Plugins for Structural Weakening and Strong Updates

Tobias Gedell   Daniel Hedin

Abstract

We present a general way of making use of may- and must-alias information to achieve flow-sensitive type systems that allow for flow-sensitivity on the heap. In particular, we show how may-alias information can be used for a limited form of flow-sensitivity — *structural weakening* — that allows type changes on the heap that are compatible with the subtype hierarchy. Further, we show how the combination of may- and must-alias information can be used to achieve *strong updates*, i.e., type changes on the heap that are not compatible with the subtype hierarchy, resembling the typical type rule for updates of variables in flow sensitive type systems. This work has been enabled by the use of our plugin framework — a framework for parameterizing type systems over the results of abstract interpretations — that allows us to abstract away from the computation of the alias information. In addition, our successful use of the plugin mechanism to extract both may- and must-alias information shows its strength.

1 Introduction

One dimension of program analyses is flow-sensitivity; the result of a flow-sensitive analysis depends on the order of the instructions of a program, whereas the result of a flow-insensitive analysis does not. Frequently, flow-sensitive analyses achieve higher precision than flow-insensitive.

Even though type systems are predominantly flow-insensitive, flow-sensitive type systems arise naturally, as shown by, for example, linear and affine type systems [Pie05]. A more well-known example of a flow-sensitive type system is the type system of Java bytecode where the limited number or registers forces (from a practical standpoint) the type system to be able to change the types of registers; each instruction is typed in a pre- and a post-type, and the assignment instruction changes the type of the target register in the post type to the type of the source value.

The static flow-sensitivity of Java bytecode is limited to registers for a reason; aliasing prohibits the flow-sensitivity to easily extend to the heap. To be able to change types on the heap, e.g., the type of a field of a certain object, it must be made sure that the types of all aliased locations are changed, otherwise a way of freely casting between the types by writing into one location and reading from another is introduced. Thus, for instance, it is not safe to extend the subtyping relation to arrays, i.e., to say that `String[]` is a subtype of `Object[]` because `String` is a subtype of `Object`, since that would allow casting from the type `Object` to the type `String`.[1]

The standard solution for this is to enforce *invariant typing* for all heap locations, involving both prohibiting type changes based on updates, and restraining the subtype relation to *invariant subtyping* for arrays, and *width subtyping* for objects [Pie02].

From a practicality standpoint, width subtyping is good enough for "standard" types. For information flow security it is not necessarily the case. Whereas the need to freely change the types of parts of the heap may seem far fetched, i.e., from a boolean to an integer, the need to change a location from holding public integers to holding secret integers is not as unreasonable, especially considering that, unlike standard types, information flow types are intrinsically flow sensitive. In addition to this, there are important extensions to basic information-flow security that rely on flow-sensitivity [HS08].

Contribution We present a general method for making use of may- and must-alias information to allow for flow-sensitive types on the heap. Whereas previous work has tried to combine the computation of the alias information and its usage, we use our recently proposed abstract-interpretation plugin framework [GH08] to decouple the computation and the usage of the alias information. This allows us to use the alias information in a general and clear way, while at the same time allowing us to instantiate the resulting type system with different alias analyses — from the most basic ones to, e.g., the most elaborate shape analyses.

The main contributions of this paper are that we 1) show how the plugin framework can be used to carry over structural information about the heap, 2) show how may-alias information can be used to formulate a structural subtyping rule, and 3) show how may- and must-alias information can be used in combination to allow for strong updates on the heap, i.e., updates that do not follow the subtype hierarchy.

Outline The paper is laid out as follows. Section 2 introduces the language, Section 3 recapitulates the needed parts of the method of parameterization, and Section 4 introduces the basic type system and the correctness statements — in particular *preservation of types*. Section 5 discusses the basic idea of flow-sensitive heap types, defines the used representation of may- and must-alias information — *structural environments* — and their semantics, and shows how structural heaps can be extracted from the parameterized alias information. Section 6 contains the first use of may-alias information to achieve a limited form of flow-sensitive heap types. The basic idea is to use the may-aliases to form a structural subtyping rule, where all symbolic locations that may be aliased are guaranteed to have the same type view, which guarantees that all

[1] Java *does* allow this, which forces a runtime check when storing objects into arrays.

concrete environments will be width well-formed. Section 7 shows how may- and must-alias information can be combined to support strong updates if the updated location is in an isolated cluster of must-pointers. Finally, Section 8 discusses related work, and Section 9 concludes.

2 Language

The language used to illustrate our method is a small imperative language with records.

Syntax Let f range over field names, b range over booleans, i range over integers, and x range over variable names. The syntax of the language is defined as follows, where A ranges over record type names.

$$\begin{array}{lcl} \text{Expressions} & e & ::= & \mathit{nil} \mid b \mid i \mid x \mid e_1 \star e_2 \mid x.f \\ \text{Commands} & c & ::= & x := e \mid x_1.f := x_2 \mid \mathit{if}\ e\ c_1\ c_2 \mid c_1; c_2 \mid \\ & & & \mathit{while}\ e\ c \mid x := \mathit{new}(A) \mid \mathit{skip} \end{array}$$

Values The environments are pairs of stores s and heaps h. The stores are maps from variables x to values v, and the heaps are maps from pointers p to records r. The records are maps from field names to values. Finally, the values are made up by booleans, integers and pointers; the *error lifted values*, ranged over by $v_\perp$, are either values or $\perp$ indicating an error. We impose the restriction that heaps may not associate the null-pointer to anything, i.e., the null-pointer must not be in the domain of any heap.

$$\begin{array}{lcl} v & ::= & b \mid i \mid p \\ E & ::= & (s, h) \end{array} \qquad \begin{array}{lcl} r & ::= & \{f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\} \\ s & ::= & \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\ h & ::= & \{p_1 \mapsto r_1, \ldots, p_n \mapsto r_n\} \end{array}$$

Let $r.f$ denote $r(f)$, and for $E = (s, h)$, let $E(x)$ denote $s(x)$, $E[x \mapsto v]$ denote $(s[x \mapsto v], h)$, $E(p)$ denote $h(p)$, and similarly for other operations on environments including variables or pointers.

Semantics We assume a simple reduction semantics for expressions of the form $\langle E, e \rangle \Downarrow v_\perp$.

The semantics of commands is given in terms of a small step semantics between configurations with transitions of the form $\langle E, c \rangle \to C$, where C is either one of the terminal configurations $\perp_E$ and E indicating abnormal and normal termination in the environment E, respectively, or a non-terminal configuration $\langle E, c \rangle$. Figure 1 contains the semantic rules for commands, where $\mathit{rec}(A)$ creates a fresh record of type A with all fields set to 0 or nil, depending on their type — we assume the existence of a map Δ from record type names to structural record types (defined in Section 4 below). The origin of this map is not significant for this work, and, thus, left unspecified, but is typically created from the program

$$\frac{}{\langle E, x := v\rangle \to \langle E[x \mapsto v], \mathit{skip}\rangle} \qquad \frac{s(x_1) = \mathit{nil}}{\langle (s,h), x_1.f := x_2\rangle \to \bot_{(s,h)}}$$

$$\frac{s(x_1) = p \quad h(p) = r \quad s(x_2) = v}{\langle (s,h), x_1.f := x_2\rangle \to \langle (s, h[p \mapsto r[f \mapsto v]]), \mathit{skip}\rangle}$$

$$\frac{}{\langle E, \mathit{if}\ \mathit{true}\ c_1\ c_2\rangle \to \langle E, c_1\rangle} \qquad \frac{}{\langle E, \mathit{if}\ \mathit{false}\ c_1\ c_2\rangle \to \langle E, c_2\rangle}$$

$$\frac{}{\langle E, \mathit{while}\ e\ c\rangle \to \langle E, \mathit{if}\ e\ (c; \mathit{while}\ e\ c)\ \mathit{skip}\rangle}$$

$$\frac{r = \mathit{rec}(A) \quad p \notin \mathit{dom}(h)}{\langle (s,h), x := \mathit{new}(A)\rangle \to \langle (s[x \mapsto p], h[p \mapsto r]), \mathit{skip}\rangle}$$

$$\frac{}{\langle E, \mathit{skip}; c\rangle \to \langle E, c\rangle}$$

Figure 1: Semantic rules for commands

source, possibly in combination with a system specific map. The record name map is invariant under the execution of the program, and is left implicit in the rest of this paper.

Following [GH08] we extend the command language with label annotations to track how environments flow in and out of commands during execution and add the following rules for reduction of labeled commands. Let l range over labels drawn from the set of labels $\mathcal{L}$. A command c can be annotated with an entry label $(c)^l$, an exit label $(c)_l$, or both $(c)^{l_1}_{l_2}$. In the following, c ranges over possibly annotated commands, i.e., $(c)^l$ denotes a command with at least an entry label l, and similarly for exit labels. For while-loops decorated with an entry label we add the following reduction rule.

$$\frac{}{\langle E, (\mathit{while}\ e\ c)^l\rangle \to \langle E, (\mathit{if}\ e\ (c; (\mathit{while}\ e\ c)^l)\ \mathit{skip})^l\rangle}$$

For commands decorated with entry labels that are not while-loops, and for *skip* decorated with an exit label we add the following reduction rules.

$$\frac{\langle E_1, c_1\rangle \to \langle E_2, c_2\rangle}{\langle E_1, (c_1)^l\rangle \to \langle E_2, c_2\rangle} \qquad \frac{}{\langle E, (\mathit{skip})_l\rangle \to \langle E, \mathit{skip}\rangle}$$

As is common for small step semantics we use evaluation contexts R to determine the position of the next computation step.

$$R \quad ::= \quad \cdot \mid x := R \mid \mathit{if}\ R\ c\ c \mid R; c \mid (R)_l$$

The accompanying standard reduction rules, found in Figure 2, allow for left-most reduction of sequences, error propagation, reduction of expressions inside commands and reduction of commands under exit labels.

$$\frac{\langle E, e\rangle \Downarrow v}{\langle E, R[e]\rangle \to \langle E, R[v]\rangle} \qquad \frac{\langle E, e\rangle \Downarrow \bot}{\langle E, R[e]\rangle \to \bot_E}$$

$$\frac{\langle E_1, c_1\rangle \to \langle E_2, c_2\rangle}{\langle E_1, R[c_1]\rangle \to \langle E_2, R[c_2]\rangle} \qquad \frac{\langle E, c\rangle \to \bot_E}{\langle E, R[c]\rangle \to \bot_E}$$

Figure 2: Semantic Rules for Contexts

3 Parameterization

Before presenting the type system we recapture the fundamental parts of the method of parameterization. For a more thorough exposition see [GH08].

Abstract Environment Maps Let $\mathbb{E}$ range over some form of abstract environments with an associated concretization function γ, mapping abstract environments to sets of concrete environments. An abstract environment map is a map from labels to abstract environments. We say that an abstract environment map $\mathbb{M}$ is an entry/exit solution with respect to a command c_1 and a concrete environment E_1 if it represents all environments flowing into/out of each command as follows where *is* is the predicate for entry solutions and *os* the predicate for exit solutions.

$$\begin{aligned} is_{c_1}^{E_1}(\mathbb{M}) \equiv \forall E_2, c_2.\ \langle E_1, c_1\rangle \to^* \langle E_2, R[(c_2)^l]\rangle \Longrightarrow \\ E_2 \in \gamma(\mathbb{M}(l)) \end{aligned}$$

$$\begin{aligned} os_{c_1}^{E_1}(\mathbb{M}) \equiv \forall E_2.\ \langle E_1, c_1\rangle \to^* \langle E_2, R[(skip)_l]\rangle \Longrightarrow \\ E_2 \in \gamma(\mathbb{M}(l)) \end{aligned}$$

The definition is lifted to sets of initial environments $\mathcal{C}$ in the obvious way.

Plugins Properties and Plugins A *plugin property* $R^\diamond$ is a family of expression liftings of a relation R on values, defined in the following way.

$$(e_1, \dots, e_n) \in R_E^\diamond \equiv \langle E, e_1\rangle \Downarrow v_1 \wedge \cdots \wedge \langle E, e_n\rangle \Downarrow v_n \Longrightarrow (v_1, \dots, v_n) \in R$$

Plugin properties define the meaning of the *plugins*, in the sense that the plugins are approximations of the plugin properties.

A plugin is a family of relations on expressions indexed by abstract environments. $R^\sharp$ is said to be a plugin for $R^\diamond$ given that the following property holds.

$$(e_1, \dots, e_n) \in R_{\mathbb{E}}^\sharp \Longrightarrow \forall E \in \gamma(\mathbb{E}).\ (e_1, \dots, e_n) \in R_E^\diamond$$

The plugin framework cannot be used to directly transfer alias information. In Section 5 we show algorithmically how equality and inequality information about pointers can be used to build may- and must-alias views of the heap, respectively. Thus, in this paper we will be using two plugins: one for may-alias extraction corresponding to lifted pointer inequality, and one for must-alias extraction corresponding to lifted pointer equality (excluding the null-pointer). Let $\mathcal{R}^{\neq}$ denote plugins for pointer inequality, and let $\mathcal{R}^{-}$ denote plugins for pointer equality. For convenience, let $\mathcal{R}^{+}$ denote the negation of the pointer inequality plugin — two pointers that cannot be shown to be unequal must be assumed to be aliased.

4 Type System

The type system introduced in this section is used as the base for two different extensions: structural weakening and strong updates which we investigate separately in Section 6 and Section 7 respectively.

Type Language The primitive types, ranged over by τ, are the type of booleans *bool*, the type of natural numbers *nat*, the type of integers *int*, and the pointer types, represented by record type names A. The record types ω are maps from fields to primitive types. As mentioned above, Δ is a map from record type names to record types. The store types, ranged over by Σ, are maps from variables to primitive types. The exception types, ranged over by ξ, are $\perp_\Sigma$, indicating the possibility that an exception is thrown in the environment type Σ, and $\top$ indicating that no exception is thrown. This is a simplification from typical models of exceptions, where multiple types are used to indicate the reason for the exception. However, for our purposes this model suffices — the results are easily extended to a richer model.

$$\begin{array}{lcl} \tau & ::= & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid A \\ \xi & ::= & \perp_\Sigma \mid \top \end{array} \qquad \begin{array}{lcl} \omega & ::= & \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\} \\ \Sigma & ::= & \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \\ \Delta & ::= & \{A_1 \mapsto \omega_1, \ldots, A_n \mapsto \omega_n\} \end{array}$$

Subtyping We define two standard subtype relations: *width subtyping* $<:_w$ and *width-depth subtyping* $<:_d$. For brevity, we will use the term *depth subtyping* to refer to width-depth subtyping in the rest of this paper. Most of this paper is only concerned with width subtyping; thus, when not explicitly marked otherwise, $<:$ refers to width subtyping.

