Binding-Time Analysis and the Taming of C Pointers *

Lars Ole Andersen
DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
E-mail: lars@diku.dk

Abstract

The aim of binding-time analysis is to determine when variables, expressions, statements, *etc.* in a program can be evaluated by classifying these into static (compile-time) and dyamic (run-time). Explicit separation of binding times has turned out to be crucial for successful self-application of partial evaluators, and apparently, it is also an important stepping-stone for profitable specialization of imperative languages with pointers and dynamic memory allocation. In this paper we present an automatic binding-time analysis for a substantial subset of the C language.

The paper has two parts. In the first part, the semantic issues of binding-time separation is discussed with emphasis on pointers and classification of these. This leads to the introduction of a two-level C language where binding times are explicit in the syntax. Finally, well-annotatedness rules are given which excludes non-consistently annotated programs.

In the second part, an automatic binding-time analysis based on constraint system solving is developed. The constraints capture the binding-time dependencies between expressions and subexpressions, and a solution to the system gives the binding times of all variables and expressions. We give rules for the generation of constraints, provide normalization rules, and describe how a solution can be found. Given the binding times of expressions, a well-annotated two-level version of the program can easily be construted. A two-level program can *e.g.* be input to an offline partial evaluator.

1 Introduction

A binding-time analysis takes a program and a binding-time description of the input parameters, and classifies all variables, expressions, statements and functions into static (compile-time) or dynamic (run-time). A binding-time analysis can *e.g.* be used to guide program transformations and optimizations. For example, a partial evaluator will evaluate all the static expressions, and generate runtime code for the rest. It is well-known that explicit, offline separation of binding times is essential for successful self-application of program specializers [5,7,9,13], but it also seems crucial for automatic specialization of imperative languages with features such as pointers and dynamic memory allocation.

In this paper we present an automatic binding-time analysis for a subset of the C programming language [12,14]. The analysis has been developed, implemented and integrated as part of a partial evaluator for the C language, but is general applicable.

1.1 Previous work

Historically, binding-time analyses were introduced into partial evaluation as a means to achieve efficient self-application [13]. The first analyses, which were based on abstract interpretation, treated first-order Lisp-like programs, and classified data structures as either completely static or completely dynamic. For instance, an alist, where the key was static but the associated value dynamic, would come out completely dynamic. Various binding-time analyses coping with partially static data structures [15,16] and higher-order languages [5,7,11] has later been developed; all for functional languages and based on abstract interpretation.

Recently, binding-time analyses based on non-standard type inference has attracted much attention [6,8,10,17]. The use of types to describe binding times works well with both higher-order languages and partially static data structures, and is well-understood.

1.2 Efficient binding-time analysis

Based on the ideas behind type inference and unification, Henglein developed an efficient binding-time analysis for an untyped lambda calculus with constants and a fixed-point operator, implemented via *constraint solving* [10]. The analysis can be implemented to run in almost linear time (in size of the input program), and is thus far more efficient than the most previous analyses with typical exponential run times.

The general idea in the analysis is to capture the binding time dependencies between an expression and its subexpressions by a *constraint system*. For example, for an expression `if (`e_1`) then` e_2 `else` e_3, a (simplified) constraint system could be $\{S \leq T_{e_1}, T_{e_2} \leq T_e, T_{e_3} \leq T_e, T_{e_1} \rhd T_e\}$, meaning that e_1 is at least static S, the binding time of the whole expression e is greater than the binding times of the two subexpressions, and if the test e_1 is dynamic then e must be dynamic too (the latter constraint). A solution to a constraint system is a substitution which maps all type variables T_e to a binding time, and hence a solution gives a binding time annotated program.

*Supported by the Danish Research Counsil STVF

The analysis proceeds in four phases. First, the constraint system is generated by a syntax directed traversal over the program. Next, the system is normalized by a set of rewrite rules. Subsequently, a solution is determined — which in case of a normalized system turns out to be straightforward — and finally the solution is used to annotate the program with the desired information.

1.3 The present work

This work generalizes and extends Henglein's analysis to a substantial subset of the C programming language. Based on the constraint solving method, we develop an analysis which can handle imperative data structures such as multi-dimensional arrays, structs and pointers; and dynamic memory allocation. The analysis classifies pointers into static and dynamic. Static pointers are those which can be dereferenced at compile-time. The use of types allows binding-time descriptions such as: `p` is a static pointer to a dynamic object. We give an example of how this kind of information can be utilized in a partial evaluator to replace dynamic memory allocation by static allocation.

Moreover, we consider the handling of externally defined functions, and show how the static type information can be utilized to prevent passing of partially static data structures to suspended applications. Our analysis will make completely dynamic a (potential) partially static data structure, which is passed to an operator.

Example 1.1 Consider the program fragment below which dynamically allocates a list, initializes it, and then looks up a key element.

```
int main(int n, int key, int data)
{
   struct List { int key, data; struct List *next; }
      list, *p;
   /* Make a list ... */
   for (p = &list; n; n--) {
        p = p->next = alloc(List);
        p->key = n;
        p->data = data++;
     }
   /* Look up the key element */
   for (p = list.next; p->key != key; p = p->next);
   return p->data;
}
```

Given the information that `n` and `key` are static, and that `data` is dynamic, our analysis will find that `struct List` is a partially static struct where `key` and `next` are static; that both `list` and `p` are static, and hence that the `alloc()` can be performed at compile-time. The references to `data` are all dynamic. **End of Example**

1.4 Outline

Section 2 discusses binding-time separation of data structures with emphasis on pointers and dynamic memory allocation. Section 3 presents Core C — an abstract intermediate language which we analyze. That section also contains some semantical considerations.

The semantic definition of well-annotatedness is given in Section 4. We introduce binding-time types and state well-annotatedness rules. This gives the background for the binding-time analysis which is developed in Section 5. In Section 6 implementation issues are considered, Section 7 discusses related work, and Section 8 concludes.

2 Binding-time separation of data structures

In this section we consider binding-time separation of data structures and motivate the definitions we introduce later. The emphasis is on pointers as they are central to C, and they form major obstacles for efficient optimizations.

2.1 Classifying pointers

Classify a pointer as *static* if it (when instantiated) solely points to objects with statically known addresses. Otherwise it is said to be *dynamic*. Operational speaking, a static pointer can be dereferenced at compile-time, a dynamic one cannot.

Example 2.1 Suppose `p` and `q` are pointers.

```
p = &x;
q = &a[ dyn-exp];
```

Even though `x` is a dynamic variable, `p` can be classified static since the symbolic run time address *loc-x* of `x` is known. On the other hand, `q` must inevitably be classified dynamic since `q = &a[`*dyn-exp*`]` corresponds to `q = (a +` *dyn-exp*`)` and this cannot be evaluated at compile-time due to the dynamic expression. **End of Example**

In the following we assume that all dynamic variables contain their symbolic run time address, *e.g.* a dynamic variable `x` contains *loc-x*.

Example 2.2 Consider partial evaluation of the statements below, where `x` is assumed dynamic.

```
p = &x; *p = 2;
```

First, the address operator can be evaluated since `p` is static. Secondly, `p` can be dereferenced since `p` is a static pointer, and finally a residual statement `x = 2` can be generated, as expected. **End of Example**

Similar applies for calls to `malloc()`. If the assigned pointer in `p = malloc()` is static, then the allocation is classified static. Otherwise it must be suspended.

2.2 Improving the binding-time separation

Every variable can be assigned a separate binding time, but since the number of dynamically allocated objects is unknown at "analysis-time", an approximation must do. A tempting idea is to collapse all objects of the same type, and give these a single uniform binding time. However, a more precise description can be obtained if objects *birth-places* are taken into account.

Suppose for simplicity that all dynamically memory allocation is accomplished via a library function `alloc()`. As statically allocated variables can be distinguished by their (unique) names, dynamically allocated objects can be referred back to the `alloc()` calls which created them. Let all allocation calls be labelled uniquely $\texttt{alloc}_l\texttt{()}$.

A call $\texttt{alloc}_l\texttt{(}S\texttt{)}$ is called the l'th *birth-place* of an S struct. Generalize the notation such that the birth-place of a statically allocated variable is its name.

We will use the notion of birth-place as the degree of precision in our analysis. Thus we assign a unique binding-time

description to each birth-place under the following restriction. For all pointers `p`, all the objects to which `p` can point to must possess the same binding times. An assignment of binding times fulfilling these conditions, and that no static variable depends of a dynamic value, is said to be *consistent*.

