Go Against the Flow for Typeless Programming

Sean McDirmid

Microsoft Research, Beijing, China
smcdirm@microsoft.com

Abstract. This paper introduces a novel type system that can infer an object's composition of traits (mixin-style classes) according to its usage in a program. Such "trait inference" is achieved with two new ideas. First, through a polyvariant treatment of assignments (`a := b`) and implied field sub-assignments, assignment can usefully encode trait extensions, method calls, overriding, and so on. Second, rather than unify term types or solve inequality constraints via intersection, we instead propagate trait requirements "backward" to assignees, ensuring that trait requirements are compatible only for terms used in the program. We show how this system is feasible without type annotations while supporting mutability, sub-typing, and parametricity.

1 Introduction

Traits like those in Scala [22, 6] are classes that support a mixin-like form of linearized multiple inheritance [5, 19]. Although traits allow for much finer-grained modularizations than classes, they incur a huge burden on programmers who must often manually compose many traits in program type annotations and object definitions. Programmers must also pre-commit to annotations that limit what options appear in code completion, preventing them from discovering useful functionality that is contained in more obscure traits [16].

The goal of our language, called YinYang, is to infer what traits an object extends based only on how they are used, rather than restrict usage based on type annotations. Global type inference that would achieve this has been elusive in object-oriented languages where subtyping leads to inequalities that defy standard Hindley-Milner unification [10, 18] and are expensive to solve as constraints, especially when mutable fields are considered (as in [2]). More significantly, most OO type inference systems work in the wrong direction by intersecting types toward assigned terms and away form object compositions.

Using two new ideas, this paper rethinks type inference, and typing in general, to enable object trait inference. First, a novel polyvariant treatment of term assignments along with implied field sub-assignments precisely captures the flow of objects through a program. Assignment and fields can then completely encode trait extensions, method calls, overriding, and so on, transforming the problem of type checking and inferring types into one of reachability within a data flow graph. Second, rather than solve inequality constraints via intersection in the direction of assigned terms, we instead solve them via union in the direction of

assignee terms. What traits an object extends can then be inferred, but comes at the expense of not being able to directly verify programmer intent; e.g. did they really mean for that object to be a duck? We temper this problem with trait compatibility where certain traits cannot be extended by the same object, e.g. Duck and number cannot be extended by the same object, which is checked only for terms referenced in the program.

We are implementing these ideas in our type inference system for our YinYang language; earlier broken versions of this system have already been prototyped, gaining us the understanding that we present here. The paper continues in Section 2 to describe a novel sound treatment of assignment and fields. This treatment is then applied to YinYang in Section 3, showing how, when combined with a backward propagation of trait requirements in the direction of assignees, object extensions can be inferred while subtyping, overriding, covariance, and parametricity are accounted for correctly. The challenges of decidability and scalability are tackled in Section 4 by treating inference as a problem of reachability within a tree of nodes connected together by assignment edges. Section 5 presents open issues; e.g. YinYang cannot currently deal with dynamic type checks. Related work is covered in Section 6 while Section 7 concludes.

2 A Better Treatment of Assignments and Fields

Reference assignment and subtyping are difficult to handle in most type inference systems [7, 11, 24, 33, 34] and is often heavily restricted compared to functional programming styles. Semantically, subtyping and assignment are very similar when considering the sets of objects that can be bound to extensible traits (or classes) and assigned terms. For an assignment $\mathsf{A} \coloneqq \mathsf{B}$ in a typical Andersen's points-to analysis [4], the set of objects that term B can be bound to, which we denote as B_δ, is a subset of those that can be bound to term A, so $\mathsf{B}_\delta \subseteq \mathsf{A}_\delta$. Likewise, for a trait/class extension S extends T, $\mathsf{S}_\delta \subseteq \mathsf{T}_\delta$. This suggests that inheritance and assignment can be modeled the same in a type system: if we define $\mathsf{A} \coloneqq \mathsf{B}$ to be type-wise equivalent to $\mathsf{B}_\delta \subseteq \mathsf{A}_\delta$, then $\mathsf{T} \coloneqq \mathsf{S}$ also expresses an extension of T by S, and $\mathsf{T} \coloneqq \mathsf{A}$ expresses that term A must be of a type that contains trait T. Rather than shoehorn subtyping or assignment in a system of quantified and constrained type variables, we instead focus on just assignment.

Field sensitivity [36] complicate the semantics of how objects flow across an assignment: given an assignment $\mathsf{A} \coloneqq \mathsf{B}$, a store to a field via A only affects the objects pointed to by A, such as those in B, and only in that field. Most previous work on field sensitivity [12, 13, 31, 36] deals with fields indirectly: rather than determine how fields of A and B are related by assignment, object flows between fields are "expanded" out based on object flows to their parents. While this might be appropriate to compute points-to relationships, it does not make sense for a type system that does not. We propose instead a treatment of fields that decodes their flow behavior into assignments to avoid actual object propagation.

For some field F present in A where $\mathsf{A} \coloneqq \mathsf{B}$, we could assume that $\mathsf{A.F} \coloneqq \mathsf{B.F}$ to capture the intuition that $\mathsf{B.F}_\delta \subseteq \mathsf{A.F}_\delta$. However, consider:

① $\mathsf{A} \coloneqq \mathsf{B}$ | ② $\mathsf{A.F} \coloneqq \mathsf{D}$ | ③ $\mathsf{A} \coloneqq \mathsf{C}$

With assignment ②, objects bound to $\mathsf{D}_\delta \subseteq \mathsf{A.F}_\delta$, but since $\mathsf{B}_\delta \subseteq \mathsf{A}_\delta$, we must conservatively assume that $\mathsf{D}_\delta \subseteq \mathsf{B.F}_\delta$ as well. Perhaps sub-field assignment is bi-directional, unifying $\mathsf{B.F}$ and $\mathsf{A.F}$ (i.e. $\mathsf{B.F} = \mathsf{A.F}$). Assignment ③ would then indirectly unify $\mathsf{C.F}$ and $\mathsf{A.F}$ ($\mathsf{B.F}_\delta = \mathsf{C.F}_\delta$): because A can only bound to at most one object from either B or C at a time, there is no intuitive flow of objects between $\mathsf{B.F}$ or $\mathsf{C.F}$, meaning unification is too conservative! Because A could be bound to either an object in B or C, the assignments $\mathsf{B.F} \coloneqq \mathsf{D}$ and $\mathsf{C.F} \coloneqq \mathsf{D}$ are still conservatively necessary, but any assignments between $\mathsf{B.F}$ and $\mathsf{C.F}$ are not.

For more precise reasoning about how objects flow across field subassignments, we define a new *weak* assignment operator $:\!-$ in addition to conventional "strong" assignment ($\coloneqq$) to propagate field sub-assignment operations as being "different" even if it still represents the flow of objects through terms. Weak assignment is defined and used as follows:

$$\text{ⓐ}\ \frac{\Gamma \vdash e_a \coloneqq e_b}{\Gamma \vdash e_a :\!- e_b} \qquad \text{ⓑ}\ \frac{\Gamma \vdash e_a :\!- e_b}{\Gamma \vdash e_a.\mathsf{F} :\!- e_b.\mathsf{F}} \qquad \text{ⓒ}\ \frac{\Gamma \vdash e_a :\!- e_b, \quad e_a.\mathsf{F} \coloneqq e_c}{\Gamma \vdash e_b.\mathsf{F} \coloneqq e_c}$$

Γ is an original program of assignments expressed by the programmer; e are terms in this program of the form $e ::\!= (\mathcal{R} \mid e).\mathsf{F} \mid \mathsf{new}$; F identifies a unique object field; $\mathcal{R}$ is a root term bound to a root object; and new are object allocation sites. Strong assignment derives weak assignment in ⓐ; in ⓑ weak assignment can then be derived between the fields of terms related by weak assignment, which we refer to as *field subassignment*. Finally, *reassignment* in ⓒ matches term weak assignments and child field strong assignments to derive new strong assignments, capturing appropriate object flow while avoiding conservative field unifications.

A program Γ is defined as follows:

$$\Gamma ::\!= \overline{g.\mathsf{F} \coloneqq h} \qquad \begin{array}{l} g ::\!= g.\mathsf{F} \mid \mathcal{R} \\ h ::\!= g.\mathsf{F} \mid \mathsf{new} \end{array} \qquad \Gamma = \begin{array}{l} \overline{g.\mathsf{F} \coloneqq h} \vdash \\ \overline{g.\mathsf{F} \coloneqq h} \end{array}$$

Note on syntax: we use $=$ to deconstruct symbols in derivations; overline means a list, and $\Gamma \vdash \overline{x}$ expands out to $\Gamma \vdash x_0, x_1, \ldots$, meaning each element of the list is its own derived judgment. Because $g.\mathsf{F}$ and h terms can be used as e terms, program assignments can be derived directly as $e \coloneqq e$ relationships.

A proof of soundness requires the use of a transitive weak assignment operator ($:\!-^*$) to express *object reachability*: if $e :\!-^* \mathsf{new}$, then new could be assigned indirectly to e. Transitive weak assignment ($:\!-^*$) is defined as:

$$\Gamma \vdash e :\!-^* e \qquad \frac{\Gamma \vdash e_a :\!-^* e_b, \quad e_b :\!- e_c}{\Gamma \vdash e_a :\!-^* e_c}$$

Lemma 1 shows that reassignment is still valid with transitivity.

Lemma 1. $\Gamma \vdash e_a :\!-^* e_b,\ e_a.\mathsf{F} \coloneqq e_c \ \Rightarrow\ \Gamma \vdash e_b.\mathsf{F} \coloneqq e_c$

Proof. Straightforward by applying reassignment inductively on $e_a :\!-^* e_b$ relationships, with a base case of one $:\!-$ relationship.

