Shape-Region and Effect Inference for C(*IL*)

Iago Abal[*]

April 19, 2016

Abstract

I describe a type-and-effect system for the C programming language based on Talpin-Jouvelot's previous work on *Polymorphic Type Region and Effect Inferece* (1992). The purpose of such an effect system is not to enforce a typing discipline, but to serve as a program abstraction to be model checked in search of bugs. Without loss of generality, I formulate this system on CIL, the *C Intermediate Language*. Any C program can be transformed into CIL, which is itself a subset of C, crucially, without implicit type conversions.

1 Introduction

As part of my PhD thesis, I am interested in efficient and effective approaches to bug finding. Purely syntax-based code scanners are fast but do not trivially find deep bugs, such as those that span multiple function calls. Semantic static analyzers do find deep bugs, but are generally slow and may not fit well into the programmers work-flow. The analysis of 42 Linux bugs from a previous study [1] suggests that conceptually simple resource manipulation bugs occur very often in practice. While simple, if these bugs span multiple functions they are virtually impossible to spot without computing some amount of semantic information. I propose the use of computational effects as a lightweight program abstraction, that enables simple code scanning techniques to find such bugs. I argue that side-effect information can be inferred efficiently, and that such a *lightweight semantic code scanning* technique would offer a good compromise between efficiency and precision.

For this I have adapted Talpin-Jouvelot's work on type-and-effect inference [5] to the C programming language. Given that type casts make C types uninformative of objects' runtime representation, and inspired by standard work on pointer analysis [4, 7], I define a lower-level type language based on memory *shapes*. In my system shapes replace C types, and this have lead us to refer to it as a *shape*-and-effect system. Thanks to *shape polymorphism*, shapes are unaffected by well-defined type casts, such as those used to workaround type genericity in C, and are suitable for tracking points-to relations. This shape-and-effect system is not intended to enforce a typing discipline that would rule out resource manipulation errors. It is however intended to build a program

[*]Working on my PhD at IT University of Copenhagen, under the supervision of Andrzej Wąsowski and Claus Brabrand.

abstraction based on memory shapes and computational effects, and to do so efficiently. All in all, my contributions are:

1. A pragmatic adaptation of Talpin-Jouvelot's type-and-effect inference system [5] to the C programming language. Without loss of generality, this system is formulated on CIL —the *C Intermediate Language*.
2. An implementation of this system, which available online under an open source license.[1]
3. A discussion of practical implementation considerations.

Note. For brevity we will not discuss the CIL representation here, but I refer the reader to CIL's 1.7.3 abstract syntax.[2] The base C type system is also not discussed in detail here, but I will comment on specific issues when necessary.

2 The Shape Language

Memory regions

Shapes are annotated with memory regions ρ, which are abstractions of the concrete locations where objects are stored in memory. The shape-and-effect system integrates a *flow-insensitive may-alias analysis*, in the spirit of Steengaards's [4]. If two terms are assigned the same shape and memory regions, that indicates that such terms may denote the same object in memory —i.e. they are aliased.

$$\textit{regions} \quad \rho \quad : \quad \varrho \mid \rho.x_i$$

A region ρ has the form $\varrho\overline{.x_i}$ where ϱ is the *host region*, and $\overline{.x_i}$ a (possibly empty) string of field accessors. The host abstracts the memory region where the object is stored, while the fields specify the *offset* of interest. Note that regions' hosts are abstracted, while offsets are concrete. For practical reasons, my system tracks aliasing between hosts, but not between offsets. Aliasing between offsets is mostly introduced through type casts, and the result is often implementation dependent. A sensible heuristic is to assume that $\varrho.x$ and $\varrho.y$ are probably not aliased if x and y are fields of the same structure type, but that may be aliased if they belong to different structures.

Memory shapes

A *shape* approximates the memory representation of an object [3, 4]. Shapes record points-to relations between references, as identified by their associated regions, allowing to identify which references may be manipulated by an expression. We split shapes into r-value (Z^R), f-value (Z^F), and l-value (Z^L) shapes. The shape language resembles C's type language, without atomic types like `int`, but with *bottom shapes* ($\bot$) and *shape variables* (ζ). We use the following terms to represent shapes:

[1]`https://eba.wikit.itu.dk`

[2]`https://github.com/cil-project/cil/blob/cil-1.7.3/src/cil.mli`

$$\begin{array}{llcl}
\textit{r-value shapes} & Z^R & : & \bot \;\mid\; \mathsf{ptr}\ Z^L \;\mid\; \mathtt{struct}\ t\ \mathtt{\{}\ \overline{Z^R{}_i\ x_i}\ \mathtt{\}} \;\mid\; \zeta \\
\textit{f-value shapes} & Z^F & : & Z^R \;\mid\; Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R \\
\textit{l-value shapes} & Z^L & : & \mathsf{ref}_\rho\ Z^F
\end{array}$$

R-value shapes denote the shape of r-value objects. An *atomic* shape $\bot$ denotes objects that have no relevant structure. Floating-point values, for instance, have $\bot$ shape. Integer values do not have predefined shapes in this system, because integers can be interpreted as pointers. The shape assigned to an integer will depend on how this integer value is used. Pointer expressions have pointer shapes, $\mathsf{ptr}\ Z^L$, where Z^L is the shape of the target reference cell of the pointer. A pointer represents the *address* of a reference cell, and therefore a pointer shape necessarily points to a reference shape. (An integer value may have a pointer shape if used as a pointer.) Arrays are flattened and treated as regular pointers, as we trade precision for simplicity, but preserve soundness. Structure shapes are composed by fields with associated r-value shapes. All fields of the structure are stored in the same memory region, and we use the own fields' name to *symbolically* refer to the offset where the field objects are stored. (Structure shapes do not capture the precise memory layout of structure objects.) *Shape variables* ζ enable safe type genericity via shape polymorphism. Our shape-and-effect system can assign polymorphic shapes to functions that manipulate data of arbitrary shapes through the masquerading of pointers as integers. For instance, functions to manipulate a linked list of integers are effectively shape polymorphic, since integers can encode pointers to other objects. (In C we commonly use pointers to `void` for this, but it can be done with integers too. Our implementation handles both forms.)

F-value shapes extend r-value shapes with function shapes. A function shape $Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R$ maps a tuple of reference shapes, corresponding to the formal parameters, to a value shape, corresponding to the result type. The base type system is only interested in the values that are passed to the function, thus function parameters are described by r-value types. Whereas our shape-and-effect system knows that actual parameters are in fact stored, for instance, in stack variables, and wants to track effects on them, hence giving function parameters l-value shapes. The returned value is an r-value expression (hence Z^R). Function shapes carry a so-called *latent effect*, φ, which accounts for the actions that *may* be performed upon invocation of the function.

L-value shapes, $\mathsf{ref}_\rho\ Z^R$, denote *references* to r-value objects. (A reference can point to another reference by holding a pointer to it.) Here, ρ is the memory region that identifies the location of the reference cell in the heap, and Z^R is the shape of the objects that it holds.

Computational effects

Types describe values, while *effects* describe computational properties of evaluation. For instance, from types perspective `++y * x` evaluates to an integer value. From memory effects perspective, it reads from locations ρ_x and ρ_y, and writes to ρ_y. Effects are a framework to reason about such and similar computational behaviors of programs.

Let F_X be a set of discrete distinguishable effects, and let $\textsc{Effect} = \langle \mathcal{P}(F_X), \sqsubseteq \rangle$ be the complete *power set lattice* of F_X. An example set of effects $\varphi \in \textsc{Effect}$ can be $\varphi = \{\underline{\mathit{read}}_{\rho_x}, \underline{\mathit{read}}_{\rho_y}, \underline{\mathit{write}}_{\rho_y}\}$. It records reading variables x and y, and writing y; where x and y have shape $\mathsf{ref}_{\rho_x}\ Z_1$ and $\mathsf{ref}_{\rho_y}\ Z_2$ respectively, for some $Z_{\{1,2\}}$. A set of effects specifies the effects that *may* result from an evaluation, disregarding the order—so it is a *flow-insensitive over-approximation.* We describe effects syntactically using the following terms:

$$\begin{array}{llcl} & F_X\ \ f_X & : & \varepsilon(\overrightarrow{\rho}) \\ \text{EFFECT} & \varphi & : & \emptyset \ \mid\ \{f_X\}\ \mid\ \xi\ \mid\ \varphi_1 \cup \varphi_2 \end{array}$$

We do not make further assumptions regarding the elements of F_X, or individual effects. Any effect constructor ε applied to a number of memory locations ρ makes a valid effect. We do assume the existence of $\underline{read}_\rho$ and $\underline{write}_\rho$ effects—representing reading and writing of memory locations. Finally, effect variables (ξ) allow for effect polymorphism.

3 Typing rules

Environments

An environment Γ maps regular variables x to reference shapes.

$$\Gamma(x) = \mathsf{ref}_\rho\ Z$$

and function variables f (introduced by function definitions) to *shape schemes*:

$$\Gamma(f) = \forall\ \overrightarrow{\zeta\varrho\xi}.\ \mathsf{ref}_{\varrho_0}\ (Z_1^L \times \cdots \times Z_n^L\ \xrightarrow{\varphi}\ Z_0^R) \qquad \text{where } \varrho_0 \notin \overrightarrow{\varrho}$$

A function shape scheme, or polymorphic function shape, is a function shape that is quantified over those type (shape, region, effect) variables on which the function's definition poses no constrain, and therefore can be freely instantiated at each call site. If φ is of the form $\varphi' \cup \xi_0$ where $\xi_0 \in \overrightarrow{\xi}$, we say that f is effect polymorphic: the effect of f is extended by the instantiation of ξ_0. In general it is unsound to generalize reference types [6]. But we can safely generalize function references because they are immutable. (In practice functions' code resides in a read-only *text segment.*) The memory region ϱ_0 identifies the function; it is used to track calls to it through function pointers, and it cannot be generalized (thus $\varrho_0 \notin \overrightarrow{\varrho}$).