Our analysis, which is based on a non-standard type inference, will assign to each pointer variable a unique type such as $*D$ meaning "a static pointer to a dynamic object", and will hence capture the above requirements. For practical purposes, this is not always sufficient, though.

2.3 Explicit pointer information needed

The output of the binding-time analysis is a description of the form that "the pointer is dynamic", or it is a "static pointer to an object of type T". However, this says nothing about the *location* of the objects with binding time T.

In practise it is often necessary to distinguish between intra- and interprocedural optimizations. To this, explicit pointer information is needed. We will therefore in the following assume the existence of a *pointer analysis function* $\mathcal{P}$, such that for every pointer variable `p`, $\mathcal{P}$(`p`) approximates the set objects `p` may point to during program execution.

Example 2.3 The C program specializer described in the Partial Evaluation book [4] does not allow both recursion and non-local side-effects. Thus, in recursive residual function, pointers referring to non-local objects must be suspended. In Example 5.1 it is shown how such an additional requirement can be coped with. **End of Example**

2.4 Case-study: replacing dynamic allocation by static allocation

As an example of how a partial evaluator can utilize the information that a pointer to an object is static, consider the list allocation program in Example 1.1. Recall that `p` and `next` are both static pointers. The key insight is that eventually the whole list will be allocated — otherwise the `next` field could not have been classified static. This means that the (static) calls to `alloc()` can be replaced by statically allocated objects in the residual program, where each object contains the dynamic field of `struct List`, that is, `data`.

Suppose that `n` is 10 and that `key` is 7. Then the following residual program could be generated.

```
struct List_key { int data; } alloc_0, ..., alloc_9;
int main_0(int data)
 {
     /*  Make a list  */
     alloc_0.data = data++;
     ...
     alloc_9.data = data++;
     /*  Look up the key element  */
     return alloc_3.data;
 }
```

All the calls to `alloc()` have been evaluated during the specialization and replaced by statically allocated variables `alloc_0`, ..., `alloc_9`. All pointer traversal has been performed, and in the residual program the desired node is referred to directly. This is in fact the program the specializer described in [4] would generate.

3 The C programming language

Our subject language is a substantial subset of the C programming language [12,14], including global variables and functions; multi-dimensional arrays, structs and pointers; and dynamically memory allocations.

3.1 The semantics of C

The recurring problem when analyzing C programs is the lack of a formal semantic definition. If an automatic, safe analysis has to cope with *e.g.* use of uninitialized pointers, it necessarily must be overly conservative, and in the worst case assume that pointers can point to any dynamic objects in the program.

We exclude programs exploiting type casts and `void` pointers. Pointer arithmetic is allowed but restricted to the standard, that is, a pointer is not allowed to be deferenced if it points outside an aggregate, and the location of aggregates is in general unknown. Moreover, we do not consider `union`s and function pointers. This paper does not contain a formal definition of the core language we analyze, but we assume the semantics given in the standard and nothing more [12].

In the following, calls to "`extern`" defined functions are explicitly named *externally calls*, whereas calls to user defined functions are simply referred to as calls. For simplicity, all dynamic memory allocation is assumed accomplished via library function `alloc(`S`)`, where S is the name of the struct to be allocated.[1] For example, `alloc(List)` will return a pointer to a struct `List`.

3.2 The Core C language

The abstract syntax of Core C is displayed in Figure 1, and is similar the language employed in the C specializer described in the literature [2,3].

A Core C program consists of an optional number of global variable declarations followed by at least one function definition.

A variable can be of base type (`int`, `double`, *etc.*), of struct type, a multi-dimensional array, or of pointer type. We do neither consider `union`s nor `void` pointers. A function can declare both parameters and local variables, and the body is made up by a number of labelled statements. A statement can be an expression, the `if` conditional which jumps to one of two labelled statements, an unconditional `goto`, a function `call`, or a function `return`. In Core C function calls are at the statement level rather than at the expression level. This is convenient for technical reasons, but is of no importance to the binding-time analysis.[2] C function calls can automatically be lifted to the statement level in Core C by introduction of temporary local variables.

An expression can be a constant, a variable reference, a struct or array indexing, a pointer indirection (`indr`), the address operator & (`addr`), an application of unary, binary or external functions, an assignment, or a call to the memory allocation function `alloc()`.

Example 3.1 An (unreadable) Core C representation of the list allocation program in Example 1.1 is shown below.

[1] The discussion carries over to dynamically allocation of *e.g.* integers and arrays without modification.

[2] When function calls are lifted to the statement level, evaluation of an expression cannot cause a control-flow shift. This is convenient when Core C is used as input to a partial evaluator.

CoreC	::= decl* fundef$^+$	Core C programs
decl	::= typespec dec	Declarations
typespec	::= *base* | `struct` { decl$^+$ }	
dec	::= *id* | $*$ dec | dec [*int*]	
fundef	::= typespec *id* { decl* stmt$^+$ }	Functions
stmt	::= *lab*: `expr` exp	Statements
	| *lab*: `goto` *lab*	
	| *lab*: `if` (exp) *lab lab*	
	| *lab*: `return` exp	
	| *lab*: `call` exp = *id* (exp*)	
exp	::= `cst` *const* | `var` *id* | `struct` exp.*id*	Expressions
	| `index` exp [exp] | `indr` exp | `addr` exp	
	| `unary` *uop* exp | `binary` exp *bop* exp | `ecall` *id* (exp*)	
	| `alloc` (*id*) | `assign` exp = exp	

Figure 1: Abstract syntax of Core C.

```
int main (int n int key int data)
{
   struct List list
   struct List (*p)
   1:  expr assign var p = addr var list
   2:  if (var n) 3 8
   3:  expr assign var p
       = assign struct List indr var p.next
       = alloc(List)
   4:  expr assign struct List indr var p.key = var n
   5:  expr assign struct List indr var p.data
       = assign var data = binary (var data) - (cst 1)
   6:  expr assign var n = binary (var n) - (cst 1)
   7:  goto 2
   8:  expr assign var p = struct List var list.next
   9:  if (binary (var key) !=
               (struct List indr var p.key)) 10 12
  10:  expr assign var p = struct List indr var p.next
  11:  goto 9
  12:  return struct List indr var p.data
}
```

End of Example

Obviously, most Ansi C conforming programs can automatically be transformed into an equivalent Core C representation.

4 Well-annotated two-level C

This section defines *well-annotated two-level* Core C programs. A two-level Core C program is a program where the binding times are explicit in the syntax. For example, there is a static `assign`, and a dynamic `assign`. A two-level program where the binding times are separated consistently is called *well-annotated*. For example, in a well-annotated program where will be no static `if` with a dynamic test expression.

4.1 Binding times made explicit

Aiming at making the binding times explicit, we extend the Core C language from Section 3 into a *two-level* version where binding times are present in the syntax. The abstract syntax is depicted in Figure 2. The actual annotation of data structure declarations is immaterial for the rest of the presentation, and is thus left out.

In the two-level language, almost all Core C constructs exist in two versions: a static version (*e.g.* `assign`) and an underlined dynamic version (*e.g.* `assign`). The intuition is: non-underlined constructs are static and can be evaluated at compile-time, whereas underlined constructs are dynamic and must be suspended to run-time.

The body of a two-level Core C function consists of a sequence of two-level statements. A two-level statement can be an ordinary Core C statement (see Figure 1), or underlined `expr`, `goto`, `if`, `return`, or `call`. Analogous, a two-level expression can be an ordinary expression, or a similar underlined one.

Let variables be bound to their (symbolic) run time addresses. Then a variable reference `var` x can always be evaluated statically, — to *loc-x*, say — and there is hence no need for a `var`.