Operational Semantics. Soundness is shown over *traces* of Γ program executions where assignments in Γ are mixed up in repeating orders to account for control flow and concurrency in our semantics. A trace $\bar{b}$ has an arbitrary length where $b_i \in \Gamma$ is the i^{th} assignment to execute in the trace. A non-root object is encoded as $\langle \mathsf{new}_x, j \rangle$, which is created by the j^{th} trace assignment $b_j = g_i.\mathsf{F} := \mathsf{new}_x$, where many objects can be created from allocation site new_x.

A store Θ_i tracks object field assignments to other objects before executing b_i of the trace; $\Theta_i((\langle \mathsf{new}_a, j \rangle \mid \mathcal{R}).\mathsf{F}) = \langle \mathsf{new}_b, k \rangle$ evaluates to what $\langle \mathsf{new}_a, j \rangle.\mathsf{F}$ (or $\mathcal{R}.\mathsf{F}$) was most recently assigned to ($\langle \mathsf{new}_b, k \rangle$). Application of Θ_i is defined on e's possible forms: $\Theta_i(e.\mathsf{F}) = \Theta_i(\Theta_i(e).\mathsf{F})$, $\Theta_i(\mathsf{new}_a) = \langle \mathsf{new}_a, i \rangle$ as object allocation, and $\Theta_i(\mathcal{R}) = \mathcal{R}$. Execution of an assignment is as follows:

$$\Theta_i \times b_i = g_i.\mathsf{F} := h_i \rightarrow \Theta_{i+1} = \Theta_i[\Theta_i(g_i).\mathsf{F} \mapsto \Theta_i(h_i)]$$

Where $\Theta_{i+1}(\Theta_i(g_i).\mathsf{F}) = \Theta_i(h_i)$. Let Θ_0 be an empty store without assigned object fields, then $\Theta_i \times b_i \rightarrow \Theta_{i+1}$ for the i^{th} assignment b_i. Access to an unassigned object field in Θ_i causes execution of a trace to abort in an error, which we use in Lemma 2 to aid in showing soundness.

Lemma 2. For $b_i = g_i.\mathsf{F}_i := h_i$ we say that g_i and h_i are *referenced* in b_i. Given a term of the form $g_x.\mathsf{F}_x$ referenced in a non-faulting b_i, there exists a $b_j < b_i$ in the trace where $b_j = g_j.\mathsf{F}_x := h_j$, $\Theta_j(g_j) = \Theta_i(g_x)$, and $\Theta_j(h_j) = \Theta_i(g_x.\mathsf{F}_x)$.

Proof. Because b_i exists in a non-faulting trace, there must be an assignment b_j with a Θ_j where $j < i$ and $\Theta_{j+1} = \Theta_j[\Theta_i(g_x).\mathsf{F}_x \mapsto \Theta_i(g_x.\mathsf{F}_x)]$.

To show soundness in Theorem 1, we need a Υ operator to extract object allocation sites from objects in the store: $\Upsilon_i(e_a) = \mathsf{new}_a$ if $\Theta_i(e_a) = \langle \mathsf{new}_a, j \rangle$ and $\Upsilon_i(\mathcal{R}) = \mathcal{R}$; by this definition if $\Theta_i(e_a) = \Theta_j(e_b)$ then $\Upsilon_i(e_a) = \Upsilon_j(e_b)$.

Theorem 1. $\Gamma \vdash h_i :-^* \Upsilon_i(h_i)$ for a term h_i referred to in b_i and an assigned $\Theta_i(h_i)$ in a trace of Γ that does not abort before or during b_i.

Proof. h_i could either be $\Upsilon_i(h_i)$, which is then trivial, or $g_i.\mathsf{F}$ where by Lemma 2, there must be a $b_j < b_i$ of the form $b_j = g_j.\mathsf{F} := h_j$, $\Theta_j(g_j) = \Theta_i(g_i)$, and $\Theta_j(h_j) = \Theta_i(h_i)$. Because base cases are eventually reached ($h = \mathsf{new}$) and because j is **decreasing** in a finite trace, we can perform induction on i where our lemma holds for $j < i$; the proof then proceeds as:

$$\begin{aligned}
\Gamma \vdash g_j :-^* \Upsilon_j(g_j) = \Upsilon_i(g_i) \quad & \text{Inductive hyp./Lemma 2: ①} \\
\Gamma \vdash h_j :-^* \Upsilon_j(h_j) = \Upsilon_i(h_i) \quad & \text{Inductive hyp./Lemma 2: ②} \\
\Gamma \vdash \Upsilon_i(g_i).\mathsf{F} := h_j \quad & \text{Lemma 1 on ①, } b_j\text{: ③} \\
\Gamma \vdash \Upsilon_i(g_i).\mathsf{F} :-^* \Upsilon_i(h_i) \quad & \text{Transitivity on ②, ③: ④}
\end{aligned}$$

If we can show $\Gamma \vdash g_i :-^* \Upsilon_i(g_i)$, subassignment yields $\Gamma \vdash g_i.\mathsf{F} :-^* \Upsilon_i(g_i).\mathsf{F}$ and transitivity yields $\Gamma \vdash g_i.\mathsf{F} :-^* \Upsilon_i(h_i)$, proving the lemma. Our inductive hypothesis alone does not give us this premise, but g_i is either $\mathcal{R}$, in which case we have reached another base case, or is of the form $g_x.\mathsf{F}_x$, where the base case $\mathcal{R}$ is eventually reached by decomposing g_x and its parents recursively, and so we can take $\Gamma \vdash g_i :-^* \Upsilon_i(g_i)$ as a second inductive hypothesis. □

$$e_a \times e_t \downarrow \epsilon \to e_a \times e_t \qquad \frac{e_t \preceq e_b}{e_a \times e_t \downarrow [e_b \coloneqq e_g] \to (e_g + (e_a - e_b)) \times (e_g + (e_t - e_b))}$$

$$e_a \times e_t \downarrow (p_0 \circ p_1) \to (e_a \times e_t \downarrow p_0) \downarrow p_1 \qquad \frac{e_t \not\preceq e_b}{e_a \times e_t \downarrow [e_b \coloneqq e_g] \to e_a \times (e_t \bowtie e_b \bowtie e_g)}$$

$$e_a \preceq e_a \qquad e_a \preceq e_b \leftarrow e_a \prec e_b \qquad e_b.\mathsf{F} \prec e_a \leftarrow e_b \preceq e_a$$

$$e_a - e_a \to \epsilon \qquad e_b.\mathsf{F} - e_a \to \mathsf{F} + (e_b - e_a) \qquad e_a + \epsilon \to e_a \qquad e_a + (\mathsf{F} + f) \to e_a.\mathsf{F} + f$$

$$\frac{e_a \preceq e_b}{e_a \bowtie e_b \to e_b} \qquad \frac{e_b \preceq e_a}{e_a \bowtie e_b \to e_a} \qquad \frac{e_b.\mathsf{F}_b \not\preceq e_a.\mathsf{F}_a, e_a.\mathsf{F}_a \not\preceq e_b.\mathsf{F}_b}{e_a.\mathsf{F}_a \bowtie e_b.\mathsf{F}_b \to e_a \bowtie e_b} \qquad e \bowtie \mathsf{new} \to \mathcal{R}$$

Fig. 1. Definitions for term rewriting $\downarrow$, term child of $\preceq$ (strict $\prec$), term field subtraction $-$ and addition $+$, and common parent term join $\bowtie$.

Parametricity

We model traits as assignable constructs as fields hanging off $\mathcal{R}$. Consider then a definition and use of a trait $\mathcal{R}.\mathsf{T}$ in a program:

① $\mathcal{R}.\mathsf{T}.\mathsf{V} \coloneqq \mathcal{R}.\mathsf{T}.\mathsf{W}$ | ② $\mathcal{R}.\mathsf{T} \coloneqq \mathcal{R}.\mathsf{A}$ | ③ $\mathcal{R}.\mathsf{D} \coloneqq \mathcal{R}.\mathsf{A}.\mathsf{V}$

Subassignment gives us $\mathcal{R}.\mathsf{T}.\mathsf{V} :- \mathcal{R}.\mathsf{A}.\mathsf{V}$, reassignment $\mathcal{R}.\mathsf{A}.\mathsf{V} \coloneqq \mathcal{R}.\mathsf{T}.\mathsf{W}$, and transitivity $\mathcal{R}.\mathsf{D} \coloneqq \mathcal{R}.\mathsf{T}.\mathsf{W}$. The result is that term $\mathcal{R}.\mathsf{D}$ is then assigned indirectly from every term that assignable from the $\mathcal{R}.\mathsf{T}.\mathsf{W}$ field, which is way too restrictive–in particular it prevents us from usefully modeling parametric traits or methods as fields. To solve this problem, we could copy trait $\mathcal{R}.\mathsf{T}$'s implementation so that its fields, which are essentially type variables, are not shared by independent references of $\mathcal{R}.\mathsf{T}$. However, integrating disparate $\mathcal{R}.\mathsf{T}$ copies that happen to become eventually assigned to the same term would be quite messy!

