{S}}$ must fulfill the following requirements which we justify below.

Definition 4.3 *A* pointer abstraction $\tilde{\mathcal{S}} : \text{ALoc} \to \wp(\text{ALoc})$ *is a map satisfying:*

1. *If* $o \in \text{ALoc}$ *is of base type:* $\tilde{\mathcal{S}}(o) = \{\text{Unknown}\}$.
2. *If* $s \in \text{ALoc}$ *is of struct/union type:* $\tilde{\mathcal{S}}(s) = \{\}$.
3. *If* $f \in \text{ALoc}$ *is a function designator:* $\tilde{\mathcal{S}}(f) = \{\}$.
4. *If* $a \in \text{ALoc}$ *is of type array:* $\tilde{\mathcal{S}}(a) = \{a[]\}$.
5. $\tilde{\mathcal{S}}(\text{Unknown}) = \text{Unknown}$.

□

The first condition requires that objects of base types are abstracted by Unknown. The motivation is that the value may be cast into a pointer, and is hence Unknown (in general). The second condition stipulates that the abstract value of a struct object is the empty set. Notice that a struct object is uniquely identified its type. The fourth condition requires that an array variable points to the content.[9] Finally, the content of an unknown location is unknown.

Define for $s \in \text{ALoc} \setminus \{\text{Unknown}\} : \{s\} \subseteq \{\text{Unknown}\}$. Then two pointer abstractions are ordered by set inclusion. A program has a minimal pointer abstraction. Given a program, we desire a minimal *safe* pointer abstraction.

[9] In reality, 'a' in 'a[10]' is not an lvalue. It is, however, convenient to consider 'a' to be a pointer to the content.

4.4.3 Safe pointer abstraction

Intuitively, a pointer abstraction for a program is *safe* if for all input, every object a pointer may point to at runtime is captured by the abstraction.

Let the abstraction function $\alpha : \text{Loc} \to \text{ALoc}$ be defined the obvious way. For example, if l_x is the location of parameter '`x`' in an invocation of a function '`f`' corresponding to the i'th variant, then $\alpha(l_x) = x^i$. An execution path from the initial program point p_0 and an initial program store $\mathcal{S}_0$ is denoted by

$$\langle p_0, \mathcal{S}_0 \rangle \to \cdots \to \langle p_n, \mathcal{S}_n \rangle$$

where $\mathcal{S}_n$ is the store at program point p_n.

Let p be a program and $\mathcal{S}_0$ an initial store (mapping the program input to the parameters of the goal function). Let p_n be a program point, and $\mathcal{L}_n$ the locations of all visible variables. A *pointer abstraction* $\tilde{\mathcal{S}}$ is *safe* with respect to p if

$$l \in \mathcal{L}_n : \alpha(\mathcal{S}_n(l)) \subseteq \tilde{\mathcal{S}}(\alpha(l))$$

whenever $\langle p_0, \mathcal{S}_0 \rangle \to \cdots \to \langle p_n, \mathcal{S}_n \rangle$.

Every program has a *safe* pointer abstraction. Define $\tilde{\mathcal{S}}_{triv}$ such that it fulfills Definition 4.3, and extend it such that for all $o \in \text{ALoc}$ where o is of pointer type, $\tilde{\mathcal{S}}_{triv}(o) = \{\text{Unknown}\}$. Obviously, it is a safe — and useless — abstraction.

The definition of safety above considers only monolithic programs where no external functions nor variables exist. We are, however, interested in analysis of translation units where parts of the program may be undefined.

Example 4.15 Consider the following piece of code.

```
extern int *o;
int *p, **q;
q = &o;
p = *q;
```

Even though '`o`' is an external variable, it can obviously be established that $[q \mapsto \{o\}]$. However, '`p`' must inevitably be approximated by $[p \mapsto \{\text{Unknown}\}]$. **End of Example**

Definition 4.4 *Let $p \equiv m_1, \ldots, m_m$ be a program consisting of the modules m_i. A pointer abstraction $\tilde{\mathcal{S}}$ is* safe *for m_{i_0} if for all program points p_n and initial stores $\mathcal{S}_0$ where $\langle p_0, \mathcal{S} \rangle \to \cdots \to \langle p_n, \mathcal{S}_n \rangle$, then:*

- *for $l \in \mathcal{L}_n$: $\alpha(\mathcal{S}_n(l)) \subseteq \tilde{\mathcal{S}}(\alpha(l))$ if l is defined in m_{i_0},*
- *for $l \in \mathcal{L}_n$: $\tilde{\mathcal{S}}(l) = \{\text{Unknown}\}$ if l is defined in $m_i \neq m_{i_0}$*

where $\mathcal{L}_n$ is the set of visible variables at program point n. □

For simplicity we regard $a[]$, given an array a, to be a "visible variable", and we regard the labels of '`alloc()`' calls to be "pointer variables.

Example 4.16 Suppose that we introduced an abstract location Extern to denote the contents of external variables. Example 4.15 would then be abstracted by: $[p \mapsto \text{Extern}]$. There is no operational difference between Extern and Unknown. **End of Example**

We will compute an approximation to a safe pointer abstraction. For example, we abstract the result of an implementation-defined cast, *e.g.* '`(int *)x`' where '`x`' is an integer variable, by Unknown, whereas the definition may allow a more accurate abstraction.

4.4.4 Pointer analysis specification

We specify a *flow-insensitive* (summary), *intra-procedural* pointer analysis. We postpone extension to inter-procedural analysis to Section 4.6.

The specification can be employed to check that a given pointer abstraction $\tilde{\mathcal{S}}$ is *safe* for a program. Due to lack of space we only present the rules for declarations and expressions (the interesting cases) and describe the other cases informally. The specification is in the form of inference rules $\tilde{\mathcal{S}} \vdash p : \bullet$.

We argue (modulo the omitted part of the specification) that if the program fulfills the rules in the context of a pointer abstraction $\tilde{\mathcal{S}}$, then $\tilde{\mathcal{S}}$ is a safe pointer abstraction. Actually, the rules will also fail if $\tilde{\mathcal{S}}$ is not a pointer abstraction, *i.e.* does not satisfy Definition 4.3. Let $\tilde{\mathcal{S}}$ be given.

Suppose that $d \equiv x : T$ is a definition (*i.e.*, not an 'extern' declaration). The safety of $\tilde{\mathcal{S}}$ with respect to d depends on the type T.

Lemma 4.1 *Let* $d \in \text{Decl}$ *be a definition. Then* $\tilde{\mathcal{S}} : \text{ALoc} \to \wp(\text{ALoc})$ *is a* pointer abstraction *with respect to* d *if*

$$\tilde{\mathcal{S}} \vdash^{pdecl} d : \bullet$$

where $\vdash^{pdecl}$ *is defined in Figure 35,* and $\tilde{\mathcal{S}}(\text{Unknown}) = \{\text{Unknown}\}$.

Proof It is straightforward to verify that Definition 4.3 is fulfilled. □

To the right of Figure 35 the rules for external variables are shown. Let $d \equiv x : T$ be an (extern) declaration. Then $\tilde{\mathcal{S}}$ is a *pointer abstraction* for d if $\tilde{\mathcal{S}} \vdash^{petype} \langle T, l \rangle : \bullet$. Notice the rules require external pointers to be approximated by Unknown, as stipulated by Definition 4.4.

The (omitted) rule for function definitions $T_f\ f(d_i)\{d_j\ S_k\}$ (would) require $\tilde{\mathcal{S}}(f) = \{f_0\}$.

Since we specify a flow-insensitive analysis, the safety of a pointer abstraction with respect to an expression e is independent of program points. A map $\tilde{\mathcal{S}} : \text{ALoc} \to \wp(\text{ALoc})$ is a *pointer abstraction* with respect to an expression e, if it is a pointer abstraction with respect to the variables occurring in e.

Lemma 4.2 *Let* $e \in \text{Expr}$ *be an expression and* $\tilde{\mathcal{S}}$ *a pointer abstraction with respect to* e. *Then* $\tilde{\mathcal{S}}$ *is* safe *provided there exist* $V \in \wp(\text{ALoc})$ *such*

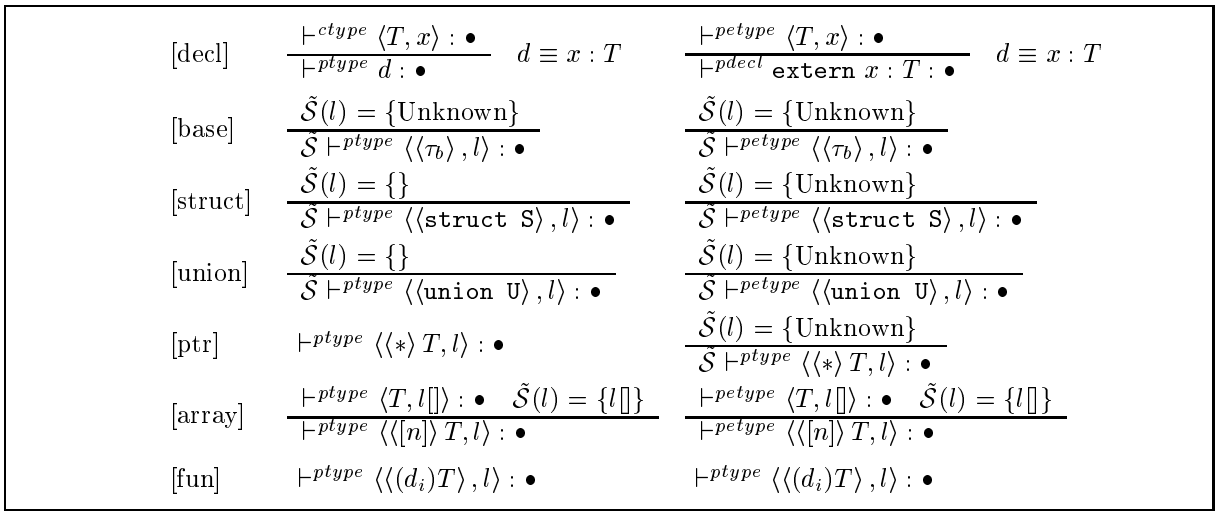

Figure 35: Pointer abstraction for declarations

$$\tilde{\mathcal{S}} \vdash^{pexp} e : V$$

where $\vdash^{pexp}$ *is defined in Figure 36.*

Intuitively, the the rules infer the *lvalues* of the expression e. For example, the lvalue of a variable v is $\{v\}$; recall that we consider intra-procedural analysis only.[10]

An informal justification of the lemma is given below. We omit a formal proof.