Example 4.1 Assume that `n` and `key` are static. Then a two-level version of the list allocation is as follows (where an underscore replaces underlines):

```
int main (int n int key int data)
{
   struct List list
   struct List (*p)
   1:   expr assign var p = addr var list
   2:   if (var n) 3 8
   3:   expr assign var p
        = assign struct List indr var p.next
        = alloc(List)
   4:   expr assign struct List indr var p.key = var n
   5:   _expr _assign _struct List indr var p.data
        = _assign var data
        = _binary (var data) - (lift(cst 1))
   6:   expr assign var n = binary (var n) - (cst 1)
   7:   goto 2
   8:   expr assign var p = struct List var list.next
   9:   if (binary (var key) !=
                (struct List indr var p.key)) 10 12
  10:  expr assign var p = struct List indr var p.next
  11:  goto 9
  12:  _return _struct List indr var p.data
}
```

End of Example

Moreover, a two-level expression can be an application of a `lift` operator. The aim of `lift` is to convert a value into

```
2CoreC ::= 2decl* 2fundef+                                   Two-level Core C programs
2fundef ::= 2typespec id { 2decl* 2stmt+ }                   Two-level functions
2stmt   ::= stmt                                             Two-level statements
          | lab: expr 2exp
          | lab: goto lab
          | lab: if ( 2exp ) lab lab
          | lab: return 2exp
          | lab: call 2exp = id ( 2exp* )
2exp    ::= exp | lift exp                                   Two-level expressions
          | struct 2exp.id
          | index 2exp [2exp] | indr 2exp | addr 2exp
          | unary uop 2exp | binary 2exp bop 2exp | ecall id ( 2exp* )
          | alloc (id) | assign 2exp = 2exp
```

Figure 2: Abstract syntax of two-level Core C (see also Figure 1).

a corresponding constant expression. For example, `lift 2` is the constant expression 2. This is useful when a static value appears in a dynamic context. For example, an expression `binary` e `+ 2`, where e is dynamic is wrong: the static constant 2 appears in a dynamic context. By applying `lift`: `binary` e `+ lift 2` the expression becomes "right". For pragmatic reasons we restrict the use of `lift` to base type values only. Thus, values of struct or pointer type cannot be lifted.

There are many syntactically legal two-level versions of a program. We are in particular interested in the *well-annotated* programs.

4.2 Two-level binding-time types

In this section we give a set of rules which two-level expressions, statements, functions and finally programs must fulfill in order to be well-annotated. Suppose a two-level expression e is given. We are interested in the binding time of the value to which e evaluates. If it is *e.g.* dynamic, we will write $\vdash^{exp} e : D$, considering binding times as *types* in the two-level language. This way, well-annotatedness is a matter of well-typedness.

Let a binding-time type $\mathcal{T}$ be defined inductively as follows:

$$\mathcal{T} ::= S \mid D \mid \mathcal{T} \times \cdots \times \mathcal{T} \mid *\mathcal{T} \mid T$$

where S and D are ground types.

The ground type S (static) represents static base type values, *e.g.* the integer value 2. The base type D (dynamic) denotes dynamic values, *e.g.* a pointer which cannot be dereferenced, or the value of struct type where the fields cannot be accessed. The constructor $*\mathcal{T}$ denotes a static pointer to an object of binding-time $\mathcal{T}$. For example, if `p` is a static pointer to integers, we have $\vdash$ `p` $: *S$. The product constructor describes the fields of a value of struct type. For example, if `s` if of type `struct { int x, y; }` where `x` is static but `y` dynamic, we can write $\vdash$ `s` $: S \times D$.

In the following we use the convention that T is a type variable ranging over binding-time types, whereas $\mathcal{T}$ denotes an arbitrary binding-time type.

Consider again at the `struct List`

```
struct List { int key, data; struct List *next;  }
```

where `next` is a pointer to a `struct List`. Writing T_{List} for the type of `next`, we have

$$T_{List} = S \times D \times (*T_{List})$$

which is a recursively defined type.[3]

To express this formally in our type system, we extend it with a fixed point operator:

$$\mathcal{T} ::= \ldots \mid \mu T, \mathcal{T}$$

defined by unfolding $\mu T.\mathcal{T} = \mathcal{T}[\mu T.\mathcal{T}/T]$.

A *type assignment*, or a *division*, $\tau : \mathrm{Id} \to \mathrm{BTT}$ is a finite function from identifiers to binding-time types. An *initial type assignment*, or an *initial division*, τ_0, is a type assignment defined solely on the parameters to the goal function. An division is said to agree with an initial division if they are equal on the domain of initial division. Let an *extended type assignment* $\tau : \mathrm{Id} \cup \mathrm{Label} \to \mathrm{BTT}$ be a type assignment which is also defined on labels. The rationale behind extended type assignments is to capture the type of objects allocated via `alloc`$_l$`()` calls. In the following, type assignment means extended type assignment.

For example, an initial division to the list allocation (Example 1.1) could be $\tau_0 = [\mathtt{n} \mapsto S, \mathtt{key} \mapsto S, \mathtt{data} \mapsto D]$. A division for the whole program is $\tau = \tau_0 \circ [\mathtt{list}, \mathtt{1} \mapsto \mu T.S \times D \times *T, \mathtt{p} \mapsto *(\mu T.S \times D \times *T)]$ where 1 is the label of the `alloc()` call.

An *operator assignment*: $\mathcal{O} : \mathrm{OId} \to \mathrm{BTT}^* \to \mathrm{BTT}$ is a map from operators and external functions to their *static* binding-time types. For example, since + is a function from two integers to an integer, we have $\mathcal{O}(+) = (S, S) \to S$.

4.3 Well-annotated two-level Core C

Not all two-level Core C programs are meaningful. For example, an occurrence of a static `if` with a dynamic test expression can be considered as a *type error*.

In this section a collection of typing rules is imposed on the set of two-level programs restricting it to a set of *well-annotated* programs. Intuitively, for a program to be well-annotated, it must hold that the transformation using the annotations does not commit a "binding-time error". That is, goes wrong *e.g.* due to a static `if` with a dynamic

[3] We could do without recursive types, but then all cyclic definitions had to be dynamic.

test. Strictly speaking, this implies that the notion of well-annotatedness must be defined with respect to a particular optimization/transformation semantics. In this paper, though, we will give an independent definition which is intuitively "correct", and we will not address pragmatics in detail.

Suppose in the following that a two-level program p is given as well as an initial division τ_0, and let τ be a type assignment for p which agrees with τ_0.

To capture that a static value can be lifted to a dynamic expression, we introduce a subtype relation $\preceq^{lift}$ between binding-time types, defined by $S \preceq^{lift} D$ and $\mathcal{T} \preceq^{lift} \mathcal{T}$, for all $\mathcal{T}$. We can then write $\vdash e : T, T \preceq^{lift} D$ meaning that T must be either S or D.

An expression e in p is *well-annotated* with respect to τ if there exists a two-level type $\mathcal{T}$ such that $\tau \vdash^{exp} e : \mathcal{T}$ where the relation $\vdash^{exp}$ is defined in Figure 3.

The rules are justified as follows.

A constant is static, and type of a variable is given by the type assignment. Consider the struct field selector. If it is static, the type of the subexpression must be a product. If it is dynamic, the subexpression must be dynamic D.

An index e_1`[`e_2`]` is semantically defined by $*(e_1 + e_2)$. Thus, in case of a static index, the left-expression must be a static pointer and the index static. Otherwise both the subexpressions must be dynamic. Notice the use of $\preceq^{lift}$. The rules for the pointer indirection `indr` are similar. In the rule for the static address operator &, it is required that the operand is non-dynamic. This rules out the case `&a[`*dyn* `]` which would be wrong to assign the type $*D$.[4]

The rules for operators and external functions are analogous. In the static case, the binding times of the arguments must equal the types provided by the operator assignment $\mathcal{O}$. Otherwise the application must be dynamic, and all the arguments dynamic. Notice the use of $\preceq^{lift}$ which allows a static base type value to be lifted to dynamic. Observe that the operator assignment $\mathcal{O}$ effectively prevents applications with partially static arguments. The rationale is that an operator or external function is a "black-box" which either can be fully evaluated at compile-time, or fully suspended until run-time.

The static `alloc()` returns a static pointer to an object of the desired type, given by the type assignment. An assignment can only be static if both the expressions are static. Otherwise both expressions must be dynamic, where the assigned expression can be lifted, though.

Consider now the annotation of two-level statements. To static statements we assign the type S, and to dynamic statements the type D. Let s be a statements in a function f; $\tau : \mathrm{Id} \to \mathrm{BTT}$ a type assignment defined on all variables in f; and let $\pi : \mathrm{FId} \to \mathrm{BTT}$ be a function type assignment mapping function identifiers to their return types. For example, $\pi(\texttt{main}) = D$.