Instead we make another important observation about reassignment semantics: given an assignment of the form $\mathcal{R}.\mathsf{T}.\mathsf{V} \coloneqq \mathcal{R}.\mathsf{T}.\mathsf{W}$, what object $\mathcal{R}.\mathsf{T}$ is assigned to will remain constant during execution of the entire assignment, meaning an object reference is just being moved within $\mathcal{R}.\mathsf{T}$ between its fields. An assignment is then **naturally parametric** with respect to a sub-tree that fully contains the assignment; e.g. $\mathcal{R}.\mathsf{T}$ contains $\mathcal{R}.\mathsf{T}.\mathsf{V} \coloneqq \mathcal{R}.\mathsf{T}.\mathsf{W}$. Leveraging this phenomena requires new judgment definitions, which are based on the rewriting of Fig. 1:

$$ⓐ \ \frac{\Gamma \vdash e_a \coloneqq e_f \times e_t}{\Gamma \vdash e_a :- e_f \uparrow [e_a \coloneqq e_f]} \qquad ⓑ \ \frac{\Gamma \vdash e_a :- e_f \uparrow p}{\Gamma \vdash e_a.\mathsf{F} :- e_f.\mathsf{F} \uparrow p}$$

$$ⓒ \ \frac{\Gamma \vdash e_a :- e_f \uparrow p, \quad e_a.\mathsf{F} \coloneqq e_g \times e_t}{\Gamma \vdash e_f.\mathsf{F} \coloneqq e_g \times e_t \downarrow p}$$

Each strong assignment is annotated with a *scope* parent term (e_t in $e_a \coloneqq e_f \times e_t$) that will stay constant when the assignment executes. A *rewrite path* ($\uparrow p$) of

strong assignments is also generated and propagated in the direction of assignment and the term's field subassignments via $:-$ relations ($p ::= [e := e] \mid \epsilon \mid p \circ p$). This path is used to rewrite ($\downarrow$) the assignee term on reassignment, where rewrites of the assignee via the rewrite path must occur only through the scope term as defined in Fig. 1; otherwise reassignment could miss possible object flows.

The scope assignment term of encoded program assignments is the common parent of assignment terms computed using the join ($\bowtie$) operator of Fig. 1:

$$\Gamma = \overline{g.\mathsf{F} := h} \vdash \overline{g.\mathsf{F} := h \times (g \bowtie h)}$$

The assignment in our previous example derives $\mathcal{R}.\mathsf{T}.\mathsf{V} := \mathcal{R}.\mathsf{T}.\mathsf{W} \times \mathcal{R}.\mathsf{T}$ where $\mathcal{R}.\mathsf{T}$ is the common parent. The new judgments derive $\mathcal{R}.\mathsf{A}.\mathsf{V} := \mathcal{R}.\mathsf{A}.\mathsf{W} \times \mathsf{A}$ given a rewrite path of $[\mathsf{T} := \mathsf{A}]$, preventing $\mathcal{R}.\mathsf{D}$ from being assigned through $\mathcal{R}.\mathsf{T}.\mathsf{W}$.

A scope becomes progressively weaker (i.e. closer to $\mathcal{R}$) through multiple reassignments to account for "breaks" in the rewrite path where stable evaluation of an assignment's original common parent can no longer be assumed; consider:

① $\mathcal{R}.\mathsf{T}.\mathsf{A}.\mathsf{F} := \mathcal{R}.\mathsf{T}.\mathsf{G}$ | ② $\mathcal{R}.\mathsf{T}.\mathsf{A} := \mathcal{R}.\mathsf{B}$ | ③ $\mathcal{R}.\mathsf{B} := \mathcal{R}.\mathsf{T}.\mathsf{C}$ | ④ $\mathcal{R}.\mathsf{T} := \mathcal{R}.\mathsf{S}$

On assignment ②, the rewrite scope used for reassignment of assignment ① into $\mathcal{R}.\mathsf{B}.\mathsf{F}$ has already weakened to $\mathcal{R}$, where $\mathcal{R}.\mathsf{B}.\mathsf{F} := \mathcal{R}.\mathsf{T}.\mathsf{G}$ is derived. On assignment ③, reassignment derives $\mathcal{R}.\mathsf{T}.\mathsf{C}.\mathsf{F} := \mathcal{R}.\mathsf{T}.\mathsf{G}$; however the rewrite scope is still $\mathcal{R}$, which can be fully appreciated after assignment ④, when reassignment derives $\mathcal{R}.\mathsf{S}.\mathsf{C}.\mathsf{F} := \mathcal{R}.\mathsf{T}.\mathsf{G}$! A reassignment of $\mathcal{R}.\mathsf{S}.\mathsf{C}.\mathsf{F} := \mathcal{R}.\mathsf{S}.\mathsf{G}$ would have been incorrect because of assignment ③: in an arbitrary ordering of interleaved assignments, $\mathcal{R}.\mathsf{T}$ could change what object it is assigned to after this assignment occurs, requiring a weaker scope as reassignment propagates through $\mathcal{R}.\mathsf{B}$.

To show soundness, reassignment transitivity must be shown to work with rewrite paths. Transitivity of $:-$ is defined to concat rewrite paths using the $\circ$ operator that is expanded in the definition of $\downarrow$ in Fig. 1:

$$\Gamma \vdash e :-^* e \uparrow \epsilon \circ \epsilon \qquad \frac{\Gamma \vdash e_a :-^* e_f \uparrow p_0 \circ \epsilon, \quad e_f :- e_g \uparrow p_1}{\Gamma \vdash e_a :-^* e_g \uparrow p_0 \circ p_1 \circ \epsilon}$$

Lemma 1 is redone as Lemma 3 to show the transitivity of reassignment.

Lemma 3. $$\Gamma \vdash e_a :-^* e_f \uparrow p,\ e_a.\mathsf{F} := e_g \times e_t \ \Rightarrow \ \Gamma \vdash e_f.\mathsf{F} := e_g \downarrow p \times e_t$$

Proof. With rewrite paths, this is slightly less straightforward than in Lemma 1. Starting with reassignment base cases of $e_a :- e_0 \uparrow p_0$, $e_0 :- e_1 \uparrow p_1$:

$$\Gamma \vdash e_0.\mathsf{F} := e_g \times e_t \downarrow p_0$$
$$\Gamma \vdash e_1.\mathsf{F} := (e_g \times e_t \downarrow p_0) \downarrow p_1$$

Using the concat definition in Fig. 1, we can rewrite the last derivation as:

$$\Gamma \vdash e_1.\mathsf{F} := (e_g \times e_t \downarrow p_0 \circ p_1)$$

Taking $p = p_0 \circ p_1$, the proof then proceeds easily by induction. □

$$\epsilon \times e_t \to \epsilon \qquad (\epsilon \circ p) \times e_t \to p \times e_t$$

$$\frac{e_t \preceq e_a}{([e_a \coloneqq e_f] \circ p) \times e_t \to [e_a \coloneqq e_f] \circ (p \times (e_f + (e_t - e_a)))} \qquad \frac{e_t \not\preceq e_a}{([e_a \coloneqq e_f] \circ p) \times e_t \to p \times (e_t \bowtie e_a \bowtie e_f)}$$

$$p \div e_t \to p \div e_t \,\%\, \epsilon \qquad \epsilon \div e_t \,\%\, p \to p \qquad (\epsilon \circ p_0) \div e_t \,\%\, p_1 \to p_0 \div e_t \,\%\, p_1$$

$$\frac{e_t \preceq e_a}{([e_a \coloneqq e_f] \circ p_0) \div e_t \,\%\, p_1 \to ((p_1 \downarrow [e_a \coloneqq e_f]) \circ p_0) \div (e_f + (e_t - e_a))} \qquad \frac{e_t \not\preceq e_a}{([e_a \coloneqq e_f] \circ p_0) \div e_t \,\%\, p_1 \to p_0 \div (e_t \bowtie e_a \bowtie e_b) \,\%\, (p_1 \circ [e_a \coloneqq e_f])}$$

$$\epsilon \downarrow [a] \to \epsilon \qquad p_0 \circ p_1 \downarrow [a] \to (p_0 \downarrow [a]) \circ (p_1 \downarrow [a])$$

$$[e_a \coloneqq e_f] \downarrow [e_b \coloneqq e_g] \to [e_g + (e_a - e_b) \coloneqq e_g + (e_f - e_b)]$$

Fig. 2. Definitions of rewrite path multiply ($\times$) and divide ($\div$).

To show soundness, the rewrite paths that now come along with object reachability must be reasoned about. To this end, a multiply $\times$ rewrite path operator is defined in Fig. 2 to aid in path reasoning by allowing rewrite paths to be broken up based on rewrite scopes. Rewrite path multiplication is equivalent to scoped rewriting as shown in Lemma 4.

Lemma 4. $e_a \times e_t \downarrow p = e_a \times e_a \downarrow p \times e_t$ for any e_a and e_t where $e_a \preceq e_t$.

Proof. Based directly on the definition of rewriting in Fig. 1 and rewrite path multiplication in Fig. 2, which simply filters out any edges not scoped by e_t.

Rewrite path multiplication characterizes reachability between terms and their rewritten forms as expressed in Lemma 5.

Lemma 5. $\Gamma \vdash e_a :\!-^* e_f \uparrow p,\ e_a \times e_t \downarrow p \to e_b \times e_s \Rightarrow \Gamma \vdash e_a :\!-^* e_b \downarrow p \times e_t$ for any p, e_a, and e_t, $e_a \preceq e_t$.

Proof. By Lemma 4, e_a rewritten with $p \times e_t$ is also e_b; rewrite path multiplication then allows us to move the scope from being associated with the rewrite operation to an equivalent rewrite path that can qualify $:\!-^*$ relationships.

Notice in Lemma 5 that if $e_t = e_a$, then $e_b = e_f$, and if $e_t = \mathcal{R}$, then $e_b = e_a$, which allows for *path interpolation* between e_a and e_f using varying rewrite scopes between e_a and $\mathcal{R}$. This implies that e_b can then somehow reach e_f even if it is not e_a or e_f, which is captured through rewrite path division ($\div$) also defined in Fig. 2. A rewrite path division includes any assignment edges filtered out by rewrite path multiplication, where these edges are rewritten into the space that rewrite path multiplication rewrites to, leading to Lemma 6.

Lemma 6. $p = (p \times e_t) \circ (p \div e_t)$ for any p, e_t.