Shape-type compatibility

An fvalue shape Z is *compatible* with a type T, written $Z \leq T$, if the shape of an object with C type T may be correctly described by Z. There may be multiple shapes compatible with a given type, and vice-versa. For instance, a value of type `int` may have shape $\bot$, if it is a plain integer number, or shape $\mathsf{ptr}\ \mathsf{ref}_\rho\ Z$ (for some Z) if it denotes a memory address.

Figure 1 shows the rules that define $Z \leq T$. Intuitively, shape-type compatibility requires that the given shape and type are structurally equivalent, with three singularities. First, any r-value shape is compatible with a C integer type (INT); or, in other words, integer values can be used to encode arbitrary r-value objects at runtime. Second, any pointer shape is compatible with `void*` (VOID-PTR). Third, function shapes capture the storage location of function parameters, which is ignored by function types (FUN).

$$\textsc{Bot-Float}\ \frac{T \in \{\texttt{float}, \texttt{double}\}}{\bot \leq T} \qquad \textsc{Bot-Void}\ \frac{}{\bot \leq \texttt{void}}$$

$$\textsc{Int}\ \frac{T \in \{\texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{long long}\}}{Z \leq T} \qquad \textsc{Void-Ptr}\ \frac{}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \leq \texttt{void*}}$$

$$\textsc{Ptr}\ \frac{Z \leq T}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \leq T\texttt{*}} \qquad \textsc{Struct}\ \frac{Z_i \leq T_i / i \in [0, n]}{\texttt{struct}\ t\ \texttt{\{}\ \overline{Z_i\ x_i}\ \texttt{\}} \leq \texttt{struct}\ t\ \texttt{\{}\ \overline{T_i\ x_i}\ \texttt{\}}}$$

$$\textsc{Fun}\ \frac{Z_i \leq T_i / i \in [0, n]}{\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n\ \xrightarrow{\varphi}\ Z_0 \leq T_1 \times \cdots \times T_n\ \rightarrow\ T_0}$$

Figure 1: Shape-type compatibility, $\leq\ \subseteq$ Shape $\times$ Type.

Castable shapes

A shape Z is *castable* to (or *compatible* with) another shape Z', written $Z \rightsquigarrow Z'$, if an object with shape Z could also be interpreted as having shape Z'. Figure 2 shows how to determine whether two shapes are castable. The $\rightsquigarrow$ relation is reflexive (Refl) and transitive (Tran). Two pointer shapes are compatible if the former's pointee shape is castable to the latter's pointee shape (Ptr). C structure types add three tricky scenarios to consider: *a)* a pointer to a structure can be converted into a pointer to one of its fields; *b)* a pointer to a structure field can be used to obtain a pointer to the structure itself; and *c)* a pointer to a structure can be cast to a pointer of another arbitrary structure.

The first case allows us to zoom into a field of a structure, and it is simple to support (Struct-Field). The second case allows us to zoom out, if we have previously zoomed into a field; this seems to require a more complex shape language than the one presented here. This system accepts casts to a container structure, but it cannot track whether the source shape has been previously obtained by zooming in the same structure shape (Field-Struct). The third case requires a precise knowledge of the layout of structures in memory, which is implementation dependent. This system accepts a cast between arbitrary structure shapes if there is a (possibly empty) prefix of their fields that are castable (Struct). (If there is no such prefix then the cast is still accepted!) In practice this means that, if rules Struct-Field and Field-Struct are used, we will loose track of certain bits of shape-region information. The concrete details will be covered in Sect. 4.

Strictly speaking, C does not directly allow these casts between structure types, but it does allow them when the casts are between pointers to structure types. Once we assume that programs have been type-checked according to C rules, we can drop the requirement of having to cast between pointers. In practice, most of these casts require adding an appropriate offset to a base pointer. For instance, from a pointer to one field in a structure, and by subtracting the offset of that field, we can recover a pointer to the container structure (see Linux macro `container_of`). Pointer arithmetic is not captured by the shape language hence the $\rightsquigarrow$ relation assumes that the programmer has done the right pointer arithmetic.

$$\textsc{Refl}\ \frac{}{Z \rightsquigarrow Z} \qquad \textsc{Tran}\ \frac{Z_1 \rightsquigarrow Z_2 \qquad Z_2 \rightsquigarrow Z_3}{Z_1 \rightsquigarrow Z_3} \qquad \textsc{Ptr}\ \frac{Z \rightsquigarrow Z'}{\mathsf{ptr}\ \mathsf{ref}_\rho\ Z \rightsquigarrow \mathsf{ptr}\ \mathsf{ref}_{\rho'}\ Z'}$$

$$\textsc{Fun}\ \frac{Z'_i \rightsquigarrow Z_i\ /\ i \in [1,n] \qquad \varphi' \sqsupseteq \varphi \qquad Z_0 \rightsquigarrow Z'_0}{\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi} Z_0 \rightsquigarrow \mathsf{ref}_{\rho'_1}\ Z'_1 \times \cdots \times \mathsf{ref}_{\rho'_n}\ Z'_n \xrightarrow{\varphi'} Z'_0}$$

$$\textsc{Struct-Field}\ \frac{\exists j.\ Z_j \rightsquigarrow Z'}{\mathtt{struct}\ t\ \{\ \overline{Z_i\ x_i}\ \} \rightsquigarrow Z'} \qquad \textsc{Field-Struct}\ \frac{\exists j.\ Z' \rightsquigarrow Z_j}{Z' \rightsquigarrow \mathtt{struct}\ t\ \{\ \overline{Z_i\ x_i}\ \}}$$

$$\textsc{Struct}\ \frac{\forall k \in [1,n].\ Z_k \rightsquigarrow Z'_k}{\mathtt{struct}\ t\ \{\ \overline{Z_i\ x_i}\ \} \rightsquigarrow \mathtt{struct}\ u\ \{\ \overline{Z'_j\ y_j}\ \}}$$

Figure 2: Castable shapes, $\rightsquigarrow\ \subseteq \textsc{Shape} \times \textsc{Shape}$.

$$\textsc{Var}\ \frac{\Gamma(x) = \mathsf{ref}_\rho\ Z}{\Gamma \vdash_L x : \mathsf{ref}_\rho\ Z\ \&\ \emptyset} \qquad \textsc{Deref}\ \frac{\Gamma \vdash_E E : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z\ \&\ \varphi}{\Gamma \vdash_L *E : \mathsf{ref}_\rho\ Z\ \&\ \varphi}$$

$$\textsc{Fun}\ \frac{\Gamma(f) = \forall\ \overrightarrow{\zeta\varrho\xi}.\ \mathsf{ref}_{\varrho_0}\ Z \qquad Z = Z^L_1 \times \cdots \times Z^L_n \xrightarrow{\varphi} Z^R_0}{\Gamma \vdash_L f : \mathsf{ref}_{\varrho_0}\ (Z[\overrightarrow{\zeta \mapsto Z'}][\overrightarrow{\varrho \mapsto \rho'}][\overrightarrow{\xi \mapsto \varphi'}])\ \&\ \emptyset}$$

$$\textsc{Index}\ \frac{\Gamma \vdash_L L : \mathsf{ref}_{\rho_1}\ Z_1\ \&\ \varphi_1 \qquad \Gamma \vdash_E E : Z_2\ \&\ \varphi_2}{\Gamma \vdash_L L\mathtt{[}E\mathtt{]} : \mathsf{ref}_{\rho_1}\ Z_1\ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Field}\ \frac{\Gamma \vdash_L L : \mathsf{ref}_\rho\ \mathtt{struct}\ t\ \{\ \overline{Z_i\ x_i}\ \}\ \&\ \varphi}{\Gamma \vdash_L L.x_j : \mathsf{ref}_{\rho.x_j}\ Z_j\ \&\ \varphi}$$

Figure 3: Typing rules for lvalues, $\vdash_L\ \subseteq \textsc{Env} \times \textsc{Lval} \times \textsc{Shape} \times \textsc{Effect}$.

Lvalues

An *lvalue* always denotes a memory location, therefore has shape $\mathsf{ref}_\rho\ Z$. An lvalue is formed by a *host* (or *base*) and an *offset*. Figure 3 shows the typing rules for lvalues. The judgment $\Gamma \vdash_L L : Z\ \&\ \varphi$ states that, under the environment Γ, the lvalue expression L has shape Z, and its evaluation produces φ effects.

The shape of a variable x is obtained directly from the environment (Var). Pointer dereferencing proceeds by evaluating an expression E, which produces φ side effects, and obtaining the reference object associated to the resulting memory address (Deref). For instance, given $p : \mathsf{ref}_{\rho_2}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_1}\ Z$, evaluating $*p$ requires fetching the pointer value stored in ρ_2, which has effect $\underline{read}_{\rho_2}$. Dereferencing a memory address has no additional effect: conceptually we use the address to look up the reference in a table. Every use of a function variable is given an arbitrary instance of the function's shape scheme (Fun). This instance is generated by substituting quantified variables with concrete shapes, regions and effects. In a typing derivation, these will depend on the calling context: the actual parameters passed to the function, and the expected shape of the function's result in that context.

A reference can be indexed obtaining a new reference to a given integer offset (Index). Note that an array is indexed by first obtaining a reference to its elements by rule Deref. (Array shapes are flattened and array objects have regular pointer shapes.) For simplicity, the index offset is ignored in the final region; in the future, we may consider the introduction of *indexed regions* (ρ`[`E`]`). For this, I must find a reasonable way to handle the occurrence of arbitrary expressions in regions. We can also obtain a reference to any field of a structured object (Field). Contrary to arrays, structure shapes are not collapsed: each field is stored at an offset of the structure memory location, identified by the field name ($\rho.x_j$). Such a precise treatment of structures is of key importance when analyzing real C programs [3, 7].