Width subtyping provides a uniform type view of the heap, see, for instance, invariant subtyping for `ML` references [Pie02], which is needed to support updates in the presence of aliases[2]; see below for a more thorough explanation.

Common to both width subtyping and depth subtyping are the rules for primitive types, exception types, and store types.

$$\tau <:_{w/d} \tau \quad \mathit{nat} <:_{w/d} \mathit{int}$$

$$\xi <:_{w/d} \xi \quad \top <:_{w/d} \perp_\Sigma \quad \frac{\Sigma_1 <:_{w/d} \Sigma_2}{\perp_{\Sigma_1} <:_{w/d} \perp_{\Sigma_2}}$$

$$\frac{\forall x \in \mathit{dom}(\Sigma_2).\ \Sigma_1(x) <:_{w/d} \Sigma_2(x)}{\Sigma_1 <:_{w/d} \Sigma_2}$$

[2]In the absence of additional analyses.

The difference between the relations is captured by the rules for subtyping of record types. Width subtyping allows records to be seen as smaller records while retaining the types of the remaining fields, i.e., bigger record types are subtypes of smaller given that all fields in the domain of the smaller record type are mapped to the same types in both the smaller and the bigger, while depth subtyping demands that all fields in the domain of the smaller are mapped to types that are depth subtypes of the corresponding types in the smaller record type.

$$\frac{\forall f \in \mathit{dom}(\omega_2).\ \omega_1.f = \omega_2.f}{\omega_1 <:_w \omega_2} \qquad \frac{\forall f \in \mathit{dom}(\omega_2).\ \omega_1.f <:_d \omega_2.f}{\omega_1 <:_d \omega_2}$$

The subtyping relations naturally induce a subtype relation on record type names. Thus, even though the system introduced so far is nominal, we use structural subtyping, defined on record identifiers as the smallest relation on record names $<:_{w/d}$ satisfying:

$$\frac{\Delta(A_1) <:_{w/d} \Delta(A_2)}{A_1 <:_{w/d} A_2}$$

This is in contrast to the more frequent use of pure nominal subtyping, where the programs explicitly declare what record names are subtypes of each other.

Width versus Depth Subtyping The subtyping relation defines when objects of one type can be safely seen as having another type. For instance, it is perfectly safe to view a natural number as an integer, since the set of integers include all natural numbers. It may seem natural to extend the subtyping relation to records based on the same subset argument; after all, the set of records with a field f holding a natural numbers is included in the set of records with the same field f holding an integer.

Such an extension of the subtyping relation to records is provided by depth-subtyping and is, in fact, perfectly sound in the presence of aliases as long as we only *read* from the records. However, in the presence of aliases and updates, depth subtyping is not sound, as illustrated by the following program where $\Delta(A) = \{f : \mathit{nat}\}$ and $\Delta(B) = \{f : \mathit{int}\}$.

```
A x := new A; B y := (B) x; y.f := -1; nat z := x.f;
```

In this example and the following examples we will use A, B, C, ... as record type names, and x, y, z, ... as variable names. In addition, for clarity, we will allow type annotations, type casts, and field assignment of constants — neither is necessary, but allows the examples to be more concise. For example, in the above program the cast in the assignment `y := (B) x` is needed to change the type of x to B before the assignment. Otherwise, the type of y would simply be overwritten by the type of x, i.e., A, since variable updates are flow-sensitive.

The example first creates a record with a field f of type natural numbers. Using depth subtyping we create an alias to the record with an integer field type. As noted above, reading the field via x and y is still sound - x has a more

$$\frac{\Sigma \vdash e : \tau, \xi}{\Sigma \vdash^{\dagger} x := e \Rightarrow \Sigma[x \mapsto \tau], \xi} \qquad \frac{\Sigma(x_1) = A \quad \Sigma(x_2) = \tau \quad \tau <: \Delta(A).f}{\Sigma \vdash^{\dagger} x_1.f := x_2 \Rightarrow \Sigma, \bot_{\Sigma}}$$

$$\frac{\Sigma_1 \vdash e : \mathit{bool}, \xi \quad \Sigma_1 \vdash^{\dagger} c_1 \Rightarrow \Sigma_2, \xi \quad \Sigma_1 \vdash^{\dagger} c_2 \Rightarrow \Sigma_2, \xi}{\Sigma_1 \vdash^{\dagger} \mathit{if}\ e\ c_1\ c_2 \Rightarrow \Sigma_2, \xi}$$

$$\frac{\Sigma \vdash e : \mathit{bool}, \xi \quad \Sigma \vdash^{\dagger} c \Rightarrow \Sigma, \xi}{\Sigma \vdash^{\dagger} \mathit{while}\ e\ c \Rightarrow \Sigma, \xi}$$

$$\frac{\Sigma_1 \vdash^{\dagger} c_1 \Rightarrow \Sigma_2, \xi \quad \Sigma_2 \vdash^{\dagger} c_2 \Rightarrow \Sigma_3, \xi}{\Sigma_1 \vdash^{\dagger} c_1; c_2 \Rightarrow \Sigma_3, \xi}$$

$$\frac{}{\Sigma \vdash^{\dagger} x := \mathit{new}(A) \Rightarrow \Sigma[x \mapsto A], \top}$$

$$\frac{}{\Sigma \vdash^{\dagger} \mathit{skip} \Rightarrow \Sigma, \top} \qquad \frac{\Sigma_2 \vdash^{\dagger} c \Rightarrow \Sigma_3, \xi_1 \quad \Sigma_1 <: \Sigma_2 \quad \Sigma_3 <: \Sigma_4 \quad \xi_1 <: \xi_2}{\Sigma_1 \vdash^{\dagger} c \Rightarrow \Sigma_4, \xi_2}$$

Figure 3: Type Rules for Commands

precise type, viewing the field as a natural number while y views it as an integer. However, the types permit us to update the field with an integer via y, and to read the written integer as a natural number via x, effectively introducing a cast going the opposite direction of the subtype hierarchy. Thus, a weakening rule as the one found in Figure 3 but based on depth subtyping rather than width subtyping is unsound.

It should be pointed out that writing is sound using a depth subtyping rule with the subtype of the fields inverted, i.e., we can view a record with an integer field as a record with a natural number field — all natural numbers are also integers and limiting the values that can be written into the field is unproblematic. For this reason reading is known as being *co-variant*, i.e., that sound subtyping with respect to reading extends structurally in the same way, and writing is known as being *contra-variant*, i.e., that sound subtyping with respect to writing extends structurally in the opposite way [Pie02]. In a system where the same type governs both reading and writing, the types of the fields must be both co-variant and contra-variant, i.e., they must be *invariant*. This is represented by the width subtyping that demands *equality* on the types of the fields, which implies invariance, since the subtyping relation is reflexive.

Expression Type Rules The typing judgment for expressions, $\Sigma \vdash e : \tau, \xi$, is read as the expression e is well-typed in the environment type Σ, with return type τ possibly resulting in exceptions as indicated by ξ. The type rules for the expressions are entirely standard, and omitted for brevity.

$$\frac{}{\delta \vdash b : \mathit{bool}} \qquad \frac{}{\delta \vdash i : \mathit{int}} \qquad \frac{i \geq 0}{\delta \vdash i : \mathit{nat}}$$

$$\frac{}{\delta \vdash \mathit{nil} : A} \qquad \frac{\delta(p) <: A}{\delta \vdash p : A}$$

$$\frac{\forall (f, \tau) \in \omega.\ \delta \vdash r.f : \tau}{\delta \vdash r : \omega} \qquad \frac{\forall (x, \tau) \in \Sigma.\ \delta \vdash s(x) : \tau}{\delta \vdash s : \Sigma}$$

$$\frac{\forall (p, A) \in \delta.\ \delta \vdash h(p) : \Delta(A)}{\delta \vdash h} \qquad \frac{\delta \vdash s : \Sigma \quad \delta \vdash h}{\delta \vdash (s, h) : \Sigma}$$

$$\frac{\delta \vdash v : \tau}{\delta \vdash v : \tau, \xi} \qquad \frac{}{\delta \vdash \bot : \tau, \bot_\Sigma}$$

$$\frac{\delta \vdash E : \Sigma_2}{\delta \vdash^\dagger \bot_E : \Sigma_1, \bot_{\Sigma_2}} \qquad \frac{\Sigma_1 \vdash^\dagger c \Rightarrow \Sigma_2, \xi \quad \delta \vdash E : \Sigma_1}{\delta \vdash^\dagger \langle E, c \rangle : \Sigma_2, \xi}$$

Figure 4: Well-formedness

Command Type Rules The type system for commands is flow sensitive; each command is typed with respect to a pre and a post environment type. The typing judgment for commands, $\Sigma_1 \vdash^{\mathbb{M}_I, \mathbb{M}_O, \mathcal{R}^+, \mathcal{R}^-} c \Rightarrow \Sigma_2, \xi$ is read as the command c is well-typed with respect to the abstract environment maps $\mathbb{M}_I$, and $\mathbb{M}_O$, the plugin $\mathcal{R}^+$, and the plugin $\mathcal{R}^-$ in the environment type Σ_1 resulting in the environment type Σ_2, possibly resulting in an exception as indicated by ξ. The standard type rules for commands are shown in Figure 3, where $\vdash^\dagger$ is used as short form for $\vdash^{\mathbb{M}_I, \mathbb{M}_O, \mathcal{R}^+, \mathcal{R}^-}$. The standard type system serves as the foundation for the extension with rules for structural weakening and strong updates.

4.1 Correctness

As is done by Pierce [Pie02] we split the correctness argument into two theorems, *progress* — intuitively, that well-typed commands and expressions are able to execute in all environments that conform to, i.e., are *well-formed* with respect to, the entry environment type of the command or expression — and *preservation* — intuitively, that the result of running the command or expression conforms to the exit environment type of the same. Together, progress and preservation guarantee proper execution of well-typed programs in all well-formed environments; progress and preservation repeatedly guarantee one step of execution, and that the result is well-formed.

Well-formedness We define two well-formedness relations, one corresponding to width subtypes, and one corresponding to depth subtypes. More precisely, well-formedness is formulated as a family of relations between values and types, indexed over pointer typings. The pointer typings, ranged over by δ, are maps from pointers to record type names and make the well-formedness relation inductively definable also for cyclic heaps. The rules for well-formedness are found in Figure 4, where $\vdash^{\dagger}$ is used as short form for $\vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-}$. The well-formedness relations are entirely standard; the interaction between the well-formedness relation for stores and heaps, together with the well-formedness relation for records guarantees that if an environment is well-formed with respect to a store type and a pointer typing, then the pointer typing types at least all live (reachable) pointers.

Progress and Preservation of Expressions Preservation of expressions expresses that well-typed expressions preserve well-formedness under execution, i.e., for an expression e such that $\Sigma \vdash e : \tau, \xi$, running e in Σ-well-formed environments will result in τ, ξ-well-formed values.

Theorem 4.1 *Preservation of Types of Expressions*

$$\Sigma \vdash e : \tau, \xi \Longrightarrow \delta \vdash E : \Sigma \wedge \langle E, e \rangle \Downarrow v_\perp \Longrightarrow \delta \vdash v_\perp : \tau, \xi$$

Proof 4.1 *By induction over the derivation of* $\Sigma \vdash e : \tau, \xi$. *The proof is entirely standard and is omitted for brevity.*

Progress of expressions expresses that well-typed expressions are able to execute in correspondingly well-formed environments, i.e., for an expression e such that $\Sigma \vdash e : \tau, \xi$ it is possible to run e in any Σ-well-formed environment.

Theorem 4.2 *Progress of Expressions*

$$\Sigma \vdash e : \tau, \xi \Longrightarrow \delta \vdash E : \Sigma \Longrightarrow \exists v_\perp . \langle E, e \rangle \Downarrow v_\perp$$

Proof 4.2 *By induction over the derivation of* $\Sigma \vdash e : \tau, \xi$. *The proof is entirely standard and is omitted for brevity.*

Together progress and preservation for expressions guarantee that well-typed expressions are able to run in correspondingly well-formed environments, and that the results are well-formed.

Progress and Preservation of Commands Preservation of types of commands is formulated in essentially the same way as for expressions, i.e., for a command c such that $\Sigma_1 \vdash^{\dagger} c \Rightarrow \Sigma_2, \xi$, running c in any Σ_1-well-formed environment that is in any set of initial environments $\mathcal{C}$ which makes $\mathbb{M}_I$ an entry solution and $\mathbb{M}_O$ an exit solution for c will result in a Σ_2, ξ-well-formed configuration.

Theorem 4.3 *Preservation of Types of Commands*

$$\begin{aligned} &\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi \wedge is^{\mathcal{C}}_c(\mathbb{M}_I) \wedge os^{\mathcal{C}}_c(\mathbb{M}_O) \Longrightarrow \\ &\qquad E \in \mathcal{C} \wedge \delta_1 \vdash E : \Sigma_1 \wedge \langle E, c \rangle \to C \Longrightarrow \exists \delta_2 .\ \delta_2 \vdash^{\dagger} C : \Sigma_2, \xi \end{aligned}$$

Proof 4.3 *By induction on the derivation of* $\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2,\xi$. *The proof is entirely standard since the standard type rules do not make use of the parameterized information and is omitted for brevity.*

Progress of commands expresses that well-typed commands are able to run in correspondingly well-formed environments, i.e., for a command c such that $\Sigma_1 \vdash^{\dagger} c \Rightarrow \Sigma_2, \xi$, it is possible to run c in any Σ_1-well-formed environment.

Theorem 4.4 *Progress of Commands*

$$\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2,\xi \wedge is^{\mathcal{C}}_{c}(\mathbb{M}_I) \wedge os^{\mathcal{C}}_{c}(\mathbb{M}_O) \Longrightarrow$$
$$E \in \mathcal{C} \wedge \delta \vdash E : \Sigma_1 \Longrightarrow \exists C.\ \langle E, c\rangle \rightarrow C$$

Proof 4.4 *By induction on the derivation of* $\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2,\xi$. *The proof is entirely standard since the standard type rules do not make use of the parameterized information and is omitted for brevity.*

As above progress and preservation interact to guarantee successful execution of well-typed commands in well-formed environments. Let $\langle E, c\rangle \rightarrow^n C$ be the obvious lifting of the small step evaluation to evaluation of n consecutive steps. We formulate the top-level correctness of commands, that well-typed commands terminate in well-formed environments or result in well-formed configurations regardless of the number of execution steps, in the following way, where T ranges over terminal configurations.