Proof. Directly by definitions in Fig. 1 and Fig. 2. Equivalence here means that for all terms e_a and scopes e_s where $e_a \preceq e_s$, then $e_a \times e_s \downarrow p = e_a \times e_s \downarrow (p \times e_t) \circ (p \div e_t)$.

We can now relate term rewriting and object reachability in Lemma 7.

Lemma 7. *$\Gamma \vdash e_a :\!-^* e_f \uparrow p,\ e_a \times e_t \downarrow p \to e_b \times e_s \Rightarrow \Gamma \vdash e_a :\!-^* e_b \downarrow p \times e_t, e_b :\!-^* e_f \downarrow p \div e_t$ for any p, e_a, and e_t, $e_a \preceq e_t$.*

Proof. Directly from Lemma 5 and Lemma 6.

Soundness Theorem 1 is now redone as Theorem 2.

Theorem 2. *$\Gamma \vdash h_i :\!-^* \Upsilon_i(h_i) \uparrow p_x$ can be derived for a term h_i referred to in b_i, an assigned $\Theta_i(h_i)$ in a non-aborting i-length trace of Γ.*

Proof. Again the base case where $h_i = \Upsilon_i(h_i)$ is trivial. As in Theorem 1, we focus on the non-base case of $h_i = g_i.\mathsf{F}$ where by Lemma 2, there must be a $b_j = g_j.\mathsf{F} \coloneqq h_j$ where $\Theta_j(g_j) = \Theta_i(g_i)$ and $\Theta_j(h_j) = \Theta_i(h_i)$. We also take the same inductive hypothesis with the addition of rewriting paths:

$$\begin{aligned}
\Gamma \vdash g_j :\!-^* \Upsilon_i(g_i) \uparrow p_y &\qquad \text{Inductive hyp./Lemma 2: ①}\\
\Gamma \vdash h_j :\!-^* \Upsilon_i(h_i) \uparrow p_z &\qquad \text{Inductive hyp./Lemma 2: ②}\\
\Gamma \vdash \Upsilon_i(g_i).\mathsf{F} \coloneqq h_j \times (g_j \bowtie h_j) \downarrow p_y &\qquad \text{Lemma 3 on ① and } b_j\text{: ③}\\
\Gamma \vdash h_j \times (g_j \bowtie h_j) \downarrow p_z :\!- \Upsilon_i(h_i) \uparrow p_z \div (g_j \bowtie h_j) &\qquad \text{Lemma 7 on ②: ④}
\end{aligned}$$

To go further with this proof, we observe that $\Theta_j(g_j \bowtie h_j)$ necessarily evaluates to the same object during b_j's execution, so we take another inductive hypothesis where the rewrite paths p_y and p_z are equivalent in the scope of $g_j \bowtie h_j$.

$$\begin{aligned}
p_y \times (g_j \bowtie h_j) = p_z \times (g_j \bowtie h_j) &\qquad \text{Inductive hyp.: ⑤}\\
\Gamma \vdash \Upsilon_i(g_i).\mathsf{F} \coloneqq h_j \times (g_j \bowtie h_j) \downarrow p_z &\qquad \text{Replacement from ③ using ⑤: ⑥}
\end{aligned}$$

Finally, by Lemma 7 and transitivity on ④ and ⑥, we get this crazy judgment:

$$\Gamma \vdash \Upsilon_i(g_i).\mathsf{F} :\!-^* \Upsilon_i(h_i) \uparrow [\Upsilon_i(g_i).\mathsf{F} \coloneqq h_j \downarrow p_z \times (g_j \bowtie h_j)] \circ p_z \div (g_j \bowtie h_j)$$

As in Theorem 1, the proof then finishes where we perform induction around the depth of g_i with $g_i = \mathcal{R}$ as a base case:

$$\begin{aligned}
&\Gamma \vdash \mathcal{R} :\!-^* \Upsilon_i(\mathcal{R}) \uparrow \epsilon\\
&\Gamma \vdash \mathcal{R}.\mathsf{F} :\!-^* \Upsilon_i(h_i) \uparrow [(\Upsilon_i(g_i) \to \mathcal{R}).\mathsf{F} \coloneqq h_j \downarrow p_z \times (g_j \bowtie h_j)] \circ p_z \div (g_j \bowtie h_j)
\end{aligned}$$

Now generalize for depths of 1 ($\mathcal{R}.\mathsf{F}_1$), 2 ($\mathcal{R}.\mathsf{F}_1.\mathsf{F}_2$), and so on, where

$$p_k = [\Upsilon_i(h_{k-1}).\mathsf{F}_k \coloneqq h_{j.k} \downarrow p_{z.k} \times (g_{j.k} \bowtie h_{j.k})] \circ p_{z.k} \div (g_{j.k} \bowtie h_{j.k})$$

for a k-depth term $\mathcal{R}...\mathsf{F}_k$ referenced in b_i and a b_k in the trace that assigns it; note that $\Upsilon_i(h_0) = \mathcal{R}$:

$$\Gamma \vdash \mathcal{R} :-^* \Upsilon_i(\mathcal{R}) \uparrow \epsilon$$
$$\Gamma \vdash \mathcal{R}.\mathsf{F}_1 :-^* \Upsilon_i(h_1) \uparrow p_1$$
$$\Gamma \vdash \mathcal{R}.\mathsf{F}_1.\mathsf{F}_2 :-^* \Upsilon_i(h_2) \uparrow p_1 \circ p_2$$
$$\Gamma \vdash \mathcal{R}.\mathsf{F}_1.\mathsf{F}_2...\mathsf{F}_k :-^* \Upsilon_i(h_k) \uparrow p_1 \circ p_2 \circ ... \circ p_k$$

To satisfy the new inductive hypothesis, we must show that all these paths are equal with respect to scope ($g_x \bowtie h_x$) if $b_i = g_x.\mathsf{F} := h_x$. It suffices to show that any successive path concatenation $k+1$ does not interfere with the path for k given a scope of $\mathcal{R}...\mathsf{F}_k$. Given that the first part of p_k is an assignment from an object term ($\Upsilon_i(h_{k-1}).\mathsf{F}_k$), the rewrite path becomes broken for any non-object terms hanging off $\mathcal{R}$, showing non interference. □

Using this treatment of assignment, traits can be polymorphic with respect to their fields and members, which achieves parametric polymorphism without explicit type variables; i.e. **all fields are effectively open type variables**.

3 YinYang

Our treatment of assignment and fields pays off in that all of YinYang's typing properties can be modeled in terms of assignment and fields, which would otherwise be difficult in other field sensitive anlayses (e.g. [36]):

- Traits are fields of root object $\mathcal{R}$;
- this term of a trait is the trait itself (i.e. the field that model's the trait);
- actual trait fields are fields of the trait;
- trait methods are fields of the trait;
- method arguments and return values are fields of the method; and
- method local variables are fields of the method.

All field accesses must be safe; e.g. accessing a trait's methods or fields in a term requires that the term be assigned to that trait either directly or indirectly. We achieve this firstly by forgoing name overloading, which is an unsurprising tradeoff in global type inference approaches; e.g. it is restricted in Haskell type classes [9]. In cases where a term's name is not obviously scoped (local or private variables), the IDE can help by resolving ambiguous short names into longer unique ones through type feedback or code completion [16]. A method call simply assigns to the arguments of a method field and assigns from the method's return value; other more primitive language features are implemented in the same straightforward way.

Additional assignments in YinYang are derived according to Section 2. By itself, this approach lacks modularity, scaling, and is not even decidable—we show how how assignment relationships can be feasibly derived in Section 4. However, we must first do something typeful with these assignments.

Going Against the Flow of Assignment

A term X's type X_τ in YinYang consists of a set of traits that are extended by each object bindable to A. If $\mathsf{Y}_\delta \subseteq \mathsf{X}_\delta$, then the traits extended by all objects in X are then extended by all objects in Y, or $\mathsf{X} :\!\!- \mathsf{Y} \Rightarrow \mathsf{Y}_\delta \subseteq \mathsf{X}_\delta \Rightarrow \mathsf{X}_\tau \subseteq \mathsf{Y}_\tau$. As in Agesen's [1] work on using constraints in type inference, a program then consists of multiple constraints derived from assignments that must be solved; consider:

Animal := Dog Cat := C A := B
Animal := Cat Dog := B A := C

The most conventional way to solve the $\mathsf{A}_\tau \subseteq \mathsf{B}_\tau$ and $\mathsf{A}_\tau \subseteq \mathsf{C}_\tau$ constraints generated in this code involves restricting the type of the assigned term; e.g. $\mathsf{A}_\tau \leftarrow \mathsf{B}_\tau \cap \mathsf{C}_\tau$. If B_τ is (Dog, Animal) and C_τ is (Cat, Animal), then A_τ must be (Animal). The advantage of this approach is that the type of A is driven by what is assigned to it; programmer intent is assumed to originate from the definitions of B and C where uses of A that violate these definitions is considered bad. Solving constraints in this direction, however, has two problems:

- Objects are allocated in the direction of assignees so it is not suitable for our goal of inferring object definitions based on usage.
- Assignments from traits to their extending terms must be special cased in our treatment given that their extensions should remain fixed.

Alternatively, a constraint of the form $\mathsf{A}_\tau \subseteq \mathsf{B}_\tau$ could be solved **by expanding what traits are required for** B_τ rather than removing what traits are provided by A_τ. In doing so, objects are defined based on usage and assignments from traits do not need to be special cased. Expanding extension sets can also be easily modeled as reachability by promoting transitive trait assignments:

$$\frac{\Gamma \vdash \mathsf{T} := \mathsf{e}_a, \mathsf{e}_a :\!\!- \mathsf{e}_b}{\Gamma \vdash \mathsf{T} := \mathsf{e}_b} \qquad \frac{\Gamma \vdash \mathsf{T} := \mathsf{e}}{\Gamma \vdash \mathsf{T} \in \mathsf{e}_\tau}$$

The type of e, e_τ, is then whatever traits it is assigned to. In our example, A has an empty type because it is not used as an assignee. Adding the code Wild := A to the program would then cause A_τ, as well as B_τ and C_τ, to include Wild.