Justification A formal proof would be by induction after "evaluation length". We argue that if $\tilde{\mathcal{S}}$ is safe before evaluation of e, it is also safe after.

A constant has an Unknown lvalue, and the lvalue of a string is given by its name. The motivation for approximating the lvalue of a constant by Unknown, rather than the empty set, if obvious from the following example: '`p = (int *)12`'. The lvalue of a variable is approximated by its name.

Consider a struct indexing $e.i$. Given the type S of the objects the subexpression denotes, the lvalues of the fields are $S.i$. The rules for pointer dereference and array indexing use the pointer abstraction to describe the lvalue of the dereferenced objects. Notice: if '`p`' points to '`x`', that is $\tilde{\mathcal{S}}(p) = \{x\}$, when the lvalue of '`*p`' is the lvalue of '`x`' which is approximated by $\{x\}$. The rule for the address operator uses the label as a "placeholder" for the indirection created.

The effect of unary and binary operator applications is described by the means of $\tilde{O} : \text{Op} \times \wp(\text{ALoc})^* \to \wp(\text{ALoc})$. We omit a formal specification.

Example 4.17 Suppose that '`p`' and '`q`' both point to an array and consider pointer subtraction '`p - q`'.[11] We have $\tilde{O}(-_{*int,*int}, \{p\}, \{q\}) = \{\text{Unknown}\}$ since the result is an integer. Consider now '`p - 1`'. We then get $\tilde{O}(-_{*int,int}, \{p\}, \{\text{Unknown}\}) = \{p\}$ since pointer arithmetic is not allowed to shuffle a pointer outside an array. **End of Example**

[10] That is, there is one "variant" of each function.

[11] Recall that operator overloading is assumed resolved during parsing.

An external function delivers its result in an unknown location (and the result itself is unknown).

Consider the rules for functions calls. The content of the argument's abstract lvalue must be contained in the description of the formal parameters.[12] The result of the application is returned in the called function's abstract return location. In the case of indirect calls, all possible functions are taken into account.

Example 4.18 In case of the program fragment

```
int (*fp)(int), x;
fp = &foo;
fp = &bar;
(*fp)(&x)
```

where '`foo()`' and '`bar()`' are two functions taking an integer pointer as a parameter, we have:

$$[fp \mapsto \{foo, bar\}]$$

due to the first two applications of the address operator, and

$$[foo{:}x \mapsto \{x\}, bar{:}x \mapsto \{x\}]$$

due to the indirect call. The 'lvalue' of the call is $\{foo_0, bar_0\}$. **End of Example**

The rules for pre- and post increment expressions are trivial.

Consider the rule for assignments. The content of locations the left hand side must contain the content of the right hand side expression. Recall that we assume that no Unknown pointers are dereferenced.

Example 4.19 Consider the following assignments

```
extern int **q;
int *p;
*q = p;
```

Since '`q`' is extern, it is Unknown what it points to. Thus, the assignment may assign the pointer '`p`' to an Unknown object (of pointer type). This extension is shown in Section 4.3.3. **End of Example**

The abstract lvalue of a comma expression is determined by the second subexpression. A `sizeof` expression has no lvalue and is approximated by Unknown.

Finally, consider the rule for casts. It uses the function Cast : Type $\times$ Type $\times$ $\wp$(ALoc) $\to$ $\wp$(ALoc) defined as follows.

[12] Recall that we consider intra-procedural, or sticky analysis.

[const] $\tilde{\mathcal{S}} \vdash^{pexp} c : \{\text{Unknown}\}$

[string] $\tilde{\mathcal{S}} \vdash^{pexp} s : \{s\}$

[var] $\tilde{\mathcal{S}} \vdash^{pexp} v : \{v\}$

[struct] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \quad \text{TypOf}(o \in O_1) = \langle \text{struct S} \rangle}{\tilde{\mathcal{S}} \vdash^{pexp} e_1.i : \{S.i\}}$$

[indr] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} *e_1 : \bigcup_{o \in O_1} \tilde{\mathcal{S}}(o)}$$

[array] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}} \vdash^{pexpr} e_2 : O_2}{\tilde{\mathcal{S}} \vdash^{pexp} e_1[e_2] : \bigcup_{o \in O_1} \tilde{\mathcal{S}}(o)}$$

[address] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}}(l) \supseteq O_1}{\tilde{\mathcal{S}} \vdash^{pexp} \&^l e_1 : \{l\}}$$

[unary] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} o\ e_1 : \tilde{O}(o, O_1)}$$

[binary] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_i : O_i}{\tilde{\mathcal{S}} \vdash^{pexp} e_1\ op\ e_2 : \tilde{O}(o, O_i)}$$

[alloc] $\tilde{\mathcal{S}} \vdash^{pexp} \texttt{alloc}^l(T) : \{l\}$

[extern] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_i : O_i}{\tilde{\mathcal{S}} \vdash^{pexp} ef(e_1, \ldots, e_n) : \{\text{Unknown}\}}$$

[user] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_i : O_i \qquad \tilde{\mathcal{S}}(f{:}x_i) \supseteq \tilde{\mathcal{S}}(O_i)}{\tilde{\mathcal{S}} \vdash^{pexp} f(e_1, \ldots, e_n) : \tilde{\mathcal{S}}(f_0)}$$

[call] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_0 : O_0 \qquad \forall o \in O_0 : \tilde{\mathcal{S}}(o{:}x_i) \supseteq \tilde{\mathcal{S}}(O_i)}{\tilde{\mathcal{S}} \vdash^{pexp} e_0(e_1, \ldots, e_n) : \bigcup_{o \in O_0} \tilde{\mathcal{S}}(o_0)}$$

[preinc] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} {+}{+}e_1 : O_1}$$

[postinc] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} e_1{+}{+} : O_1}$$

[assign] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}} \vdash^{pexp} e_2 : O_2 \quad \forall o \in O_1 : \tilde{\mathcal{S}}(o) \supseteq \tilde{\mathcal{S}}(O_2)}{\tilde{\mathcal{S}} \vdash^{pexp} e_1\ aop\ e_2 : O_2}$$

[comma] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}} \vdash^{pexp} e_2 : O_2}{\tilde{\mathcal{S}} \vdash^{pexp} e_1, e_2 : O_2}$$

[sizeof] $\tilde{\mathcal{S}} \vdash^{pexp} \texttt{sizeof}(T) : \{\text{Unknown}\}$

[cast] $$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} (T)e_1 : \text{Cast}(T, \text{TypOf}(e_1), O_1)}$$

Figure 36: Pointer abstraction for expressions

$$\begin{array}{lll}
\text{Cast}(T_{to}, T_{from}, O_{from}) = \text{case } (T_{to}, T_{from}) \text{ of} & & \\
\quad (\langle \tau_b \rangle, \langle \tau_b' \rangle) & : O_{from} & \\
\quad (\langle * \rangle T, \langle \tau_b \rangle) & : \{\text{Unknown}\} & \\
\quad (\langle \tau_b \rangle, \langle * \rangle T) & : \{\text{Unknown}\} & \\
\quad (\langle * \rangle T, \langle * \rangle \langle \texttt{struct S} \rangle) & : \begin{cases} \{o.1 \mid o \in O_{from}\} & T \text{ type of first member of } S \\ O_{from} & \text{Otherwise} \end{cases} & \\
\quad (\langle * \rangle T', \langle * \rangle T'') & : O_{from} &
\end{array}$$

Casts between base types do not change an object's lvalue. Casts from a pointer type to an integral type, or the opposite, is implementation-defined, and approximated by Unknown.

Recall that a pointer to a struct object points, when suitably converted, also to the first member. This is implemented by the case for cast from struct pointer to pointer. We denote the name of the first member of S by '1'. Other conversions do not change the lvalue of the referenced objects. This definition is in accordance with the Standard [ISO 1990, Paragraph 6.3.4]. **End of Justification**

The specification of statements uses the rules for expressions. Further, in the case of a '`return` e':

$$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e : O \quad \tilde{\mathcal{S}}(f_0) \supseteq \tilde{\mathcal{S}}(O)}{\tilde{\mathcal{S}} \vdash^{pstmt} \texttt{return } e : \bullet}$$

which specifies that the abstract return location of function f (encapsulating the statement) must containt the value of the expression e.

We conjecture that given a program p and a map $\tilde{\mathcal{S}} : \text{ALoc} \to \wp(\text{ALoc})$, then $\tilde{\mathcal{S}}$ is a *safe pointer abstraction* for p iff the rules are fulfilled.

4.5 Intra-procedural pointer analysis

This section presents a *constraint-based* formulation of the pointer analysis specification. The next section extends the analysis to an *inter-procedural* analysis, and Section 4.7 describes constraint solving.

4.5.1 Pointer types and constraint systems

A *constraint system* is defined as a set of constraints over *pointer types*. A solution to a constraint system is a substitution from pointer type variables to sets of abstract locations, such that all constraints are satisfied.

The syntax of a *pointer type* T is defined inductively by the grammar

$$\begin{array}{lcll}
\mathcal{T} & ::= & \{o_j\} & \text{locations} \\
& | & *\mathcal{T} & \text{deference} \\
& | & \mathcal{T}.i & \text{indexing} \\
& | & (\mathcal{T}) \to \mathcal{T} & \text{function} \\
& | & T & \text{type variable}
\end{array}$$

where $o_j \in \text{ALoc}$ and i is an identifier. A pointer type can be a *set* of abstract locations, a *dereference type*, an *indexing* type, a *function* type, or a *type variable*. Pointer types $\{o_j\}$ are *ground types*. We use $\mathcal{T}$ to range over pointer types.

To every object $o \in \text{ALoc}$ of non-functional type we assign a type variable T_o; this includes the abstract return location f_0 for a function f. To every object $f \in \text{ALoc}$ of function type we associate the type $(T_d) \to T_{f_0}$, where T_d are the type variables assigned to parameters of f. To every type specifier τ we assign a type variable T_τ.

The aim of the analysis is to instantiate the type variables with an element from $\wp(\text{ALoc})$, such that the map $[o \mapsto T_o]$ becomes a safe pointer abstraction.

A *variable assignment* is a substitution $S : \text{TVar} \to \text{PType}$ from type variables to ground pointer types. Application of a substitution S to a type $\mathcal{T}$ is denoted by juxtaposition $S \cdot \mathcal{T}$. The *meaning* of a pointer type is defined relatively to a variable assignment.