Theorem 4.5 *Top-level Correctness of Commands*

$$\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c_1 \Rightarrow \Sigma_2,\xi \wedge is^{\mathcal{C}}_{c_1}(\mathbb{M}_I) \wedge os^{\mathcal{C}}_{c_1}(\mathbb{M}_O) \Longrightarrow$$
$$E_1 \in \mathcal{C} \wedge \delta_1 \vdash E_1 : \Sigma_1 \wedge \langle E_1, c_1\rangle \rightarrow^n C \Longrightarrow$$
$$\exists \delta_2\ .\ \delta_2 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} C : \Sigma_2,\xi$$

Proof 4.5 *The proof of top-level correctness for commands proceeds by induction over the number of execution steps. The proof is completely independent on the parameterized information, and thus valid for all possible parameterizations and proceed by induction over the number of execution steps. For brevity we refer to [GH08] for the details of the proof.*

5 Heap Types and Aliases

The type system presented in Section 4 allows the types of the variables to change; after an assignment to a variable, the type of the variable becomes the type of the expression that was assigned to the variable. This is an example of a *flow-sensitive* type system. As discussed, this type scheme does not immediately extend to records. Instead, it is common to demand type invariance for heap locations to guarantee a uniform type view of the heap.

Let *sl* range over symbolic locations, defined as follows.

$$sl ::= x \mid sl.f$$

The meaning of the symbolic locations with respect to a concrete environment is defined by recursive dereference. In a given environment, each symbolic location refers at most one concrete location, but different symbolic locations may refer to the same concrete location. When this occurs, we say that the symbolic locations are aliased.

This section explores the details of the interaction between aliases and subtyping in the presence of reading and writing, and how alias information can be obtained using the plugin framework. The following two sections show how this information can be used to achieve a certain degree of the freedom enjoyed by variable types — simplified, the alias information is used to make sure that all aliased locations agree on the type.

5.1 Flow-sensitive Heap Types

Consider the flow-sensitive type rule for variable assignment from Section 4.

$$\frac{\Sigma \vdash e : \tau, \xi}{\Sigma \vdash^{\dagger} x := e \Rightarrow \Sigma[x \mapsto \tau], \xi}$$

The soundness of this rule relies on the fact that each variable is stored at a unique place in the store, which is not shared by any other variables. Thus, writing to a variable does not modify the value, and hence neither the type, of any of the other variables.

A similar rule for records on the heap is not immediately possible in the presence of aliasing. Similar to above, if such a rule is provided we have a direct way of making a single concrete location seen as having two different types. As shown above, such a situation is equivalent to having a way of freely casting between the two types by writing into the concrete location via one symbolic location, and reading from the other symbolic location, as illustrated by the following program assuming that $\Delta(A) = \{f : \mathit{int}\}$.

```
A x := new A;  A y := x; y.f := true; int z := x.f;
```

First, x and y are initialized to point to the same record. Thereafter, the field of that record is updated with a boolean via y, causing the type of y to become $\{f : \mathit{bool}\}$. Integers written via x can now be read as booleans via y, and vice versa.

5.2 Alias Information

Alias information is information about which symbolic locations may be or are guaranteed to be aliased with each other. There are two forms of aliases, *may*- and *must*-aliases, corresponding to whether two symbolic locations may be aliased, i.e., it cannot be ruled out that they are aliased, or must be aliased, i.e., they are always aliased. Intuitively, if x and y are may-aliased then it may be the case that in one of the program runs x and y contain the same pointer. On the other hand if x and y are must-aliased then it must be the case that they contain the same pointer in every program run.

There are two common forms of alias information: as a map from program points to 1) relations on symbolic locations, or 2) structural environments — see [Deu94] for references of both methods.

In the former, if a pair of symbolic locations (sl_1, sl_2) are related for some program point labeled with l then, in the may-alias case, it cannot be excluded that sl_1 and sl_2 contain the same concrete pointer at l in some program run, and, in the must-alias case, it must be the case that sl_1 and sl_2 contain the same pointer at l in all program runs. To keep the alias information finite in the presence of cycles, two important properties of (both may- and must-) alias information are used. The first property is a form of substitutivity property

$$(sl_1, sl_2) \wedge (sl_3, sl_4) \Longrightarrow (sl_3[sl_1/sl_2], sl_4[sl_1/sl_2]) \tag{1}$$

which expresses that if sl_1 and sl_2 are aliased then one can form new aliases by replacing sl_1 with sl_2 in other aliases. For example, if (x, y) and $(x, x.f)$, then we know from this rule that $(y, y.f)$. This property subsumes transitivity, since if (sl_1, sl_2) and (sl_2, sl_3) then (sl_1, sl_3). The second property

$$(sl_1, sl_2) \Longrightarrow (sl_1.f, sl_2.f) \tag{2}$$

expresses that anything reachable from an alias is also an alias.

For our purposes the second form of alias information — the structural environments — is more convenient to work with. The idea behind structural environments is to have an abstract representation that captures information about the common structure of a set of concrete environments. This is achieved by using abstract structural pointers with the property that (may- and must-) aliased symbolic locations contain the same structural pointer.

Syntax of Structural Environments The syntax of the structural environments follows the syntax of the values, with pointers represented by abstract pointers, and all other values represented by an abstract dummy.

$$\begin{array}{lcl} \widehat{v} & ::= & \widehat{p} \mid \bullet \\ \widehat{E} & ::= & (\widehat{s}, \widehat{h}) \end{array} \qquad \begin{array}{lcl} \widehat{r} & ::= & \{f_1 \mapsto \widehat{v}_1, \ldots, f_n \mapsto \widehat{v}_n\} \\ \widehat{s} & ::= & \{x_1 \mapsto \widehat{v}_1, \ldots, x_n \mapsto \widehat{v}_n\} \\ \widehat{h} & ::= & \{\widehat{p}_1 \mapsto \widehat{r}_1, \ldots \widehat{p}_n \mapsto \widehat{r}_n\} \end{array}$$

That is, the structural values $\widehat{v}$ are either structural pointers, ranged over by $\widehat{p}$, or any other value represented by $\bullet$. A structural record $\widehat{r}$ is a map from field names to structural values, a structural store $\widehat{s}$ is a map from variable names to structural values, and a structural heap $\widehat{h}$ is a map from structural pointers to structural records. Finally, the structural environments $\widehat{E}$ are pairs of structural stores and heaps.

May-alias interpretation We define the may-alias meaning of the structural heaps in terms of a may-alias concretization function γ^+. In the following we will drop the superscript when possible without risk of confusion. Let ξ range over maps from structural pointers to non-empty sets of concrete pointers. To make sure that different structural pointers get mapped to different concrete pointers we parameterize the concretization function over a structural pointer valuation with pairwise disjoint codomain, excluding the null-pointer. We say that such pointer valuations are *may-alias sound*.

Definition 5.1 (May-alias sound pointer valuation) *A pointer valuation ξ is may-alias sound if it does not map anything to the null-pointer and has a pairwise disjoint codomain, i.e.*

$$\forall \widehat{p}_1, \widehat{p}_2 \in dom(\xi) \;.\; \widehat{p}_1 \neq \widehat{p}_2 \Longrightarrow \xi(\widehat{p}_1) \cap \xi(\widehat{p}_2) = \emptyset$$

Let ξ^+ range over the set of may-alias sound pointer valuations.

Let $\mathcal{V}_{\backslash p}$ be the set of concrete values excluding the pointers. The concretization for primitive values is defined as follows.

$$\gamma^+_{\xi^+}(\bullet) = \mathcal{V}_{\backslash p} \qquad \gamma^+_{\xi^+}(\widehat{p}) = \xi^+(\widehat{p}) \cup \{nil\}$$

The concretization function is extended structurally while allowing all possible combinations of concrete pointers as defined by the pointer valuation. For records we let $\gamma^+_{\xi^+}(\widehat{r}) = \{r \mid f \in dom(\widehat{r}), r.f \in \gamma^+_{\xi^+}(\widehat{r}.f)\}$, and similarly for stores $\gamma^+_{\xi^+}(\widehat{s}) = \{s \mid x \in dom(\widehat{s}), s(x) \in \gamma^+_{\xi^+}(\widehat{s}(x))\}$. For heaps we let $\gamma^+_{\xi^+}(\widehat{h}) = \{h \mid \widehat{p} \in dom(\widehat{h}), p \in \xi^+(\widehat{p}), h(p) \in \gamma^+_{\xi^+}(\widehat{h}(\widehat{p}))\}$. Finally, we define $\gamma^+_{\xi^+}(\widehat{E})$ for environments by combining the results from the concretization functions for heaps and stores using the same pointer valuation.

The set of concrete environments associated with one particular structural environment is the union over all may-alias sound pointer valuations.

$$\gamma^+(\widehat{E}) = \bigcup_{\xi^+} \{\gamma^+_{\xi^+}(\widehat{E})\}$$

Must-alias interpretation The concretization function above defines the meaning of may-aliases; with a small change in the interpretation of the structural values we can define the meaning of must-aliases. Let γ^- denote the concretization function for must-aliases. As above when there is no risk of confusion the superscript is dropped. A pointer valuation is *must-alias sound* if all sets in its codomain are singleton.

Definition 5.2 (Must-alias sound pointer valuation) *A pointer valuation is must-alias sound if all sets in its codomain are singleton and do not contain the null-pointer, i.e.*

$$\forall \widehat{p} \in dom(\xi) \;.\; |\xi(\widehat{p})| = 1$$

Let ξ^- range over must-alias sound pointer valuations.

For structural values we define the must-alias concretization function as

$$\gamma^-_{\xi^-}(\bullet) = \mathcal{V} \qquad \gamma^-_{\xi^-}(\widehat{p}) = \xi^-(\widehat{p})$$

with the difference that, unlike $\mathcal{V}_{\backslash p}$, $\mathcal{V}$ ranges over the set of all concrete values including the pointers. The structural extension of the concretization function to structural environments is done in exactly the same way as above.

As before, the set of concrete environments associated with one particular structural environment is the union over all valid pointer valuations.

$$\gamma^-(\widehat{E}) = \bigcup_{\xi^-}\{\gamma^-_{\xi^-}(\widehat{E})\}$$

This means that the must-alias information only conveys information about which pointer locations must be equal — it does not rule out any other aliases, and considers all locations to be possibly aliased with any other location.

5.3 Plugins are Under Approximations

Sound may-alias information can be seen as an over approximation of the possibly *aliased* locations, i.e., it is safe to consider more locations to be aliased than actually are.

When designing plugin properties for probing may-alias information we must take into consideration that plugins are by definition *under approximations*, and as such not suitable for probing over approximations — if two locations are not marked as being aliases the conclusion is that they are unaliased, however a plugin may by definition freely exclude locations from the relation, making this conclusion invalid.

The solution to this is to use the dual interpretation of may-aliases — must-not aliases — i.e., the under approximation of guaranteed *unaliased* locations. Thus, instead of using equality on pointers as the relation for our plugin property we use inequality.

5.4 Extracting May-Aliases

Using the may-alias plugin $\mathcal{R}^+$ defined above we can extract may-alias information by a traversal rooted in the variables of pointer type.

The algorithm takes a may-alias plugin `R`, an abstract environment `aenv`, a list of previously visited symbolic locations `vs`, a structural environment which is modified during the traversal `env`, and a work list of symbolic locations yet to be visited `ls`. Each iteration removes the topmost symbolic location of the work list, and checks it against all previously visited symbolic locations. If it is guaranteed to be unaliased with all previous locations a fresh structural pointer is introduced and the symbolic location associated with it; otherwise, the symbolic location is associated with the structural pointer of the found previous location. Whenever a new symbolic location is introduced all relevant succeeding symbolic locations (the fields of pointer type) are added at the end of the work list.

In the pseudo code below, `lookup env v` gets the structural pointer associated with `v` in the structural environment `env`, `env[p -> *]` returns an updated version of `env` where `p` is associated with an initialization record that is type compatible with the location associated with `p`, i.e., the fields of pointer type are empty (they will be updated by subsequent iterations) and all other fields contain •, `env[l -> p]` returns an updated version of `env` where the location `l` is associated with the structural pointer `p`, and **fields of** `l` returns the list of symbolic locations of the fields of `l` of pointer type.[3]

```
extract _ _ _  env [] = env
extract aenv R vs env (l:ls) =
 case find (not . R aenv l) vs of
  None -> p    = fresh pointer
          env' = env[p -> *][l -> p]
          lfs  = fields of l
          extract aenv R (l:vs) env' (ls++lfs)

  Some v -> p = lookup env v
            env' = env[l -> p]
            extract aenv R vs env' ls
```

We define the extraction function η^+ for may-aliases in terms of the above *extract* function as follows, where $init_\Gamma$ is the list of initial symbolic locations — the variables of pointer type (as given by the store type).

$$\eta^+(\mathbb{E}, \mathcal{R}^+, \Gamma) = extract\ \mathbb{E}\ \mathcal{R}^+\ [\,]\ ([\,],[\,])\ init_\Gamma$$

Thus, $\eta^+(\mathbb{E}, \mathcal{R}^+, \Gamma)$ extracts a may-alias view of $\mathbb{E}$, using $\mathcal{R}^+$ starting in the variables given pointer types by Γ.

The correctness of the algorithm relies on the substitutivity property of the may-alias information. Termination of the algorithm relies on the abstract environment and the number of fields being finite and every non-terminated path of field references in the may-alias information containing a cycle, i.e., there must not be an infinite number of unaliased locations; a formal definition of this cyclic path property is found in Appendix A. For each underlying may-alias information there exist plugins for which these properties hold.

5.5 Extracting Must-Aliases

Similar to above, we can extract must-alias information using the must-alias plugin $\mathcal{R}^-$.

The algorithm takes a must-alias plugin `R`, an abstract environment `aenv`, a list of previously visited symbolic locations `vs`, a a structural environment which is modified during the traversal `env`, and a work list of symbolic locations yet to be visited `ls`. Each iteration removes the topmost symbolic location of

[3] All aliased locations are required to have the same type.

the work list, and checks it against all previously visited symbolic locations. If it is must-aliased with any of the previously inspected symbolic locations it is associated with the structural pointer of the found previous location; otherwise, it is checked whether it is must-aliased with itself, which would imply that it is guaranteed not to be a null-pointer.[4] If it is must-aliased with itself a fresh structural pointer is introduced and the symbolic location associated with it. If it is not then the symbolic location is associated with •.

```
extract _ _ _  env [] = env
extract aenv R vs env (l:ls) =
 case find (not . R aenv l) vs of
  None -> if R aenv l l then
           p    = fresh pointer
           env' = env[p -> *][l -> p]
           lfs  = fields of l
           extract aenv R (l:vs) env' (ls++lfs)
          else
           env' = env[l -> •]
           extract aenv R vs env' ls

  Some v -> p    = lookup env v
            env' = env[l -> p]
            extract aenv R vs env' ls
```

The extraction function η^- for must-aliases is defined in the same way as the extraction function for may-aliases and correctness and termination of the algorithm rely on the same properties.