Going backwards alone, however, provides no way of checking programmer intent because missing traits are simply added as needed. This can be quite dangerous: whereas duck typing at least rejects the usage of objects as ducks if they lack quack methods, YinYang simply infers that an object is a duck when it is asked to quack! We temper this and other dangers of missing programmer intent with three checked safety rules:

- Certain kinds of trait combinations are disallowed as nonsensical: either their value-type layouts are incompatible (e.g. Duck and Int traits), or they implement the same abstract methods (e.g. UI Button and Slider traits).
- Types of object allocations must be fully inferred within their defining traits; otherwise an object could be instantiated with traits that the programmer of a trait was not aware of.

– Object allocations must be concrete without any unimplemented abstract methods; e.g. an object cannot extend a UI Widget trait without also extending trait like Button that implements its abstract methods.

Unintended extension can still occur, but at least the object remains sensible.

The Language

Examples are now shown of our type inference system at work; consider:

```
trait Duck:
  def Quack(): ...
```

```
var a := new   # allocate Duck
a.Quack()      # Duck := a
```

YinYang is based on Python-like syntax [30] where indentation determines block structure; we precede indented lines with lightly shaded bars that indicate nesting depth. The # character precedes Python-like comments in YinYang that we use to document derived assignments. A "new" expression creates a new object whose implementation of trait extensions will be inferred; e.g. the new object assigned to the a variable extends Duck because Quack can be called on it.

As mixins, traits support fine-grained object modularizations; consider:

```
trait Bordered:
  var Thickness
trait Button:
  def Pressed(): ...
```

```
var x := new              # allocate Button, Bordered
if x.Pressed():           # Button := x
  x.Thickness := 10       # Bordered := x
```

Propagation in YinYang is control flow insensitive: only the assignment topology is relevant and condition or assignment order are ignored. The object assigned to the x variable extends the Button trait because the Pressed method is called on it, while it extends the Bordered trait because its Thickness field is assigned. With inference, programmers can ignore the traits they are composing and focus on members discovered through code completion menus [16].

```
trait Panel:
  # Panel.AddP.v.ParentW := Panel
  def AddP(v): v.ParentW := this
  abstract def LayoutP()
trait Canvas:
  implement def LayoutP(): ...
  def MovedC(v, q): ...
trait Dock:
  implement def LayoutP(): ...
  def InvalidateD(v): ...
```

```
trait Widget:
  var ParentW
trait WidgetInC:
  def SetPos(q):
    # Widget := WidgetInC, Canvas := WidgetInC.ParentW
    this.ParentW.MovedC(this, q)
trait WidgetInD:
  def ShoveLeft():
    # Widget := WidgetInD, Dock := WidgetInD.ParentW
    this.ParentW.InvalidateD(this)
```

Fig. 3. Traits that encode UI widget/panel relationships.

Programs consist of multiple objects whose collective structure can determine the capabilities of each; e.g. a Windows UI [20] FrameworkElement object can be manually position when it is in a Canvas. However, no type relationship in C#

exists between an element object and the Panel object that contains it: setting an object position that is not contained in a Canvas object is silently ignored! YinYang improves on this with reasoning over graphs of objects; consider:

```
1: var p := new, w := new
[conflict: Canvas cannot be Dock]
2: p.AddP(w)          # Panel := p, p.AddP.v := w  ⇒  w.ParentW := p
3: w.SetPos((10,10))  # WidgetInC := w  ⇒  Canvas := w.ParentW  ⇒  Canvas := p
4: w.ShoveLeft()      # WidgetInD := w  ⇒  Dock  := w.ParentW  ⇒  Dock  := p
```

This example uses UI traits defined in Fig. 3. Variables w and p respectively are assigned from (i.e. extend) the Widget and Panel traits by the AddP call. The second line AddP call also causes reassignment of p to w.ParentW, where the Panel term, which models this in Panel, is rewritten as p and Panel.AddP.v is rewritten as w in the assignment made by AddP's implementation in Figure 3 according to the rules in Section 2. The third line SetPos call causes Canvas to be assigned to w.ParentW through reassignment, which is then propagated through p to the object allocation assigned to it. Likewise, the fourth line ShoveLeft call causes Dock to be assigned to w.ParentW through the p variable.

A type compatibility error is generated on the first line of our example when p is assigned to a new object: [conflict: Canvas cannot be Dock]. This object is inferred to extend both Canvas and Dock, but these traits are incompatible because they both implement Panel's LayoutP method. Unlike Scala [22], methods in YinYang can only be implemented once per object even if they can still be overridden multiple times, which avoids non-nonsensical trait definitions. The error can be removed by eliminating the ShoveLeft call on the last line, causing the object assigned to p to be just a Canvas.

```
trait Animal:
  abstract def EatA(f)
  abstract def SoundA()

trait Cow:
  def Moo(): ...
  implement def EatA(f):
    f.MowG()  # Grass := Cow.EatA.f
  implement def SoundA(): this.Moo()

trait Grass:
  def MowG(): ...
  implement def Consumed(): ...
```

```
trait Food:
  abstract def Consumed()

trait Duck:
  def Quack(): ...
  implement def EatA(f):
    f.SquirmW()  # Worm := Duck.EatA.f
  implement def SoundA(): this.Quack()

trait Worm:
  def SquirmW(): ...
  implement def Consumed(): ...
```

Fig. 4. Traits encode animals and food.

As another example, consider the traits in Fig. 4. The Animal trait defines an abstract method EatA with one argument f that is otherwise completely unspecified. The Cow and Duck traits then implement the EatA method with more specific requirements on its f argument–it must be Grass for Cow and Worm for Duck–which is achieved simply by assigning the argument as a field to either trait; e.g. Worm :=Duck.EatA.f. Typically, object-oriented languages do not allow

such craziness: method arguments should remain invariant, or at least become weaker, not stronger! But such "polymorphic catcalls" [17] are not a problem in YinYang, which deals elegantly with covariance; consider:

```
var d := new, c := new
d.Quack() # Duck := d
c.Moo()   # Cow := c
var a := d
if ...: a := c
```

```
a.SoundA()  # Animal := a
# a.EatA.f := new, by reassignment: d.EatA.f := new
#                                   c.EatA.f := new
a.EatA(new)
[conflict: Worm cannot be Grass]
```

Variable a is assigned to either d (a Duck) or c (a Cow) according to some unknown condition. The Animal SoundA method can then be called on a without problem, but EatA cannot be called on a because its argument, an object that is both Worm and Grass, cannot exist. This is because the reassignment judgment fires on the EatA call, which is modeled as the assignment a.EatA.f := new, causing assignments of that new term to d.EatA.f and c.EatA.f, creating a situation where both Worm and Grass propagate to the new object.

```
trait Map:
  abstract def GetM(k)
  abstract def PutM(k,e)
```

```
trait HashMap:
  var Tbl := new(LenA = 100)
  implement def GetM(k):
    # HashMap.GetM.RET := HashMap.Tbl.F
    return this.Tbl[k.Hash % this.Tbl.CountA]
  implement def PutM(k,e):
    # HashMap.Tbl.F := HashMap.PutM.e
    this.Tbl[k.Hash % this.Tbl.CountA] = e
```

Fig. 5. An abstract Map trait and a simple HashMap trait that implements it.

As seen in the last example, method implementation can express new assignments missing in the original abstract method; this also applies to method overriding. Calls to an abstract method, which is modeled as just field assignments, eventually become integrated with the assignments of the implementing methods through reassignment. For example, consider a generic Map trait and its HashMap trait implementation in Fig. 3. Map is completely unannotated, delaying any type feedback about a Map object until it is known to be a HashMap or some other more concrete variant; consider:

```
trait Sample:
  # Sample.DoitA.m.PutM.e := Sample.DoitA.g
  # Duck := Sample.DoitA.g
  # Cow := Sample.DoitA.n.GetM.RET
  def DoitA(n, g):
    g.Quack()
    n.PutM(42, g)
    n.GetM(42).Moo()
```

```
var s := new, m := new, h := new
s.DoitA(m, h) # s.DoitA.n := m, s.DoitA.g := h
[conflict: Duck cannot be Cow]
m is HashMap # HashMap := m
```

In this example, the DoitA method of trait Sample manipulates an argument n that is known to extend Map, but the specific implementation of Map is left open. When DoitA is called in the right column of the example, its n is assigned

to m, which is known to be a HashMap; reassignment then integrates HashMap's assignments with those of n. A conflict then occurs on h in the call, which, being assigned to g of DoitA, must then be both a Duck and Cow.

The Map trait in Fig. 3 is abstract because it has abstract methods. As a result Map cannot be used alone; consider:

```
var m := new
[abstract Map]
m.PutM(100, ''hello'') # Map := m
```

Knowing that the new object is required to extend Map is not enough to instantiate the object, and a more specific type is needed, which is challenging if the implementation is not distinguished by its own set of unique methods. Instead, this error can be fixed by writing m is HashMap to explicitly introduce a HashMap trait requirement. Even if inference is unable to completely define the object in this case, the programmer is provided with feedback, an abstract instantiation error, that is useful in completing the definition.