Expressions

Expressions denote *values* and have either rvalue or function shapes. Note that CIL expressions are *side-effect free*, but evaluating an expression involves reading memory locations, and such reads are recorded as effects too. Figure 4 introduces the typing rules for expressions. The judgment $\Gamma \vdash_E E : Z \ \& \ \varphi$ specifies that, in the environment Γ, evaluating the expression E results in a value of shape Z and produces φ effects.

From memory shape perspective, constants are divided into three groups. Constants of C types that should not be used to encode pointers have $\perp$ shape (Con-Bot). String constants, of type `char*`, have shape $\mathsf{ptr}\ \mathsf{ref}_\rho \perp$ (Con-Str). String literals are statically allocated into some arbitrary region ρ. Constants of integer types (`short` or larger) may encode pointers and thus can be given arbitrary rvalue shapes Z (Con-Int). Similarly to lvalue typing rule Fun, the concrete instantiation for Z will depend on the context where the constant is used.

In C there is no explicit operator to read from a memory cell. Instead, when an lvalue expression appears in a rvalue position this has the (implicit) effect of fetching the object stored in the corresponding memory cell (Lval). For instance, when a variable x is used as an rvalue this results in the *read* of the memory cell denoted by x, to fetch the value stored in it. The *address-of* operator (`&`) allows to view lvalues as rvalue expressions, on which it is possible to perform arithmetic. Obtaining the address of an lvalue does not add any additional effect.

Expressions `sizeof`(T) (Sizeof-T) and `alignof`(T) (Alignof-T) are statically resolved and produce no effects at runtime. (In fact, we could consider these constants.) With the exception of variable length arrays where `sizeof(`T`[`E`])`, which can be interpreted as `(`E`)*sizeof(`T`)`, requires the evaluation of the expression E (Sizeof-A). (The base C type checker shall check the restrictions that ANSI C imposes on the type T in order to be a valid argument of `sizeof` and `alignof`.) Expressions `sizeof`(E) (Sizeof-E) and `alignof`(E) (Alignof-E) produce no effects either: only the type of the expression is considered. The semantics of these operators suggests that the result of these expressions should not be interpreted as a pointer, therefore their shape is $\perp$.

Arithmetic, bitwise and logical operators take one or two expressions (operands), and produce a new value, but no additional effects. CIL conveniently distinguishes pointer arithmetic (Ptr-A) and pointer difference (Minus-PP) expressions. When subtracting two pointers we should check that both belong to the

$$\textsc{Con-Bot}\ \dfrac{\mathrm{typeof}(c) \in \{\texttt{_Bool}, \texttt{char}, \texttt{float}, \texttt{double}\}}{\Gamma \vdash_E c : \bot \ \&\ \emptyset} \qquad \textsc{Con-Str}\ \dfrac{\mathrm{typeof}(str) = \texttt{char*}}{\Gamma \vdash_E str : \mathsf{ptr}\ \mathsf{ref}_\rho\ \bot \ \&\ \emptyset}$$

$$\textsc{Con-Int}\ \dfrac{\mathrm{typeof}(i) \in \{\texttt{int}, \texttt{short}, \texttt{long}, \texttt{long long}\}}{\Gamma \vdash_E i : Z \ \&\ \emptyset} \qquad \textsc{Lval}\ \dfrac{\Gamma \vdash_L L : \mathsf{ref}_\rho\ Z \ \&\ \varphi}{\Gamma \vdash_E L : Z \ \&\ \varphi \cup \underline{read}_\rho}$$

$$\textsc{Addr}\ \dfrac{\Gamma \vdash_L L : \mathsf{ref}_\rho\ Z \ \&\ \varphi}{\Gamma \vdash_E \texttt{\&}L : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z \ \&\ \varphi} \qquad \textsc{Sizeof-T}\ \dfrac{T \neq T'\texttt{[}E\texttt{]}}{\Gamma \vdash_E \texttt{sizeof}(T) : \bot \ \&\ \emptyset}$$

$$\textsc{Sizeof-A}\ \dfrac{\Gamma \vdash_E E : Z \ \&\ \varphi}{\Gamma \vdash_E \texttt{sizeof}(T\texttt{[}E\texttt{]}) : \bot \ \&\ \varphi} \qquad \textsc{Sizeof-E}\ \dfrac{}{\Gamma \vdash_E \texttt{sizeof}(E) : \bot \ \&\ \emptyset}$$

$$\textsc{Alignof-T}\ \dfrac{}{\Gamma \vdash_E \texttt{alignof}(T) : \bot \ \&\ \emptyset} \qquad \textsc{Alignof-E}\ \dfrac{}{\Gamma \vdash_E \texttt{alignof}(E) : \bot \ \&\ \emptyset}$$

$$\textsc{Neg}\ \dfrac{\Gamma \vdash_E E : Z \ \&\ \varphi}{\Gamma \vdash_E \texttt{-}\ E : \bot \ \&\ \varphi} \qquad \textsc{B-Not}\ \dfrac{\Gamma \vdash_E E : Z \ \&\ \varphi}{\Gamma \vdash_E \texttt{~}\ E : \bot \ \&\ \varphi} \qquad \textsc{L-Not}\ \dfrac{\Gamma \vdash_E E : Z \ \&\ \varphi}{\Gamma \vdash_E \texttt{!}\ E : \bot \ \&\ \varphi}$$

$$\textsc{Int-A}\ \dfrac{\Gamma \vdash_E E_1 : Z \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : Z \ \&\ \varphi_2 \qquad \oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash_E E_1 \oplus E_2 : Z \ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Bit-A}\ \dfrac{\Gamma \vdash_E E_1 : Z \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : Z \ \&\ \varphi_2 \qquad \otimes \in \{\texttt{\&}, \texttt{^}, \texttt{|}, \texttt{<<}, \texttt{>>}\}}{\Gamma \vdash_E E_1 \otimes E_2 : Z \ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Ptr-A}\ \dfrac{\Gamma \vdash_E E_1 : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z_1 \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : Z_2 \ \&\ \varphi_2 \qquad \oplus \in \{\texttt{+}, \texttt{-}\}}{\Gamma \vdash_E E_1 \oplus E_2 : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z_1 \ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Minus-PP}\ \dfrac{\Gamma \vdash_E E_1 : \mathsf{ptr}\ \mathsf{ref}_{\rho_1}\ Z \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z \ \&\ \varphi_2 \qquad \rho_1 \equiv \rho_2}{\Gamma \vdash_E E_1 - E_2 : \bot \ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Cmp}\ \dfrac{\Gamma \vdash_E E_1 : Z \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : Z \ \&\ \varphi_2 \qquad \trianglelefteq \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{\Gamma \vdash_E E_1 \trianglelefteq E_2 : \bot \ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Bool-A}\ \dfrac{\Gamma \vdash_E E_1 : Z \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : Z \ \&\ \varphi_2 \qquad \odot \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash_E E_1 \odot E_2 : \bot \ \&\ \varphi_1 \cup \varphi_2}$$

$$\textsc{Question}\ \dfrac{\Gamma \vdash_E E_1 : Z_1 \ \&\ \varphi_1 \qquad \Gamma \vdash_E E_2 : Z \ \&\ \varphi_2 \qquad \Gamma \vdash_E E_3 : Z \ \&\ \varphi_3}{\Gamma \vdash_E E_1\ \texttt{?}\ E_2\ \texttt{:}\ E_3 : Z \ \&\ \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

$$\textsc{Cast}\ \dfrac{\Gamma \vdash_E E : Z \ \&\ \varphi \qquad Z \rightsquigarrow Z' \qquad Z' \leq T}{\Gamma \vdash_E \texttt{(}T\texttt{)}E : Z' \ \&\ \varphi}$$

Figure 4: Typing rules for expressions, $\vdash_E\ \subseteq \textsc{Env} \times \textsc{Exp} \times \textsc{Shape} \times \textsc{Effect}$.

same memory region, otherwise the result is not defined by the standard. The other cases are for integer arithmetic not involving pointers, however, through type casts, we could be operating with pointers masqueraded as integers: for instance, `(int)p + (int)q`. So this system does not restrict the shape of integer operands, and accepts many non-standard ways of pointer arithmetic (Int-A), even bitwise operations on pointers (Bit-A). While one does not expect to find many uses of pointer multiplication, or modulo arithmetic on pointers, these could make sense in specific scenarios. Remarkably, the operands' shapes must match, so operations on incompatible pointer shapes are not allowed. The result of arithmetic negation (Neg), bitwise not (B-Not), comparisons (Cmp), and logical connectives (L-Not, Bool-A), should not be interpreted as a pointer

$$\textsc{Set}\ \frac{\Gamma \vdash_L L : \mathsf{ref}_\rho\ Z \ \&\ \varphi_1 \qquad \Gamma \vdash_E E : Z \ \&\ \varphi_2}{\Gamma \vdash_I L\ \texttt{=}\ E \ \&\ \varphi_1 \cup \varphi_2 \cup \underline{\mathit{write}}_\rho}$$

$$\textsc{Call}\ \frac{\Gamma \vdash_E E_0 : (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n) \xrightarrow{\varphi'} Z_0 \ \&\ \varphi_0 \qquad \Gamma \vdash_E E_i : Z_i \ \&\ \varphi_i / i \in [1,n]}{\Gamma \vdash_I E_0(E_1, \cdots, E_n) \ \&\ \varphi_0 \cup (\bigcup_{i\in[1,n]} \varphi_i) \cup \varphi'}$$

$$\textsc{Call-n-Set}\ \frac{\begin{array}{c}\Gamma \vdash_E E_0 : (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n) \xrightarrow{\varphi'} Z_0 \ \&\ \varphi_0 \\ \Gamma \vdash_E E_i : Z_i \ \&\ \varphi_i / i \in [1,n] \qquad \Gamma \vdash_L L : \mathsf{ref}_{\rho_0}\ Z_0 \ \&\ \varphi''\end{array}}{\Gamma \vdash_I L = E_0(E_1, \cdots, E_n) \ \&\ \varphi_0 \cup (\bigcup_{i\in[1,n]} \varphi_i) \cup \varphi' \cup \varphi'' \cup \underline{\mathit{write}}_{\rho_0}}$$

Figure 5: Typing rules for instructions, $\vdash_I\ \subseteq \textsc{Env} \times \textsc{Instr} \times \textsc{Effect}$.

and thus has $\perp$ shape.