Definition 4.5 *Suppose that S is a variable assignment. The* meaning *of a pointer type $\mathcal{T}$ is defined by*

$$\begin{array}{lcl}
[\![O]\!]\ S & = & O \\
[\![*\mathcal{T}]\!]\ S & = & \bigcup_o ST_o, \quad o \in [\![\mathcal{T}]\!]\ S \\
[\![\mathcal{T}.i]\!]\ S & = & \bigcup_o \{S(U.i) \mid \text{TypOf}(o) = \langle \text{struct U} \rangle\} \quad o \in [\![\mathcal{T}]\!]\ S \\
[\![(\mathcal{T}_i) \to \mathcal{T}]\!]\ S & = & ([\![\mathcal{T}_i]\!]\ S) \to [\![\mathcal{T}]\!]\ S \\
[\![T]\!]\ S & = & ST
\end{array}$$

where T_o is the unique type variable associated with object o. □

The meaning of a deference type $*\mathcal{T}$ is determined by the variable assignment. Intuitively, if $\mathcal{T}$ denotes objects $\{o_i\}$, the meaning is the contents of those objects: ST_{o_i}. In the case of an indexing $\mathcal{T}.i$, the meaning equals content of the fields of the object(s) $\mathcal{T}$ denote.

A *constraint system* is a multi-set of formal inclusion constraints

$$\mathcal{T} \supseteq \mathcal{T}$$

over pointer types $\mathcal{T}$. We use $\mathcal{C}$ to denote constraint systems.

A *solution* to a constraint system $\mathcal{C}$ is a *substitution* $S : \text{TVar} \to \text{PType}$ from type variables to ground pointer types which is the identity on variables but those occurring in $\mathcal{C}$, such that all constraints are satisfied.

Definition 4.6 *Define the relation $\supseteq^*$ by $O_1 \supseteq^* O_2$ iff $O_1 \supseteq O_2$ for all $O_1, O_2 \in \wp(\text{ALoc})$, and $(\mathcal{T}_i) \to \mathcal{T} \supseteq^* (\mathcal{T}_i') \to \mathcal{T}'$ iff $\mathcal{T}_i \supseteq^* \mathcal{T}_i'$ and $\mathcal{T}' \supseteq^* \mathcal{T}$.*

A substitution $S : \text{TVar} \to \text{PType}$ *solves a constraint $\mathcal{T}_1 \supseteq \mathcal{T}_2$ if it is a variable assignment and $[\![\mathcal{T}]\!]\ S \supseteq^* [\![\mathcal{T}]\!]\ S$.* □

Notice that a function type is contra-variant in the result type. The set of solutions to a constraint system $\mathcal{C}$ is denoted by $\text{Sol}(\mathcal{C})$. The constraint systems we will consider all have at least one solution.

Order solutions by subset inclusion. Then a constraint system has a minimal solution, which is a "most" accurate solution to the pointer analysis problem.

4.5.2 Constraint generation

We give a constraint-based formulation of the pointer analysis specification from the previous section.

Definition 4.7 *Let $p = \langle \mathcal{T}, \mathcal{D}, \mathcal{F} \rangle$ be a program. The pointer-analysis constraint system $\mathcal{C}_{pgm}(p)$ for p is defined by*

$$\mathcal{C}_{pgm}(p) = \bigcup_{t \in \mathcal{T}} \mathcal{C}_{tdef}(t) \cup \bigcup_{d \in \mathcal{D}} \mathcal{C}_{decl}(d) \cup \bigcup_{f \in \mathcal{F}} \mathcal{C}_{fun}(f) \cup \mathcal{C}_{goal}(p)$$

where the constraint generating functions are defined below. □

Below we implicitly assume that the constraint $T_{unknown} \supseteq \{\text{Unknown}\}$ is included in all constraint systems. It implements Condition 5 in Definition 4.3 of pointer abstraction.

Goal parameters

Recall that we assume that only a "goal" function is called from the outside. The content of the goal function's parameters is unknown. Hence, we define

$$\mathcal{C}_{goal}(p) = \bigcup \{T_x \supseteq \{\text{Unknown}\}\}$$

for the goal parameters $x : T$ of the goal function in p.

Example 4.20 For the main function '`int main(int argc, char **argv)`' we have:

$$\mathcal{C}_{goal} = \{T_{argc} \supseteq \{\text{Unknown}\}, T_{argv} \supseteq \{\text{Unknown}\}\}$$

since the content of both is unknown at program start-up. **End of Example**

Declaration

Let $d \in \text{Decl}$ be a declaration. The constraint system $\mathcal{C}_{decl}(d)$ for d is defined by Figure 37.

Lemma 4.3 *Let $d \in \text{Decl}$ be a declaration. Then $\mathcal{C}_{decl}(d)$ has a solution S, and*

$$S_{|\text{ALoc}} \vdash^{pdecl} d : \bullet$$

where $\vdash^{pdecl}$ is defined by Figure 35.

Proof To see that the constraint system $\mathcal{C}_{decl}(d)$ has a solution, observe that the trivial substitution S_{triv} is a solution.

It is easy to see that a solution to the constraint system is a pointer abstraction, cf. proof of Lemma 4.1. □

[decl]	$\dfrac{\vdash^{ctype} \langle T, x\rangle : T_t}{\vdash^{cdecl} x : T \ : \ T_x}$	$\{T_x \supseteq T_t\}$	$\dfrac{\vdash^{cetype} \langle T, x\rangle : T_t}{\vdash^{cdecl} \texttt{extern}\ x : T \ : \ T_x}$	$\{T_x \supseteq T_t\}$
[base]	$\vdash^{ctype} \langle \langle \tau_b \rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$	$\vdash^{cetype} \langle \langle \tau_b \rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$
[struct]	$\vdash^{ctype} \langle \langle \text{struct S} \rangle, l\rangle : T$	$\{T \supseteq \{\}\}$	$\vdash^{cetype} \langle \langle \text{struct S} \rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$
[union]	$\vdash^{ctype} \langle \langle \text{union U} \rangle, l\rangle : T$	$\{T \supseteq \{\}\}$	$\vdash^{cetype} \langle \langle \text{union U} \rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$
[ptr]	$\vdash^{ctype} \langle \langle * \rangle\, T', l\rangle : T$		$\vdash^{cetype} \langle \langle * \rangle\, T', l\rangle : T$	
[array]	$\dfrac{\vdash^{ctype} \langle T', l[]\rangle : T_1}{\vdash^{ctype} \langle \langle [n] \rangle\, T', l\rangle : T}$	$\{T \supseteq \{l[]\}\}$	$\dfrac{\vdash^{cetype} \langle T', l[]\rangle : T_1}{\vdash^{cetype} \langle [n] T', l\rangle : T}$	$\{T \supseteq \{l[]\}\}$
[fun]	$\dfrac{\vdash^{cdecl} d_i : T_{d_i}}{\vdash^{ctype} \langle \langle (d_i) \rangle\, T', l\rangle : T}$		$\dfrac{\vdash^{cdecl} d_i : T_{d_i}}{\vdash^{cetype} \langle \langle (d_i) \rangle\, T', l\rangle : T}$	

Figure 37: Constraint generation for declarations

[struct]	$\dfrac{\vdash^{cdecl} d_i : T}{\vdash^{ctdef} \texttt{struct}\ S\{\ d_i\ \} : \bullet}$
[union]	$\dfrac{\vdash^{cdecl} d_i : T}{\vdash^{ctdef} \texttt{union}\ U\{\ d_i\ \} : \bullet}$
[enum]	$\vdash^{ctdef} \texttt{enum}\ E\{e\} : \bullet$

Figure 38: Constraint generation for type definitions

Type definitions

The constraint generation for a type definition t, $\mathcal{C}_{tdef}(t)$, is shown in Figure 38.

Lemma 4.4 *Let $t \in TDef$ be a type definition. Then the constraint system $\mathcal{C}_{tdef}(t)$ has a solution S, and it is a pointer abstraction with respect to t.*

Proof Follows from Lemma 4.3. □

Example 4.21 To implement sharing of common initial members of unions, a suitable number of inclusion constraints are added to the constraint system. **End of Example**

Expressions

Let e be an expression in a function f. The constraint system $\mathcal{C}_{exp}(e)$ for e is defined by Figure 39.

The constraint generating function O_c for operators is defined similarly to O used in the specification for expressions. We omit a formal definition.

Example 4.22 For the application '`p - q`', where '`p`' and '`q`' are pointers, we have $O_c(-_{*int,*int}, T_e, T_{e_i}) = \{T_e \supseteq \{\text{Unknown}\}\}$. In the case of an application '`p - 1`', we have $O_c(-_{*int,int}, T_e, T_{e_i}) = \{T_e \supseteq T_{e_1}\}$, cf. Example 4.17. **End of Example**

[const]	$\vdash^{cexp} c : T_e$	$\{T_e \supseteq \{\text{Unknown}\}\}$
[string]	$\vdash^{cexp} s : T_e$	$\{T_e \supseteq \{s\}\}$
[var]	$\vdash^{cexp} v : T_e$	$\{T_e \supseteq \{v\}\}$
[struct]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} e_1.i : T_e}$	$\{T_e \supseteq T_{e_1}.i\}$
[indr]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} *e_1 : T_e}$	$\{T_e \supseteq *T_{e_1}\}$
[array]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1\,[e_2] : T_e}$	$\{T_e \supseteq *T_{e_1}\}$
[addr]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} \&^l e_1 : T_e}$	$\{T_e \supseteq \{l\}, T_l \supseteq T_e\}$
[unary]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} o\ e_1 : T_e}$	$O_c(o, T_e, T_{e_1})$
[binary]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1\ o\ e_2 : T_e}$	$O_c(o, T_e, T_{e_i})$
[ecall]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} ef(e_1, \ldots, e_n)}$	$\{T_e \supseteq \{\text{Unknown}\}\}$
[alloc]	$\vdash^{cexp} \texttt{alloc}^l\texttt{(}T\texttt{)} : T_e$	$\{T_e \supseteq \{T_l\}\}$
[user]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} f^l(e_1, \ldots, e_n) : T_e}$	$\{*\{f\} \supseteq (*T_{e_i}) \to T_l, T_e \supseteq \{l\}\}$
[call]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_0^l(e_1, \ldots, e_n) : T_e}$	$\{*T_{e_0} \supseteq (*T_{e_i}) \to T_l, T_e \supseteq \{l\}\}$
[pre]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} \texttt{++}e_1 : T_e}$	$\{T_e \supseteq T_{e_1}\}$
[post]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} e_1\texttt{++} : T_e}$	$\{T_e \supseteq T_{e_1}\}$
[assign]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1\ aop\ e_2 : T_e}$	$\{*T_{e_1} \supseteq *T_{e_2}, T_e \supseteq T_{e_2}\}$
[comma]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1, e_2 : T_e}$	$\{T_e \supseteq T_{e_2}\}$
[sizeof]	$\vdash^{cexp} \texttt{sizeof(}T\texttt{)} : T_e$	$\{T_e \supseteq \{\text{Unknown}\}\}$
[cast]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} (T)e_1 : T_e}$	$\text{Cast}_c(T, \text{TypOf}(e_1), T_e, T_{e_1})$

Figure 39: Constraint generation for expressions

To represent the lvalue of the result of a function application, we use a "fresh" variable T_l. For reasons to be seen in the next section, calls are assumed to be labeled.