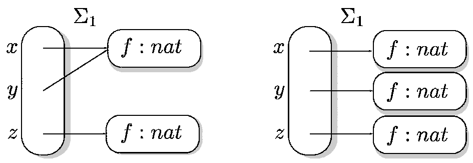
6 Structural Weakening

This section details how may-alias information can be used to safely weaken the types of heap locations. The section begins with two examples that show why a naive extension of the standard weakening rule is unsound, and how structural alias information can be used to provide a sound weakening rule by demanding that all aliased locations are subject to the same type changes. Thereafter, we introduce the basis for the weakening, the decorating structural well-formedness — essentially a well-formedness relation for structural values — and show how it can be used to create a sound weakening rule. The section ends with a small example illustrating the use of the weakening rule.

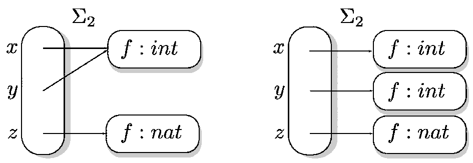
As we saw in Section 4, a weakening rule based on depth-subtyping is not sound in the presence of aliases and updates. The problem is that depth-subtyping makes it possible to create different type views of aliased symbolic locations. Without alias information we are forced to impose an invariant type view of all locations that may be aliased using width-subtyping; with may-alias information it is possible to relax this demand and demand an invariant type view only on the locations that are may-aliased.

[4]This is the case since the plugin for must-alias excludes the null-pointer.

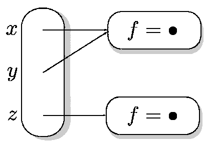
To illustrate this, assume a set of concrete environments S, e.g., the set of environments reaching a certain program point in a particular program. Assume that x and y are may-aliased and that x and y are unaliased with z, i.e., there exist at least one environment in S, where x and y point to the same record, and in no environments in S does z point to the same record as x or y. Assume further that all of x, y and z point to records of one field f holding a natural number, i.e., that the environments in S are well-formed in $\Sigma_1 = \{x : A, y : A, z : A\}$, where $\Delta(A) = \{f : nat\}$. The possible heaps in S (up-to type constrained isomorphism and omitting null-pointers) can be illustrated as follows.



It is clear that we can safely change the types of x and y to hold an integer as long as we change both, i.e., all environments in S are well-formed in $\Sigma_2 = \{x : B, y : B, z : A\}$, where $\Delta(B) = \{f : int\}$.



To see how structural may-alias information can be used to achieve this consider the structural representation of the situation above: $\widehat{s} = \{x \mapsto \widehat{p}_1, y \mapsto \widehat{p}_1, z \mapsto \widehat{p}_2\}$, with $\widehat{h} = \{\widehat{p}_1 \mapsto \{f \mapsto \bullet\}, \widehat{p}_2 \mapsto \{f \mapsto \bullet\}\}$.



As can be seen in the picture and as was described in Section 5.2 all locations that may be aliased contain the same structural pointer. Thus, the same ideas underlying width well-formedness for concrete environments can be used for structural may-alias information to ensure a uniform type-view for all may-aliased locations.

Decorated Structural May-Aliases We define the well-formedness relation for structural may-aliases following the standard well-formedness; in addition we let the structural well-formedness produce a type decorated version of the structural environment — the use of the decoration will become apparent below.

$$\Omega \vdash \bullet : \tau \curvearrowright \bullet_\tau,\ \tau \in \{nat, int, bool\} \qquad \frac{\Omega(\widehat{p}) <: A \qquad A <: \Omega(\widehat{p})}{\Omega \vdash \widehat{p} : A \curvearrowright \widehat{p}}$$

$$\frac{\begin{array}{c}(f,\tau) \in \omega.\ \Omega \vdash \widehat{r}.f : \tau \curvearrowright \widehat{r}_d.f \\ dom(\widehat{r}) = dom(\widehat{r}_d) = dom(\omega)\end{array}}{\Omega \vdash \widehat{r} : \omega \curvearrowright \widehat{r}_d}$$

$$\frac{\begin{array}{c}\forall (x,\tau) \in \Sigma.\ \Omega \vdash \widehat{s}(x) : \tau \curvearrowright \widehat{s}_d(x) \\ dom(\widehat{s}) = dom(\widehat{s}_d) = dom(\Sigma)\end{array}}{\Omega \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d}$$

$$\frac{\begin{array}{c}\forall (\widehat{p},A) \in \Omega.\ \Omega \vdash \widehat{h}(\widehat{p}) : \Delta(A) \curvearrowright \widehat{h}_d(\widehat{p}) \\ dom(\widehat{h}) = dom(\widehat{h}_d) = dom(\Omega)\end{array}}{\Omega \vdash \widehat{h} \curvearrowright \widehat{h}_d}$$

$$\frac{\Omega \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d \qquad \Omega \vdash \widehat{h} \curvearrowright \widehat{h}_d}{\Omega \vdash (\widehat{s},\widehat{h}) : \Sigma \curvearrowright (\widehat{s}_d,\widehat{h}_d)}$$

Figure 5: Decorating Structural Well-formedness

The language for the decorated structural may-aliases is identical to the language for the structural may-aliases with the addition of a type decoration on all occurrences of $\bullet$:

$$\widehat{v} \ ::= \ \widehat{p} \mid \bullet_\tau$$

When needed we use $\widehat{v}_d$, $\widehat{r}_d$, $\widehat{s}_d$, $\widehat{h}_d$, and $\widehat{E}_d$ to distinguish decorated values, records, stores, heaps and environments from the undecorated structural may-alias counterparts.

Concretization of Decorated Structural May-Aliases The concretization is constrained to the meaning of the type annotation, instead of all values apart from the pointers

$$\gamma^+_{\xi^+}(\bullet_\tau) = [\![\tau]\!]$$

where $[\![nat]\!]$ is the set of natural numbers, $[\![int]\!]$ the set of integers, and $[\![bool]\!]$ the set of booleans.

Decorating Structural Well-formedness Let Ω range over structural pointer typings, i.e., maps from structural pointers to record identifiers. The rules for the *decorating structural well-formedness* are found in Figure 5, and are easily extended to decorated structural values by replacing the rule for $\bullet$ with:

$$\frac{\tau_1 <: \tau_2}{\Omega \vdash \bullet_{\tau_1} : \tau_2 \curvearrowright \bullet_{\tau_1}} \quad \tau_1, \tau_2 \in \{nat, int, bool\}$$

The decorating structural well-formedness has two important properties. First, the type decoration does not exclude any well-formed environments.

Lemma 6.1 (Stability of Type Decoration)

$$\Omega_{max} \vdash \widehat{E} : \Sigma \curvearrowright \widehat{E}_d \wedge \delta \vdash E : \Sigma \wedge E \in \gamma^+(\widehat{E}) \Longrightarrow E \in \gamma^+(\widehat{E}_d)$$

where Ω_{max} *denotes the* maximal Ω *with respect to which* $\widehat{E}$ *is well-formed in* Σ*. See Appendix B for details.*

Proof 6.1 *First,* $E \in \gamma^+(\widehat{E})$ *implies the existence of* ξ^+ *such that* $E \in \gamma^+_{\xi^+}(\widehat{E})$*. Now, Lemma B.2 gives us that* $E \in \gamma^+_{\xi^+_{E,\Sigma}}(\widehat{E})$*. The result is immediate from Lemma B.7, and Lemma B.9.*

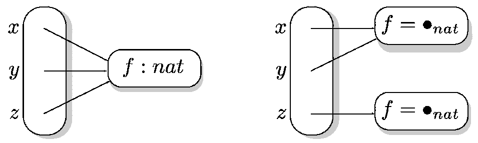
Second, well-formedness is preserved by concretization of the *decorated* structural environment. Let δ_{Ω,ξ^+} be the pointer typing induced by Ω and ξ^+, i.e., $\delta_{\Omega,\xi^+}(p) = \Omega({\xi^+}^{-1}(p))$. We know that ${\xi^+}^{-1}$ exists, since all may-alias sound pointer valuations are injective.

Lemma 6.2 (Preservation of Well-formedness under Concretization)

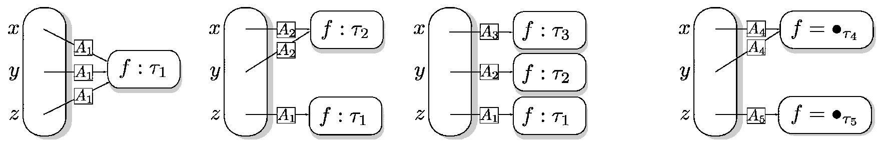
$$\Omega \vdash \widehat{E}_d : \Sigma \wedge E \in \gamma^+_{\xi^+}(\widehat{E}_d) \Longrightarrow \delta_{\Omega,\xi^+} \vdash E : \Sigma$$

Proof 6.2 *Given that* $E = (s, h)$ *we must show that* $\delta_{\Omega,\xi^+} \vdash s : \Sigma$*, and that* $\delta_{\Omega,\xi^+} \vdash h$*. The result is immediate from Lemma C.3, and Lemma C.5.*

One way of viewing the decorated structural may-alias representation is as a may-alias aware environment type, i.e., an environment type where types have been specialized to the may-alias structure. The following picture of the standard type view to the left and the decorated structured may-alias environment from the example above to the right illustrates this idea.



It is easy to see how the structural may-alias information limits the freedom of the types as illustrated by the following picture showing three different environment types and a structural environment (rightmost). The structural environment gives a limit to the maximal possible type structure in the sense that it defines which symbolic locations must have the same types. Thus, intuitively, the two leftmost environment types are compatible, but not the third, since the third tries to give x and y different types — A_3 and A_2 respectively.



For a more detailed explanation, assume that A_1, A_2, and A_3 are different. The first example can be accommodated by choosing $A_4 = A_5 = A_1$ and thus $\tau_4 = \tau_5 = \tau_1$, the second example by choosing $A_4 = A_2$ and $A_5 = A_1$, and thus $\tau_4 = \tau_2$ and $\tau_5 = \tau_1$, whereas the third example cannot be accommodated, since that would imply choosing A_4 and τ_4 to two different types corresponding to different type views for aliased locations.

With this view the well-formedness for decorated structural values can be used to formulate something that can be seen as a constrained depth-subtype relation. To see this let $\Omega_1 \vdash \widehat{E} : \Sigma_1 \curvearrowright \widehat{E}_d \wedge \Omega_2 \vdash \widehat{E}_d : \Sigma_2$ be written $\Sigma_1 <:_{\widehat{E}} \Sigma_2$. We have the following semantic property.

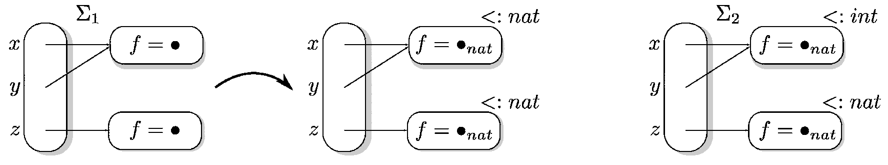
Lemma 6.3 (Structural Subtype)

$$\Sigma_1 <:_{\widehat{E}} \Sigma_2 \wedge E \in \gamma^{+}_{\xi^+}(\widehat{E}) \wedge \delta \vdash E : \Sigma_1 \Longrightarrow \exists \delta .\ \delta \vdash E : \Sigma_2$$

Proof 6.3 *We have that* (1) $\Omega_1 \vdash \widehat{E} : \Sigma_1 \curvearrowright \widehat{E}_d$*,* (2) $\Omega_2 \vdash \widehat{E}_d : \Sigma_2$*,* (3) $E \in \gamma^{+}_{\xi^+}(\widehat{E})$*, and* (4) $\delta \vdash E : \Sigma_1$

First, Lemma 6.1 together with $(1,4,3)$ *gives us that* (5) $E \in \gamma^{+}_{\xi^+}(\widehat{E}_d)$*. Now, Lemma 6.2 together with* $(2,5)$ *gives us that* $\delta_{\Omega_2,\xi^+} \vdash E : \Sigma$ *and we are done.*

To illustrate the use consider the example from above where $\widehat{s} = \{x \mapsto \widehat{p}_1, y \mapsto \widehat{p}_1, z \mapsto \widehat{p}_2\}$, with $\widehat{h} = \{\widehat{p}_1 \mapsto \{f \mapsto \bullet\}, \widehat{p}_2 \mapsto \{f \mapsto \bullet\}\}$, $\Sigma_1 = \{x : A, y : A, z : A\}$ and $\Sigma_2 = \{x : B, y : B, z : A\}$, where $\Delta(A) = \{f : nat\}$, and $\Delta(B) = \{f : int\}$. In this example the two steps of $\Sigma_1 <:_{(\widehat{s},\widehat{h})} \Sigma_2$ are $\{\widehat{p}_1 \mapsto A, \widehat{p}_2 \mapsto A\} \vdash (\widehat{s}, \widehat{h}) : \Sigma_1 \curvearrowright (\widehat{s}_d, \widehat{h}_d)$, and $\{\widehat{p}_1 \mapsto B, \widehat{p}_2 \mapsto A\} \vdash (\widehat{s}_d, \widehat{h}_d) : \Sigma_2$, for $\widehat{s}_d = \widehat{s}$ and $\widehat{h}_d = \{\widehat{p}_1 \mapsto \{f \mapsto \bullet_{nat}\}, \widehat{p}_2 \mapsto \{f \mapsto \bullet_{nat}\}\}$, since $nat <: int$, as illustrated by the following picture, where the subtyping annotations express the demands put on the structural environments by Σ_1 in the middle structural environment and Σ_2 in the rightmost structural environment.



The picture illustrates how Σ_1 has decorated the the original structural environment $(\widehat{s}, \widehat{h})$, and how Σ_2 is able to change the type view of x and y to a super type of the previous type recorded by the decorated structural environment.