The evaluation of a conditional expression E_1 `?` E_2 `:` E_3 first evaluates the guard E_1, and then it evaluates either to E_2 (if $E_1 \to \texttt{1}$) or to E_3 (if $E_1 \to \texttt{0}$). The shape of both branches, E_2 and E_3, must coincide. Since this system computes *may* effects, the overall effects are the sum of evaluating the three expressions, even though E_2 and E_3 will not be simultaneously evaluated.

Type casts allow us to interpret an object as having any arbitrary type, as long as the object's runtime representation matches the type. ANSI C defines which of these casts are guaranteed well-defined but, ultimately, the base C type checker will trust the programmer's judgment. Since this system tracks the memory shape of objects, it can do better and check that the target shape is compatible with the object's inferred type (Cast). We say that one shape is *castable* to another, written $Z \rightsquigarrow Z'$, if an object of shape Z' can be *view* as having shape Z'. The relation $\rightsquigarrow$ is reflexive, but not symmetric. Because multiple types are shape-compatible (for instance, $\mathsf{ptr}\ \mathsf{ref}_\rho\ Z$ is compatible with `int`, `int*`, `char*` and `void*`), most type casts are trivially accepted using reflexivity of $\rightsquigarrow$. The trickiest type casts are those involving structure types. For instance, in C one can cast a pointer to a structure as a pointer to the first field of the structure, and back. Further, it is possible to cast a pointer to a structure type as a pointer to a different structure type. This system does not handle all these casts gracefully, since that would require a more complex shape language and probably even more complex type inference algorithm.

Instructions

CIL instructions describe basic program steps without control flow. These correspond to C side-effectful expressions: assignments and function calls. Figure 5 shows the typing rules for instructions. The judgment $\Gamma \vdash_I I \ \&\ \varphi$ states that, under Γ, instruction I is valid and, when evaluated, produces effects φ.

An instruction lv `=` E writes the value resulting from the evaluation of E, into the memory cell denoted by lv (Set). The shape of E must match the shape of objects that lv stores. The base C type system is responsible for forbidding illegal writes to read-only references, such as function and `const` variables.

CIL distinguishes function calls where the result is ignored (Call), and

$$\textsc{Instr}\ \frac{\Gamma \vdash_I I \ \&\ \varphi}{\Gamma \vdash_S^Z I\texttt{;}\ \&\ \varphi} \qquad \textsc{Return}\ \frac{}{\Gamma \vdash_S^{\perp} \texttt{return;}\ \&\ \emptyset} \qquad \textsc{Return-E}\ \frac{\Gamma \vdash_E E : Z \ \&\ \varphi}{\Gamma \vdash_S^Z \texttt{return}\ E\texttt{;}\ \&\ \varphi}$$

$$\textsc{Label}\ \frac{\Gamma \vdash_S^Z S \ \&\ \varphi}{\Gamma \vdash_S^Z L\texttt{:}\ \ S\texttt{;}\ \&\ \varphi} \qquad \textsc{Goto}\ \frac{}{\Gamma \vdash_S^Z \texttt{goto}\ L\texttt{;}\ \&\ \emptyset} \qquad \textsc{Goto-E}\ \frac{\Gamma \vdash_E E : \mathsf{ptr}\ \mathsf{ref}_\rho\ Z' \ \&\ \varphi}{\Gamma \vdash_S^Z \texttt{goto}\ E\texttt{;}\ \&\ \varphi}$$

$$\textsc{Break}\ \frac{}{\Gamma \vdash_S^Z \texttt{break;}\ \&\ \emptyset} \qquad \textsc{Continue}\ \frac{}{\Gamma \vdash_S^Z \texttt{continue;}\ \&\ \emptyset}$$

$$\textsc{If}\ \frac{\Gamma \vdash_E E : Z_0 \ \&\ \varphi_0 \qquad \Gamma \vdash_S^Z S_1 \ \&\ \varphi_1 \qquad \Gamma \vdash_S^Z S_2 \ \&\ \varphi_2}{\Gamma \vdash_S^Z \texttt{if}\ E\ S_1\ \texttt{else}\ S_2 \ \&\ \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

$$\textsc{Switch}\ \frac{\Gamma \vdash_E E : Z_0 \ \&\ \varphi_0 \qquad \Gamma \vdash_S^Z S_i \ \&\ \varphi_i \ /\ i \in [1,n]}{\Gamma \vdash_S^Z \texttt{switch (}E\texttt{) \{}\ S_1 \cdots\ S_n\ \texttt{\}}\ \&\ \varphi_0 \cup (\bigcup_{i\in[1,n]} \varphi_i)} \qquad \textsc{Loop}\ \frac{\Gamma \vdash_S^Z S \ \&\ \varphi}{\Gamma \vdash_S^Z \texttt{while (1)}\ S \ \&\ \varphi}$$

$$\textsc{Seq}\ \frac{\Gamma \vdash_S^Z S_1 \ \&\ \varphi_1 \qquad \Gamma \vdash_S^Z S_2 \ \&\ \varphi_2}{\Gamma \vdash_S^Z S_1 S_2 \ \&\ \varphi_1 \cup \varphi_2}$$

Figure 6: Typing rules for statements, $\vdash_S\ \subseteq \textsc{Env} \times \textsc{Shape} \times \textsc{Stmt} \times \textsc{Effect}$.

function calls where the result is assigned to an lvalue (Call-n-Set). The reason being that function calls may have side-effects and thus cannot be CIL expressions. An expression $E_0(E_1, \cdots, E_n)$ invokes the function denoted by E_0 passing it the result of evaluating expressions E_1 to E_n as arguments. The shape of the actuals must be compatible with the shape of the formals. Such invocation introduces the function's *latent* effects (φ'). In the case of Call-n-Set, there are additional effects from evaluating and writing to the lvalue.

Statements

Statements add control flow to CIL instructions. Figure 6 shows the typing rules for statements. The judgment $\Gamma \vdash_S^Z S \ \&\ \varphi$ specifies that, in the context Γ of a function returning values of shape Z, the statement S is valid, and its evaluation may produce effects φ. Because this is a flow-insensitive *may* analysis, control flow is ignored, and the typing of statements is fairly straightforward. The effects of an statement are computed as the sum of the effects resulting from the evaluation of all its sub-expressions and sub-statements.

A semicolon converts an instruction into an statement (Instr). When returning the value of an expression, the shape of the expression must match the result shape of the enclosing function (Return-E). No restriction applies to functions returning `void`, ie. nothing (Return). Unstructured control-flow is basically ignored in a flow-insensitive analysis like this one (cf. Label, Goto, Goto-E, Break, and Continue). A *computed goto*, `goto` E`;`, is a GCC extension that allows to jump to a label through a pointer (Goto-E). The effects computed for a branching statement consider the potential evaluation of all its branches (If, Switch). Similarly, the effects of a looping statement are computed independently of how many times the loop is entered, or whether the entire loop body gets executed (Loop). Finally, statements may be executed in sequence (Seq).

$$\mathrm{Gen}_{\Gamma}^{\varrho_0}(Z) = \forall\ \overrightarrow{\zeta\rho\xi}.\ \mathsf{ref}_{\varrho_0}\ Z \text{ where } \overrightarrow{\zeta\rho\xi} = \mathrm{FV}(Z) \setminus (\mathrm{FV}(\Gamma) \cup \{\varrho_0\})$$

$$\mathrm{Observe}_{\Gamma,Z}(\varphi) = \{\varepsilon(\overrightarrow{\rho}) \in \varphi \mid \rho_i \in \mathrm{FV}(\Gamma) \cup \mathrm{FV}(Z)\} \cup \{\xi \in \varphi \mid \xi \in \mathrm{FV}(\Gamma) \cup \mathrm{FV}(Z)\}$$

$$\textsc{Def}\ \frac{\begin{array}{c}\Gamma' = \Gamma; x_1 : \mathsf{ref}_{\rho_1}\ Z_1; \cdots ; x_n : \mathsf{ref}_{\rho_n}\ Z_n \\ \Gamma'; x_{n+1} : \mathsf{ref}_{\rho_{n+1}}\ Z_{n+1}; \cdots ; x_m : \mathsf{ref}_{\rho_m}\ Z_m \vdash_S^{Z_0} S \,\&\, \varphi \\ \varphi' \sqsupseteq \mathrm{Observe}_{\Gamma',Z_0}(\varphi) \qquad Z_i \leq T_i\ /\ i \in [0,m]\end{array}}{\Gamma \vdash_D T_0\ \ f\texttt{(}T_1\ \ x_1, \cdots, T_n\ \ x_n\texttt{)}\ \texttt{\{}\ \overrightarrow{T_i\ x_i}^{n+1:m}\texttt{;}\ \ S\ \texttt{\}} : \mathrm{Gen}_{\Gamma}^{\varrho_0}(\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi'} Z_0)}$$

Figure 7: Typing of function definitions, $\vdash_D\ \subseteq \textsc{Env} \times \textsc{Def} \times \textsc{Shape-Scheme}$.