The function Cast_c implementing constraint generation for casts is defined as follows.

$$
\begin{array}{lcl}
\mathrm{Cast}_c(T_{to}, T_{from}, T_e, T_{e_1}) & = & \text{case } (T_{to}, T_{from}) \text{ of} \\
(\langle \tau_b \rangle, \langle \tau_b \rangle) & : & \{T_e \supseteq T_{e_1}\} \\
(\langle * \rangle T, \langle \tau_b \rangle) & : & \{T_e \supseteq \{\mathrm{Unknown}\}\} \\
(\langle \tau_b \rangle, \langle * \rangle T) & : & \{T_e \supseteq \{\mathrm{Unknown}\}\} \\
(\langle * \rangle T, \langle * \rangle \langle \text{struct S} \rangle) & : & \begin{cases} \{T_e \supseteq T_{e_1}.1\} & T \text{ type of first member of } S \\ \{T_e \supseteq T_{e_1}\} & \text{Otherwise} \end{cases} \\
(\langle * \rangle T_1, \langle * \rangle T_2) & : & \{T_e \supseteq T_{e_1}\}
\end{array}
$$

Notice the resemblance with function Cast defined in Section 4.4.

Lemma 4.5 *Let $e \in$ Expr be an expression. Then $\mathcal{C}_{exp}(e)$ has a solution S, and*

$$S_{|\mathrm{ALoc}} \vdash^{pexp} e : V$$

where $\vdash^{pexp}$ is defined by Figure 36.

Proof To see that $\mathcal{C}_{exp}(e)$ has a solution, observe that S_{triv} is a solution.

That S is a safe pointer abstraction for e follows from definition of pointer types (Definition 4.5) and solution to constraint systems (Definition 4.6). □

Example 4.23 Consider the call '$\mathtt{f}^1\mathtt{(\&}^2\mathtt{x)}$'; a (simplified) constraint system is

$$\{T_{\&x} \supseteq \{2\}, T_2 \supseteq \{x\}, T_f \supseteq \{f\}, *T_f \supseteq (*T_{\&x}) \to T_1, T_{f()} \supseteq \{1\}\}$$

cf. Figure 39. By "natural" rewritings (see Section 4.7) we get

$$\{(T_{f_1}) \to T_{f_0} \supseteq (*\{2\}) \to T_1, T_{f()} \supseteq \{1\}\}$$

(where we have used that T_f is bound to $(T_{f_1}) \to T_{f_0}$) which can be rewritten to

$$\{(T_{f_1}) \to T_{f_0} \supseteq (T_2) \to T_1, T_{f()} \supseteq \{1\}\}$$

(where we have used that $*\{2\} \Rightarrow T_2$) corresponding to

$$\{T_{f_1} \supseteq \{x\}, T_{f()} \supseteq T_1\}$$

that is, the parameter of f may point to '$\mathtt{x}$', and f may return the value in location '1'. Notice that use of contra-variant in the last step. **End of Example**

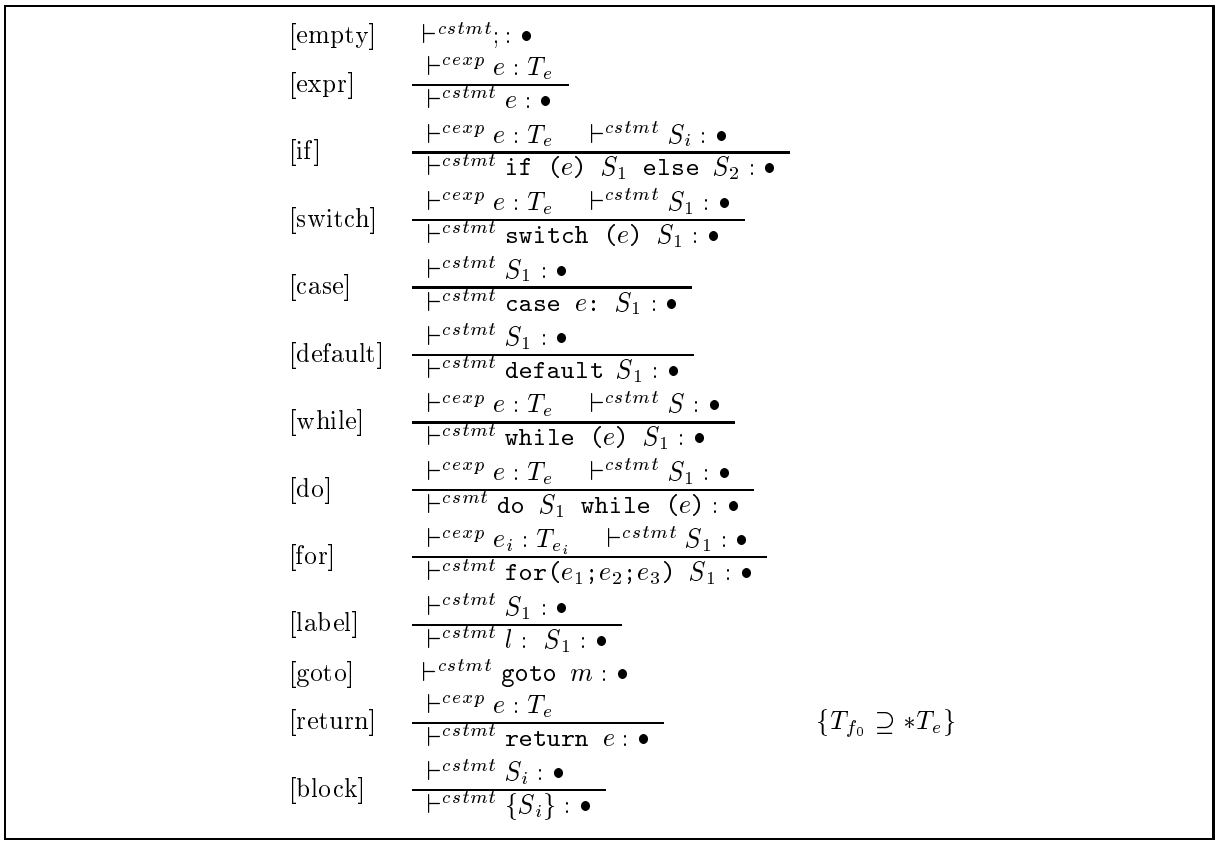

Figure 40: Constraint generation for statements

Statements

Suppose $s \in$ Stmt is a statement in a function f. The constraint system $\mathcal{C}_{stmt}(s)$ for s is defined by Figure 40.

The rules basically collect the constraints for contained expressions, and add a constraint for the `return` statement.

Lemma 4.6 *Let $s \in$ Stmt be a statement in function f. Then $\mathcal{C}_{stmt}(s)$ has a solution S, and S is a safe pointer abstraction for s.*

Proof Follows from Lemma 4.5. □

Functions

Let $f \in$ Fun be a function definition $f = \langle T, \mathcal{D}_{par}, \mathcal{D}_{loc}, \mathcal{S} \rangle$. Define

$$\mathcal{C}_{fun}(f) = \bigcup_{d \in \mathcal{D}_{par}} \mathcal{C}_{decl}(d) \cup \bigcup_{d \in \mathcal{D}_{loc}} \mathcal{C}_{decl}(d) \cup \bigcup_{s \in \mathcal{S}} \mathcal{C}_{stmt}(s)$$

where $\mathcal{C}_{decl}$ and $\mathcal{C}_{stmt}$ are defined above.

Lemma 4.7 *Let $f \in$ Fun be a function. Then $\mathcal{C}_{fun}(f)$ has a solution S, and S is a safe pointer abstraction for f.*

Proof Obvious. □

This completes the specification of constraint generation.

4.5.3 Completeness and soundness

Given a program p. We show that $\mathcal{C}_{pgm}$ has a solution and that the solution is a safe pointer abstraction.

Lemma 4.8 *Let p be a program. The constraint system $\mathcal{C}_{pgm}(p)$ has a solution.*

Proof The trivial solution S_{triv} solves $\mathcal{C}_{pgm}(p)$. □

Theorem 4.1 *Let p be a program. A solution $S \in \mathrm{Sol}(\mathcal{C}_{pgm}(p))$ is a* safe pointer abstraction *for p.*

Proof Follows from Lemma 4.7, Lemma 4.3 and Lemma 4.4. □

4.6 Inter-procedural pointer analysis

The intra-procedural analysis developed in the previous section sacrifices accuracy at functions calls: *all* calls to a function are merged. Consider for an example the following function:

```
/* inc_ptr: increment pointer p */
int *inc_ptr(int *q)
{
   return q + 1;
}
```

and suppose there are two calls '`inc_ptr(a)`' and '`inc_ptr(b)`', where '`a`' and '`b`' are pointers. The intra-procedural analysis merges the calls and alleges a call to '`inc_ptr`' yields a pointer to either '`a`' or '`b`'

With many calls to '`inc_ptr()`' spurious point-to information is propagated to unrelated call-sites, degrading the accuracy of the analysis. This section remedies the problem by extending the analysis into an *inter-procedural*, or *context-sensitive* point-to analysis.

4.6.1 Separating function contexts

The naive approach to inter-procedural analysis is by textual copying of functions before intra-procedural analysis. Functions called from different contexts are copied, and the call-sites changed accordingly. Copying may increase the size of the program exponentially, and henceforth also the generated constraint systems.

Example 4.24 Consider the following program.

```
int main(void)                  int *dinc(int *p)
{                               {
   int *pa,*pb,a[10],b[10];        int *p1 = inc_ptr(p);
   px = dinc(a);                   int *p2 = int_ptr(p1);
   py = dinc(b);                   return p2;
}                               }
```

Copying of function '`dinc()`' due to the two calls in '`main()`' will create two variants with 4 calls to '`int_ptr()`'. **End of Example**

The problem with textual copying of functions is that the analysis is slowed down due to the increased number of constraints, and worse, the copying may be useless: copies of function may be used in "similar" contexts such that copying does not enhance accuracy. Ideally, the cloning of functions should be based on the *result* of the analysis, such that only functions that gain from copying actually are copied.