Structural Weakening Based on this we can create a new weakening rule based on well-formedness where the structural representations of the entry and exit environments of a command c are used to ensure that all aliased pointers have compatible type views. Let $\mathbb{M}_I$ and $\mathbb{M}_O$ range over entry and exit solutions, respectively, and let η^+ be the may-alias extraction function.

$$\frac{\begin{array}{c}\Sigma_2 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_3, \xi_1 \quad \xi_1 <: \xi_2 \\ \Sigma_1 <:_{\eta^+(\mathbb{M}_I(l_1),\mathcal{R}^+,\Sigma_1)} \Sigma_2 \qquad \Sigma_3 <:_{\eta^+(\mathbb{M}_O(l_2),\mathcal{R}^+,\Sigma_3)} \Sigma_4\end{array}}{\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} (c)^{l_1}_{l_2} \Rightarrow \Sigma_4, \xi_2}$$

We prove soundness of the structural weakening rule by proving preservation of types for it.

Lemma 6.4 (Preservation of Types of Structural Weakening)

$$\begin{array}{l}\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi \wedge is^{\mathcal{C}}_c(\mathbb{M}_I) \wedge os^{\mathcal{C}}_c(\mathbb{M}_O) \Longrightarrow \\ \qquad\qquad E \in \mathcal{C} \wedge \delta_1 \vdash E : \Sigma_1 \wedge \langle E, c\rangle \rightarrow C \Longrightarrow \\ \qquad\qquad\qquad\qquad \exists \delta_2.\ \delta_2 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} C : \Sigma_2, \xi\end{array}$$

Proof 6.4 *Assume* (1) $\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi$, (2) $is^{\mathcal{C}}_c(\mathbb{M}_I)$, (3) $os^{\mathcal{C}}_c(\mathbb{M}_O)$, (4) $E \in \mathcal{C}$, (5) $\delta_1 \vdash E : \Sigma_1$ *and* (6) $\langle E, c\rangle \rightarrow C$.

(1) *gives* (7) $\Sigma_1' \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2', \xi'$, (8) $\xi' <: \xi$, (9) $\Sigma_1 <:_{\eta^+(\mathbb{M}_I(l_1),\mathcal{R}^+,\Sigma_1)} \Sigma_1'$, *and* (10) $\Sigma_2' <:_{\eta^+(\mathbb{M}_O(l_2),\mathcal{R}^+,\Sigma_2')} \Sigma_2$.

(2) *and* (4) *gives* (13)$E \in \gamma^+(\mathbb{M}_I(l_1))$ *which together with soundness of the extraction function for may-aliases gives* (14)$E \in \gamma^+_{\xi_1^+}(\eta^+(\mathbb{M}_I(l_1), \mathcal{R}^+, \Sigma_1))$ *for some* ξ_1^+. *From this Lemma 6.3* (11)$\delta_1' \vdash E : \Sigma_1'$ *for some* δ_1'. *Now the induction hypothesis is applicable, which gives* (12) $\delta_2 \vdash C : \Sigma_2', \xi$ *for some* δ_2. *We proceed with an analysis of* (12).

abnormal termination *This case gives* $\delta_2 \vdash E_2 : \Sigma_e$ *for some* Σ_3, *where* $C = \bot_{E_3}$, *which immediately gives* $\delta_2 \vdash \bot_{E_2} : \Sigma_2', \bot_{\Sigma_3}$.

termination *This case gives* $\delta_2 \vdash E_2 : \Sigma_2'$, *where* $C = E_2$. *From* $os^{\mathcal{C}}_c(\mathbb{M}_O)$, *we get that* $E_2 \in \gamma^+(\mathbb{M}_O(l_2))$, *and, thus, from the soundness of the extraction function that* $E_2 \in \gamma^+_{\xi_2^+}(\eta^+(\mathbb{M}_O(l_2), \mathcal{R}^+, \Sigma_2'))$ *for some* ξ_2^+. *Similar to above Lemma 6.3 gives* $\delta_2' \vdash E_2 : \Sigma_2$, *for some* δ_2' *which gives us that* $\delta_2' \vdash E_2 : \Sigma_2, \xi$ *from which the result is immediate.*

non-termination *This case gives* $\Sigma_3 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c' \Rightarrow \Sigma_2'$ *for some* Σ_3, *and* $\delta_2 \vdash E : \Sigma_3$. *In the same way as above we establish that* $\delta_2' \vdash E_2 : \Sigma_2$ *for some* δ_2'. *It now remains to show that* $\Sigma' \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c' \Rightarrow \Sigma_2$, *which is immediate from the structural weakening rule.*

Example Use We end this section with a small example of the use of structural weakening for achieving a limited form of flow-sensitive types on the heap. Assuming $\Delta(A) = \{f : nat\}$ consider the following program, which is not typable in the standard type system since *int* is not a subtype of *nat*.

```
A x = new A; A y := x; x.f := 0; x.f := -1;
```

However, even a simplistic alias analysis is able to determine that x and y are may-aliased, and unaliased with all other locations. This means that before $x.f := -1$ the type of x and y can be weakened to B, given that $\Delta(B) = \{f : \mathit{int}\}$, which allows us to perform the update. Since all aliases have their types changed uniformly, the pitfall of introducing the possibility of casting values is avoided.

7 Strong Updates

This section shows how may-alias information and must-alias information can be combined to allow for heap updates that do not follow the subtype hierarchy — similar to the updates of variables where the variable type environment is updated with the type of the value written into the variable. This differs from the structural weakening of the previous section, where the structure of the environments was used to express which types soundly described the environments, and the update was supported by finding a type in which the update was supported. For strong updates the actual update is more central; the old environment is typically not well-formed in the new environment type, and vice versa. Consider the following tiny program, which is typable in pre-type $\Sigma_1 = \{x : \tau\}$ for any type τ, and post-type $\Sigma_2 = \{x : \mathit{bool}\}$ using the flow sensitive type rule for variables.

```
x := 0; x := true;
```

Clearly, after assigning a boolean to x the environment is not typable in the post-type of $x := 0$, in which the type of x is nat.

The soundness of the flow-sensitive type rule for variables comes from the fact that no variables are aliased. In the same way, if a structural (may-) pointer is uniquely associated with a symbolic location we know that that symbolic location is alias free and can safely be strongly updated. However, demanding that a location is completely unaliased to support strong updates is unnecessarily restrictive. For instance, if we know that all aliases to the symbolic location are must-aliases, we know that a strong update is safe, given that we change the type of all must-aliases accordingly. Thus, it would be natural to expect the following program to be typable in the empty pre-type $\{\ \}$ and post-type $\{x : B, y : B\}$, where $\Delta(A) = \{f : \tau\}$ for some τ, and $\Delta(B) = \{f : \mathit{bool}\}$.

```
A x := new A; A y := x; x.f := true;
```

The previous section showed how may-alias information can be used to support weakening, and how weakening can be used to support a limited form of flow sensitive types. This was achieved by the use of a structural width well-formedness relation that guaranteed concrete width well-formedness. To fit with the results of the previous section, and with the correctness proof of the standard type system we will use preservation of width well-formedness as the base for the correctness argument of this section. This restricts the result to updates of must-aliased location *that are not reachable via may-aliases.* This restriction is justified by the fact that the presence of may-aliases would constrain the update to follow the sub-type hierarchy, which together with the demand of concrete width well-formedness would result in the same expressive power as structural weakening.

Combined May- and Must-alias Information The approach we consider is based on a sound merge of may- and must-alias information that guarantees that the must-aliased heap locations are not reachable via may-aliases. In short, this is achieved by annotating the pointers in the structural environment as either may-alias pointers or must-alias pointers and making sure that must and may aliases never concretize to the same concrete pointer.

As before we introduce the syntax, the semantics in form of a concretization function and decorating structural well-formedness; for brevity we only present the changes to what has previously been presented. First, the syntax for structural values is extended to contain both structural may-alias pointers $\widehat{p}^{+}$ and structural must-alias pointers $\widehat{p}^{-}$.

$$\widehat{v} ::= \widehat{p}^{-} \mid \widehat{p}^{+} \mid \bullet$$

The separation between must and may aliases is achieved by demanding shared pointer valuations with pairwise disjoint codomains with the additional demand that the pointer valuations map all must-alias pointers to singleton sets, and to limit the concretization of $\bullet$ to non-pointer values. The demand that the pointer valuations are pairwise disjoint also for must-aliased pointers is not a restriction since two must-aliased pointers that concretize to the same concrete pointer is by definition may-aliased and may-aliases have priority over must-aliases in the merged alias information.

Definition 7.1 (May- and Must-alias sound pointer valuations) *Let $\widehat{p}$ range over structural may-alias pointers and structural must-alias pointers.*

$$\widehat{p} ::= \widehat{p}^{+} \mid \widehat{p}^{-}$$

A pointer valuation ξ is may- and must-alias sound if it has pairwise disjoint codomain that does not contain the null-pointer and maps all structural must-alias pointers to singleton sets.

$$\forall \widehat{p}_1, \widehat{p}_2 \in dom(\xi) \ . \ \widehat{p}_1 \neq \widehat{p}_2 \Longrightarrow \xi(\widehat{p}_1) \cap \xi(\widehat{p}_2) = \emptyset \wedge \forall \widehat{p}^{-} \in dom(\xi) \ . \ |\xi(\widehat{p}^{-})| = 1$$

Let ξ^* range over may- and must-alias sound pointer valuations.

The meaning of the combined may- and must-alias information is formulated in terms of a concretization function γ^*.

$$\gamma^*_{\xi^*}(\bullet) = \mathcal{V}_{\setminus p} \qquad \gamma^*_{\xi^*}(\widehat{p}^{+}) = \xi^*(\widehat{p}^{+}) \cup \{nil\} \qquad \gamma^*_{\xi^*}(\widehat{p}^{-}) = \xi^*(\widehat{p}^{-})$$

Similarly to before we form a decorated version of the structural values, stores and heaps by annotating $\bullet$ with a type. The concretization is changed accordingly to $\gamma^*_{\xi^*}(\bullet_\tau) = [\![\tau]\!]$, where $[\![int]\!]$ is the set of integers, $[\![nat]\!]$ the set of natural numbers, and $[\![bool]\!]$ the set of booleans. The set of concrete environments associated with one particular combined structural environment is the union over all may- and must-alias sound pointer valuations.

$$\gamma^*(\widehat{E}) = \bigcup_{\xi^*}\{\gamma^*_{\xi^*}(\widehat{E})\}$$

The decorating structural well-formedness for the combined may- and must-alias information is immediate both for the undecorated syntax and the decorated syntax, using the rule for structural may pointers of Figure 5 for structural may pointers, and the rules for the undecorated $\bullet$, and the decorated $\bullet_\tau$ from Section 6 above.

$$\Omega \vdash \bullet : \tau \curvearrowright \bullet_\tau,\ \ \tau \in \{nat, int, bool\}$$

$$\frac{\tau_1 <: \tau_2}{\Omega \vdash \bullet_{\tau_1} : \tau_2 \curvearrowright \bullet_{\tau_2}}\ \ \tau_1, \tau_2 \in \{nat, int, bool\}$$

$$\frac{\Omega(\widehat{p}^-) = A_1 \quad A_1 <: A_2}{\Omega \vdash \widehat{p}^- : A_2 \curvearrowright \widehat{p}^-} \qquad \frac{\Omega(\widehat{p}^+) <: A \quad A <: \Omega(\widehat{p}^+)}{\Omega \vdash \widehat{p}^+ : A \curvearrowright \widehat{p}^+}$$

The structural well-formedness of the extended structural language have the same properties as the structural well-formedness for may-aliases of the previous section.

Lemma 7.1 (Stability of Type Decoration)

$$\Omega_{max} \vdash \widehat{E} : \Sigma \curvearrowright \widehat{E}_d \wedge \delta \vdash E : \Sigma \wedge E \in \gamma^*(\widehat{E}) \Longrightarrow E \in \gamma^*(\widehat{E}_d)$$

Proof 7.1 *First, $E \in \gamma^*(\widehat{E})$ implies the existence of ξ^* such that $E \in \gamma^*_{\xi^*}(\widehat{E})$. Now, Lemma B.2 gives us that $E \in \gamma^*_{\xi^*_{E,\Sigma}}(\widehat{E})$. The result is immediate from Lemma B.7, and Lemma B.9.*

Lemma 7.2 (Preservation of Well-formedness under Concretization)

$$\Omega \vdash \widehat{E} : \Sigma \wedge E \in \gamma^*_{\xi^*}(\widehat{E}) \Longrightarrow \delta_{\Omega,\xi^*} \vdash E : \Sigma$$

Proof 7.2 *Given that $E = (s, h)$ we must show that $\delta_{\Omega,\xi^*} \vdash s : \Sigma$, and that $\delta_{\Omega,\xi^*} \vdash h$. The result is immediate from Lemma C.3, and Lemma C.5 below.*

Merging May and Must Alias Information A structural may-alias environment, and a structural must-alias environment are *mergeable* with respect to a merge function f if the concretization of the merged result is conservative. Let $\widehat{E}^+$ range over structural may-alias environments, and $\widehat{E}^-$ range over structural must-alias environments.

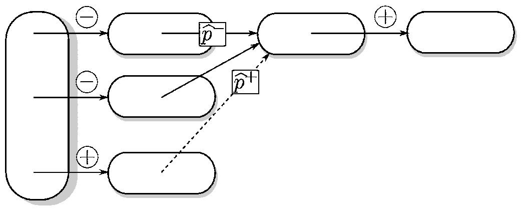
$$\gamma^+(\widehat{E}^+) \cap \gamma^-(\widehat{E}^-) \subseteq \gamma^*(f(\widehat{E}^+, \widehat{E}^-))$$

An abstract environment $\mathbb{E}$ is *mergeable* with respect to a merge function f, a may plugin $\mathcal{R}^+$, a must plugin $\mathcal{R}^-$, and an environment type Σ, if the extracted may- and must-alias environments are mergeable. We define the function *merge* as $\mathit{merge}(\mathbb{E}, \mathcal{R}^+, \mathcal{R}^-, \Sigma) = f(\eta^+(\mathbb{E}, \mathcal{R}^+, \Sigma), \eta^-(\mathbb{E}, \mathcal{R}^-, \Sigma))$ given that $\mathbb{E}$ is mergeable with respect to f, $\mathcal{R}^+$, $\mathcal{R}^-$, and Σ, and undefined otherwise. It is always possible to find merge functions, e.g., simply using the may-alias information is sound, possibly with the additional optimization that unique may-aliases are replaced by must-aliases as discussed above. For generality, in the following we parameterize over the merge function.