Then s is *locally well-annotated* in function f, if there exists a type $\mathcal{T}$ such that $\tau, \pi \vdash^{stmt} s : \mathcal{T}$ where the relation $\vdash^{stmt}$ is defined in Figure 4.

The binding times of a statement is mainly determined by the binding times of the subexpressions. However, in a residual function, that is, a function which returns a dynamic value, all `return` must be dynamic since a run-time function cannot return a value at compile-time. This is assured by inspecting the function environment $\pi(f)$. In the case of a `call` statement, the arguments must possess the same binding times as the formal parameters. Due to the use of the lift coercion it is possible to lift a static value to dynamic.

The following definition states the conditions for a program to be well-annotated. Informal speaking, it must hold that all statements are locally well-annotated, and that if a function contains a dynamic statement or a dynamic parameter, then the function is dynamic too. Furthermore, we will restrict user defined function to take only completely static arguments or fully dynamic (D) arguments, similar to the restrictions enforced on operators and external functions. This is convenient in connection with partial evaluation, but can of course be liberated in other applications.

Definition 4.1 *Let p be a two-level Core C program and τ_0 an initial division. Suppose that τ is a type assignment defined on all free identifiers in p which respects the initial division; and that π is a function type assignment defined on all function identifiers in p. Then p is* well-annotated *if*

1. *For all parameters v: $\tau(v)$ is either D or completely static.*
2. *For all functions f: $\pi(f)$ is either D or completely static.*
3. *For all functions f: for all s: $\tau, \pi \vdash^{stmt} s : \mathcal{T}$*
4. *For all functions f, if there is a statement s in f such that $\tau, \pi \vdash^{stmt} s : D$, or a parameter v in f such that $\tau(v) = D$, then $\pi(f) = D$.*

□

There are two-level programs where the binding-time are well-separated but those contained in the definition of well-annotatedness. In particular, notice that the above definition imposes a mono-variant binding time assignment to functions: all calls to a function must have the same binding time.

5 Binding-time analysis by constraint set solving

In the previous section a set of type inference rules was employed to see whether a two-level program is well-annotated. The aim of binding-time analysis is the opposite: given a program and an initial division, to compute a corresponding two-level program. We proceed in two phases. First we compute the binding times — or two-level types — of all variables and expressions by the means of a non-standard type inference. Next, we convert the program into a well-annotated two-level version. The latter is merely a question of presenting the types to the world, and we will therefore not consider that part in detail.

5.1 Constraints and constraint systems

A *constraint system* is multiset of constraints of the following form:

$$T_1 =^? T_2,\ T_1 \preceq^? T_2,\ T_1 \sqsubseteq^? T_2,\ T_1 \trianglelefteq^? T_2,\ T_1 \rhd^? T_2$$

where T_i range over binding-time types.

Let the partial orders $\prec$ and $\sqsubset$ be defined over binding-time types as follows:

$$\begin{array}{rcl} S & \prec & D \\ *D & \sqsubset & D \\ D \times \cdots \times D & \sqsubset & D \end{array}$$

[4] But unfortunately also the case &x where x is dynamic.

[Const] $\tau \vdash^{exp} \texttt{cst}\ c : S$

[Var] $\tau \vdash^{exp} \texttt{var}\ v : \tau(v)$

[Struct]
$$\frac{\tau \vdash^{exp} e_1 : T_1 \times \cdots \times T_n}{\tau \vdash^{exp} \texttt{struct}\ e_1.i : T_i} \qquad \frac{\tau \vdash^{exp} e_1 : D}{\tau \vdash^{exp} \underline{\texttt{struct}}\ e_1.i : D}$$

[Index]
$$\frac{\tau \vdash^{exp} e_1 : *T \quad \tau \vdash e_2 : S}{\tau \vdash^{exp} \texttt{index}\ e_1[e_2] : T} \qquad \frac{\tau \vdash^{exp} e_1 : D \quad \tau \vdash e_2 : T \quad T \preceq^{lift} D}{\tau \vdash^{exp} \underline{\texttt{index}}\ e_1[e_2] : D}$$

[Indr]
$$\frac{\tau \vdash^{exp} e : *T}{\tau \vdash \texttt{indr}\ e : T} \qquad \frac{\tau \vdash^{exp} e : D}{\tau \vdash^{exp} \underline{\texttt{indr}}\ e : D}$$

[Addr]
$$\frac{\tau \vdash^{exp} e : T, \quad T \neq D}{\tau \vdash^{exp} \texttt{addr}\ e : *T} \qquad \frac{\tau \vdash^{exp} e : D}{\tau \vdash^{exp} \underline{\texttt{addr}}\ e : D}$$

[Unary]
$$\frac{\tau \vdash^{exp} e_1 : T_1 \quad \mathcal{O}(op) = (T_1) \to T}{\tau \vdash^{exp} \texttt{unary}\ op\ e_1 : T} \qquad \frac{\tau \vdash^{exp} e_1 : T_1 \quad T_1 \quad T_1 \preceq^{lift} D}{\tau \vdash^{exp} \underline{\texttt{unary}}\ e_1 : D}$$

[Binary]
$$\frac{\tau \vdash^{exp} e_i : T_i \quad \mathcal{O}(op) = (T_1, T_2) \to T}{\tau \vdash^{exp} \texttt{binary}\ e_1\ op\ e_2 : T} \qquad \frac{\tau \vdash^{exp} e_i : T_i \quad T_i \preceq^{lift} D}{\tau \vdash^{exp} \underline{\texttt{binary}}\ e_1\ op\ e_2 : D}$$

[Ecall]
$$\frac{\tau \vdash^{exp} e_i : T_i \quad \mathcal{O}(f) = (T_1, \ldots, T_n) \to T}{\tau \vdash^{exp} \texttt{ecall}\ f(e_1, \ldots, e_n) : T} \qquad \frac{\tau \vdash^{exp} e_i : T_i \quad T_i \preceq^{lift} D}{\tau \vdash^{exp} \underline{\texttt{ecall}}\ f(e_1, \ldots, e_n) : D}$$

[Alloc] $\tau \vdash^{exp} \texttt{alloc(S)} : *\tau(l) \qquad \tau \vdash^{exp} \underline{\texttt{alloc(S)}} : D$

[Assign]
$$\frac{\tau \vdash^{exp} e_1 : T \quad \tau \vdash e_2 : T \quad T \neq D}{\tau \vdash^{exp} \texttt{assign}\ e_1 = e_2 : T} \qquad \frac{\tau \vdash^{exp} e_1 : D \quad \tau \vdash e_2 : T_2 \quad T_2 \preceq^{lift} D}{\tau \vdash^{exp} \underline{\texttt{assign}}\ e_1 = e_2 : D}$$

Figure 3: Well-annotatedness rules for two-level expressions with explicit lift.

[Expr]
$$\frac{\tau \vdash^{exp} e : T \quad T \neq D}{\tau, \pi \vdash^{stmt} \texttt{expr}\ e : S} \qquad \frac{\tau \vdash^{exp} e : D}{\tau, \pi \vdash^{stmt} \underline{\texttt{expr}}\ e : D}$$

[Goto] $\tau, \pi \vdash^{stmt} \texttt{goto}\ m : S \qquad \tau, \pi \vdash^{stmt} \underline{\texttt{goto}}\ m : D$

[If]
$$\frac{\tau \vdash^{exp} e : S}{\tau, \pi \vdash^{stmt} \texttt{if (e) } m\ n : S} \qquad \frac{\tau \vdash^{exp} e : D}{\tau, \pi \vdash^{stmt} \underline{\texttt{if}}\ \texttt{(e)}\ m\ n : D}$$

[Return]
$$\frac{\tau \vdash^{exp} e : \pi(f), \quad \pi(f) \neq D}{\tau, \pi \vdash^{stmt} \texttt{return}\ e : S} \qquad \frac{\tau \vdash^{exp} e : T, T \quad \preceq^{lift} \pi(f), \quad \tau(f) = D}{\tau, \pi \vdash^{stmt} \underline{\texttt{return}}\ \texttt{e} : D}$$

[Call]
$$\frac{\tau \vdash^{exp} e_i : \tau(f_i'), \quad \tau(f_i') \neq D \quad \tau \vdash^{exp} x : \pi(f'), \quad \pi(f') \neq D}{\tau, \pi \vdash^{stmt} \texttt{call}\ x\ \texttt{=}\ f'\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} : S} \qquad \frac{\tau \vdash^{exp} e_i : T_i, \quad T_i \preceq^{lift} \tau(f_i') \quad \tau \vdash^{exp} x : \pi(f'), \quad \pi(f') = D}{\tau, \pi \vdash^{stmt} \underline{\texttt{call}}\ x\ \texttt{=}\ f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} : D}$$

Figure 4: Well-annotatedness rules for two-level statements.

and let $T_1 \preceq T_2$ iff $T_1 \prec T_2$ or $T_1 = T_2$, and similar for $\sqsubseteq$ and $\trianglelefteq$.