Example 4.25 The solution to intra-procedural analysis of Example 4.24 is given below.

$$\begin{array}{lcl} T_{pa} & \mapsto & \{a, b\} \\ T_{pb} & \mapsto & \{a, b\} \\ T_{p} & \mapsto & \{a, b\} \\ T_{q} & \mapsto & \{a, b\} \end{array}$$

where the calls to '`dinc()`' have been collapsed. By copying of '`inc_ptr()`' four times, the pointers '`a`' and '`b`' would not be mixed up. **End of Example**

4.6.2 Context separation via static-call graphs

We employ the program's static-call graph to differentiate functions in different contexts. Recall that a program's static-call graph is a function $\mathcal{SCG}$: CallLabel $\times$ Variant $\rightarrow$ Id $\times$ Variant mapping a call-site and a variant number of the enclosing function to a function name and a variant. The static-call graph of the program in Example 4.24 is shown in Figure 41. Four variants of '`inc_ptr()`' exist due to the two call-sites in '`dinc()`' which again is called twice from '`main()`'.

Explicit copying of functions amounts to creating the variants as indicated by Figure 41. However, observe: the constraint systems generated for the variants are *identical* except for constraints for calls and `return`. The idea is to generate constraints over *vectors* of pointer types corresponding to the number of variants. For example, the constraint

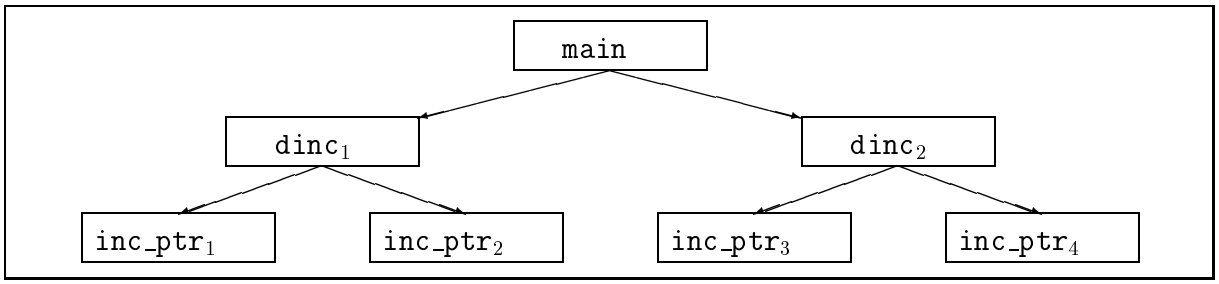

Figure 41: Static-call graph for the example program

system for '`inc_ptr()`' will use vectors of length 5, since there are four variants. Variant 0 is used as a *summary* variant, and for indirect calls.

After the analysis, procedure cloning can be accomplished on the basis of the computed pointer information. Insignificant variants can be eliminated and replaced with more general variants, or possibly with the summary variant 0.

4.6.3 Constraints over variant vectors

Let an *extended constraint system* be a multi-set of extended constraints

$$\mathcal{T}^n \supseteq \mathcal{T}^n$$

where $\mathcal{T}$ range over pointer types. Satisfiability of constraints is defined by component-wise extension of Definition 4.6.

Instead of assigning a single type variable to objects and expressions, we assign a *vector* $\overline{T}$ of type variables. The length is given as the number of variants of the encapsulating function (plus the 0 variant) or 1 in the case of global objects.

Example 4.26 Consider again the program in Example 4.24. Variable '`p`' of '`dinc`' is associated with the vector $p \mapsto \left\langle T_p^0, T_p^1, T_p^2 \right\rangle$ corresponding to variant 1 and 2, and the summary 0. The vector corresponding to the parameter of '`inc_ptr()`' has five elements due to the four variants. **End of Example**

The vector of variables associated with object o is denoted by $\overline{T_o} = \langle T_o^0, T_o^1, \ldots, T_o^n \rangle$. Similarly for expressions and types.

Example 4.27 An inter-procedural solution to the pointer analysis problem in Example 4.24:

$$\begin{array}{lcl}
\left\langle T_{pa}^0, T_{pa}^1 \right\rangle & \mapsto & \langle \{a\}, \{a\} \rangle \\
\left\langle T_{pb}^0, T_{pb}^1 \right\rangle & \mapsto & \langle \{b\}, \{b\} \rangle \\
\left\langle T_p^0, T_p^1, T_p^2 \right\rangle & \mapsto & \langle \{a,b\}, \{a\}, \{b\} \rangle \\
\left\langle T_q^0, T_q^1, T_q^2, T_q^3, T_q^4 \right\rangle & \mapsto & \langle \{a,b\}, \{a\}, \{a\}, \{b\}, \{b\} \rangle
\end{array}$$

where the context numbering is shown in Figure 41. **End of Example**

In the example above, it would be advantageous to merge variant 1, 2, and 3, 4, respectively.

4.6.4 Inter-procedural constraint generation

The inter-procedural constraint generation proceeds almost as in the intra-procedural analysis, Section 4.5.2, except in the cases of calls and `return` statements. Consider constraint generation in a function with n variants.

The rule for constants:

$$\{\overline{T_e} \supseteq \langle\{\text{Unknown}\}, \{\text{Unknown}\}, \ldots, \{\text{Unknown}\}\rangle\}$$

where the length of the vector is $n+1$. The rule for variable references:

$$\begin{array}{ll} \{\overline{T_e} \supseteq \langle\{v\}, \{v\}, \ldots, \{v\}\rangle\} & \text{if } v \text{ is global} \\ \{\overline{T_e} \supseteq \langle\{v^0\}, \{v^1\}, \ldots, \{v^n\}\rangle\} & \text{if } v \text{ is local} \end{array}$$

where v^i denote the i'th variant of object v. This rule seems to imply that there exists n versions of v. We describe a realization below. (The idea is that an object is uniquely identified by its associated variable, so in practice the rule reads $\overline{T_e} \supseteq \langle\{T_v^0\}, \{T_v^1\}, \ldots, \{T_v^n\}\rangle$.)

Consider a call $g^l(e_1, \ldots, e_m)$ in function f. The constraint system is

$$\bigcup_{i=1,\ldots,n} \{T_{g_j}^{k^i} \supseteq *T_{e_j}^i\} \cup \bigcup_{u=1,\ldots,n} \{T_{l^i} \supseteq T_{g_0}^{k^i}\} \cup \{\overline{T_e} \supseteq \overline{\{l^i\}}\}$$

where $\mathcal{SCG}(l,i) = \langle g, k^i\rangle$.

The rule is justified as follows. The i'th variant of the actual parameters are related to the corresponding variant k^i of the formal parameters, cf. $\mathcal{SCG}(l,i) = \langle g, k^i\rangle$. Similarly for the result. The abstract location l abstracts the lvalue(s) of the call.

The rule for an indirect call $e_0(e_1, \ldots, e_n)$ uses the summary nodes:

$$\{*T_{e_0}^0 \supseteq (*T_{e_i}^0) \to T_l^0, \overline{T_e} \supseteq \left\langle\{l^0\}, \{l^0\}, \ldots, \{l^0\}\right\rangle\}$$

cf. the rule for intra-procedural analysis. Thus, no context-sensitivity is maintained by indirect calls.

Finally, for every definition '$x : T$' that appears in n variants, the constraints

$$\bigcup_{i=1,\ldots,n} \{T_x^0 \supseteq T_x^i\}$$

are added. This assures that variant 0 of a type vector summarizes the variants.

Example 4.28 The first call to '`inc_ptr()`' in Example 4.24 gives rise to the following constraints.

$$\begin{array}{ll} T_{inc\_ptr}^1 \supseteq T_{dinc}^1, T_p^1 \supseteq T_{dinc_0}^1 & \text{variant 1} \\ T_{inc\_ptr}^2 \supseteq T_{dinc}^2, T_p^2 \supseteq T_{dinc_0}^2 & \text{variant 2} \end{array}$$

where we for the sake of presentation have omitted "intermediate" variables, and rewritten the constraints slightly. **End of Example**

A constraint system for inter-procedural analysis consists of only a few more constraints than in the case of intra-procedural analysis. This does not mean, naturally, that a inter-procedural solution can be found in the same time as an intra-procedural solution: the processing of each constraint takes more time. The thesis is the processing of an extended constraint takes less time than processing of an increased number of constraints.

4.6.5 Improved naming convention

As a side-efect, the inter-procedural analysis improves on the accuracy with respect to heap-allocated objects. Recall that objects allocated from the same call-site are collapsed.

The constraint generation in the inter-procedural analysis for '`alloc`^{l}`()`' calls is

$$\{\overline{T_e} \supseteq \left\langle \{l^0\}, \{l^1\}, \ldots, \{l^n\} \right\rangle\}$$

where l_i are $n+1$ "fresh" variables.

Example 4.29 An intra-procedural analysis merges the objects allocated in the program below even though they are unrelated.

```
int main(void)                    struct S *allocate(void)
{                                 {
   struct S *s = allocate();         return alloc¹(S);
   struct S *t = allocate();      }
}
```

The inter-procedural analysis creates two variants of '`allocate()`', and separates apart the two invocations. **End of Example**

This gives the analysis the same accuracy with respect to heap-allocated objects as other analyses, *e.g.* various invocations of a function is distinguished [Choi *et al.* 1993].

4.7 Constraint solving

This section presents a set of solution-preserving rewrite rules for constraint systems. We show that repeated application of the rewrite rules brings the system into a form where a solution can be found easily. We argue that this solution is minimal.

For simplicity we consider intra-procedural constraints only in this section. The extension to inter-procedural systems is straightforward: pairs of types are processed component-wise. Notice that the same number of type variables always appear on both sides of a constraint. In practice, a constraint is annotated with the length of the type vectors.

4.7.1 Rewrite rules

Let $\mathcal{C}$ be a constraint system. The application of rewrite rule l resulting in system $\mathcal{C}'$ is denoted by $\mathcal{C} \Rightarrow^l \mathcal{C}'$. Repeated application of rewrite rules is written $\mathcal{C} \Rightarrow \mathcal{C}'$. Exhausted application[13] is denoted by $\mathcal{C} \Rightarrow^* \mathcal{C}'$ (we see below that exhausted application makes sense).