Strong Updates Using the combined may- and must-alias information we can create a flow-sensitive field update rule from the flow-insensitive counterpart. First, consider the type rule for field updates from Section 4 above.

$$\frac{\Sigma(x_1) = A \qquad \Sigma(x_2) = \tau \qquad \tau <: \Delta(A).f}{\Sigma \vdash x_1.f := x_2 \Rightarrow \Sigma, \bot_\Sigma}$$

Using the ideas outlined above, we can extract the merged alias-information; if the merge succeeds we know that the result is an accurate representation of the concrete environments reaching the command. Further, we know by construction that must-alias pointers form semi-isolated subgraphs in the heap in the sense that no may alias pointer points into the subgraphs, but may very well point out from it, as illustrated below, where $+$ indicates may-aliases and $-$ must-aliases. For the may-alias pointer $\widehat{p}^+$, represented by the dashed line in the figure, to point to the same position as the must-alias pointer $\widehat{p}^-$, they must be equal, i.e., $\widehat{p}^+ = \widehat{p}^-$, which is clearly not possible.



The basic idea is to perform the update in the type decorated structural representation of the environment and making sure that the new structural environment is well-formed with respect to the exit type. Let $\mathit{upd}_f(\widehat{p}^-, \widehat{v}, \widehat{E}) = (\widehat{s}_1, \widehat{h}_2)$ given that $\widehat{E} = (\widehat{s}_1, \widehat{h}_1)$, $\widehat{r}_1 = \widehat{h}_1(\widehat{p}^-)$, $\widehat{r}_2 = \widehat{r}_1[f \mapsto \widehat{v}]$, and $\widehat{h}_2 = \widehat{h}_1[\widehat{p}^- \mapsto \widehat{r}_2]$, i.e., the result of updating the field f with $\widehat{v}$ in the record pointed to by $\widehat{p}^-$ in $\widehat{E}$, defined identically for concrete environments. The rule for strong updates is defined as follows.

$$\frac{\begin{array}{c}\Omega_1 \vdash \mathit{merge}(\mathbb{M}_I(l), \mathcal{R}^+, \mathcal{R}^-, \Sigma_1) : \Sigma_1 \curvearrowright (\widehat{s}_1, \widehat{h}_1) \\ \widehat{s}_1(x_1) = \widehat{p}^- \qquad\qquad \widehat{s}_1(x_2) = \widehat{v} \\ \Omega_2 \vdash \mathit{upd}_f(\widehat{p}^-, \widehat{v}, (\widehat{s}_1, \widehat{h}_1)) : \Sigma_2\end{array}}{\Sigma_1 \vdash^{\mathbb{M}_I, \mathbb{M}_O, \mathcal{R}^+, \mathcal{R}^-} (x_1.f := x_2)^l \Rightarrow \Sigma_2, \bot_{\Sigma_1}}$$

The soundness of this rule relies on one important property for updates over must-aliases that expresses the soundness of the structural update.

Lemma 7.3 (Stability of Must-update under Concretization)

$$\{upd_f(p, v, E) \mid E \in \gamma^*_{\xi^*}(\widehat{E}), p \in \gamma^*_{\xi^*}(\widehat{p}^-), v \in \gamma^*_{\xi^*}(\widehat{v})\} = \gamma^*_{\xi^*}(upd_f(\widehat{p}^-, \widehat{v}, \widehat{E}))$$

Proof 7.3 *Since the store is unaffected by the update is suffices to show that the property holds for heaps, which is shown in the following lemma.*

Lemma 7.4 (Stability of Must-update under Heap Concretization)

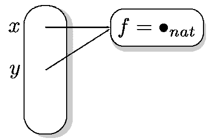
$$\{h[p \mapsto r] \mid h \in \gamma^*_{\xi^*}(\widehat{h}), p \in \gamma^*_{\xi^*}(\widehat{p}^-), r \in \gamma^*_{\xi^*}(\widehat{r})\} = \gamma^*_{\xi^*}(\widehat{h}[\widehat{p}^- \mapsto \widehat{r}])$$

Proof 7.4 *The proof relies on the facts that ξ^* has a pairwise disjoint codomain and $\gamma^*_{\xi^*}(\widehat{p}^-)$ is a singleton set not containing the null-pointer, since $\widehat{p}^-$ is a must-alias. Let $\{p\}$ be this set. Intuitively, the reasoning is as follows. On the left hand side we have that in all heaps in the concretization of $\widehat{h}$, p points to one of the concretizations of $\widehat{h}(\widehat{p}^-)$. Similarly, on the right hand side we have that in all heaps in the concretization of $\widehat{h}[\widehat{p}^- \mapsto \widehat{r}]$, p points to one of the concretizations of $\widehat{r}$. Now, if we take the concretization of $\widehat{h}$ and update p to point to a record in the concretization of $\widehat{r}$ then we get the same set as the concretization of $\widehat{h}[\widehat{p} \mapsto \widehat{r}]$.*

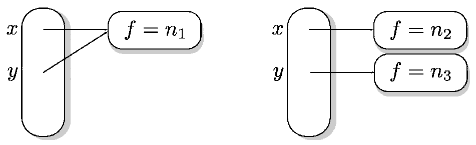
If $\widehat{p}^-$ was concretized to more than one concrete pointer, we would on the left hand side add heaps where only one of the concrete pointers is updated to point to records in the concretization of $\widehat{r}$, the other would still point to records in the concretization of $\widehat{h}(\widehat{p}^-)$.

If ξ^* did not have a pairwise disjoint codomain, p might be in the concretization of more structural pointers than $\widehat{p}$. This would require r to not only be in the concretization of $\widehat{r}$ but also in the concretization of each structural record pointed to by the additional structural pointers, something that in general would not be the case.

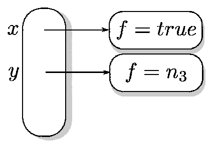
To see why the proof does not hold for may-aliases, it helps to illustrate why it holds for must-aliases. In fact, we can easily justify that $\{upd_f(p, v, E) \mid E \in \gamma^*_{\xi^*}(\widehat{E}), p \in \gamma^*_{\xi^*}(\widehat{p}^+), v \in \gamma^*_{\xi^*}(\widehat{v})\} \supseteq \gamma^*_{\xi^*}(upd_f(\widehat{p}^+, \widehat{v}, \widehat{E}))$, i.e., that the structural update is no longer guaranteed to be a sound approximation of the update. This comes from the fact that $\gamma^*_{\xi^*}(upd_f(\widehat{p}^+, \widehat{v}, \widehat{E}))$ only generates heaps where the records pointed to by all pointers $p \in \gamma^*_{\xi^*}(\widehat{p}^+)$ are updated whereas $\{upd_f(p, v, E) \mid E \in \gamma^*_{\xi^*}(\widehat{E}), p \in \gamma^*_{\xi^*}(\widehat{p}^+), v \in \gamma^*_{\xi^*}(\widehat{v})\}$ only updates the record pointer to by p. Consider the following example, where $\widehat{s} = \{x \mapsto \widehat{p}_1^+, y \mapsto \widehat{p}_1^+\}$, with $\widehat{h} = \{\widehat{p}_1^+ \mapsto \{f \mapsto \bullet_{nat}\}\}$.



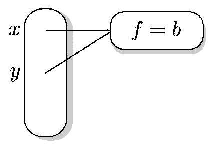
As described, the possible heaps concretized from $(\widehat{s}, \widehat{h})$ (ignoring null-pointers) can be illustrated as follows.



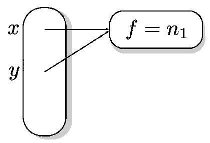
Updating the rightmost heap by $x.f := \mathit{true}$, i.e., $s_1 \mapsto \{x \mapsto p_1, y \mapsto p_2\}$, and $h_1 = \{p_1 \mapsto \{f \mapsto n_2\}, p \mapsto \{f \mapsto n_3\}\}$ results in a new heap $h_2 \mapsto \{p_1 \mapsto \{f \mapsto \mathit{true}\}, p \mapsto \{f \mapsto n_3\}\}$, illustrated below, i.e., where the record pointed to by y remains unchanged, since x and y were not aliased in this particular environment.



However, $\mathit{upd}_f(\widehat{p}^+, \widehat{v}, \widehat{E}) = (\widehat{s}, \widehat{h}_3)$, where $\widehat{h}_2 \mapsto \{\widehat{p}_1 \mapsto \{f \mapsto \bullet_{\mathit{bool}}\}$, will only produce environments of the following form.



This problem does not occur with must-aliases, since must-aliased locations are guaranteed to alias, i.e., $\widehat{s} = \{x \mapsto \widehat{p_1^-}, y \mapsto \widehat{p_1^-}\}$, with $\widehat{h} = \{\widehat{p_1^-} \mapsto \{f \mapsto \bullet_{\mathit{nat}}\}\}$ will only concretize to environments of the following structure, which are correctly modeled by the structural update.



With this, we prove the soundness of the strong update rule by proving its case in a preservation of types proof in the same way as in Section 6 above. Again, since the rule preserves width well-formedness under the assumption that the weakened command also preserves width well-formedness it can safely be added to the type system in Figure 3.

Lemma 7.5 (Preservation of Types of Strong Updates)

$$\begin{aligned}
&\Sigma_1 \vdash^{\mathbb{M}_1, \mathbb{M}_2, \eta^+, \eta^-} x_1.f := x_2 \Rightarrow \Sigma_2, \xi \wedge \\
&\qquad\qquad is_c^{\mathcal{C}}(\mathbb{M}_I) \ \wedge \ os_c^{\mathcal{C}}(\mathbb{M}_O) \Longrightarrow \\
&\qquad\qquad \forall E \in \mathcal{C}.\ \delta_1 \vdash E : \Sigma_1 \wedge \langle E, x_1.f := x_2 \rangle \to C \Longrightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad \exists \delta_2.\ \delta_2 \vdash C : \Sigma_2, \xi
\end{aligned}$$

Proof 7.5 *Assume an $E \in \mathcal{C}$, such that* (1) $\delta_1 \vdash E : \Sigma_1$, *and* (2) $\langle E, x_1.f := x_2 \rangle \to C$. *We must show that* $\delta_2 \vdash C : \Sigma_2, \xi$ *for some* δ_2.

(2) *gives two possible cases: 1) the execution fails due to* x_1 *containing a null-pointer, and 2) the execution succeeds and* $C = upd_f(p, v, E) = (s, h[p \mapsto r[f \mapsto v]]$ *for* $E = (s, h)$. *The first case is a simple exception propagation, and we focus on the second case in the following.*

First, the soundness of the merge function gives us that together with (1) *and Lemma 7.1 gives that* $E \in \gamma^*_{\xi^*}(\widehat{s}_1, \widehat{h}_1)$ *for some pointer valuation* ξ^*, *which also gives that* $p \in \gamma^*_{\xi^*}(\widehat{p}^-)$, *and* $v \in \gamma^*_{\xi^*}(\widehat{v})$, *since* $\widehat{p}^- = \widehat{s}(x_1)$, *and* $\widehat{v} = \widehat{s}(x_2)$.

We have that $upd_f(p, v, E) \in \gamma^*_{\xi^*}(upd_f(\widehat{p}^-, \widehat{v}, (\widehat{s}_1, \widehat{h}_1)))$ *from Lemma 7.3, and the result is immediate from Lemma 7.2.*

Example Use We end this section with a variation of the example of the previous section. Assuming $\Delta(A) = \{f : \mathit{nat}\}$ we saw how the following program was typable using structural weakening, by weakening the type of x and y to B, where $\Delta(B) = \{f : \mathit{int}\}$.

```
A x = new A; A y := x; x.f := 0; x.f := -1;
```

As discussed, structural weakening is limited to type changes that are supported by the subtype hierarchy. Thus, the following minor modification to the program makes the program untypable using structural weakening.

```
A x = new A; A y := x; x.f := 0; x.f := true;
```

As in the case above, even a simplistic alias analysis is able to determine that x and y are not only may-aliases but also must-aliases, and unaliased with all other locations. This means that the type rule for strong updates can be used to type $x.f := \mathit{true}$ which results in x and y getting the type B, where $\Delta(B) = \{f : \mathit{bool}\}$. The strong update is safe, since we know that x and y contain the same pointer in all program runs, and that this pointer is different from all other pointers.

8 Related Work

Using alias information to improve the precision of other analysis is widespread, e.g., [ABB06, CG93, CCL+96, LH98, PC04, FTA01, DF01, SWM99, WM01]. Common to most of these analyses is that they compute the needed alias information; our approach allows for the alias analysis to be parameterized, allowing different alias analyses to be plugged in with relative ease. Of the above mentioned work only the work on extending single static assignment (SSA) to non-scalar variables [CCL+96, CG93, LH98] uses parameterized information. It would be interesting to investigate to which extent the plugins framework could benefit the rest of the analyses.

Most closely related is the work by Smith, Walker and Morrisett [SWM99]. Therein they develop a pseudo-linear type system for alias types allowing for limited flow-sensitive types of aliased locations, and safe deallocation. With respect to the type change, our work is a generalization of theirs; their flow-sensitivity is limited to linear types, and a very specific extension using dynamic type checking.

Also related is the work by Foster, Terauchi and Aiken [FTA01] on inferring flow-sensitive type qualifiers. Even though they limit their work to type qualifiers nothing seems to prohibit their method to be applied to the full types instead of only the qualifiers. Again our work generalizes their work with respect to the use of alias information for flow-sensitivity, since they restrict the flow-sensitivity to linear types; it would be interesting to see to which extent our ideas could be used to generalized their approach.

In [ABB06] Amtoft, Bandhakavi and Banerjee develop a hoare-style logic for reasoning about noninterference. In particular, the logic contains region assertions — a simple form of alias analysis — used to increase the precision of the analysis.

In [HS06] Hunt and Sands study a flow-sensitive type system for information flow security; their work shows us that there exists a most general lattice for each program — the powerset lattice of the variables — and that, for a simple imperative language with variables, one can form a type based transformation from the flow-sensitive type system to a flow-insensitive one. This suggests that flow-sensitivity might not be necessary for information flow security. The transformation does, however, rely on the ability of easily cloning the contents of variables, and statically allocating more variables to hold the values of different types. This is not always possible, or practical. For instance, in many JVM implementations the number of simultaneously live variable is limited.

With respect to the computation of alias information, see the work on shape analysis by Sagiv, Reps and Wilhelm [SRW96], or Walker and Morrisett [WM01] on recursive alias types. For a more recent result on shape analysis see the work by Yang et al. [YLB$^+$08]. This work focuses on combining precision and scalability for use in the verification of device drivers and contains many interesting references to real world application of pointer analyses. For a more standard exposition of alias analysis see, e.g., [Deu94].