Let C be a constraint system. A *solution* to C is a substitution $S : \text{TVar} \to \text{BTT}$ from type variables to binding-time types such that for all $c \in C$ it holds:

c is $T_1 =^? T_2$	implies	$ST_1 = ST_2$
c is $T_1 \preceq^? T_2$	implies	$ST_1 \preceq ST_2$
c is $T_1 \sqsubseteq^? T_2$	implies	$ST_1 \sqsubseteq ST_2$
c is $T_1 \trianglelefteq^? T_2$	implies	$ST_1 \trianglelefteq ST_2$
c is $T_1 \rhd^? T_2$	implies	$T_1 = D \Rightarrow T_2 = D$

and S is the identity on type variables not occurring in C. The set of solutions to a constraints system C is denoted by $\text{SOL}(C)$. If a constraint system has a solution, there is a "most" static one. This is called a *minimal solution*.

5.2 Capturing binding times by constraints

To each variable v, expression e, statement s, and function f we assign a unique type variable T_v, T_e, T_s and T_f, respectively. The output of the binding-time analysis is a program where all the type variables have been instantiated consistently.

Let e be a Core C expression. The set of constraints $\mathcal{C}_{exp}(e)$ generated for e is inductively defined in Figure 5.

A constant e is static, and we generate the constraint $T_e =^? S$. The binding time of a variable v is given by the unique type variable T_v so we add $T_e =^? T_v$ to the constraint set.

Consider the rules [Struct] in Figure 3. In the static case, the subexpression e_1 possesses a product type, and in the dynamic case it is D. This is captured via the constraint $\sqsubseteq^?$. Recall that a solution to this constraint must satisfy that either the two sides are equal, or otherwise the right hand side is dynamic and the subtypes to the left are all

$\mathcal{C}_{exp}(e)$ = case e of

$[\![\texttt{cst } c\ ]\!]$	$\Rightarrow$	$\{S = T_e\}$
$[\![\texttt{var } v\ ]\!]$	$\Rightarrow$	$\{T_v = T_e\}$
$[\![\texttt{struct } e_1.i\ ]\!]$	$\Rightarrow$	$\{T_1 \times \cdots \times T_e \times \cdots \times T_n \sqsubseteq^? T_{e_1}\} \cup \mathcal{C}_{exp}(e_1)$
$[\![\texttt{index } e_1[e_2]\ ]\!]$	$\Rightarrow$	$\{*T_e \sqsubseteq^? T_{e_1}, T_{e_2} \rhd^? T_{e_1}\} \cup \mathcal{C}_{exp}(e_i)$
$[\![\texttt{indr } e_1\ ]\!]$	$\Rightarrow$	$\{*T_e \sqsubseteq^? T_{e_1}\} \cup \mathcal{C}_{exp}(e_1)$
$[\![\texttt{addr } e_1\ ]\!]$	$\Rightarrow$	$\{*T_{e_1} \sqsubseteq^? T_e, T_{e_1} \rhd^? T_e\} \cup \mathcal{C}_{exp}(e_1)$
$[\![\texttt{unary } op\ e_1\ ]\!]$	$\Rightarrow$	$\{T_{e_1} \preceq^? \overline{T}_{e_1}, T_{op_1^{stat}} \trianglelefteq^? \overline{T}_{e_1}, \overline{T}_{e_1} \rhd^? T_e, T_{op^{stat}} \trianglelefteq^? T_e, T_e \rhd^? \overline{T}_{e_1}\} \cup \mathcal{C}_{exp}(e_1)$
$[\![\texttt{binary } e_1\ op\ e_2\ ]\!]$	$\Rightarrow$	$\{T_{e_i} \preceq^? \overline{T}_{e_i}, T_{op_i^{stat}} \trianglelefteq^? \overline{T}_{e_i}, \overline{T}_{e_i} \rhd^? T_e, T_{op^{stat}} \trianglelefteq^? T_e, T_e \rhd^? \overline{T}_{e_i}\} \cup \mathcal{C}_{exp}(e_i)$
$[\![\texttt{ecall } f(e_1,\ldots,e_n)\ ]\!]$	$\Rightarrow$	$\{T_{e_i} \preceq^? \overline{T}_{e_i}, T_{f_i^{stat}} \trianglelefteq^? \overline{T}_{e_i}, \overline{T}_{e_i} \rhd^? T_e, T_{f^{stat}} \trianglelefteq^? T_e, T_e \rhd^? \overline{T}_{e_i}\} \cup \mathcal{C}_{exp}(e_i)$
$[\![\texttt{alloc(S)}\ ]\!]$	$\Rightarrow$	$\{*T_s \sqsubseteq^? T_e\}$
$[\![\texttt{assign } e_1 \texttt{ = } e_2\ ]\!]$	$\Rightarrow$	$\{T_e = T_{e_1}, T_{e_2} \preceq^? T_{e_1}\} \cup \mathcal{C}_{exp}(e_i)$

Figure 5: Constraints for expressions.

dynamic.

The reasoning behind the constraint for `index`, `indr` and `addr` are similar. In case of the `index` expression, a dependency constraint is added to force the left expression e_1 to be dynamic if the index e_2 is dynamic, cf. the definition of $\rhd^?$.

Now consider the rules for [Unary], [Binary] and [Ecall]. In the static case, the binding times of the arguments must equal the static type T_{op_i} (given by $\mathcal{O}$ in Figure 3). Otherwise it must be lift-able to D. This is captured in the constraint system as follows. For each argument e_i, a new "lifted" type variable $\overline{T}_{e_i}$ is introduced. The type of the actual expression T_{e_i} is put in lift-relation to the lifted argument $\overline{T}_{e_i}$. Furthermore, the "lifted" variables are constrained via the $\trianglelefteq^?$ constraint. Recall that either $\overline{T}_{e_i}$ must be the completely static (binding-time type) of the operator, or D. This is exactly the definition of a solution to $T_{op_i} \trianglelefteq^? \overline{T}_{e_i}$. Dependency constraints are added to assure that if one of the arguments are dynamic, then the application becomes dynamic too, and vice versa.

In the case of an `alloc(`St`)`, the type of the expression e is either a pointer to the type of St, or D. This is captured by $*T_{st} \sqsubseteq^? T_e$. Finally, for an assignment $e_1 = e_2$ it must hold that the type of e_2 is lift-less than the type of e_1, and the type of the whole expression equals the type of e_1.

Let s be a two-level statement in a two-level function f. The constraints $\mathcal{C}_{stmt}(s)$ generated for s is defined in Figure 6.

For all statements containing an expression e we add a dependency $T_e \rhd^? T_f$. This implements the first part of item 4 in Definition 4.1.

In the case of a `return` statement, the returned expression must have a binding time which is lift-able to the return type T_f, cf. the rule [Return] in Figure 4. Consider now the call statement. The actual expression e_i must be lift-able to the formal parameters $T_{f_i'}$ of the called function f'. Furthermore, the binding time of the assigned variable x must equal the binding time of the return type of f'.

Let p be a Core C program, and assume that the parameters `main`$_1$,...,`main`$_n$ of the `main()` function are of base type.[5] Suppose that $\tau_0 = [\texttt{main}_1 \mapsto T_1, \ldots \texttt{main}_n \mapsto T_n]$ is an initial division. We will say that a constraint system $\mathcal{C}_0 = \{T_0 \preceq^? T_{main_0}, \ldots . T_n \preceq^? T_{main_n}\}$ *agrees* with τ_0, since the minimal solution to it "is" τ_0.

[5] In order to allow parameters of struct and pointer types, the constraint set generation must be changed slightly to guarantee the existence of a solution.