A rewrite rule l is *solution preserving* if a substitution S is a solution to $\mathcal{C}$ if and only if it is a solution to $\mathcal{C}'$, when $\mathcal{C} \Rightarrow^l \mathcal{C}'$. The aim of constraint rewriting is to propagate point-to sets through the type variables. The rules are presented in Figure 42, and make use of an auxiliary function Collect : TVar $\times$ CSystem $\rightarrow \wp$(ALoc) defined as follows.

[13] Application until the system stabilizes.

Type normalization			
1.a	$\mathcal{C} \equiv \mathcal{C}' \cup \{T \supseteq \{s\}.i\}$	$\Rightarrow$	$\mathcal{C} \cup \{T \supseteq T_{S_i}\}$ $\quad \text{TypOf}(s) = \langle \text{struct S} \rangle$
1.b	$\mathcal{C} \equiv \mathcal{C}' \cup \{T \supseteq *\{o\}\}$	$\Rightarrow$	$\mathcal{C} \cup \{T \supseteq T_o\}$
1.c	$\mathcal{C} \equiv \mathcal{C}' \cup \{*\{o\} \supseteq \mathcal{T}\}$	$\Rightarrow$	$\mathcal{C} \cup \{T_0 \supseteq \mathcal{T}\}$ $\quad o \mapsto T_o$
1.d	$\mathcal{C} \equiv \mathcal{C}' \cup \{(T_i) \to T \supseteq (\mathcal{T}_i') \to T'\}$	$\Rightarrow$	$\mathcal{C} \cup \{T_i \supseteq \mathcal{T}_i', T' \supseteq T\}$
Propagation			
2.a	$\mathcal{C} \equiv \mathcal{C}' \cup \{T_1 \supseteq T_2\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \text{Collect}(T_2, \mathcal{C})} \{T_1 \supseteq \{o\}\}$
2.b	$\mathcal{C} \equiv \mathcal{C}' \cup \{T_1 \supseteq T_2.i\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \text{Collect}(T_2, \mathcal{C})} \{T \supseteq \{o\}.i\}$
2.c	$\mathcal{C} \equiv \mathcal{C}' \cup \{T_1 \supseteq *T_2\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \text{Collect}(T_2, \mathcal{C})} \{T_1 \supseteq *\{o\}\}$
2.d	$\mathcal{C} \equiv \mathcal{C}' \cup \{*T \supseteq \mathcal{T}\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \text{Collect}(T, \mathcal{C})} \{*\{o\} \supseteq \mathcal{T}\}$

Figure 42: Solution preserving rewrite rules

Definition 4.8 *Let $\mathcal{C}$ be a constraint system. The function* $\text{Collect} : \text{TVar} \times \text{CSystem} \to \wp(\text{ALoc})$ *is defined inductively by:*

$$\text{Collect}(T, \mathcal{C}) = \{o_i \mid T \supseteq \{o_i\} \in \mathcal{C}\} \cup \{o \mid T \supseteq T_1 \in \mathcal{C}, o_i \in \text{Collect}(T_1, \mathcal{C})\}$$

□

Notice that constraints may be self-dependent, *e.g.* a constraint system may contain constraints $\{T_1 \supseteq T_2, T_2 \supseteq T_1\}$.

Lemma 4.9 *Let $\mathcal{C}$ be a constraint system and suppose that T is a variable appearing in $\mathcal{T}$. Then* $\text{Sol}(\mathcal{C}) = \text{Sol}(\mathcal{C} \cup \{T \supseteq \text{Collect}(T, \mathcal{C})\})$.

Proof Obvious. □

For simplicity we have assumed abstract location sets $\{o\}$ consist of one element only. The generalization is straightforward. Constraints of the form $\{o\}.i \supseteq \mathcal{T}$ can never occur; hence no rewrite rule.

Lemma 4.10 *The rules in Figure 42 are solution preserving.*

Proof Assume that $\mathcal{C}_l \Rightarrow^l \mathcal{C}_r$. We show: S is a solution to $\mathcal{C}$ iff it is a solution to $\mathcal{C}'$.
Cases 1: The rules follow from the definition of pointer types (Definition 4.5). Observe that due to static well-typedness, "s" in rule 1.a denotes a struct object.
Case 2.a: Due to Lemma 4.9.
Case 2.b: Suppose that S is a solution to $\mathcal{C}_l$. By Lemma 4.9 and definition of pointer types, S is a solution to $\mathcal{C}_l \cup \{T_1 \supseteq \{o\}.i\}$ for $o \in \text{Collect}(T_2, \mathcal{C}_l)$. Suppose that S' is a solution to $\mathcal{C}_r$. By Lemma 4.9, S' is a solution to $\mathcal{C}_r \cup \{T_2 \supseteq \{o\}\}$ for $o \in \text{Collect}(T_2, \mathcal{C}_r)$.
Case 2.c: Similar to case 2.b.
Case 2.d: Similar to case 2.b. □

Lemma 4.11 *Consider a constraint system to be a* set *of constraint. Repeated application of the rewrite rules in Figure 42* $\mathcal{C} \Rightarrow \mathcal{C}'$ *terminates.*

Proof All rules add constraints to the system. This can only be done a finite number of times. □

Thus, when considered as a set, a constraint system $\mathcal{C}$ has a *normal form* $\mathcal{C}'$ which can be found by exhaustive application $\mathcal{C} \Rightarrow \mathcal{C}'$ of the rewrite rules in Figure 42.

Constraint systems in normal form have a desirable property: a solution can be found directly.

4.7.2 Minimal solutions

The proof of the following theorem gives a constructive (though inefficient) method for finding a minimal solution to a constraint system.

Theorem 4.2 *Let* $\mathcal{C}$ *be a constraint system. Perform the following steps:*

1. *Apply the rewrite rules in Figure 42 until the system stabilizes as system* $\mathcal{C}'$.
2. *Remove from* $\mathcal{C}'$ *all constraints but constraints of the form* $T \supseteq \{o\}$ *giving* $\mathcal{C}''$.
3. *Define the substitution* S *by* $S = [T \mapsto \text{Collect}(T, \mathcal{C}'')]$ *for all* T *in* $\mathcal{C}''$.

Then $S_{|\text{ALoc}} \in \mathit{Sol}(\mathcal{C})$*, and* S *is a minimal solution.*

Proof Due to Lemma 4.10 and Lemma 4.9 it suffices to show that S is a solution to $\mathcal{C}'$.

Suppose that S is not a solution to $\mathcal{C}'$. Clearly, S is a solution to the constraints added during rewriting: constraints generated by rule 2.b are solved by 1.a, 2.c by 1.b, and 2.d by 1.c. Then there exists a constraint $c \in \mathcal{C} \setminus \mathcal{C}'$ which is not satisfied. Case analysis:

- $c = T_1 \supseteq \{o\}$: Impossible due to Lemma 4.9.
- $c = T_1 \supseteq T_2$: Impossible due to exhaustive application of rule 2.a and Lemma 4.9.
- $c = T_1 \supseteq T_2.i$: Impossible due to rewrite rule 2.b and Lemma 4.9.
- $c = T_1 \supseteq *T_2$: Impossible due to rewrite rule 2.c and Lemma 4.9.
- $c = *T_1 \supseteq \mathcal{T}$: Impossible due to rewrite rules 2.d and 1.c, and Lemma 4.9.

Hence, S is a solution to $\mathcal{C}'$.

To see that S is minimal, notice that no inclusion constraints $T_1 \supseteq \{o\}$ than needed are added; thus S must be a minimal solution. □

The next section develops an iterative algorithm for pointer analysis.

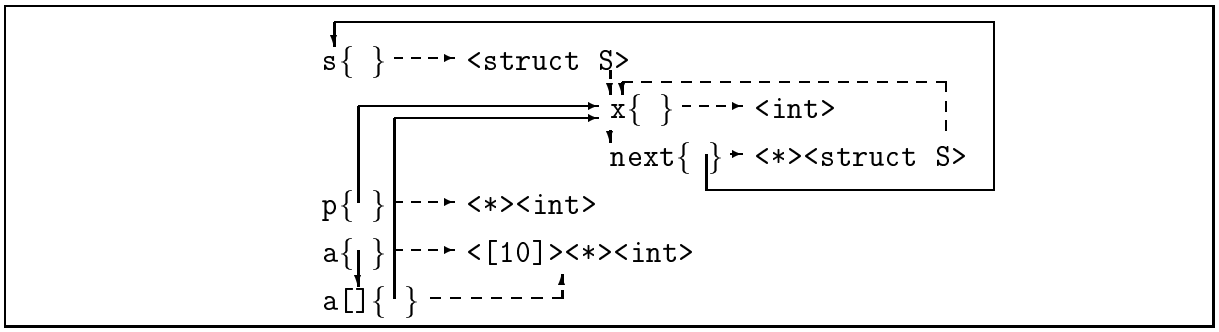

Figure 43: Pointer type representation

4.8 Algorithm aspects

In this section we outline an algorithm for pointer analysis. The algorithm is similar to classical iterative fixed-point solvers [Aho *et al.* 1986,Kildall 1973]. Further, we describe a convenient representation.

4.8.1 Representation

To every declarator in the program we associate a pointer type. For abstract locations that do not have a declarator, *e.g.* $a[]$ in the case of an array definition '`int a[10]`', we create one. A object is uniquely identified by a pointer to the corresponding declarator. Thus, the constraint $T \supseteq *\{o\}$ is represented as $T \supseteq *\{T_o\}$ which can be rewritten into $T \supseteq T_o$ in constant time.

Example 4.30 The "solution" to the pointer analysis problem of the program below is shown in Figure 43.

```
struct S { int x; struct S *next; } s;
int *p, *a[10];
s.next = &s;
p = a[1] = &s.x;
```

The dotted lines denotes representation of static types.

End of Example

To every type variable '`T`' we associate a set '`T.incl`' of (pointers to) declarators. Moreover, a boolean flag '`T.upd`' is assumed for each type variable. The field '`T.incl`' is incrementally updated with the set of objects '`T`' includes. The flag '`T.upd`' indicates whether a set has changed since "last inspection".

4.8.2 Iterative constraint solving

Constraints of the form $T \supseteq *\{T_o\}$ can be "pre-normalized" to $T \supseteq T_o$ during constraint generation, and hence do not exists during the solving process. Similar for constraint generated for user-function call.

The constraint solving algorithm is given as Algorithm 4.1 below.