9 Conclusion

We have presented a way to allow for flow-sensitive types on the heap based on our plugin framework. In particular we have shown how may-alias information can be used to support *structural weakening*, where information about may-aliases is used to allow for a safe use of depth-subtyping in the subtyping rule, which made it possible to change the types of heap locations while retaining a uniform type view of all may-aliases, thus guaranteeing conformance with the concrete width well-formedness. Structural weakening only supports changing heap location type to more general types.

We have also shown how the combination of may- and must-alias information can be used to support strong updates, i.e., updates that do not have to follow the subtyping hierarchy. This was done by using a combined representation of may-alias and must-alias information that guaranteed that no location was both must-and may-aliased with any other location.

In addition to this we have shown how may- and must-alias information can be extracted using our plugin framework. The use of the plugin framework has very much contributed to the generality of this work by forcing us to think abstractly about may- and must-aliases, and by allowing us a flexibility of exploring many different type rules with relative ease coming from the fact that the rules are free from the computation of alias information, only containing its usage.

Bibliography

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Symposium on Principles of Programming Languages*, New York, NY, USA, 2006. ACM Press.

[CCL$^+$96] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267, London, UK, 1996. Springer-Verlag.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in ssa form. *SIGPLAN Not.*, 28(6):36–45, 1993.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 29(6), 1994.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, New York, NY, USA, 2001. ACM.

[FTA01] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. Technical report, Berkeley, CA, USA, 2001.

[GH08] Tobias Gedell and Daniel Hedin. Abstract interpretation plugins for type systems. In *12th International Conference on Algebraic Methodology and Software Technology AMAST 2008*, LNCS, 2008.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06, Proceedings of the 33rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages*, January 2006.

[HS08] S. Hunt and D. Sands. Just forget it – the semantics and enforcement of information erasure. In *Programming Languages and Systems. 17th*

European Symposium on Programming, ESOP 2008, number 4960 in LNCS, 2008.

[LH98] Christopher Lapkowski and Laurie J. Hendren. Extended ssa numbering: Introducing ssa properties to language with multi-level pointers. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 128–143, London, UK, 1998. Springer-Verlag.

[PC04] Corneliu Popeea and Wei-Ngan Chin. A type system for resource protocol verification and its correctness proof. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 135–146, New York, NY, USA, 2004. ACM.

[Pie02] Benjamin C. Pierce, editor. *Types and Programming Languages.* MIT Press, 2002.

[Pie05] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages.* MIT Press, 2005.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–31, New York, NY, USA, 1996. ACM.

[SWM99] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical report, Ithaca, NY, USA, 1999.

[WM01] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *TIC '00: Selected papers from the Third International Workshop on Types in Compilation*, pages 177–206, London, UK, 2001. Springer-Verlag.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

A Cyclic Path Property

The path property $\mathit{path}(sl_1 \ldots sl_2)$, expressing that the symbolic location sl_2 can be reached from the symbolic location sl_1 by a number of field references, is defined as follows where $\mathit{fs}(sl)$ returns the set of pointer fields of the symbolic location sl.

$$\begin{array}{lcl} \mathit{path}(sl) & \equiv & \mathit{true} \\ \mathit{path}(sl_1 \ldots sl_n) & \equiv & \mathit{path}(sl_1 \ldots sl_{n-1}) \wedge \\ & & \exists f \in \mathit{fs}(sl_{n-1}).\ sl_{n-1}.f = sl_n \end{array}$$

The cyclic property $\mathit{cyclic}(sl_1 \ldots sl_n, \mathcal{R})$, expressing that there exists two unique locations sl_i and sl_j in $sl_1 \ldots sl_n$ such that they are related by the plugin $\mathcal{R}$, is defined in the following way.

$$\mathit{cyclic}(sl_1 \ldots sl_n, \mathcal{R}) \quad \equiv \quad \exists i, j \in [1 \ldots n].\ i \neq j \ \wedge \ (sl_i, sl_j) \in \mathcal{R}$$

Finally, we define the cyclic path property $\mathit{cyclicp}(\mathcal{R})$, expressing that the plugin $\mathcal{R}$ has an upper limit n on the length of acyclic paths.

$$\mathit{cyclicp}(\mathcal{R}) \quad \equiv \quad \exists n.\ \forall sl_1 \ldots sl_n.\ \mathit{path}(sl_1 \ldots sl_n) \Longrightarrow \mathit{cyclic}(sl_1 \ldots sl_n, \mathcal{R})$$

B Stability of Type Decoration

This section contains the proofs of stability of type decoration of the structural may-aliases of Section 6, and the combined structural may- and must-aliases of Section 7 — the former language is a sublanguage of the latter. In this section all pointer valuations are may- and must-alias sound, why the $*$ superscript is dropped from ξ^* throughout.

Definition B.1 (Ω order) *We define an order $\leq$ on abstract pointer typings as follows.*

$$\frac{\widehat{p} \in \mathit{dom}(\Omega_2).\ \Omega_1(\widehat{p}) <: \Omega_2(\widehat{p})}{\Omega_1 \leq \Omega_2}$$

The intuition behind the order is that bigger abstract pointer typings place less demands on structural environment.

Lemma B.1 (Maximal Ω) *For a given structural environment $\widehat{E}$, there exists a unique maximal (up to type constrained isomorphism) Ω_{max}, such that*

$$\Omega \vdash \widehat{E} : \Sigma \Longrightarrow \Omega \leq \Omega_{max} \wedge \Omega_{max} \vdash \widehat{E} : \Sigma$$

Proof B.1 *The intuition is that increasing abstract pointer typings place less demands on the structural environment, and that the maximal abstract-pointer typing for a given well-formed structural environment is given by Σ.*

First, it is clear that for a given Ω there exists a maximal abstract-pointer typing Ω_{max} obtainable by repeatedly choosing bigger pointer typings until no bigger exists in which $\widehat{E}$ is still Σ well-formed.

Now, assume that there exists two different maximal pointer typings Ω_{max_1} Ω_{max_2}. It is clear that neither $\Omega_{max_1} \leq \Omega_{max_2}$, nor $\Omega_{max_2} \leq \Omega_{max_1}$, since in such case one of them would not be maximal. Thus, there exists at least one abstract pointer $\widehat{p}$ occurring at a symbolic location l, such that $\Omega_{max_1}(\widehat{p}) <: \Sigma(l)$, and $\Omega_{max_2}(\widehat{p}) <: \Sigma(l)$, but with $\Omega_{max_1}(\widehat{p})$ and $\Omega_{max_2}(\widehat{p})$ incomparable. Because of the use of width-subtyping we know that all shared fields of $\Omega_{max_1}(\widehat{p})$, and $\Omega_{max_2}(\widehat{p})$ must be equal.

This gives us that $\Omega_{max_1}(\widehat{p})$ has a field not in $\Omega_{max_2}(\widehat{p})$ and vise versa. However, the incompatible fields are not forced by $\Sigma(l)$, given by $\Omega_{max_1}(\widehat{p}) <: \Sigma(l)$, and $\Omega_{max_2}(\widehat{p}) <: \Sigma(l)$, which means $\Omega_{max_1}(\widehat{p})$ and $\Omega_{max_2}(\widehat{p})$ are not maximal — they can both be replaced by $\Omega_{max_1}(\widehat{p}) \sqcap \Omega_{max_2}(\widehat{p}) <: \Sigma(l)$.

The maximal abstract pointer typing Ω_{max} is easily obtained by a fixed-point iteration.

Pointers not being live require special care when establishing stability of type decoration. To illustrate this, consider the case where $\widehat{s}(x) = \widehat{p}$, $\widehat{h}(\widehat{p}) = \{f \mapsto \bullet\}$, $\xi(\widehat{p}) = \{p_1, p_2\}$, $\Sigma(x) = A$ and $\Delta(A) = \{f : int\}$.

All heaps in the concretization of $\widehat{h}$ will have both p_1 and p_2 in its domain. In all heaps that are well-formed with respect to Σ, the well-formedness relation will require that the pointer that x contains will point to a record containing a value that is well-formed with respect to the type *int*. The other pointer will, however, not have any requirements placed on it since it will not be live.

However, in the concretization of the type decorated version of the heap, $\widehat{h}(\widehat{p}) = \{f \mapsto \bullet_{int}\}$, both p_1 and p_2 will be required to point to records containing values that are well-formed with respect to the type *int*.

This means that type decoration does not preserve concretizations for all pointer valuations. In order to work around this, we restrict ourselves to only consider the live pointers. This is reasonable to do, since a pointer that is not live is semantically safe to ignore. More, specifically, noting that the well-formedness relation only places requirements on pointers that are typed by Σ, we limit ourselves to only consider pointers that are live and that are given a type by Σ. We do this by defining Σ-*reachability.*

Definition B.2 (Σ-reachability) *We say that p is Σ-reachable in E, written $p \in E_\Sigma$, if there exists a δ such that $\delta \vdash E : \Sigma$ and there exists a symbolic location l such that $\Sigma(l)$ is defined and $E(l) = p$.*

When establishing stability of type decoration this is done with respect to a pointer valuation ξ whose codomain only consists of Σ-reachable pointers. We call such a ξ *minimal* and define it in the following way.

Definition B.3 (Minimal ξ) *For each environment $E \in \gamma_\xi(\widehat{E})$ such that $\delta \vdash E : \Sigma$, we define the minimal pointer valuation $\xi_{E,\Sigma}$ to be the sub-valuation of ξ that only contains the pointers Σ-reachable in E.*

$$\forall \widehat{p} \in dom(\xi), p \in \xi_{E,\Sigma}(\widehat{p}).\ p \in \xi(\widehat{p}) \wedge p \in E_\Sigma$$

Lemma B.2 ($\xi_{E,\Sigma}$ **preserves** E)

$$\Omega \vdash \widehat{E} : \Sigma \wedge E \in \gamma_\xi(\widehat{E}) \wedge \delta \vdash E : \Sigma \Longrightarrow E \in \gamma_{\xi_{E,\Sigma}}(\widehat{E})$$

Proof B.2 *By construction - $\xi_{E,\Sigma}$ contains all pointers that are live, and typed in E, but no else. Thus, it removes all demands on things that are not live or typed, while allowing for the same concretization of all live and typed locations.*

Lemma B.3 (Σ-reachable locations are well-formed)

$$\delta \vdash E : \Sigma \wedge E(l) = v \wedge \Sigma(l) = \tau \Longrightarrow \delta \vdash v : \tau$$

Proof B.3 *By straightforward induction on the symbolic location.*

Lemma B.4 (Σ-reachable pointers are well-formed)

$$\delta \vdash E : \Sigma \wedge p \in \gamma_{\xi_{E,\Sigma}} \Longrightarrow \exists A \ . \ \delta(p) = A$$

Proof B.4 *The result is immediate, since $\xi_{E,\Sigma}$ forms a subset of the Σ-reachable pointers.*

By definition if $p \in \gamma_{\xi_{E,\Sigma}}$ there exists a symbolic location l such that $E(l) = p$ and $\Sigma(l) = \tau$. From Lemma B.3 we have that $\delta \vdash p : \tau$, which gives us $\tau = A_1$, and $\delta(p) = A_2 <: A_1$.

Lemma B.5

$$\Omega_{max} \vdash \widehat{E} : \Sigma \wedge \delta \vdash E : \Sigma \wedge E \in \gamma_{\xi_{E,\Sigma}}(\widehat{E}) \Longrightarrow$$
$$p \in \xi_{E,\Sigma}(\widehat{p}) \wedge \widehat{p} \in dom(\Omega_{max}) \wedge \Longrightarrow \delta(p) <: \Omega_{max}(\widehat{p})$$

Proof B.5 *First we state some properties needed.*

1. *$\Omega_{max} \vdash \widehat{E} : \Sigma$ gives us that all symbolic locations l such that $\Sigma(l)$ is defined we either that $\widehat{E}(l) = \widehat{p}^+$, $\widehat{E}(l) = \widehat{p}^-$, or that $\widehat{E}(l) = \bullet$.*

2. *Let $L_{\widehat{p}^+}$ be the set of symbolic locations such that for $l \in L_{\widehat{p}^+}$, $\Sigma(l)$ is defined and $\widehat{E}(l) = \widehat{p}$. Since $E \in \gamma_\xi(\widehat{E})$ we have for each $p \in \xi(\widehat{p}^+)$ that the set $L_{\widehat{p}}$ such that $l \in L_p$, $\Sigma(l)$ is defined and $E(l) = p$ is a subset of $L_{\widehat{p}^+}$.*

3. *Let $L_{\widehat{p}^-}$ be the set of symbolic locations such that for $l \in L_{\widehat{p}^-}$, $\Sigma(l)$ is defined and $\widehat{E}(l) = \widehat{p}$. Since $E \in \gamma_\xi(\widehat{E})$ we have for the unique $p \in \xi(\widehat{p}^-)$ that the set L_p such that $l \in L_p$, $\Sigma(l)$ is defined and $E(l) = p$ is a equal to the set $L_{\widehat{p}^-}$.*

4. *From, $\Omega_{max} \vdash \widehat{E} : \Sigma$ we have that for all $l \in L_{\widehat{p}^+}$ it holds that $\Omega_{max}(\widehat{p}) <: \Sigma(l)$ and $\Sigma(l) <: \Omega_{max}(\widehat{p})$, i.e., all may-aliased locations have exactly the same type view (up to renaming). Together with $E \in \gamma_\xi(\widehat{E})$ this gives us that all occurrences of concrete pointers $p \in \xi(\widehat{p})$ have the same type view.*

5. *From, $\Omega_{max} \vdash \widehat{E} : \Sigma$ we have that for all $l \in L_{\widehat{p}^-}$ it holds that $\Omega_{max}(\widehat{p}) <: \Sigma(l)$, i.e., all must-aliased locations have compatible type views Together with $E \in \gamma_\xi(\widehat{E})$ this gives us that all occurrences of concrete pointers $p \in \xi(\widehat{p})$ have compatible type views.*

First, using $\delta \vdash E : \Sigma$ and $p \in \xi_{E,\Sigma}(\widehat{p})$, Lemma B.4 gives us $\delta(p) = A$ for some A, i.e., $\delta(p)$ is defined. The proof has two cases.