Function definitions

Figure 7 shows typing rules for non-recursive function definitions ($\textsc{Def}$). The judgment $\Gamma \vdash_D T_0\ \ f\texttt{(}T_1\ \ x_1, \cdots, T_n\ \ x_n\texttt{)}\ \texttt{\{}\ \overrightarrow{T_i\ x_i}^{n+1:m}\texttt{;}\ \ S\ \texttt{\}} : \forall\ \overrightarrow{\zeta\rho\xi}.\ \mathsf{ref}_{\varrho_0}\ Z$ states that, in the environment Γ, the definition of function f is valid and has shape scheme $\forall\ \overrightarrow{\zeta\rho\xi}.\ \mathsf{ref}_{\varrho_0}\ Z$. (Function definitions do not introduce side-effects.) Here, T_0 is the function's return type, $\overrightarrow{x_i}^{0:n}$ are the function's formal parameters, and $\overrightarrow{x_i}^{n+1:m}$ are local variables used in the function's body S. For this to hold, the function's body statement S must be valid in the environment Γ extended with the function's formal parameters and local variables. The shape of f's result and formal parameters are chosen to be compatible with f's type signature. The shape of f's local variables are chosen to fit their usage within S. The shape scheme of f is obtained by quantifying over the type (shape, region, effect) variables that are local to f's definition, that is, not known in Γ. Shape-region and effect generalization allows to infer more precise shape schemes, that can be instantiated (ie. specialized) at call site. Recursive definitions are typed in the usual way using monomorphic recursion.

Observable effects and subeffecting. $\mathrm{Observe}_{\Gamma',Z_0}(\varphi)$ masks any effect in φ that is not *observable* from a caller's perspective. An effect is observable if it can interfere with the program state at a call site: this happens if f performs operations on global regions known at definition site, on its own arguments, or on any object that is part of f's result. Operations on local variables such as loop counters can be safely masked as they cannot be observed from the outside. This type system supports subeffecting by taking as latent effect of f a *superset* of the observable effects of evaluating S. In other words, it is allowed to enlarge the latent effects of a function, in order to satisfy a subsequent typing constraint when using that function. Crucially, we can enlarge the effects of f with a fresh unconstrained effect variable ξ, so that f is given a shape scheme of the form $\forall\ \overrightarrow{\zeta\rho\xi}.\ \mathsf{ref}_{\varrho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\xi\cup\varphi'} Z_0)$. This allows φ' to be a precise approximation of f's evaluation effects, while ξ can be conveniently instantiated at call site to satisfy typing constraints.

4 Inference rules

In this section we derive an *inference algorithm* for the type system presented above, following the recipe given in [2]. The inference algorithm shall infer principal types and minimum sets of effects. A standard step is to replace all the *guesses* made in the declarative typing rules with the introduction of fresh type variables. Whenever shape equivalence is required, we collect equality constraints and solve them using an unification algorithm. A second crucial step for obtaining this inference algorithm is to replace latent effects in functions with effect variables [2], and separately collect (and solve) subeffecting constraints on those variables. This makes solving of shape equations tractable using a simple unification algorithm.

Subeffecting constraints

A *subeffecting constraint* written $\xi \sqsupseteq \varphi$ states that any solution for the effect variable ξ must include at least the effects φ.

A *system of subeffecting constraints* is denoted by κ. By construction, a system κ always admit at least one solution; we are interested in the *least* solution of κ. A system of subset inequalities can be solved using *chaotic iteration.*

The *restriction* of a constraint system κ on the effect variables $\overrightarrow{\upsilon}$ is defined as $\kappa_{\overrightarrow{\upsilon}} = \{\xi \sqsupseteq \varphi \in \kappa \mid \xi \in \overrightarrow{\upsilon}\}$.

Environment

An environment Γ maps regular variables x to reference shapes.

$$\Gamma(x) = \mathsf{ref}_\rho\ Z$$

and function variables f (introduced by function definitions) to *shape schemes*:

$$\Gamma(f) = \forall\ \overrightarrow{\zeta\varrho\xi}.\ \kappa \Rightarrow \mathsf{ref}_{\varrho_0}\ (Z_1^L \times \cdots \times Z_n^L \ \xrightarrow{\xi}\ Z_0^R) \qquad \text{where } \varrho_0 \notin \overrightarrow{\varrho}$$

where $\xi_i \sqsupseteq \varphi_i \in \kappa$ forces any instantiation of ξ_i to include *at least* the effects φ_i. If $\xi_0 \in \overrightarrow{\xi}$ the function is effect polymorphic.

Most general shape

The typing rules make guesses about shapes in several places, and these shapes are often required to be compatible with the given C types, written $Z \leq T$. In the inference system we instead compute the *most-general shape* of a C type T, written shape-of(T). Figure 8 shows the algorithm described by inference rules that must be applied from top to bottom.

Unification

Equality constraints on shapes are solved by a simple Robinson-like unification algorithm. We write $Z_1 \sim Z_2 = \theta$ to denote that Z_1 and Z_2 unify, and their most general unifier is the substitution θ. Figure 9 shows the unification algorithm for shapes. Although it is not completely sound, here unification

$$\textsc{Bot-Float}\ \frac{T \in \{\texttt{float}, \texttt{double}\}}{\text{shape-of}(T) = \bot} \qquad \textsc{Bot-Void}\ \frac{}{\text{shape-of}(\texttt{void}) = \bot}$$

$$\textsc{Int}\ \frac{T \in \{\texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{long long}\} \qquad \zeta \text{ fresh}}{\text{shape-of}(T) = \zeta}$$

$$\textsc{Void-Ptr}\ \frac{\varrho, \zeta \text{ fresh}}{\text{shape-of}(\texttt{void*}) = \mathsf{ptr}\ \mathsf{ref}_\varrho\ \zeta} \qquad \textsc{Ptr}\ \frac{\text{shape-of}(T) = Z \qquad \varrho \text{ fresh}}{\text{shape-of}(T\texttt{*}) = \mathsf{ptr}\ \mathsf{ref}_\varrho\ Z}$$

$$\textsc{Struct}\ \frac{\text{shape-of}(T_i) = Z_i / i \in [0, n]}{\text{shape-of}(\texttt{struct}\ t\ \texttt{\{}\ \overline{T_i\ x_i}\ \texttt{\}}) = \texttt{struct}\ t\ \texttt{\{}\ \overline{Z_i\ x_i}\ \texttt{\}}}$$

$$\textsc{Fun}\ \frac{\text{shape-of}(T_i) = Z_i \qquad \varrho_i, \xi \text{ fresh} \qquad i \in [0, n]}{\text{shape-of}(T_1 \times \cdots \times T_n\ \rightarrow\ T_0) = \mathsf{ref}_{\varrho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\varrho_n}\ Z_n\ \xrightarrow{\xi}\ Z_0}$$

Figure 8: Most general shape, shape-of(T) : Shape.

not only captures shape equality but also the *castability* relation on shapes introduced in the previous section (cf. rules Struct-Field, Field-Struct and Struct).

Unification of latent effects is trivial given that these are always captured by effect variables (Fun). Effect variables are subject to subeffecting constraints which are solved in a different step. Unification of reference shapes forces the unification of memory regions (Ref), which constitutes a flow-insensitive bidirectional alias analysis [4]. When unifying regions, field offsets are constants that cannot be unified (Fld-Fld): only aliasing between base regions is inferred. While potentially unsound, we have found this to be a reasonable implementation choice that simplifies unification of structure shapes and reduces the number of region variables to track.

Lvalues

Inference of lvalues takes an environment Γ, a constraint system κ, and an lvalue L; and computes a substitution θ, the shape $\mathsf{ref}_\rho\ Z$ of L, the effects φ that result from evaluating L, and a new constraint system κ'. We write this inference step as $\Gamma; \kappa \vdash^{\uparrow}_{L} L : \theta\ \&\ \mathsf{ref}_\rho\ Z\ \&\ \varphi\ \&\ \kappa'$. Figure 10 shows the inference rules for lvalues. There are two main differences with respect to the typing rules. First, as a result of unification, the inference rules produce and propagate subtitutions. Second, in rule Fun shape schemes are instantiated with fresh type variables rather than with arbitrary (*guessed*) shapes. Perhaps surpringly, there is no explicit usage of unification in these rules. The reason is that CIL makes all type conversions explicit, and thus it is guaranteed that lvalues and expressions will have the shape expected from the context. It is when inferring the shape of cast expressions that unification is performed.