Limiting Inference. As mentioned previously, objects must be completely defined in the traits that create them given that object cannot be modified by the programmer to account for additional constructor parameters or abstract methods introduced by required traits they do not know about; consider:

```
trait Sample:
  def DoitB():
    var d = new
    d.Quack()
    return d

trait Wild:
  DoWildThing(): ...
```

```
var s := new
var a := s.DoitB()
a.DoWildThing()
[frozen: cannot be Wild]
```

In this example, a Duck object is created and returned by the Sample's trait DoitB method. A client calling DoitB then tries to also have that object extend the Wild trait, which results in an error. Detecting such constructions presents an additional challenge to modularity that is discussed in Section 4.

Parametricity Revisited. Traits in YinYang are automatically parametric with the judgments of Section 2. All methods in this Section have been non-parametric as they have been modeled as direct assignments into the target object. Methods in YinYang can be parametric with an instantiation step that involves creating a new field to call the method from, assigning this field to the called method to account for overriding correctly. Parametric reassignment then does the rest of the work as it does for traits. Unfortunately, doing this creates more fields whose assignments can be costly to integrate in the next Section.

4 Feasibility

Section 3 demonstrated the usefulness of our type system, but is it really feasible? There are reasons to be pessimistic: the assignment/field judgments of

Section 2 is basically undecidable, seems to examine the whole program; and is intrinsically polyvariant, which is often not scalable. However, our analysis need only compute trait extension requirements **for terms actually referenced by the programmer**, meaning the analysis does not need to be exhaustive on all potential terms, which can be infinitely many via fields and recursion. Also, assignment connectivity for a trait is often routed through a few fields. This section shows how these properties can be leveraged to design a feasible system. Note that with a complete assignment graph, a very basic algorithm can be used to propagate trait extension requirements as specified in Section 3; this section then focuses on how such an assignment graph can be computed efficiently.

Given that only terms referenced by the programmer are of concern, we start by "skipping" field accesses unreferenced in the program; consider:

① A ≔ B ⎪ ③ B ≔ C ⎪ ② A.F ≔ D ⎪ ④ E ≔ C.F

The term B "inherits" field F from being assigned to A but B.F is not referred to in the program. In this case, subassignment and reassignment skip B.F to derive A.F :– C.F and C.F ≔ D for a term C.F that is referenced by the programmer. By skipping, the decidability problem is avoided given that the analysis becomes more demand based. Field skipping can be expressed as a modification to subassignment defined in Section 2:

$$\text{ⓐ}\ \frac{\exists(e_a.\mathsf{F})}{\Gamma \vdash e_a.\mathsf{F} :\approx e_a \uparrow \epsilon} \qquad \text{ⓑ}\ \frac{\Gamma \vdash\ e_a.\mathsf{F} :\approx e_b \uparrow p_0, \quad e_b :\!- e_c \uparrow p_1, \exists(e_c.\mathsf{F})}{\Gamma \vdash e_a.\mathsf{F} :\!- e_c.\mathsf{F} \uparrow p_0 \circ p_1}$$

$$\text{ⓒ}\ \frac{\Gamma \vdash\ e_a.\mathsf{F} :\approx e_b \uparrow p_0, \quad e_b :\!- e_c \uparrow p_1, \nexists(e_c.\mathsf{F})}{\Gamma \vdash e_a.\mathsf{F} :\approx e_c \uparrow p_0 \circ p_1}$$

The $:\approx$ operator propagates skipped field accesses, while $\exists(e.\mathsf{F})$ is true if the term $e.\mathsf{F}$ is referenced in either an explicit or derived assignment; subassignment can then be defined in terms of potentially skipped fields. A new definition of reassignment is also needed to handle skipped fields:

$$\text{ⓐ}\ \frac{\begin{array}{r}\Gamma \vdash\ e_a.\mathsf{F} \coloneqq e_g \times e_t,\\ e_a.\mathsf{F} :\approx e_b \uparrow p_0,\\ e_b :\!- e_c \uparrow p_1,\ \exists(e_c.\mathsf{F})\end{array}}{\Gamma \vdash e_c.\mathsf{F} \coloneqq e_g \times e_t \downarrow p_0 \circ p_1} \qquad \text{ⓑ}\ \frac{\begin{array}{r}\Gamma \vdash\ e_a.\mathsf{F} \coloneqq e_g \times e_t,\\ e_a.\mathsf{F} :\approx e_b \uparrow p_0, \quad e_b :\!- e_c \uparrow p_1,\\ e_g \times e_t \downarrow p_0 \circ p_1 \to e_h \times e_s,\ \exists(e_h)\end{array}}{\Gamma \vdash e_c.\mathsf{F} \coloneqq e_h \times e_s}$$

When a term is referenced, its assignment connectivity with other "materialized" terms, i.e. those that have not been skipped, must be revealed to allow for correct reasoning about object flow in the analysis. Judgment ⓐ reveals reassignments from the perspective of an assignment, while judgment ⓑ applies reassignment from the perspective of an existing rewritten assignee to ensure incoming assignment edges are revealed.

Nice to Meet You. Field skipping comes with more complexity in ensuring all assignment relationships are reproduced as if no terms were skipped. Consider the example in Fig. 6: reassignment would derive that x.value ≔ y; even

```
trait NLink:
  var next, value
trait TLink:
  def DoT():
    value.Quack()     # Duck := TLink.value
    if ...: next.DoS() # SLink := TLink.next
trait SLink:
  def DoS():
    value.Moo()       # Cow := SLink.value
    next.DoT()        # TLink := SLink.next
```

```
var x := new
x.DoT()    # TLink := x
var i := x
while i ≠ null:
  var y := new
  i.value = y
  [conflict: Duck cannot be Cow]
  i := i.next
```

Fig. 6. YinYang code where terms meeting causes lots of trouble.

though x.value is missing, an indirect weak assignment TLink.value :−* y must still be derived to account for the flow of the reassignment. Even more tricky, reassignment derives that i.next.value := y, and field subassignment would derive SLink.value :− i.next.value, leading to Cow :−* y and causing a conflict with Duck on y. Considering recursion, it is very important to avoid "materializing terms," i.e. creating unskipped terms, like i.next.value as doing so would beg the question of when to stop. In contrast, computing a fix-point of indirect assignments like SLink.value :−* y is decidable. We incorporate these indirect assignments into our judgments with a "meet" $\rightleftharpoons$ operator that detects when terms would meet at skipped fields to derive the type error in Fig. 6:

$$\text{ⓐ}\ \frac{\Gamma \vdash\ e_a.\mathsf{F} :\approx e_c \uparrow p_0,\quad e_b.\mathsf{F} :\approx e_c \uparrow p_1,\quad \nexists(e_c.\mathsf{F})}{\Gamma \vdash\ e_a.\mathsf{F} \times e_a \diamond p_0 \rightleftharpoons e_b.\mathsf{F} \times e_b \diamond p_1} \qquad \text{ⓑ}\ \frac{\Gamma \vdash\ e_a \times e_c \diamond p_0 \rightleftharpoons e_b \times e_d \diamond p_1,\quad e_f.\mathsf{F} :\approx e_a \uparrow p_2,\quad e_g.\mathsf{F} :\approx e_b \uparrow p_3,}{\Gamma \vdash\ e_f.\mathsf{F} \times e_c \diamond p_2 \circ p_0 \rightleftharpoons e_g.\mathsf{F} \times e_d \diamond p_3 \circ p_1}$$

$$\text{ⓒ}\ \frac{\Gamma \vdash\ e_a \times e_c \diamond p_0 \rightleftharpoons e_b \times e_d \diamond p_1,\quad e_d := e_f \times e_t,\quad e_a \preceq e_d,\ e_t \preceq e_c}{\Gamma \vdash\ (e_f + (e_a - e_d)) \times e_c \diamond [e_f := e_d] \circ p_0 \rightleftharpoons e_b \times e_d \diamond p_1}$$

$$\text{ⓓ}\ \frac{\Gamma \vdash\ e_a \times e_c \diamond p_0 \rightleftharpoons e_b \times e_d \diamond p_1}{\Gamma \vdash\ e_a \times e_c \diamond p_0 \rightleftharpoons^* e_b \times e_d \diamond p_1,\quad e_b \times e_d \diamond p_1 \rightleftharpoons^* e_a \times e_c \diamond p_0}$$

A meeting begins in ⓐ when two terms of the same field meet at some term that skips this field. Meeting is transitive in two ways. First in ⓑ, if two skipped terms of the same field meet at some term, then recursively their fields and skipped fields also meet as well. Second in ⓒ, an assignment that can rewrite the meet term (e_a) under the scope of the meet (e_c) is followed to create a new meet relationship since, due to reassignment, these terms would actually meet. Finally, $\rightleftharpoons$ is made reflexive in ⓓ in a way after transitivity is applied. A scope and rewrite path is included in each side of a $\rightleftharpoons$ relation to accommodate indirect

reassignments derived as:

$$\frac{\Gamma \vdash\ e_a.\mathsf{F} \times e_c \diamond p_0 \rightleftharpoons^* e_b.\mathsf{F} \times e_d \diamond p_1, \quad e_a.\mathsf{F} \coloneqq e_g \times e_t, \quad e_t \not\preceq e_c, \\ e_g \times e_t \downarrow p_0 \to e_h \times e_s, \qquad e_b \times e_b \downarrow p_1 \to e_f \times e_f}{\Gamma \vdash e_b.\mathsf{F} :\!- e_h \uparrow p_1 \circ [e_f.\mathsf{F} \coloneqq e_h]}$$

If the scope of the assignment e_t is a child of e_a's scope e_c in the meeting, then rewriting e_g would result in a node internal to the term of meeting, where reassignment would not provide any useful information to the analysis. If, however, e_t is not under e_c, then a derived reassignment is meaningful with respect to the assignment connectivity of $e_b.\mathsf{F}$ and must be derived, which is necessarily weak because the skipped term would be assigned to $e_b.\mathsf{F}$ weakly.