Algorithm 4.1 *Iterative constraint solving.*

```
do
   fix = 1;
   for (c in clist)
      switch (c) {
         case T1 ⊇ 0: update(T1,0); break;
         case T1 ⊇ T2: update(T1,T2.incl); break;
         case T1 ⊇ T2.i: update(T1,struct(T2.incl,i)); break;
         case T1 ⊇ *T2:
            update(T1,indr(T2.incl)));
            if (Unknown in T2.incl) abort("Unknown dereferenced");
            break;
         case *T1 ⊇ *T2:
            if (T1.upd || T2.upd) {
               for (o in T1.incl)
                  update(To,indr(T2.incl));
               }
            break;
         case *T0 ⊇ (*T'i)->T':
            if (T0.upd) {
               for ((Ti)->T in T0.incl)
                  clist ∪= { Ti ⊇ *T'i, T' ⊇ T };
               }
             break;
      }
while (!fix);

/* update: update content T.incl with O */
update(T,0)
{
   if (T.incl ⊈ 0) { T.incl ∪= 0; fix = 0; }
}
```

Functions '`indr()`*' and '*`struct()`*' are defined the obvious way. For example, '*`indr()`*' dereferences (looks up the binding of) a declarator pointer (location name) and returns the point-to set.* □

Notice case for pointer deference. If Unknown is dereferenced, the algorithm aborts with a "worst-case" message. This is more strict than needed. For example, the analysis yields "worst-case" in the case of an assignment '`p = *q`', where '`q`' is approximated by Unknown. In practice, constraints appearing at the left hand side of assignments are "tagged", and only those give rise to abortion.

4.8.3 Correctness

Algorithm 4.1 terminates since the '`incl`' fields only can be update a finite number of times. Upon termination, the solution is given by $S = [T \mapsto \texttt{T.incl}]$.

Lemma 4.12 *Algorithm 4.1 is correct.*

Proof The algorithm implements the rewrite rules in Figure 42. □

4.8.4 Complexity

Algorithm 4.1 is polynomial in the size of program (number of declarators). It has been shown that inter-procedural may-alias in the context of multi-level pointers is P-space hard [Landi 1992a].[14] This indicates the degree of approximation our analysis make. On the other hand, it is fast and the results seem reasonable.

4.9 Experiments

We have implemented a pointer analysis in the *C-Mix* system. The analysis is similar to the one presented in this chapter, but deviates it two ways: it uses a representation which reduces the number of constraints significantly (see below), and it computes summary information for all indirections of pointer types.

The former decreases the runtime of the analysis, the latter increases it. Notice that the analysis of this chapter only computes the lvalues of the first indirection of a pointer; the other indirections must be computed by inspections of the objects to which a pointer may point.

The value of maintaining summary information for all indirections depends on the usage of the analysis. For example with summary information for all indirections, the side-effect analysis of Chapter 6 does not need to summarize pointers at every indirection node; this is done during pointer analysis. On the other hand, useless information may be accumulated. We suspect that the analysis of this chapter is more feasible in practice, but have at the time of writing no empirical evidence for this.

We have applied the analysis to some test programs. All experiments were conducted on a Sun SparcStation II with 64 Mbytes of memory. The results are shown below. We refer to Chapter 9 for a description of the programs.

Program	Lines	Constraints	Solving
Gnu strstr	64	17	$\approx$ 0.0 sec
Ludcmp	67	0	0.0 sec
Ray tracer	1020	157	0.3 sec
ERSEM	$\approx$ 5000	465	3.3 sec

As can be seen, the analysis is fast. It should be stressed, however, that none of the programs use pointers extensively. Still, we believe the analysis of this chapter will exhibit comparable run times in practice. The quality of the inferred information is good. That is, pointers are approximated accurately (modulo flow-insensitivity). In average, the points-to sets for a pointer are small.

Remark: The number of constraints reported above seems impossible! The point is that most of the superset constraints generated can be solved by equality. All of these constraints are pre-normalized, and hence the constraint system basically contains only constraints for assignments (calls) involving pointers.

[14] This has only been shown for programs exhibiting more than four levels of indirection.

4.10 Towards program-point pointer analysis

The analysis developed in this chapter is flow-insensitive: it produces a summary for the entire body of a function. This benefits efficiency at the price of precision, as illustrated by the (contrived) function to the left.

```
int foo(void)
{
   if (test) {
      p = &x;
      foobar(p);
   } else {
      p = &y;
      foobar(p);
   }
}
```

```
int bar(void)
{
   p = &x;
   foobar(p);
   p = &y;
   foobar(p);
 }
```

The analysis ignores the branch and information from one branch may influence the other. In this example the loss of accuracy is manifested by the propagation of the point-to information $[p \mapsto \{x, y\}]$ to *both* calls.

The example to the right illustrates lack of program-point specific information. A program-point specific analysis will record that '`p`' will point to '`x`' at the first call, and to '`y`' at the second call. In this section we consider program-point specific, flow-sensitive pointer analysis based on constraint solving.

4.10.1 Program point is sequence point

The aim is to compute a pointer abstraction for each program point, mapping pointers to the sets of objects they may point to at that particular program point. Normally, a program point is defined to be "between two statements", but in the case of C, the notion coincides with *sequence points* [ISO 1990, Paragraph 5.1.2.3]. At a sequence point, all side-effects between the previous and the current point shall have completed, *i.e.* the store updated, and no subsequent side-effects have taken place. Further, an object shall be accessed at most once to have its value determined. Finally, an object shall be accessed only to determine the value to be stored. The sequence points are defined in Annex C of the Standard [ISO 1990].

Example 4.31 The definition renders undefined an expression such as '`p = p++ + 1`' since '`p`' is "updated" twice between two sequence points.

Many analyses rely on programs being transformed into a simpler form, *e.g.* 'e_1 `=` e_2 `=` e_3' to 'e_2 `=` e_3`;` e_1 `=` e_2'. This introduces new sequence points and may turn an undefined expression into a defined expression, for example '`p = q = p++`'. **End of Example**

In the following we for simplicity ignore sequence points in expressions, and use the convention that if S is a statement, then m is the program immediately before S, and n is the program after. For instance, for a sequence of statements, we have $m_1 S_1 n_1 m_2 S_2 n_2 \ldots m_n S_n n_n$.

4.10.2 Program-point constraint-based program analysis

This section briefly recapitulates constraint-based, or set-based program analysis of imperative language, as developed by Heintze [Heintze 1992].

To every program points m, assign a vector of type variables $\overline{T}^m$ representing the abstract store.

Example 4.32 Below the result of a program-point specific analysis is shown.

```
int main(void)
{
   int x, y, *p;
   /* 1: ⟨T_x^1, T_y^1, T_p^1⟩ ↦ ⟨{}, {}, {}⟩ */
   p = &x;
   /* 2: ⟨T_x^2, T_y^2, T_p^2⟩ ↦ ⟨{}, {}, {x}⟩ */
   p = &y;
   /* 3: ⟨T_x^3, T_y^3, T_p^3⟩ ↦ ⟨{}, {}, {y}⟩ */
   x = 4;
   /* 4: ⟨T_x^4, T_y^4, T_p^4⟩ ↦ ⟨{}, {}, {y}⟩ */

}
```

Notice that T_p^3 does not contain $\{x\}$. **End of Example**

The corresponding constraint systems resemble those introduced in Section 4.5. However, extra constraints are needed to *propagate* the abstract state through the program points. For example, at program point 4, the variable T_p^4 assumes the same value as T_p^3, since it is not updated.

Example 4.33 Let $\overline{T}^n \supseteq \overline{T}^m[x \mapsto O]$ be a short hand for $T_o^n \supseteq T_o^m$ for all o except x, and $T_x^n \supseteq O$. Then the following constraints abstracts the pointer usage in the previous example:

$$
\begin{array}{ll}
2: & \overline{T}^2 \supseteq \overline{T}^1[p \mapsto \{x\}] \\
3: & \overline{T}^3 \supseteq \overline{T}^2[p \mapsto \{y\}] \\
4: & \overline{T}^4 \supseteq \overline{T}^3[x \mapsto \{\}]
\end{array}
$$

End of Example

The constraint systems can be solved by the rewrite rules in Figure 42, but unfortunately the analysis cannot cope with multi-level pointers.

4.10.3 Why Heintze's set-based analysis fails

Consider the following program fragment.

```
int x, y, *p, **q;
/* 1: ⟨T_x^1, T_y^1, T_p^1, T_q^1⟩ ↦ ⟨{}, {}, {}, {}⟩ */
p = &x;
/* 2: ⟨T_x^2, T_y^2, T_p^2, T_q^2⟩ ↦ ⟨{}, {}, {x}, {}⟩ */
q = &p;
/* 3: ⟨T_x^3, T_y^3, T_p^3, T_q^3⟩ ↦ ⟨{}, {}, {x}, {p}⟩ */
*q = &y;
/* 4: ⟨T_x^4, T_y^4, T_p^4, T_q^4⟩ ↦ ⟨{}, {}, {y}, {p}⟩ */
```

The assignment between program point 3 and 4 updates the abstract location p, but '`p`' does not occur syntactically in the expression '`*q = &y`'. Generating the constraints $\overline{T}^4 \supseteq \overline{T}^3[*T_q^3 \mapsto \{y\}]$ will incorrectly leads to $T_p^4 \supseteq \{x, y\}$.

There are two problems. First, the values to be propagated through states are not *syntactically* given by an expression, *e.g.* that '`p`' will be updated between program points 3 and 4. Secondly, the indirect assignment will be modeled by a constraint of the form $*T_q^4 \supseteq \{y\}$ saying that the indirection of '`q`' (that is, '`p`') should be updated to contain '`y`'. However, given $T_q^4 \mapsto \{p\}$, it is not apparent from the constraint that $*T_q^4 \supseteq \{y\}$ should be rewritten to $T_p^4 \supseteq \{y\}$; program points are not a part of a constraint (how is the "right" type variable for '`p`' chosen?).

To solve the latter problem, constraints generated due to assignments can be equipped with program points: $\mathcal{T}^n \supseteq^m \mathcal{T}$ meaning that program point n is updated from state m. For example, $*T_q^{4\ 4} \supseteq^3 \{y\}$ would be rewritten to $T_p^4 \supseteq \{y\}$, since $T_q^4 \mapsto \{p\}$, and the update happens at program point 4.

The former problem is more intricate. The variables *not* to be updated depend on the solution to T_q^4. Due to loops in the program and self-dependences, the solution to T_q^4 may depend on the variables propagated through program points 3 and 4.

Currently, we have no good solution to this problem.