Expressions

Inference of expressions takes an environment Γ, a constraint system κ, and an expression E; and computes a substitution θ, the shape Z of E, the effects

Unification of regions $\sim_R$: $\textsc{Region} \times \textsc{Region} \to \textsc{Subst}$.

$$\textsc{Eps-Fld}\ \frac{}{\varrho_1.\epsilon \ \sim_R\ \varrho_2.\overrightarrow{y} \ = \ \{\varrho_1 \mapsto \varrho_2.\overrightarrow{y}\}} \qquad \textsc{Fld-Eps}\ \frac{}{\varrho_1.\overrightarrow{x} \ \sim_R\ \varrho_2.\epsilon \ = \ \{\varrho_2 \mapsto \varrho_1.\overrightarrow{x}\}}$$

$$\textsc{Fld-Fld}\ \frac{}{\varrho_1.\overrightarrow{x} \ \sim_R\ \varrho_2.\overrightarrow{y} \ = \ \{\varrho_1 \mapsto \varrho_2\}}$$

Unification of shapes $\sim$: $\textsc{Shape} \times \textsc{Shape} \to \textsc{Subst}$.

$$\textsc{Bot}\ \frac{}{\bot \ \sim\ \bot \ = \ \mathrm{id}} \qquad \textsc{Var-L}\ \frac{\zeta \notin \mathrm{FV}(Z)}{\zeta \ \sim\ Z \ = \ \{\zeta \mapsto Z\}} \qquad \textsc{Var-R}\ \frac{\zeta \notin \mathrm{FV}(Z)}{Z \ \sim\ \zeta \ = \ \{\zeta \mapsto Z\}}$$

$$\textsc{Ptr}\ \frac{Z_1 \ \sim\ Z_2 \ = \ \theta}{\mathsf{ptr}\ Z_1 \ \sim\ \mathsf{ptr}\ Z_2 \ = \ \theta} \qquad \textsc{Ref}\ \frac{\rho_1 \ \sim_R\ \rho_2 \ = \ \theta_\rho \qquad \theta_\rho Z_1 \ \sim\ \theta_\rho Z_2 \ = \ \theta}{\mathsf{ref}_{\rho_1}\ Z_1 \ \sim\ \mathsf{ref}_{\rho_2}\ Z_2 \ = \ \theta\,\theta_\rho}$$

$$\textsc{Fun}\ \frac{\begin{array}{c} Z'_1 \ \sim\ Z_1 \ = \ \theta_1 \qquad \cdots \qquad \theta_{n-1}\cdots\theta_1 Z'_n \ \sim\ \theta_{n-1}\cdots\theta_1 Z_n \ = \ \theta_n \\ \theta' = \{\theta_n\cdots\theta_1\xi' \mapsto \theta_n\cdots\theta_1\xi\}\theta_n\cdots\theta_1 \qquad \theta' Z_0 \ \sim\ \theta' Z'_0 \ = \ \theta \end{array}}{\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \ \xrightarrow{\xi}\ Z_0 \ \sim\ \mathsf{ref}_{\rho'_1}\ Z'_1 \times \cdots \times \mathsf{ref}_{\rho'_n}\ Z'_n \ \xrightarrow{\xi'}\ Z'_0 \ = \ \theta}$$

$$\textsc{Struct-Field}\ \frac{\exists j.\ Z_j \ \sim\ Z' \ = \ \theta}{\mathtt{struct}\ t\ \mathtt{\{}\ \overline{Z_i\ x_i}\ \mathtt{\}} \ \sim\ Z' \ = \ \theta} \qquad \textsc{Field-Struct}\ \frac{\exists j.\ Z' \ \sim\ Z_j \ = \ \theta}{Z' \ \sim\ \mathtt{struct}\ t\ \mathtt{\{}\ \overline{Z_i\ x_i}\ \mathtt{\}} \ = \ \theta}$$

$$\textsc{Struct}\ \frac{\forall k \in [1,n].\ \theta_{k-1}\cdots\theta_1 Z_k \ \sim\ \theta_{k-1}\cdots\theta_1 Z'_k \ = \ \theta_k}{\mathtt{struct}\ t\ \mathtt{\{}\ \overline{Z_i\ x_i}\ \mathtt{\}} \ \sim\ \mathtt{struct}\ u\ \mathtt{\{}\ \overline{Z'_j\ y_j}\ \mathtt{\}} \ = \ \theta_k\cdots\theta_1}$$

Figure 9: Unification algorithm.

$\vdash^{\uparrow}_{L}$: $\textsc{Env} \times \mathrm{K} \times \textsc{Lval} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$

$$\textsc{Var}\ \frac{\Gamma(x) = \mathsf{ref}_\rho\ Z}{\Gamma;\kappa \vdash^{\uparrow}_{L} x : \mathrm{id}\ \&\ Z\ \&\ \emptyset\ \&\ \kappa} \qquad \textsc{Deref}\ \frac{\Gamma;\kappa \vdash^{\uparrow}_{E} E : \theta\ \&\ \mathsf{ptr}\ \mathsf{ref}_\rho\ Z\ \&\ \varphi\ \&\ \kappa'}{\Gamma;\kappa \vdash^{\uparrow}_{L} {*E} : \theta\ \&\ \mathsf{ref}_\rho\ Z\ \&\ \varphi\ \&\ \kappa'}$$

$$\textsc{Fun}\ \frac{\begin{array}{c} \Gamma(f) = \forall\ \overrightarrow{\zeta\varrho\xi}.\ \kappa_0 \Rightarrow \mathsf{ref}_{\varrho_0}\ Z_1 \times \cdots \times Z_n \ \xrightarrow{\xi_0}\ Z_0 \\ \theta = \{\overrightarrow{\zeta \mapsto \zeta'}, \overrightarrow{\varrho \mapsto \varrho'}, \overrightarrow{\xi \mapsto \xi'}\} \qquad \overrightarrow{\zeta'\varrho'\xi'}\ \text{fresh} \end{array}}{\Gamma;\kappa \vdash^{\uparrow}_{L} f : \mathrm{id}\ \&\ \mathsf{ref}_{\varrho_0}\ \theta(Z_1 \times \cdots \times Z_n \ \xrightarrow{\xi_0}\ Z_0)\ \&\ \emptyset\ \&\ \kappa \sqcup \theta\kappa_0}$$

$$\textsc{Index}\ \frac{\Gamma;\kappa \vdash^{\uparrow}_{L} L : \theta\ \&\ \mathsf{ref}_{\rho_1}\ Z_1\ \&\ \varphi_1\ \&\ \kappa' \qquad \theta\Gamma;\kappa' \vdash^{\uparrow}_{E} E : \theta'\ \&\ Z_2\ \&\ \varphi_2\ \&\ \kappa''}{\Gamma;\kappa \vdash^{\uparrow}_{L} L\mathtt{[}E\mathtt{]} : \theta'\theta\ \&\ \theta'(\mathsf{ref}_{\rho_1}\ Z_1)\ \&\ \theta'\varphi_1 \cup \varphi_2\ \&\ \kappa''}$$

$$\textsc{Field}\ \frac{\Gamma;\kappa \vdash^{\uparrow}_{L} L : \theta\ \&\ \mathsf{ref}_\rho\ \mathtt{struct}\ t\ \mathtt{\{}\ \overline{Z_i\ x_i}\ \mathtt{\}}\ \&\ \varphi\ \&\ \kappa'}{\Gamma;\kappa \vdash^{\uparrow}_{L} L.x_j : \theta\ \&\ \mathsf{ref}_{\rho.x_j}\ Z_j\ \&\ \varphi\ \&\ \kappa'}$$

Figure 10: Inference rules for lvalues.

φ that result from evaluating E, and a new constraint system κ'. We write this inference step as $\Gamma;\kappa \vdash^{\uparrow}_{E} E : \theta\ \&\ Z\ \&\ \varphi\ \&\ \kappa'$. Figure 11 shows the inference rules for non-arithmetic expressions, and Fig. 12 shows the inference rules for arithmetic expressions. The derivation of the inference algorithm from the typing rules follows the same principles as for lvalues. Remarkably, the result shape Z' of a cast will be an instance of the most general shape of the target type T (Cast). The connection between the source shape Z and Z' is established through unification.

$$\vdash^{\uparrow}_{E} \;:\; \textsc{Env} \times \textsc{K} \times \textsc{Exp} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \textsc{K}$$

$$\textsc{Con-Bot}\quad \frac{\text{typeof}(c) \in \{\texttt{\_Bool}, \texttt{char}, \texttt{float}, \texttt{double}\}}{\Gamma; \kappa \vdash^{\uparrow}_{E} c : \text{id} \;\&\; \bot \;\&\; \emptyset \;\&\; \kappa}
\qquad
\textsc{Con-Str}\quad \frac{\text{typeof}(str) = \texttt{char*} \qquad \rho \text{ fresh}}{\Gamma; \kappa \vdash^{\uparrow}_{E} str : \text{id} \;\&\; \textsf{ptr ref}_{\rho}\ \bot \;\&\; \emptyset \;\&\; \kappa}$$

$$\textsc{Con-Int}\quad \frac{\text{typeof}(i) \in \{\texttt{int}, \texttt{short}, \texttt{long}, \texttt{long long}\} \qquad \zeta \text{ fresh}}{\Gamma; \kappa \vdash^{\uparrow}_{E} i : \text{id} \;\&\; \zeta \;\&\; \emptyset \;\&\; \kappa}
\qquad
\textsc{Lval}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{L} L : \theta \;\&\; \textsf{ref}_{\rho}\ Z \;\&\; \varphi \;\&\; \kappa'}{\Gamma; \kappa \vdash^{\uparrow}_{E} L : \theta \;\&\; Z \;\&\; \varphi \cup \underline{read}_{\rho} \;\&\; \kappa'}$$

$$\textsc{Addr}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{L} L : \theta \;\&\; \textsf{ref}_{\rho}\ Z \;\&\; \varphi \;\&\; \kappa'}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{\&}L : \theta \;\&\; \textsf{ptr ref}_{\rho}\ Z \;\&\; \varphi \;\&\; \kappa'}
\qquad
\textsc{Sizeof-T}\quad \frac{T \neq T'\texttt{[}E\texttt{]}}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{sizeof}(T) : \text{id} \;\&\; \bot \;\&\; \emptyset \;\&\; \kappa}$$

$$\textsc{Sizeof-A}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta \;\&\; Z \;\&\; \varphi \;\&\; \kappa'}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{sizeof}(T\texttt{[}E\texttt{]}) : \theta \;\&\; \bot \;\&\; \varphi \;\&\; \kappa'}
\qquad
\textsc{Sizeof-E}\quad \frac{}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{sizeof}(E) : \text{id} \;\&\; \bot \;\&\; \emptyset \;\&\; \kappa}$$