Definition 5.1 *Let p be a Core C program and suppose that τ_0 is an initial division. The constraint system $\mathcal{C}(p)$ for program p is defined by:*

$$\mathcal{C}(p) = \mathcal{C}_0 \cup \bigcup_{f \in p_{func}} \{T_{f_i^{stat}} \trianglelefteq^? T_{f_i}, T_{f_i} \rhd^? T_f, T_{f^{stat}} \trianglelefteq^? T_f\} \cup \bigcup_{f \in p_{func}} \bigcup_{s \in f_{stmt}} \mathcal{C}_{stmt}(s)$$

where f_i are the formal parameters of function f, $T_{f_i^{stat}}$ are the static binding time types of formal parameter i of f, and $\mathcal{C}_0$ agree on τ_0. □

The first part is the constraints corresponding to the initial division. The next set captures item 1, 2 and the last part of 4 in Definition 4.1. The last set corresponds to item 3 in the definition.

To prove the correctness of the transformation, we have to prove items 1 to 4 in Definition 4.1. However, the annotation of statements is clearly determined by the binding times of expressions, and we have already argued that the rules [Return] and [Call] are correctly transformed into constraints, so it suffice to consider expressions.

Theorem 5.1 *Let the scenario be as in Definition 4.1, i.e. p is a well-annotated two-level program. Let p' be the corresponding Core C program. Suppose S is a minimal solution to $\mathcal{C}(p')$.*

1. *For all variables v: $\tau(v) = \mathcal{T} \Leftrightarrow S(T_v) = \mathcal{T}$*
2. *For all functions f: $\pi(f) = \mathcal{T} \Leftrightarrow S(T_f) = \mathcal{T}$*
3. *For all expressions s: $\tau \vdash^{exp} e : \mathcal{T}, \mathcal{T} \preceq^{lift} \mathcal{T}' \Leftrightarrow S(T_e) = \mathcal{T}'$*

Thus, S is a minimal solution to $\mathcal{C}(p')$ iff p is well-annotated.

The 1 and 2 correspond to 1, 2 and 4 in Definition 4.1.

Proof 1, 2: We have already argued for 1 and 2 in the definition, and 4 is captured by the dependency $T_e \rhd^? T_f$ which are added for all statements, cf. Figure 6.

$\mathcal{C}_{stmt}(s)$ = case s of

$[\![$`expr` e $]\!]$	$\Rightarrow$	$\{T_e \rhd^? T_f\} \cup \mathcal{C}_{exp}(e)$
$[\![$`goto` m $]\!]$	$\Rightarrow$	$\{\}$
$[\![$`if (`e`)` m n $]\!]$	$\Rightarrow$	$\{T_e \rhd^? T_f\} \cup \mathcal{C}_{exp}(e)$
$[\![$`return` e $]\!]$	$\Rightarrow$	$\{T_e \preceq^? T_f\} \cup \mathcal{C}_{exp}(e)$
$[\![$`call` x `=` f'`(`e_1`,...,`e_n`)` $]\!]$	$\Rightarrow$	$\{T_{e_i} \preceq^? T_{f'_i}, T_{f'} = T_x, T_x \rhd^? T_f\} \cup \mathcal{C}_{exp}(x) \cup \bigcup_i \mathcal{C}_{exp}(e_i)$

Figure 6: Constraints for statements.

3. The left to right implication is by induction on the height of the type inference tree. The right to left implication is by structural induction on the expression. □

Example 5.1 Suppose we want to suspend all references to non-local objects in residual recursive functions. This can accomplished by adding constraints $T_f \rhd^? T_p$ for all recursive functions f and pointers variables p which refer to non-local objects. The pointer analysis $\mathcal{P}(p)$ tells whether a pointer may point to a non-local object. **End of Example**

Observe that the left hand side of $\trianglelefteq^?$-constraints are always fully instantiated; that the constructor of types to the left of $\sqsubseteq^?$ constraints are either $*$ or $\times$, and that $\preceq^?$-constriants are (initially) over S, D or type variables only.

Below we show that every constraint set generated by $\mathcal{C}$ can be brought into a *normal form* by a set of solution preserving transformations, and that every constraint system in normal form has a solution.

5.3 Normal form and normalization

This section presents a set of rewrite rules which simplifies a constraint system so that a solution easily can be found. Let $\mathcal{C}$ be a constraint system. By $\mathcal{C} \Rightarrow^S \mathcal{C}'$ we denote the application of a rewriting rule resulting in system $\mathcal{C}'$ under substitution S (which may be the identity). Repeated applications are denoted by $\mathcal{C} \Rightarrow +^S\mathcal{C}'$. Exhaustive application is denoted by $\mathcal{C} \Rightarrow *^S\mathcal{C}'$[6] and the system $\mathcal{C}'$ is said to be a normal form of $\mathcal{C}$.

A transformation $\mathcal{C} \Rightarrow^S \mathcal{C}'$ is *solution preserving* if for all substitutions S', $S' \circ S$ is a solution to $\mathcal{C}$ iff S' is a solution to $\mathcal{C}'$.

In Figure 7 a set of weakly normalizing rewrite rules is shown. As usually, the T's are type variables whereas the $\mathcal{T}$'s represent binding-time types. Observe that no substitution with types but S, D and type variables is made.

The following theorem states that the rules in Figure 7 are normalizing, that is, every constraint system $\mathcal{C}$ has a normal form $\mathcal{C}'$ and it can be found by an exhaustive application. Furthermore, it characterizes the normal form which is unique (with respect to the rules in Figure 7).

Theorem 5.2 *Let p be a program and $\mathcal{C} = \mathcal{C}(p)$.*

1. *The rewrite rules in Figure 7 are solution preserving.*
2. *Every constraint system $\mathcal{C}$ has a normal form $\mathcal{C}'$, and it can be found by exhaustive application of the rules in Figure 7.*
3. *A normal form constraint system possesses the form*
   - $\mathcal{T} \preceq^? T$ *where* $\mathcal{T} \in \{S, T\}$.
   - $*\mathcal{T} \sqsubseteq^? T$ *where* $\mathcal{T} \in \{S, D, T\}$
   - $\mathcal{T}_1 \times \cdots \times \mathcal{T}_n \sqsubseteq^? T$ *where* $\mathcal{T}_i \in \{S, D, T\}$
   - $\mathcal{T}_1 \trianglelefteq^? \mathcal{T}_2$ *where* $\mathcal{T}_2 \neq D$
   - $\mathcal{T}_1 \rhd^? \mathcal{T}_2$ *where* $\mathcal{T}_1 \neq D$

   and there are no constraints $\mathcal{T} \trianglelefteq^? T$, $\mathcal{T}_1 \sqsubseteq^? T$, such that D is less than or equal to a subtype of $\mathcal{T}_1$.

[6] where, however, the constraint system is considered as a set and not as a multiset.

Proof 1. By case analysis of the rules. Suppose $\mathcal{C} \Rightarrow^S \mathcal{C}'$, and that S' is a substitution.

Rule 3.b: "⇒". If $S \circ S'(T) = \mathcal{T}$, then $S \circ S'(T_1) = S \circ S'(T)$ since static binding-time types subsumes static types. Thus, S' is a solution to $\mathcal{T} \trianglelefteq^? T_1$. Assume $S \circ S'(T) = D$. Then $S \circ S'(T_1) = S$ or $S \circ S'(T_1) = D$. In the former case, since binding-time types subsumes static types at the static level, $\mathcal{T}$ must be S. The latter case is trivial. "⇐". Obvious, since the domain of S and S' are disjoint.

Rule 3.c: "⇒". Suppose $S \circ S'(T) = *\mathcal{T}$. Then $S \circ S'(T_1) = \mathcal{T}$ by definition of $\sqsubseteq^?$. Recall that $\mathcal{T} \neq D$, so $\mathcal{C}'$ is solved by S. Suppose that $S \circ S'(T) = D$. Then $S \circ S'(T_1) = D$, and thus S a solution to $\mathcal{C}'$. "⇐". Obvious.

2. Notice that all rules but 3.a, 3.b and 3.c either remove a constraint or replace one with an equality constraint. Equality constraints are effectively removed by rules 4.a, 4.b and 4.c. Rule 3.b, 3.c and 3.d can only be applied a finite number of times, even though $\mathcal{T}$ is recursive, since there are only finite many $\sqsubseteq^?$ constraints, and no rule generates new.