4.11 Related work

We consider three areas of related work: alias analysis of Fortran and C, the point-to analysis developed by Emami which is the closest related work, and approximation of heap-allocated data structures.

4.11.1 Alias analysis

The literature contains much work on alias analysis of Fortran-like languages. Fortran differs from C in several aspects: dynamic aliases can only be created due to reference parameters, and program's have a purely static call graph.

Banning devised an efficient inter-procedural algorithm for determining the set of aliases of variables, and the side-effects of functions [Banning 1979]. The analysis has two steps. First all trivial aliases are found, and next the alias sets are propagated through the call graph to determine all non-trivial aliases. Cooper and Kennedy improved the complexity of the algorithm by separating the treatment of global variables from reference

parameters [Cooper and Kennedy 1989]. Chow has designed an inter-procedural data flow analysis for general single-level pointers [Chow and Rudmik 1982].

Weihl has studied inter-procedural flow analysis in the presence of pointers and procedure variables [Weihl 1980]. The analysis approximates the set of procedures to which a procedure variable may be bound to. Only single-level pointers are treated which is a simpler problem than multi-level pointers, see below. Recently, Mayer and Wolfe have implemented an inter-procedural alias analysis for Fortran based on Cooper and Kennedy's algorithm, and report empirical results [Mayer and Wolfe 1993]. They conclude that the cost of alias analysis is cheap compared to the possible gains. Richardson and Ganapathi have conducted a similar experiment, and conclude that aliases only rarely occur in "realistic" programs [Richardson and Ganapathi 1989]. They also observe that even though inter-procedural analysis theoretically improves the precision of traditional data flow analyses, only a little gain is obtained in actual runtime performance.

Bourdoncle has developed an analysis based on abstract interpretation for computing assertions about scalar variables in a language with nested procedures, aliasing and recursion [Bourdoncle 1990]. The analysis is somewhat complex since the various aspects of interest are computed in parallel, and are not been factored out. Larus *et al.* used a similar machinery to compute inter-procedural alias information [Larus and Hilfinger 1988]. The analysis proceeds by propagating alias information over an extended control-flow graph. Notice that this approach requires the control-flow graph to be statically computable, which is not the case with C. Sagiv *et al.* computes pointer equalities using a similar method [Sagiv and Francez 1990]. Their analysis tracks both universal (must) and existential (may) pointer equalities, and is thus more precise than our analysis. It remains to extend these methods to the full C programming language. Harrison *et al.* use abstract interpretation to analyze program in an intermediate language Mil into which C programs are compiled [Harrison III and Ammarguellat 1992]. Yi has developed a system for automatic generation of program analyses [Yi 1993]. It automatically converts a specification of an abstract interpretation into an implementation.

Landi has developed an inter-procedural alias analysis for a subset of the C language [Landi and Ryder 1992,Landi 1992a]. The algorithm computes flow-sensitive, conditional may-alias information that is used to approximate inter-procedural aliases. The analysis cannot cope with casts and function pointers. Furthermore, its performance is not impressive: 396s to analyze a 3.631 line program is reported.[15] Choi *el al.* have improved on the analysis, and obtained an algorithm that is both more precise and efficient. They use a naming technique for heap-allocated objects similar to the one we have employed. Cytron and Gershbein have developed a similar algorithm for analysis of programs in static single-assignment form [Cytron and Gershbein 1993].

Landi has shown that the problem of finding aliases in a language with more than four levels of pointer indirection, runtime memory allocation and recursive data structures is P-space hard [Landi and Ryder 1991,Landi 1992a]. The proof is by reduction of the set of regular languages, which is known to be P-space complete [Aho *et al.* 1974], to the alias problem [Landi 1992a, Theorem 4.8.1]. Recently it has been shown that intra-procedural may-alias analysis under the same conditions actually not is recursive

[15] To the author knowledge, a new implementation has improved the performance substantially.

[Landi 1992b]. Thus, approximating algorithms are always needed in the case of languages like C.

4.11.2 Points-to analysis

Our initial attempt at pointer analysis was based on abstract interpretation implemented via a (naive) standard iterative fixed-point algorithm. We abandoned this approach since experiments showed that the analysis was far to slow to be feasible. Independently, Emami has developed a *point-to analysis* based on traditional gen-kill data-flow equations, solved via an iterative algorithm [Emami 1993,Emami *et al.* 1993].

Her analysis computes the same kind of information as our analysis, but is more precise: it is flow-sensitive and program-point specific, computes both may and must point-to information, and approximates calls via functions pointers more accurately than our analysis.

The analysis takes as input programs in a language resembling three address code [Aho *et al.* 1986]. For example, a complex statement as `x = a.b[i].c.d[2][j].e` is converted to

```
temp0 = &a.b;
temp1 = &temp0[i];
temp2 = &(*temp1).c.d;
temp3 = &temp2[2][j];
x = (*temp3).e;
```

where the `temp`'s are compile-time introduced variables [Emami 1993, Page 21]. A Simple language may be suitable for machine code generation, but is unacceptably for communication of feedback.

The intra-procedural analysis of statement proceeds by a standard gen-kill approach, where both may and must point-to information is propagated through the control-flow graph. Loops are approximated by a fixed-point algorithm.[16] Heap allocation is approximated very rudely using a single variable "Heap" to represent all heap allocated objects.

We have deliberately chosen to approximate function calls via pointers conservatively, the objective being that more accurate information in the most cases (and definitely for our purpose) is useless. Ghiya and Emami have taken a more advanced approach by using the point-to analysis to perform inter-procedural analysis of calls via pointers. When it has been determined that a function pointer may point to a function f, the call-graph is updated to reflect this, and the (relevant part of the) point-to analysis is repeated [Ghiya 1992].

The inter-procedural analysis is implemented via the program's extended control-flow graph. However, where our technique only increases the number of constraints slightly, Emami's procedure essentially corresponds to copying of the data-flow equations; in practise, the algorithm traverses the (representation) of functions repeatedly. Naturally, this causes the efficiency to degenerate. Unfortunately, we are not aware of any runtime benchmarks, so we can not compare the efficiency of our analysis to Emami's analysis.

[16] Unconditional jumps are removed by a preprocess.

4.11.3 Approximation of data structures

Closely related to analysis of pointers is analysis of heap-allocated data structures. In this chapter we have mainly been concerned with stack-allocated variables, approximating runtime allocated data structures with a 1-limit methods.

Jones and Munchnick have developed a data-flow analysis for inter-procedural analysis of programs with recursive data structures (essentially Lisp S-expressions). The analysis outputs for every program point and variable a regular tree grammar, that includes all the values the variable may assume at runtime. Chase *el al.* improve the analysis by using a more efficient summary technique [Chase *et al.* 1990]. Furthermore, the analysis can discover "true" trees and lists, *i.e.* data structures that contain no aliases between its elements. Larus and Hilfinger have developed a flow analysis that builds an alias graph which illustrates the structure of heap-allocated data [Larus and Hilfinger 1988].

4.12 Further work and conclusion

We have in this chapter developed an inter-procedural point-to analysis for the C programming language, and given a constraint-based implementation. The analysis has been integrated into the *C-Mix* system and proved its usefulness. However, several areas for future work remain to be investigated.

4.12.1 Future work

Practical experiments with the pointer analysis described in this chapter have convincingly demonstrated the feasibility of the analysis, especially with regard to efficiency. The question is whether it is worthwhile to sacrifice some efficiency for the benefit of improved precision. The present analysis approximates as follows:

- flow-insensitive/summary analysis of function bodies,
- arrays are treated as aggregates,
- recursive data structures are collapsed,
- heap-allocated objects are merged according to their birth-place,
- function pointers are not handled in a proper inter-procedurally way.

Consider each in turn.

We considered program-specific pointer analysis in Section 4.10. However, as apparent from the description, the amount of information both during the analysis and in the final result may be too big for practical purposes. For example, in the case of a 1,000 line program with 10 global variables, say, the output will be more than 100,000 state variables (estimating the number of local variables to be 10). Even in the (typical) case of a sparse state description, the total memory usage may easily exceed 1M byte. We identify the main problem to be the following: too much irrelevant information is maintained by the

constraint-based analysis. For example, in the state corresponding to the statement '`*p = 1`' the only information of interest is that regarding '`p`'. However, all other state variables are propagated since they may be used at later program points.

We suspect that the extra information contributes only little on realistic programs, but experiments are needed to clarify this. Our belief is that the poor man's approach described in Section 4.2 provides the desired degree of precision, but we have not yet made empirical test that can support this.

Our analysis treats arrays as aggregates. Program using tables of pointers may suffer from this. Dependence analysis developed for parallelizing Fortran compilers has made some progress in this area [Gross and Steenkiste 1990]. The C language is considerably harder to analyze: pointers may be used to reference array elements. We see this as the most promising extension (and the biggest challenge).

The analysis in this chapter merges recursive data structures.[17] In our experience elements in a recursive data structure is used "the same way", but naturally, exceptions may be constructed. Again, practical experiments are needed to evaluate the loss of precision.

Furthermore, the analysis is mainly geared toward analysis of pointers to stack-allocated objects, using a simple notion of (inter-procedural) birth-place to describe heap-allocated objects. Use of birth-time instead of birth-place may be an improvement [Harrison III and Ammarguellat 1992]. In the author's opinion discovery of for instance singly-linked lists, binary trees *etc.* may find substantial use in program transformation and optimization, but we have not investigated inference of such information in detail.

Finally, consider function pointers. The present analysis does not track down inter-procedurally use of function pointers, but uses a sticky treatment. This greatly simplifies the analysis, since otherwise the program's call graph becomes dynamic. The use of static-call graphs is only feasible when *most* of the calls are static. In our experience, function pointers are only rarely used which justifies our coarse approximation, but naturally some programming styles may fail. The approach taken by Ghiya [Ghiya 1992] appears to be expensive, though.

Finally, the relation and benefits of procedure cloning and polymorphic-based analysis should be investigated. The k-limit notions in static-call graphs give a flexible way of adjusting the precision with respect to recursive calls. Polymorphic analyses are less flexible but seem to handle program with dynamic call graphs more easily.

4.12.2 Conclusion

We have reported on an inter-procedural, flow-insensitive point-to analysis for the entire C programming language. The analysis is founded on constraint-based program analysis, which allows a clean separation between specification and implementation. We have devised a technique for inter-procedural analysis which prevents copying of constraints. Furthermore we have given an efficient algorithm.

[17] This happens as a side-effect of the program representation, but the k-limit can easily be increased.