$$\textsc{Alignof-T}\quad \frac{}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{alignof}(T) : \text{id} \;\&\; \bot \;\&\; \emptyset \;\&\; \kappa}
\qquad
\textsc{Alignof-E}\quad \frac{}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{alignof}(E) : \text{id} \;\&\; \bot \;\&\; \emptyset \;\&\; \kappa}$$

$$\textsc{Question}\quad \frac{\begin{array}{c}\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta_1 \;\&\; Z_1 \;\&\; \varphi_1 \;\&\; \kappa_1 \qquad \theta_1\Gamma; \kappa_1 \vdash^{\uparrow}_{E} E_2 : \theta_2 \;\&\; Z_2 \;\&\; \varphi_2 \;\&\; \kappa_2 \\ \theta_2\theta_1\Gamma; \kappa_2 \vdash^{\uparrow}_{E} E_3 : \theta_3 \;\&\; Z_3 \;\&\; \varphi_3 \;\&\; \kappa' \qquad \theta_3 Z_2 \sim Z_3 = \theta_4\end{array}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 \texttt{ ? } E_2 \texttt{ : } E_3 : \theta_4\theta_3\theta_2\theta_1 \;\&\; \theta_4 Z_3 \;\&\; \theta_4(\theta_3(\theta_2\varphi_1 \cup \varphi_2) \cup \varphi_3) \;\&\; \theta_4`\kappa'}$$

$$\textsc{Cast}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta \;\&\; Z \;\&\; \varphi \;\&\; \kappa' \qquad Z' = \text{shape-of}(T) \qquad Z \sim Z' = \theta'}{\Gamma; \kappa \vdash^{\uparrow}_{E} \texttt{(}T\texttt{)}E : \theta'\theta \;\&\; \theta' Z' \;\&\; \theta'\varphi \;\&\; \theta'\kappa'}$$

Figure 11: Inference rules for non-aritmetic expressions.

Instructions

Inference of instructions takes an environment Γ, a constraint system κ, and an instruction I; and computes a substitution θ, the effects φ that result from evaluating I, and a new constraint system κ'. We write this inference step as $\Gamma; \kappa \vdash^{\uparrow}_{I} I : \theta \;\&\; \varphi \;\&\; \kappa'$. Figure 13 shows the inference rules for instructions. The derivation of the inference algorithm from the typing rules follows the same principles as for lvalues and expressions.

Statements

Inference of statements takes an environment Γ, a constraint system κ, an expected return shape Z, and a statement S; and computes a substitution θ, the effects φ that result from evaluating S, and a new constraint system κ'. We write this inference step as $\Gamma; \kappa \vdash^{\uparrow Z}_{S} S : \theta \;\&\; \varphi \;\&\; \kappa'$. Figure 14 shows the inference rules for statements.

$$\vdash^{\uparrow}_{E} \ : \textsc{Env} \times \mathrm{K} \times \textsc{Exp} \to \textsc{Subst} \times \textsc{Shape} \times \textsc{Effect} \times \mathrm{K}$$

$$\textsc{Neg}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta \ \& \ Z \ \& \ \varphi \ \& \ \kappa' \qquad \ominus \in \{\texttt{-}, \texttt{~}, \texttt{!}\}}{\Gamma; \kappa \vdash^{\uparrow}_{E} \ominus E : \theta \ \& \ \bot \ \& \ \varphi \ \& \ \kappa'}$$

$$\textsc{Int-A}\quad \frac{\begin{array}{c}\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta \ \& \ Z_1 \ \& \ \varphi_1 \ \& \ \kappa' \\ \theta\Gamma; \kappa' \vdash^{\uparrow}_{E} E_2 : \theta' \ \& \ Z_2 \ \& \ \varphi_2 \ \& \ \kappa'' \qquad \theta' Z_1 \ \sim \ Z_2 \ = \ \theta'' \qquad \oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}\end{array}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 \oplus E_2 : \theta''\theta'\theta \ \& \ \theta'' Z_2 \ \& \ \theta''(\theta'\varphi_1 \cup \varphi_2) \ \& \ \theta''\kappa''}$$

$$\textsc{Bit-A}\quad \frac{\begin{array}{c}\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta \ \& \ Z_1 \ \& \ \varphi_1 \ \& \ \kappa' \\ \theta\Gamma; \kappa' \vdash^{\uparrow}_{E} E_2 : \theta' \ \& \ Z_2 \ \& \ \varphi_2 \ \& \ \kappa'' \qquad \theta' Z_1 \ \sim \ Z_2 \ = \ \theta'' \qquad \otimes \in \{\texttt{\&}, \texttt{^}, \texttt{|}, \texttt{«}, \texttt{»}\}\end{array}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 \otimes E_2 : \theta''\theta'\theta \ \& \ \theta'' Z_2 \ \& \ \theta''(\theta'\varphi_1 \cup \varphi_2) \ \& \ \theta''\kappa''}$$

$$\textsc{Ptr-A}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta \ \& \ \mathsf{ptr\ ref}_{\rho} \ Z_1 \ \& \ \varphi_1 \ \& \ \kappa' \qquad \theta\Gamma; \kappa' \vdash^{\uparrow}_{E} E_2 : \theta' \ \& \ Z_2 \ \& \ \varphi_2 \ \& \ \kappa'' \qquad \oplus \in \{\texttt{+}, \texttt{-}\}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 \oplus E_2 : \theta'\theta \ \& \ \mathsf{ptr\ ref}_{\theta'\rho} \ \theta' Z_1 \ \& \ \theta'\varphi_1 \cup \varphi_2 \ \& \ \kappa''}$$

$$\textsc{Minus-PP}\quad \frac{\begin{array}{c}\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta \ \& \ \mathsf{ptr\ ref}_{\rho_1} \ Z_1 \ \& \ \varphi_1 \ \& \ \kappa' \\ \theta\Gamma; \kappa' \vdash^{\uparrow}_{E} E_2 : \theta' \ \& \ \mathsf{ptr\ ref}_{\rho_2} \ Z_2 \ \& \ \varphi_2 \ \& \ \kappa'' \qquad \rho_1 \equiv \rho_2\end{array}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 - E_2 : \theta'\theta \ \& \ \bot \ \& \ \theta'\varphi_1 \cup \varphi_2 \ \& \ \kappa''}$$

$$\textsc{Cmp}\quad \frac{\begin{array}{c}\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta \ \& \ Z_1 \ \& \ \varphi_1 \ \& \ \kappa' \\ \theta\Gamma; \kappa' \vdash^{\uparrow}_{E} E_2 : \theta' \ \& \ Z_2 \ \& \ \varphi_2 \ \& \ \kappa'' \qquad \trianglelefteq \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}\end{array}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 \trianglelefteq E_2 : \theta'\theta \ \& \ \bot \ \& \ \theta'\varphi_1 \cup \varphi_2 \ \& \ \kappa''}$$

$$\textsc{Bool-A}\quad \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 : \theta \ \& \ Z_1 \ \& \ \varphi_1 \ \& \ \kappa' \qquad \theta\Gamma; \kappa' \vdash^{\uparrow}_{E} E_2 : \theta' \ \& \ Z_2 \ \& \ \varphi_2 \ \& \ \kappa'' \qquad \odot \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma; \kappa \vdash^{\uparrow}_{E} E_1 \odot E_2 : \theta'\theta \ \& \ \bot \ \& \ \theta'\varphi_1 \cup \varphi_2 \ \& \ \kappa''}$$

Figure 12: Inference rules for arithmetic expressions.

Function definitions

Inference of function definitions takes an environment Γ, a constraint system κ, and a definition of a function f; and computes a substitution θ, the shape scheme of f, and a new constraint system κ'. We write this inference step as $\Gamma; \kappa' \vdash^{\uparrow}_{D} T_0 \ f\texttt{(}T_1 \ x_1, \cdots, T_n \ x_n\texttt{)} \ \texttt{\{} \ \overrightarrow{T_i \ x_i}^{n+1:m}\texttt{;} \ S \ \texttt{\}} : \theta \ \& \ \forall \ \overrightarrow{\zeta\rho\xi}. \ \kappa_f \Rightarrow \mathsf{ref}_{\varrho_0} \ Z \ \& \ \kappa'$. Figure 15 shows the inference rules for function definitions. Remarkably, function's formal parameters and local variables are given their most general shape, which is refined by θ after inferring the effects of the function's body S. Similarly, for the case of recursive functions, every recursive call of f would see the same instance of its most general shape. At this point it makes sense to solve the constraint system κ_f of effect variables that are local to f.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. ASE 2014.