May-alias case *Assume $p \in \xi(\widehat{p}^+)$, and $\widehat{p}^+ \in dom(\Omega_{max})$; from above we have that all $l \in L_{\widehat{p}^+} \supseteq L_p$ have exactly the same type view. Thus, $\delta(p) <: \Sigma(l) = \Omega_{max}(\widehat{p}^+)$.*

Must-alias case *Assume $p \in \xi(\widehat{p}^-)$, and $\widehat{p}^- \in dom(\Omega_{max})$; from above we have that $l \in L_{\widehat{p}^+} = L_p$. Furthermore, since Ω_{max} is maximal we know that $\Omega_{max}(\widehat{p}^-) = \bigsqcap_{l \in L_{\widehat{p}^+}} \Sigma(l)$. Thus, since for all $l \in L_p$ it holds $\delta(p) <: \Sigma(l)$ we have that $\delta(p) <: \Omega_{max}(\widehat{p}^-) <: \Sigma(l)$.*

Lemma B.6 (Stability of Value Type Decoration)

$$\Omega_{max} \vdash \widehat{v} : \tau_1 \curvearrowright \widehat{v}_d \wedge \delta \vdash v : \tau_2 \wedge \tau_2 <: \tau_1 \wedge v \in \gamma_\xi(\widehat{v}) \Longrightarrow v \in \gamma_\xi(\widehat{v}_d)$$

Proof B.6 *Assume* (1) $\Omega_{max} \vdash \widehat{v} : \tau_1 \curvearrowright \widehat{v}_d$, (2) $\delta \vdash v : \tau_2$, (3) $\tau_2 <: \tau_1$ *and* (4) $v \in \gamma_\xi(\widehat{v})$.

The proof continues by a case analysis of (1).

case $\Omega_{max} \vdash \bullet : \tau_1 \curvearrowright \bullet_{\tau_1}$ *We must show that $v \in \gamma_\xi(\widehat{v}_d) = [\![\tau_1]\!]$ for the cases where τ_1 is one of int, nat or bool. This follows directly from* (2) *and* (3).

case $\Omega_{max} \vdash \widehat{p} : \boldsymbol{A} \curvearrowright \widehat{p}$ *Since the decoration does not affect the concretization of pointers the result follows directly from* (4).

Lemma B.7 (Stability of Store Type Decoration)

$$\Omega_{max} \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d \wedge \delta \vdash s : \Sigma \wedge s \in \gamma_\xi(\widehat{s}) \Longrightarrow s \in \gamma_\xi(\widehat{s}_d)$$

Proof B.7 *Assume* (1) $\Omega_{max} \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d$, (2) $\delta \vdash s : \Sigma$, *and* (3) $s \in \gamma_\xi(\widehat{s})$.

We must show that $\forall x \in dom(\widehat{s}_d).\ s(x) \in \gamma_\xi(\widehat{s}_d(x))$. We have that (4) $\forall x \in dom(\widehat{s}).\ s(x) \in \gamma_\xi(\widehat{s}(x))$ *from* (3), *and* (2) *gives us that* (5) $\forall x \in dom(\Sigma).\ \delta \vdash s(x) : \Sigma(x)$. *Now,* (1) *gives us that* (6) $\forall (x, \tau) \in \Sigma.\ \Omega_{max} \vdash \widehat{s}(x) : \tau \curvearrowright \widehat{s}_d(x)$, (7) $dom(\widehat{s}) = dom(\widehat{s}_d)$, *and* (8) $\forall x \in dom(\widehat{s}) \setminus dom(\Sigma).\ \widehat{s}_d(x) = \widehat{s}(x)$.

Thus, assuming $x \in dom(\widehat{s}_d)$, we either have $x \in dom(\widehat{s}) \setminus dom(\Sigma)$ in which case we are done by (8) *or $x \in dom(\Sigma)$ and $x \in dom(\widehat{s})$ by* (7). *Now,* (6) *gives $\Omega_{max} \vdash \widehat{s}(x) : \Sigma(x) \curvearrowright \widehat{s}_d(x)$,* (5) *gives $\delta \vdash s(x) : \Sigma(x)$,* (4) *gives $s(x) \in \gamma_\xi(\widehat{s}(x))$, and we reach the conclusion via Lemma B.6.*

Lemma B.8 (Stability of Record Type Decoration)

$$\Omega_{max} \vdash \widehat{r} : \omega_1 \curvearrowright \widehat{r}_d \wedge \delta \vdash r : \omega_2 \wedge \omega_2 <: \omega_1 \wedge r \in \gamma_\xi(\widehat{r}) \Longrightarrow r \in \gamma_\xi(\widehat{r}_d)$$

Proof B.8 *Assume* (1) $\Omega_{max} \vdash \widehat{r} : \omega_1 \curvearrowright \widehat{r}_d$, (2) $\delta \vdash r : \omega_2$, (3) $\omega_2 <: \omega_1$, *and* (4) $r \in \gamma_\xi(\widehat{r})$.

To show $r \in \gamma_\xi(\widehat{r}_d)$ *we need to show that* $\forall f \in dom(\widehat{r}_d).\ r.f \in \gamma_\xi(\widehat{r}_d.f)$.

(1) *gives* (5) $\forall (f, \tau) \in \omega_1.\ \Omega_{max} \vdash \widehat{r}.f : \tau \curvearrowright \widehat{r}_d.f$, (6) $dom(\widehat{r}) = dom(\widehat{r}_d)$, *and* (7) $\forall f \in dom(\widehat{r}) \setminus dom(\omega).\ \widehat{r}_d(f) = \widehat{r}(f)$. (2) *gives* (8) $\forall (f, \tau) \in \omega_2.\ \delta \vdash r.f : \tau$. (3) *gives* (9) $\forall (f, \tau) \in \omega_1.\ \omega_2.f = \tau$.

Assume $f \in dom(\widehat{r}_d)$. (6) *gives that we have either* $f \in dom(\widehat{r}) \backslash dom(\omega_1)$ *and we are done by* (7), *or* $f \in \widehat{r}$, *and* $f \in \omega_1$ *such that* $\Omega_{max} \vdash \widehat{r}.f. : \omega_1.f. \curvearrowright \widehat{r}_d.f.$ *Now,* (9) *gives us that* $\omega_2.f = \omega_1.f$, *and* (8) *gives* $\delta \vdash r.f : \omega_2.f$. *With this Lemma B.6 allows us to conclude.*

Lemma B.9 *Stability of Heap Type Decoration*

$$\Omega_{max} \vdash (s, \widehat{h}) : \Sigma \curvearrowright (\widehat{s}_d, \widehat{h}_d) \wedge \delta \vdash (s, h) \wedge h \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}) \Longrightarrow h \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}_d)$$

Proof B.9 *Assume* (1) $\Omega_{max} \vdash \widehat{h} \curvearrowright \widehat{h}_d$, (2) $\delta \vdash h$, *and* (3) $h \in \gamma_{\xi_{E,\Sigma}}(\widehat{h})$. (1) *gives* (4) $\forall (\widehat{p}, A) \in \Omega_{max}.\ \Omega_{max} \vdash \widehat{h}(\widehat{p}) : \Delta(A) \curvearrowright \widehat{h}_d(\widehat{p})$, (5) $dom(\widehat{h}) = dom(\widehat{h}_d)$, *and* (6) $\forall \widehat{p} \in dom(\widehat{h}) \setminus dom(\Omega)_{max}.\ \widehat{h}_d(\widehat{p}) = \widehat{h}(\widehat{p})$. (2) *gives* $\forall (p, A) \in \delta.\ \delta \vdash h(p) : \Delta(A)$, *and* (3) *gives* $\forall \widehat{p} \in dom(\widehat{h}), p \in \xi_{E,\Sigma}(\widehat{p}).\ h(p) \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}(\widehat{p}))$.

We must show that $\forall \widehat{p} \in dom(\widehat{h}_d), p \in \xi_{E,\Sigma}(\widehat{p}).\ h(p) \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}_d(\widehat{p}))$. *Assume* $\widehat{p} \in dom(\widehat{h}_d)$ *and* $p \in \xi_{E,\Sigma}(\widehat{p})$. (5) *gives either* $\widehat{p} \in dom(\widehat{h}) \setminus dom(\Omega_{max})$ *and we are done by* (6) *or* $(\widehat{p}, A_1) \in \Omega_{max}$ *and thus that* $\Omega_{max} \vdash \widehat{h}(\widehat{p}) : \Delta(A_1) \curvearrowright \widehat{h}_d(\widehat{p})$ *by* (4). *From Lemma B.5 we have that* $\delta(p) <: A_1$, *which implies that* δ *is defined for* p, *i.e.,* $\delta(p) = A_2$ *for some* A_2. *Now,* (2) *gives us* $\delta \vdash h(p) : \Delta(A_2)$, *and we are done by Lemma B.8.*

Lemma B.10 (Stability of Type Decoration)

$$\Omega_{max} \vdash: \widehat{E} : \Sigma \curvearrowright \widehat{E}_d \wedge \delta \vdash: E : \Sigma \wedge E \in \gamma(\widehat{E}) \Longrightarrow E \in \gamma(\widehat{E}_d)$$

Proof B.10 *First,* $E \in \gamma(\widehat{E})$ *implies the existence of* ξ *such that* $E \in \gamma_\xi(\widehat{E})$. *Now, Lemma B.2 gives us that* $E \in \gamma_{\xi_{E,\Sigma}}(\widehat{E})$. *The result is immediate from Lemma B.7, and Lemma B.9.*

Lemma B.11 *Type Decoration Preserves Well-formedness*

$$\Omega \vdash \widehat{E}_1 : \Sigma \curvearrowright \widehat{E}_2 \Longrightarrow \Omega \vdash \widehat{E}_2 : \Sigma$$

Proof B.11 *The result is immediate from inspecting the rules and noting that the only decoration takes place in the well-formedness rule for* $\bullet$ *and that the decoration is the type demanded by well-formedness, i.e.,* $\Omega \vdash \bullet : int : \bullet_{int}$.

C Preservation of Well-formedness under Concretization

This section contains the proofs that well-formedness is preserved by concretization of the structural may-aliases of Section 6, and the combined structural may- and must-aliases of Section 7. In this section all pointer valuations are may- and must-alias sound, why the $*$ superscript is dropped from ξ^* throughout.

Lemma C.1 *Preservation of Well-formedness under Concretization*

$$\Omega \vdash \widehat{E} : \Sigma \wedge E \in \gamma_\xi(\widehat{E}) \Longrightarrow \delta_{\Omega,\xi} \vdash E : \Sigma$$

Proof C.1 *Given that $E = (s, h)$ we must show that $\delta_{\Omega,\xi} \vdash s : \Sigma$, and that $\delta_{\Omega,\xi} \vdash h$. The result is immediate from Lemma C.3, and Lemma C.5 below.*

Lemma C.2 *Preservation of Well-formedness under Value Concretization*

$$\Omega \vdash \widehat{v} : \tau \wedge v \in \gamma_\xi(\widehat{v}) \Longrightarrow \delta_{\Omega,\xi} \vdash v : \tau$$

Proof C.2 *We proceed by a case analysis on $\widehat{v}$.*

case $\widehat{v} = \bullet_{\tau_1}$ *We have that $v \in [\![\tau_1]\!]$ from $v \in \gamma_\xi(\widehat{v})$, and that $\tau_1 <: \tau$ from $\Omega \vdash \widehat{v} : \tau$, and the result is immediate.*

case $\widehat{v} = \widehat{p}^+$ *We have that $v = p \in \xi(\widehat{p}^+)$ from $v \in \gamma_\xi(\widehat{p}^+)$, and that $\Omega(\widehat{p}^+) = \tau = A$ for some A. By definition $\delta_{\Omega,\xi}(p) = \Omega(\widehat{p}^+)$ given $p \in \gamma_\xi(\widehat{p}^+)$, and the result is immediate.*

case $\widehat{v} = \widehat{p}^-$ *We have that $v = p \in \xi(\widehat{p}^-)$ from $v \in \gamma_\xi(\widehat{p}^-)$, and that $\Omega(\widehat{p}^-) <: \tau = A$ for some A. By definition $\delta_{\Omega,\xi}(p) = \Omega(\widehat{p}^-)$ given $p \in \gamma_\xi(\widehat{p}^-)$, and the result is immediate.*

Lemma C.3 (Preservation of Well-formedness under Store Concretization)

$$\Omega \vdash \widehat{s} : \Sigma \wedge s \in \gamma_\xi(\widehat{s}) \Longrightarrow \delta_{\Omega,\xi} \vdash ws : \Sigma$$

Proof C.3 *We must show that $(x, \tau) \in \Sigma.\ \delta_{\Omega,\xi} \vdash s(x) : \tau$. Assume $(x, \tau) \in \Sigma$. We have that $\Omega \vdash \widehat{s}(x) : \tau$ from $\Omega \vdash \widehat{s} : \Sigma$, and $s(x) \in \gamma_\xi(\widehat{s}(x))$ from $s \in \gamma_\xi(\widehat{s})$. The result is immediate from Lemma C.2.*

Lemma C.4 (Preservation of Well-formedness under Record Concretization)

$$\Omega \vdash \widehat{r} : \omega \wedge r \in \gamma_\xi(\widehat{r}) \Longrightarrow \delta_{\Omega,\xi} \vdash r : \omega$$

Proof C.4 *We must show that $(f, \tau) \in \omega.\ \delta_{\Omega,\xi} \vdash r.f : \tau$. Assume $(f, \tau) \in \omega$. We have that $\Omega \vdash \widehat{r}.f : \tau$ from $\Omega \vdash \widehat{r} : \omega$, and $r.f \in \gamma_\xi(\widehat{r}.f)$ from $r \in \gamma_\xi(\widehat{r})$. The result is immediate from Lemma C.2.*

Lemma C.5 (Preservation of Well-formedness under Heap Concretization)

$$\Omega \vdash \widehat{h} \land h \in \gamma_\xi(\widehat{h}) \Longrightarrow \delta_{\Omega,\xi} \vdash h$$

Proof C.5 *We must show that* $(p, A) \in \delta_{\Omega,\xi}.\ \delta_{\Omega,\xi} \vdash h(p) : \Delta(A)$. *Assume* $(p, A) \in \delta_{\Omega,\xi}$. *By definition of* $\delta_{\Omega,\xi}$ *we have that there exists an abstract structural pointer (must or may)* $\widehat{p}$ *such that* $p \in \xi(\widehat{p})$, *and* $\Omega(\widehat{p}) = A$. *We have that* $\Omega \vdash \widehat{h}(\widehat{p}) : \Delta(A)$ *from* $\Omega \vdash \widehat{h}$, *and from the fact that* ξ *has pairwise disjoint codomain we have that* $\widehat{p}$ *is unique, which together* $h \in \gamma_\xi(\widehat{h})$, *gives us* $h(p) \in \gamma_\xi(\widehat{h}(\widehat{p}))$ *and the result is immediate from Lemma C.4.*