3. By inspecting the rules. □

Example 5.2 Suppose `p` is a pointer to a struct which is passed to a function `g`: `p->y = y`; `g(p)`. Recall that if the struct is dynamic, the pointer must be dynamic too. In annotated Core C we have:

```
1:  expr assign:T13 struct:T10 indr:T11 var:T9 p.y
       = var:T12 y;
2:  call var:T18 gen_0 = g(var:T15 p)
```

where the variable `y` is assumed dynamic. The following constraints (plus some more) are generated:

`D`	$=^?$	`T7`	`y` *is dynamic*
`*((SxS))`	$\trianglelefteq^?$	`T22`	
`T15`	$\preceq^?$	`T22`	
`T8`	$=^?$	`T15`	*T8 is* `p`
`T12`	$\preceq^?$	`T10`	
`T13`	$\preceq^?$	`T10`	
`T7`	$\preceq^?$	`T12`	*T7 is* `y`

1. Normalization of $\preceq^?$

a	$\mathcal{C} \cup \{S \preceq^? S\}$	$\Rightarrow$	$\mathcal{C}$	
b	$\mathcal{C} \cup \{S \preceq^? D\}$	$\Rightarrow$	$\mathcal{C}$	
c	$\mathcal{C} \cup \{D \preceq^? D\}$	$\Rightarrow$	$\mathcal{C}$	
d	$\mathcal{C} \cup \{D \preceq^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{T =^? D\}$	
e	$\mathcal{C} \cup \{T \preceq^? S\}$	$\Rightarrow$	$\mathcal{C} \cup \{T =^? S\}$	
f	$\mathcal{C} \cup \{T \preceq^? D\}$	$\Rightarrow$	$\mathcal{C}$	

2. Normalization of $\sqsubseteq^?$

a	$\mathcal{C} \cup \{*D \sqsubseteq^? D\}$	$\Rightarrow$	$\mathcal{C}$	
b	$\mathcal{C} \cup \{*T \sqsubseteq^? D\}$	$\Rightarrow$	$\mathcal{C} \cup \{T =^? D\}$	
c	$\mathcal{C} \cup \{*T_1 \sqsubseteq^? T, *T_2 \sqsubseteq^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{*T_1 \sqsubseteq^? T, T_1 =^? T_2\}$	
d	$\mathcal{C} \cup \{D \times \cdots \times D \sqsubseteq^? D\}$	$\Rightarrow$	$\mathcal{C}$	
e	$\mathcal{C} \cup \{T_1 \times \cdots \times T_n \sqsubseteq^? D\}$	$\Rightarrow$	$\mathcal{C} \cup \{T_1 =^? D, \ldots, T_n =^? D\}$	
f	$\mathcal{C} \cup \{T_1 \times \cdots \times T_n \sqsubseteq^? T, T_1' \times \cdots \times T_n' \sqsubseteq^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{T_1 \times \cdots \times T_n \sqsubseteq^? T, T_1 =^? T_1', \ldots, T_n =^? T_n'\}$	

3. Normalization of $\trianglelefteq^?$

a	$\mathcal{C} \cup \{\mathcal{T} \trianglelefteq^? D\}$	$\Rightarrow$	$\mathcal{C}$	
b	$\mathcal{C} \cup \{\mathcal{T} \trianglelefteq^? T, T_1 \preceq^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{\mathcal{T} \trianglelefteq^? T, \mathcal{T} \trianglelefteq^? T_1, T_1 \rhd^? T, T_1 \preceq^? T\}$	
c	$\mathcal{C} \cup \{*\mathcal{T} \trianglelefteq^? T, *T_1 \sqsubseteq^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{*\mathcal{T} \trianglelefteq^? T, *T_1 \sqsubseteq^? T, \mathcal{T} \trianglelefteq^? T_1, T_1 \rhd^? T\}$	
d	$\mathcal{C} \cup \{\mathcal{T}_1 \times \cdots \times \mathcal{T}_n \trianglelefteq^? T, T_1 \times \cdots \times T_n \sqsubseteq^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{\mathcal{T}_1 \times \cdots \times \mathcal{T}_n \trianglelefteq^? T, T_1 \times \cdots \times T_n \sqsubseteq^? T,$ $\mathcal{T}_1 \trianglelefteq^? T_1, \ldots, \mathcal{T}_n \trianglelefteq^? T_n, T_1 \rhd^? T, \ldots, T_n \rhd^? T\}$	

4. Normalization of $=^?$

a	$\mathcal{C} \cup \{T =^? \mathcal{T}\}$	$\Rightarrow$	$\mathcal{C}'$	where $\mathcal{C}' = [T \mapsto \mathcal{T}]\mathcal{C}$
b	$\mathcal{C} \cup \{\mathcal{T} =^? T\}$	$\Rightarrow$	$\mathcal{C}'$	where $\mathcal{C}' = [T \mapsto \mathcal{T}]\mathcal{C}$
c	$\mathcal{C} \cup \{\mathcal{T}_1 =^? \mathcal{T}_2\}$	$\Rightarrow$	$\mathcal{C}$	

5. Normalization of $\rhd^?$

a	$\mathcal{C} \cup \{D \rhd^? T\}$	$\Rightarrow$	$\mathcal{C} \cup \{D =^? T\}$	

Figure 7: Normalization rewriting rules.

```
(T25xT10) ⊑?  T11
*(T11)    ⊑?  T9
T8        =?  T9
```

where the first is due to the dynamic `y`, and the second constraints the formal parameter (`T22`) of `g`.

Observe that due to `T7` $=^?$ `D`, we have `T12` $=^?$ `D`, and then `T10` $=^?$ `D`. By applying rule 3.c, we have `*((SxS))` $\trianglelefteq^?$ `T15`. Since `T15` $=^?$ `T8`, `T8` $=^?$ `T9`, we can apply rule 3.d, which adds the constraints `T11` $\rhd^?$ `T15` and `SxS` $\trianglelefteq^?$ `T11`. Applying rule 3.e to the latter and `(T25xD)` $\sqsubseteq^?$ `T11`, we get `T11` $=^?$ `D`, and then `T15` $=^?$ `D`. That is, the formal parameter of `g` and `p` are both dynamic. Due to the definition of $\sqsubseteq^?$, also `T25` gets dynamic. Thus, we end up with the substitution:

$$S = [\texttt{T7}, \texttt{T8}, \texttt{T9}, \texttt{T10}, \texttt{T11}, \texttt{T12}, \texttt{T13}, \texttt{T15}, \texttt{T22} \mapsto D]$$

and an empty constraint set. **End of Example**

5.4 Minimal solution

Given a normalized constraint system we seek a solution which maps as many type variables as possible to a static type. However, this is easy: map all un-constrained type variables to S and solve all remaining $\preceq^?$ and $\sqsubseteq^?$ constraints by equality.

Theorem 5.3 *Let $\mathcal{C}'$ be a normal form constraint system. It has a minimal solution, and it can be found by mapping un-constrained type variables to S and solving all inequalities by equality.*

Obvious, the solution constructed by Theorem 5.3 must be a minimal solution, since all normal form constraints are solved by equality, and during the normalization, a type variable has only been made dynamic when necessary due to the definition of a solution.

Proof Due to the structure of normal form constraint systems. □

The Theorems 5.2 and 5.3 are similar to Henglein's theorems for his analysis of the untyped lambda calculus [10].

The steps of the binding-time analysis can be summarized as follows: 1) Construct the constraint system $\mathcal{C}(p)$, 2) normalize according to Theorem 5.2 to derive a normal form constraint system $\mathcal{C} \Rightarrow *^S\mathcal{C}'$, 3) find a solution S' to the normalized system by applying Theorem 5.3. The substitution $S \circ S'$ is then a solution to $\mathcal{C}$, and it is a minimal solution.

The last step is then 4) which is to construct a two-level version of the original program.