Given that $\rightleftharpoons$ is recursive, it might appear that resolving these indirect reassignments is undecidable. However, only a finite number of $\rightleftharpoons$ relationships can be derived given that they can only relate a finite number of materialized terms: a recursive meet derivation will then eventually encounter terms that have already met as a termination condition.

Frozen Objects. As mentioned in Section 3, an object's set of extended traits is fixed within the trait that defines it. This presents a technical challenge for type checking: while compatibility can be checked on demand as terms are referenced by the programmer, it is possible for an object to pick up additionally traits indirectly as skipped fields meet; consider:

```
trait Base:
  var value
trait Ext1:
  def DoT():
    value := new  # allocate Duck
    value.Quack() # Duck := value
trait Ext2:
  def DoS():
    value.DoWildThing() # Wild := value
```

```
var x := new
x.DoT()   # Ext1 := x
x.DoS()   # Ext2 := x
[frozen: x.value cannot extend Wild]
```

In the right column of this code, the value field for Ext1 and Ext2 meet at x, which by the previous judgment derives Ext2.value :– new, where new is the object created in Ext1. At this point, the Wild trait then propagates to new and an error is detected; the challenge is then to reformat the detected error in terms that the programmer can reason about; e.g. by referring to x.value when the Ext2 assignment is inferred.

Scalability

Even with our modified judgments for skipping fields, recursion can still induce the materialization of an infinite number of terms; consider:

```
trait TDef:
  var valA, next
  def DoitC():
    valA := next.valA # TDef.next := TDef, TDef.valA := TDef.next.valA
```

$$\text{ⓐ}\ \frac{\Gamma \vdash e_a \coloneqq e_b \times e_t, \quad |e_b| < |e_a|}{\Gamma \vdash e_a \rightsquigarrow e_b \times e_t \diamond \epsilon} \qquad \text{ⓑ}\ \frac{\Gamma \vdash\ e_a \rightsquigarrow e_b \times e_t \diamond p,\ e_c \coloneqq e_d \times e_s, \quad |e_c| = |e_d|,\ e_b \preceq e_c \prec e_t}{\Gamma \vdash e_a \rightsquigarrow (e_b \times e_t \downarrow [e_c \coloneqq e_d]) \bowtie e_s \diamond p}$$

$$\text{ⓒ}\ \frac{\begin{array}{l}\Gamma \vdash\ e_a \rightsquigarrow e_b \times e_t \diamond p,\\ \quad e_c \coloneqq e_d \times e_s,\ |e_c| < |e_d|,\\ \quad e_a \times e_t \downarrow p \to e_f \times e_r,\ e_b \preceq e_c \prec e_t\end{array}}{\Gamma \vdash e_f \coloneqq ((e_b \times e_t \downarrow [e_c \coloneqq e_d]) \bowtie e_s) \downarrow p} \qquad \text{ⓓ}\ (e_a \times e_t) \bowtie e_s \to e_a \times (e_t \bowtie e_s)$$

Fig. 7. Judgments for promoting shallow-deep assignments via $\rightsquigarrow$ relations.

By the reassignment judgments presented for field skipping, the assignment TDef.next.valA $\coloneqq$ TDef.next.next.valA would be derived, followed by TDef.next.next.valA $\coloneqq$ TDef.next.next.next.valA, and so on!

Beyond decidability problems, our analysis is still not very scalable because trait fields are busy nexuses of assignments with the trait's methods' arguments, return values, and local variables. Accordingly, referencing one field of a method would likely materialize a trait field and then the fields of many other methods, which is quite expensive for each term. We deal with this problem by exploiting depth in the assignment graph, expressed using $=$, $<$, and $\leq$:

$$\begin{array}{lll} |e_a| \leq |e_b| \leftarrow |e_a| = |e_b| & & |e_a.\mathsf{F}_0| = |e_a.\mathsf{F}_1| \\ |e_a| \leq |e_b| \leftarrow |e_a| < |e_b| & & |e_a.\mathsf{F}_0| = |e_b.\mathsf{F}_1| \leftarrow |e_a| = |e_b| \\ |e_a.\mathsf{F}| < |\mathcal{R}| & |\mathcal{R}| = |\mathcal{R}| & |e_a.\mathsf{F}_0| < |e_b| \leftarrow |e_a| \leq |e_b| \end{array}$$

If $|e_a| < |e_b|$, we say that e_a is *deeper* than e_b and, conversely, e_b is *shallower* than e_a. Our strategy is to avoid deriving reassignments to/from shallower terms, achieving decidability and bypassing shallower terms of less interest. The assignment chain is only followed up and over, but not down into deeper terms; reassignment is then redefined as:

$$\frac{\begin{array}{l}\Gamma \vdash\ e_a.\mathsf{F} \coloneqq e_g \times e_t,\ |e_a.\mathsf{F}| \leq |e_g|\\ \quad e_a.\mathsf{F} :\approx e_b \uparrow p_0,\\ \quad e_b :- e_c \uparrow p_1,\ \exists(e_c.\mathsf{F})\end{array}}{\Gamma \vdash e_c.\mathsf{F} \coloneqq e_g \times e_t \downarrow p_0 \circ p_1} \qquad \frac{\begin{array}{l}\Gamma \vdash\ e_a.\mathsf{F} \coloneqq e_g \times e_t,\ |e_g| \leq |e_a.\mathsf{F}|\\ \quad e_a.\mathsf{F} :\approx e_b \uparrow p_0,\ e_b :- e_c \uparrow p_1,\\ \exists(e_h),\ e_g \times e_t \downarrow p_0 \circ p_1 \to e_h \times e_s\end{array}}{\Gamma \vdash e_c.\mathsf{F} \coloneqq e_h \times e_s}$$

In other words, reassignment never causes the term graph to become deeper. Assignments between shallower terms via deeper terms must then be *promoted* so they are not missed, which we facilitate with a $\rightsquigarrow$ propagation operator, defined in Fig. 7. Strong assignments from shallower to deeper terms are transformed by ⓐ into $\rightsquigarrow$ relations that are propagated through same-depth assignments in ⓑ. When a $\rightsquigarrow$ relation hits an assignment from a deeper to a shallower term, it is transformed by ⓒ into a strong assignment that bypasses the deeper terms. Note that $\bowtie$ is also augmented in ⓓ to operate on assignment scopes directly for

brevity in expressing ⓑ and ⓒ. Also, for judgments ⓑ and ⓒ, only assignments under the rewrite scope of the $\rightsquigarrow$ relation are considered; other assignments are handled through $\rightleftharpoons$ relations as we talk about next.

Each $\rightsquigarrow$ relation is associated with a rewrite path after $\diamond$ that is initially ϵ by ⓐ and is used by ⓒ to rewrite the resulting assignment, which if ϵ is a no-op by Fig. 1. This rewrite path is used to integrate the $\rightsquigarrow$ relations of distributed sub-graphs that meet at some term:

$$\frac{\begin{array}{ll}\Gamma \vdash & e_a \times e_c \diamond p_0 \circ p_1 \rightleftharpoons^* e_b \times e_d \diamond p_2 \circ p_3, \quad e_f \rightsquigarrow e_a \times e_t \diamond p_0, \\ & e_f \times e_t \downarrow p_0 \circ p_1 = e_h \times e_s \downarrow p_2 \circ p_3, \quad e_t \preceq e_c, \quad p_3 \neq \epsilon \\ \Gamma \vdash & e_h \not\rightsquigarrow e_b \times e_s \diamond p_2 \end{array}}{\Gamma \vdash e_h \rightsquigarrow e_b \times e_s \diamond p_2 \circ p_3}$$

The $\rightleftharpoons$ relation, as defined earlier in this Section, reveals where skipped terms meet as the same term is assigned to their parents. We use it in this judgment again to "transfer" $\rightsquigarrow$ relations between different assignment subgraphs so that the search for a shallower assignment is not limited to one sub-graph. Transfer of a $\rightsquigarrow$ relation is contingent on its rewrite scope (e_t) being or being under ($\preceq$) the rewrite scope of the meet-source subgraph (e_c), in contrast to judgments ⓑ and ⓒ of Fig. 7, where this must not be the case. Additionally, the rewrite path of the $\rightsquigarrow$ relation (p_0) must begin the source subgraph's own rewrite path ($p_0 \circ p_1$): given that e_a could be involved in multiple meetings, this prevents $\rightsquigarrow$ relations from accidentally being transferred transitively to sub-graphs that they have no relationship with. Finally, their must not already be a $\rightsquigarrow$ relation of a shorter rewrite path (like $p_2 = \epsilon$) in the destination sub-graph, which is important to the efficiency of our analysis as $\rightsquigarrow$ relations that do not add new information to the destination sub-graph are not propagated.

When a $\rightsquigarrow$ relation is transferred to the other subgraph, it is rewritten in the terms of that sub-graph. We use the $\downarrow$ definition to translate terms from a rewrite path of the source subgraph ($p_0 \circ p_1$) to the equivalent terms in the destination subgraph whose rewrite path is $p_2 \circ p_3$. Finally, the rewrite path of the $\rightsquigarrow$ relation is simply that of the destination sub-graph ($p_2 \circ p_3$).

As an example of how this judgment works with those in Fig. 7, consider:

① U.A ≔ U.B.C	③ F.B ≔ F.D	⑤ F ≔ G	⑦ K ≔ G.A
② H.D.C ≔ H.E	④ H ≔ G	⑥ U ≔ G	

In this example, the relation U.A $\rightsquigarrow$ U.B.C $\times$ U $\diamond \epsilon$ is transferred to F.A $\rightsquigarrow$ F.B.C $\times$ F $\diamond$ [F ≔ G] given the meeting of U.B and F.B at G. Then by the second judgment of Fig. 7, F.A $\rightsquigarrow$ F.D.C $\times$ F $\diamond$ [F ≔ G] is derived, which is transferred to H.A $\rightsquigarrow$ H.D.C $\times$ H $\diamond$ [H ≔ G] given the meeting of F.D and H.D at G. Finally, G.A ≔ G.E $\times$ G is derived by the third judgment in Fig. 7, which is rewritten by the path [H ≔ G]. As a result, K is assigned to G.E indirectly through G.A, where this assignment is detected by our analysis.