$$\vdash_I^{\uparrow} \ : \textsc{Env} \times \mathrm{K} \times \textsc{Instr} \rightarrow \textsc{Subst} \times \textsc{Effect} \times \mathrm{K}$$

$$\textsc{Set}\quad \frac{\Gamma;\kappa \vdash_L^{\uparrow} L : \theta \ \&\ \mathsf{ref}_\rho\ Z_1 \ \&\ \varphi_1 \ \&\ \kappa' \qquad \theta\Gamma;\kappa' \vdash_E^{\uparrow} E : \theta' \ \&\ Z_2 \ \&\ \varphi_2 \ \&\ \kappa'' \qquad \theta' Z_1 \ \sim\ Z_2 \ =\ \theta''}{\Gamma;\kappa \vdash_I^{\uparrow} L \ = \ E : \theta''\theta'\theta \ \&\ \theta''(\theta'\varphi_1 \cup \varphi_2 \cup \underline{write}_{\theta'\rho}) \ \&\ \theta''\kappa''}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}\Gamma;\kappa \vdash_E^{\uparrow} E_0 : \theta_0 \ \&\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n) \xrightarrow{\varphi'} Z_0 \ \&\ \varphi_0 \ \&\ \kappa_0 \\ \theta_{i-1}\cdots\theta_0\Gamma;\kappa_{i-1} \vdash_E^{\uparrow} E_i : \theta_i \ \&\ Z'_i \ \&\ \varphi_i \ \&\ \kappa_i \qquad Z_i \ \sim\ Z'_i \ =\ \theta'_i \qquad \theta' = \theta'_{n:1} \qquad i \in [1,n]\end{array}}{\Gamma;\kappa \vdash_I^{\uparrow} E_0(E_1,\cdots,E_n) : \theta'\theta_{n:0} \ \&\ \theta'(\theta_{n:1}\varphi_0 \cup (\bigcup_{i\in[1,n]} \theta_{n:i+1}\varphi_i) \cup \theta_{n:1}\varphi') \ \&\ \theta'\kappa_n}$$

$$\textsc{Call-n-Set}\quad \frac{\begin{array}{c}\Gamma;\kappa \vdash_E^{\uparrow} E_0 : \theta_0 \ \&\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n) \xrightarrow{\varphi'} Z_0 \ \&\ \varphi_0 \ \&\ \kappa_0 \\ \theta_{i-1:0}\Gamma;\kappa_{i-1} \vdash_E^{\uparrow} E_i : \theta_i \ \&\ Z'_i \ \&\ \varphi_i \ \&\ \kappa_i \ / \ i \in [1,n] \\ \theta_{n:0}\Gamma;\kappa_n \vdash_L^{\uparrow} L : \theta'' \ \&\ \mathsf{ref}_{\rho_0}\ Z'_0 \ \&\ \varphi'' \ \&\ \kappa'' \qquad Z_i \ \sim\ Z'_i \ =\ \theta'_i \ / \ i \in [0,n] \qquad \theta' = \theta'_{n:0}\end{array}}{\Gamma;\kappa \vdash_I^{\uparrow} L = E_0(E_1,\cdots,E_n) : \theta''\theta'\theta_{n:0} \ \&\ \theta'(\theta''(\theta_{n:1}\varphi_0 \cup (\bigcup_{i\in[1,n]} \theta_{n:i+1}\varphi_i) \cup \theta_{n:1}\varphi') \cup \varphi'' \cup \underline{write}_{\rho_0}) \ \&\ \theta'\kappa''}$$

Figure 13: Inferece rules for instructions.

[2] P. Jouvelot and J.-P. Talpin. The type and effect discipline, 1993.

[3] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. CC 1996.

[4] B. Steensgaard. Points-to analysis in almost linear time. POPL 1996.

[5] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2, 7 1992.

[6] M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1), 1990.

[7] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. PLDI 1999.

$$\vdash^{\uparrow}_{S}\ : \textsc{Env} \times \textsc{K} \times \textsc{Shape} \times \textsc{Stmt} \to \textsc{Subst} \times \textsc{Effect} \times \textsc{K}$$

$$\textsc{Instr}\ \frac{\Gamma; \kappa \vdash^{\uparrow}_{I} I : \theta \ \&\ \varphi \ \&\ \kappa'}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} I\texttt{;} : \theta \ \&\ \varphi \ \&\ \kappa'} \qquad \textsc{Return}\ \frac{}{\Gamma; \kappa \vdash^{\uparrow\perp}_{S} \texttt{return;} : \mathrm{id} \ \&\ \emptyset \ \&\ \kappa}$$

$$\textsc{Return-E}\ \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta \ \&\ Z' \ \&\ \varphi \ \&\ \kappa' \qquad Z \ \sim\ Z' \ =\ \theta'}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{return}\ E\texttt{;} : \theta'\theta \ \&\ \theta'\varphi \ \&\ \theta'\kappa'}$$

$$\textsc{Label}\ \frac{\Gamma; \kappa \vdash^{\uparrow Z}_{S} S : \theta \ \&\ \varphi \ \&\ \kappa'}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} L\texttt{:}\quad S\texttt{;} : \theta \ \&\ \varphi \ \&\ \kappa'} \qquad \textsc{Goto}\ \frac{}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{goto}\ L\texttt{;} : \mathrm{id} \ \&\ \emptyset \ \&\ \kappa}$$

$$\textsc{Goto-E}\ \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta \ \&\ Z' \ \&\ \varphi \ \&\ \kappa' \qquad Z' \ \sim\ \mathsf{ptr}\ \mathsf{ref}_{\rho}\ \zeta \ =\ \theta' \qquad \rho, \zeta \text{ fresh}}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{goto}\ E\texttt{;} : \theta \ \&\ \varphi \ \&\ \kappa'}$$

$$\textsc{Break}\ \frac{}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{break;} : \mathrm{id} \ \&\ \emptyset \ \&\ \kappa} \qquad \textsc{Continue}\ \frac{}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{continue;} : \mathrm{id} \ \&\ \emptyset \ \&\ \kappa}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta_0 \ \&\ Z_0 \ \&\ \varphi_0 \ \&\ \kappa_0 \\ \theta_0\Gamma; \kappa_0 \vdash^{\uparrow Z}_{S} S_1 : \theta_1 \ \&\ \varphi_1 \ \&\ \kappa_1 \qquad \theta_1\theta_0\Gamma; \kappa_1 \vdash^{\uparrow Z}_{S} S_2 : \theta_2 \ \&\ \varphi_2 \ \&\ \kappa_2\end{array}}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{if}\ E\ S_1\ \texttt{else}\ S_2 : \theta_2\theta_1\theta_0 \ \&\ \theta_2\theta_1\varphi_0 \cup \theta_2\varphi_1 \cup \varphi_2 \ \&\ \kappa_2}$$

$$\textsc{Switch}\ \frac{\Gamma; \kappa \vdash^{\uparrow}_{E} E : \theta_0 \ \&\ Z_0 \ \&\ \varphi_0 \ \&\ \kappa_0 \qquad \theta_{i-1:0}\Gamma; \kappa_{i-1} \vdash^{\uparrow Z}_{S} S_i : \theta_i \ \&\ \varphi_i \ \&\ \kappa_i \ /\ i \in [1, n]}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{switch (}E\texttt{) \{}\ S_1 \cdots\ S_n\ \texttt{\}} : \theta_{n:0} \ \&\ \theta_{n:1}\varphi_0 \cup \Big(\bigcup_{i \in [1,n]} \theta_{n:i+1}\varphi_i \Big) \ \&\ \kappa_n}$$

$$\textsc{Loop}\ \frac{\Gamma; \kappa \vdash^{\uparrow Z}_{S} S : \theta \ \&\ \varphi \ \&\ \kappa'}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} \texttt{while (1)}\ S : \theta \ \&\ \varphi \ \&\ \kappa'}$$

$$\textsc{Seq}\ \frac{\Gamma; \kappa \vdash^{\uparrow Z}_{S} S_1 : \theta_1 \ \&\ \varphi_1 \ \&\ \kappa' \qquad \Gamma; \kappa' \vdash^{\uparrow Z}_{S} S_2 : \theta_2 \ \&\ \varphi_2 \ \&\ \kappa''}{\Gamma; \kappa \vdash^{\uparrow Z}_{S} S_1 S_2 : \theta_2\theta_1 \ \&\ \theta_2\varphi_1 \cup \varphi_2 \ \&\ \kappa''}$$

Figure 14: Inference rules for statements.

$$\vdash^{\uparrow}_{D}\ : \textsc{Env} \times \textsc{K} \times \textsc{Def} \to \textsc{Subst} \times \textsc{Shape-Scheme} \times \textsc{K}$$

$$\mathrm{Observe}_{\Gamma, Z}(\varphi) = \{\varepsilon(\overrightarrow{\rho}) \in \varphi \mid \rho_i \in \mathrm{FV}(\Gamma) \cup \mathrm{FV}(Z)\} \cup \{\xi \in \varphi \mid \xi \in \mathrm{FV}(\Gamma) \cup \mathrm{FV}(Z)\}$$

$$\textsc{Def}\ \frac{\begin{array}{c}\Gamma' = \Gamma; \overrightarrow{x_i : \mathsf{ref}_{\rho_i}\ Z_i}^{1:n} \\ \Gamma', \overrightarrow{x_i : \mathsf{ref}_{\rho_i}\ Z_i}^{n+1:m}; \kappa \vdash^{\uparrow Z_0}_{S} S : \theta \ \&\ \varphi \ \&\ \kappa' \qquad Z_i = \text{shape-of}(T_i) \ /\ i \in [0, m] \\ \varphi' = \mathrm{Observe}_{\overline{\kappa'}\theta\Gamma', \overline{\kappa'}\theta Z_0}(\overline{\kappa'}\varphi) \qquad \kappa_f = \kappa'_{\overrightarrow{\zeta\rho\xi}} \sqcup \{\xi_0 \sqsupseteq \varphi'\} \qquad \kappa'' = \kappa' \setminus \kappa_f \\ Z_f = \overrightarrow{\mathsf{ref}_{\theta\rho_i}\ \theta Z_i}^{1:n} \xrightarrow{\theta\xi_0} \theta Z_0 \qquad \overrightarrow{\zeta\rho\xi} = \mathrm{FV}(\overline{\kappa'} Z_f) \setminus (\mathrm{FV}(\overline{\kappa'}\Gamma) \cup \{\varrho_0\}) \qquad \overrightarrow{\rho_i}^{1:m} \varrho_0 \xi_0 \text{ fresh}\end{array}}{\Gamma \vdash^{\uparrow}_{D} T_0\ f\texttt{(}T_1\ x_1, \cdots, T_n\ x_n\texttt{) \{}\ \overrightarrow{T_i\ x_i}^{n+1:m}\texttt{;}\ S\ \texttt{\}} : \theta \ \&\ \forall\, \overrightarrow{\zeta\rho\xi}.\ \kappa_f \Rightarrow \mathsf{ref}_{\varrho_0}\ Z_f \ \&\ \kappa''}$$

Figure 15: Inference of function definitions.