5.5 From binding-time types to the two-level

Given a binding-time annotated Core C program, it is easy to convert it into a well-annotated two-level version. Observe that the static and dynamic rules in Figure 3 can be distinguished solely by looking at the types of the subexpressions. For example, a static `index` is characterized by a pointer constructor in the type of the left expression e_1, while in the dynamic version it is D.

```
while (constraint_list != NULL) {
  remove c from constraint_list;
  switch c {
     case m = n: union(m',n'); break;
     case m <= n: m' = find m; n = find n;
        switch (m',n') {
           case (S,S):case (D,D):case (S,D):case (T,D):
              break;
           case (T,S): case (D,T):
              union(m',n'); break;
           case (T1,T2):
              less(n') U= {m'}; dep(m') U= {n'}; break;
           case (* m1,D):
              union(find m1, n'); break;
           case (* m1,T):
              less(n') U= {m'};
              union_subtype(n'); split_type(n'); break;
           case (x ms,D):
              union(find ms,n'); break;
           case (x ms,T):
              less(n') U= {m'};
              union_subtype(n'); split_type(n'); break;
           }
      case m <| n: m' = find m; n' = find n;
         switch (m',n') {
           case (_, T):
              type(n') U= {m'}; split_type(n'); break;
           case (_,_):
              break;
           }
      }
}
```

Figure 8: Constraint normalization algorithm.

6 Experiments

We have made an implementation of (an extended version of) the binding-time analysis, and integrated it into a partial evaluator for a C subset [2,4]. The implementation uses an almost linear time normalization algorithm similar to the one invented by Henglein [10], but extended to accommodate with the $\trianglelefteq^?$ constraints, and the somewhat different normalization rules.

The general idea behind the efficient algorithm is that it is possible to normalize by looking at each constraint one time only. To this purpose, the representation of a type variable T is equipped with three additional fields: `less(`T`)`, `type(`T`)`, and `dep(`T`)`. The first one is a list of types which are $\preceq^?$- or $\sqsubseteq^?$-less-than T. The second is a list of types which are $\trianglelefteq^?$-less-than T. The third is a list of type variables T' where $T \rhd^? T'$. The latter can be initialized during the constraint generation, and there will thus be no dependency constraints in the system.

The normalization algorithm is sketched in Figure 8. The `=` represents $=^?$, `<=` implements $\preceq^?$ and $\sqsubseteq^?$, and `<|` the $\trianglelefteq^?$ constraint.

Unification is implemented by the means of find-union data structures [1,18]. Recall that there is no need for substitutions with types but S, D and type variables during the normalization. The `union()` function takes to terms and unify these. If one of the terms are D, the all type variables appearing the `dep` list is furthermore unified with D. The function `union_subtype()` implements normalization rules 2.c and 2.f by inspecting the `less` lists. Finally, the function `split_type()` implements rules 3.b, 3.c and 3.d using the `type` and `less` lists.

The latter functions may add new constraints to the constraint list, but since the number of generated constraints during the normalization is bounded by the program size, the whole normalization run in almost linear time in size of the program length.

The normalization loop is run until no more constraints remain. The instantiation step is now simple: for every type variable T, if the `type(`T`)` list is non-empty, otherwise unify with S.

Example 6.1 Analyzing the list allocation program (Example 1.1) we get program as shown in Figure 9. The dots represent the recursive type of `struct List`. The number of generated constraints was 101.

End of Example

7 Related work

The use of two-level languages to specify binding-time separation originates from the Nielsons [17], and has later been adopted to various languages, *e.g.* Scheme and C [2,6,9].

Gomard investigated binding-time analysis of an untyped lambda calculus using a modified version of Algorithm W [8], and Henglein gave an efficient algorithm based on constraint system solving [10]. Bondorf and Jørgensen has extended Henglein's algorithm to a subset of Scheme, and proved their analysis correct [6]. The analysis described in this paper stems from [2], but has later been considerably simplified and implemented.

8 Conclusion and future work

We have considered binding-time separation of a subset the pragmatically oriented C programming language, including pointers and dynamically memory allocation. In the first part, we gave a semantic definition of well-annotatedness specified in form of a two-level Core C language with explicit binding-time annotations.

Next we described binding-time analysis based on non-standard type inference and implemented via constraint system solving. The analysis is, due to its simplicity, easy to prove correct, and it can be implemented efficiently on the computer. To our knowledge it is the first binding-time analysis for a "real" imperative language including pointers and structured values. Furthermore we have implemented the analysis and found that it works fast in practise.

For practical use, a limitation in the analysis is the monovariant treatment of functions. By duplicating functions according to the number of uses, a "poor mans" polyvariance can be obtained, but this should clearly be integrated into the analysis. Furthermore, the analysis should be extended to cope with pointers to functions and the union data type. The former is easy since a C function pointer can only point to one of the user defined functions. Unions can be handled by introducing disjoint sum into the type system.

It still remain to extend the analysis to full C. A major obstacle is, however, the lack of a clear semantic definition, and to cope with all "possible" C programs, the analysis may turn out to be overly conservative.

```
main::D (n::S key::S data::D  )
{
  list::(SxDx*((SxDx*(...))))
  p::*((SxDx*((SxDx*(...)))))
  1:  expr assign:*((SxDx*((SxDx*(...))))) var:*((SxDx*((SxDx*(...))))) p
      = addr:*((SxDx*((SxDx*(...))))) var:(SxDx*((SxDx*(...)))) list;
  2:  if (var:S n) 3 8;
  3:  expr assign:*((SxDx*((SxDx*(...))))) var:*((SxDx*((SxDx*(...))))) p
      = assign:*((SxDx*((SxDx*(...))))) struct:*((SxDx*((SxDx*(...)))))
           indr:(SxDx*((SxDx*(...)))) var:*((SxDx*((SxDx*(...))))) p.next
      = alloc:*((SxDx*((SxDx*(...))))) (List);
  4:  expr assign:S struct:S indr:(SxDx*((SxDx*(...)))) var:*((SxDx*((SxDx*(...))))) p.key = var:S n;
  5:  expr assign:D struct:D indr:(SxDx*((SxDx*(...)))) var:*((SxDx*((SxDx*(...))))) p.data
      = assign:D var:D data = binary:D var:D data - cst:S 1;
  6:  expr assign:S var:S n = binary:S var:S n - cst:S 1;
  7:  goto 2;
  8:  expr assign:*((SxDx*((SxDx*(...))))) var:*((SxDx*((SxDx*(...))))) p
      = struct:*((SxDx*((SxDx*(...))))) var:(SxDx*((SxDx*(...)))) list.next;
  9:  if (binary:S var:S key != struct:S indr:(SxDx*((SxDx*(...)))) var:*((SxDx*((SxDx*(...))))) p.key) 10 12;
  10:  expr assign:*((SxDx*((SxDx*(...))))) var:*((SxDx*((SxDx*(...))))) p
       = struct:*((SxDx*((SxDx*(...))))) indr:(SxDx*((SxDx*(...)))) var:*((SxDx*((SxDx*(...))))) p.next;
  11:  goto 9;
  12:  return struct:D indr:(SxDx*((SxDx*(...)))) var:*((SxDx*((SxDx*(...))))) p.data;
}
```

Figure 9: Annotated version of the list allocation program.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] L.O. Andersen. C program specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. DIKU Student Project 91-12-17, 134 pages.

[3] L.O. Andersen. Partial evaluation of C and automatic compiler generation (extended abstract). In U. Kastens and P. Pfahler, editors, *Compiler Constructions—4th International Conference, CC'92 (LNCS 641)*, pages 251–257. Springer-Verlag, October 1992.

[4] L.O. Andersen. *Partial Evaluation of C*, chapter 11 in Partial Evaluation and Automatic Compiler Generation, N.D. Jones, C.K. Gomard, P. Sestoft. Prentice-Hall, 1993. (To appear).

[5] A. Bondorf. Automatic autoprojection of higher order recursive equations. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark. LNCS, vol 432*, pages 70–87. Springer Verlag, May 1990.

[6] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, 1993. (To appear).

[7] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. ACM, 1990.

[8] C.K. Gomard. Partial type inference for untyped functional programs. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 282–287. ACM, 1990.

[9] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[10] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (LNCS, vol. 523)*, pages 448–472. ACM, Springer Verlag, 1991.

[11] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 154–165. ACM, 1991.

[12] ISO/IEC 9899:1990 International Standard. *Programming Languages—C*, 1990.

[13] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[14] B.W. Kernighan and D.M. Ritchie. *The C programming language (Draft-Proposed ANSI C)*. Software Series. Prentice-Hall, second edition edition, 1988.

[15] J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Dep. of Computing Science, University of Glasgow, Glasgow G12 8QQ, 1990.

[16] T.Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, Dept. of Comp. Science, University of Copenhagen, Mar 1989.

[17] H.R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[18] R. Tarjan. *Data Structures and Network Flow Algorithms*, volume CMBS 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.