The term-height analysis technique presented here allows for further optimization. One of the most expensive operations during type checking is integrating two independently developed traits T_1 and T_2 that extend a common

super trait T_0 in different ways. These traits can then be pre-met in a new empty trait T_{12} such that $\mathsf{T}_1 \coloneqq \mathsf{T}_{12}$ and $\mathsf{T}_2 \coloneqq \mathsf{T}_{12}$ where an assignment from T_1 or T_2 is replaced with an assignment from T_{12} when the other trait is already know to be extended. The benefit is that the "meet" computations above are then computed only once for every case where T_1 and T_2 are used together. Additionally, this technique allows for us to deal with modularity in a very nice way: we can make "more private" terms deeper than "less private" ones so assignments between them are automatically skipped for reassignment.

Complexity. Our analysis is then at worst logarithmic with respect to the number of assignments for each term leading to a $n\log(n)$ complexity of the trait propagation graph; n being the number of assignments in Γ. However, this complexity depends on a balanced/deep tree of fields, and does not include the overhead needed to skip deeper terms as presented next. Term tree depth arises naturally by construction, i.e. by placing method local variables deeper than method arguments, which are deeper than trait fields. Better yet, deep, possibly recursive, call chains and object compositions, which analyses typically have problems with, incidentally lead to a corresponding amount of depth in the term graph. The complexity of trait propagation also does not include the cost of propagating $\rightsquigarrow$ relations, whose greatest cost in turn are being transferred through $\rightleftharpoons^*$ relations. At worst, the complexity of a $\rightleftharpoons$ analysis is n^2 for any terms that meet where n binds the number of terms under these; consider:

① A ≔ B.N	③ C ≔ D.N	⑤ E ≔ C.N	⑦ C ≔ X
② B ≔ A.N	④ D ≔ E.N	⑥ A ≔ X	

Six N field terms meet at term X: A.N $\rightleftharpoons$ C.N, B.N $\rightleftharpoons$ E.N, A.N $\rightleftharpoons$ D.N, B.N $\rightleftharpoons$ C.N, A.N $\rightleftharpoons$ E.N, and B.N $\rightleftharpoons$ D.N. However, recursively defined terms are often not out of phase as in this example (2 vs. 3 recursive hops) and complexity is closer to n in the common case. Finally, the analysis is very similar to one based on demand: assignments are mostly not copied unless a term in the assignment chain is referenced by the programmer; the exception being where $\rightsquigarrow$ relationships cause the promotion of assignments at terms where they meet.

Threats to Validity. We have yet to show that this analysis is correct; while we have tested it manually on many examples (filling up many notebooks), enough corner cases justify more rigor. Additionally, although we have implemented a previous less rigorous and less scalable version of this type system in YinYang's programming environment, at the time of this submission, the implementation of the system presented in this paper is still incomplete.

5 Open Design Challenges

From a design perspective, the type inference system presented in this paper is significantly deficient in a few ways. For one thing, without explicit trait extensions, there is no basis to linearize [5] trait compositions in object definitions;

instead we currently order traits by their own extensions with each other, where other ordering is arbitrary but stable. The type system also does not as is support dynamic type tests that will invariably arise in real programs. At first glance, dynamic type tests seem benign as they exist outside of the static type system. However, additional inferences can occur guarded by a type test that, reasonably to the programmer, should not be applied unless that test passed. To solve this problem, we plan to enhance our type system in the future to support "guarded" assignments that are not considered until their guard condition (reachability from the tested trait) are known to be satisfied.

Without considering parametric methods, the number of unique fields in a program is fixed according to the encoded structure of the program. With parametric methods, however, a field must be created under the calling trait to "instantiate" the method, where this field is then assigned to the method being called. Although this complexity is necessary to ensure that instantiations of the method meet overridden definitions correctly, it is unclear if an extra field-per-call site would make our analysis infeasible. More consideration is needed.

Finally, as mentioned in Section 3, the analysis-based nature of our type system complicates reasoning about typing properties and debugging type errors. Typing results cannot be explained in a syntax-directed fashion, but is rather obtained through reasoning over an assignment graph, leading to significant usability challenges in how programmers understand type errors or how objects are formed. The IDE can help out since it knows the assignment graph; e.g. each trait listed in a type probe or error message can be clicked on to navigate to the statement where the extended trait propagates or is seeded into the unit. However, even this is not always good enough: the programmer might need to examine the graph visually to determine what assignment or instantiation led to a type incompatibility. McAdam [15] describes how graphs can be presented to programmers. Future work on YinYang can explore how term assignment graphs can be used to help programmers debug their type errors.

6 Related Work

Realizing type inference through flow analysis is not new: it has been explored with some depth in [26] for a type system with subtyping and recursive types, though without parametricity [3]. YinYang, however, was designed specifically to be typed by analyzing assignments, leading to its non-traditional typing behavior. Our analysis is then related to that on inclusion-based points-to analyses such as Andersen's analysis [4], and more specifically, those that deal with fields in object-oriented languages. As mentioned in Section 2, much of this work [12, 13, 31, 36] depends on "looking" inside a parent term to see what objects they are bound to. Our approach instead decodes field operations by copying field assignments in the direction of the objects assigned to field parent terms, which works better for type inference by avoiding direct object propagation. To the best of our knowledge, we are the first to decode field operations into assignments; it is

not yet clear if this treatment would be useful in a points-to analysis to avoid materializing all potential, but unreferenced, fields.

Concerning work specifically on type inference, Leroy [11] investigates how to support type inference in the presence of assignment by treating type variables involved in assignments as *dangerous* and disallowing generalization, meaning they must be monomorphic. YinYang is not based on generalization and, in fact, tackles fields and assignment as the core of its type system. Agesen explores type inference for objects in the presence of parametric polymorphism with his Cartesian Product Algorithm [2] (CPA) for type inference in Self. CPA resembles YinYang by also maintaining a graph built from assignments (an innovation of Palsberg [25]); however, in CPA's case, "clone families" propagate from assignees to assignments, generalizing their type. CPA also does not support the data polymorphism necessary to model collection types; YinYang supports data polymorphism through its parametric treatment of field assignments. Other work [29, 32, 35] has dealt with data polymorphism using various context sensitive global analyses; our direct approach should be more efficient, though we have not compared against these systems directly.

Besides CPA, there is a plethora of work supporting type inference with subtyping [8, 23, 24, 27, 29, 33, 37]; such techniques are expensive and limited, in particular in the way they handle fields. Palsberg and Zhao [28] show how to detect read-only fields as they admit covariant subtyping, yet YinYang instead ensures that the assignee can take on all traits of any field that it is being assigned into, still avoiding polymorphic cat calls [17]. The fundamental difference is that subtyping in YinYang is prescriptive rather than restrictive; i.e.:

> *They're subtypes, Jim, but not as we know them.*

We do not claim that our approach to subtyping is better, just different; e.g. as mentioned in Section 3, we lose the ability to directly check programmer intent as we gain the ability to infer object compositions.

YinYang's "typeless" code appears similar to code in dynamic languages like Ruby [14] or Python [30], the latter of which is used as a role model for YinYang (in addition to Scala [22]). However, while typing in YinYang is flexible, it is limited by a control flow insensitive analysis; dynamic languages also still require "type annotations" to specify object compositions. The design of mixin-like traits in YinYang are based on Scala's [22], but are enhanced with compatibility relationships to temper multiple inheritance so that traits can fully replace the role of classes. Scala only supports local type inference [21], but then supports advanced features beyond what could be expressed in YinYang such as existential types and GADTs. Still, having inference define objects in YinYang makes fined-grained traits more viable as they do not need to be composed explicitly.

7 Conclusion

We introduced a new point in the type system design space based on assignment, fields, and backward (toward assignee) propagation of trait extensions. With this

type system, YinYang is able to infer object compositions based on usage without type annotations, allowing for the use of finer-grained traits in programming and improving how code completion can aide programmers. The resulting type system is able to support an OO language without many restrictions, and is also simple from the programmer's point of view: all feedback (including three kinds of checks) is derived directly from trait propagation across assignments.

To realize this type system, we invented a novel field-sensitive analysis that avoided the need for object propagation. We found that an assignment between fields of a common term is naturally parametric for assignments of that common term, and exploited this property to model all typing aspects of YinYang with assignments and fields. Finally, we demonstrated how this analysis could be feasible (decidable and scalable) using field skipping and an approach that exploited the "height" of a rooted tree of fields.

There is still much work to be done. We are actively re-implementing our type system based on what we learned from this paper's rigorous treatment. Additionally, we must still show that our scalable analysis is equivalent to our sound but undecidable analysis. Finally, YinYang should be improved to support dynamic typing, efficient parametric methods, and better type debugging.

References

1. Agesen, O.: Constraint-based type inference and parametric polymorphism. In: Proc. of SAS. pp. 78–100 (1994)
2. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Proc. of ECOOP. pp. 2–26 (1995)
3. Amadio, R.M., Cardelli, L.: Subtyping recursive types. In: Proc. of POPL. pp. 104–118 (1991)
4. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (1994)
5. Bracha, G., Cook, W.: Mixin-based inheritance. In: Proc. of OOPSLA/ECOOP. pp. 303–311 (1990)
6. Curry, G., Baer, L., Lipkie, D., Lee, B.: Traits: An approach to multiple-inheritance subclassing. In: Proc. of SIGOA. pp. 1–9 (Jun 1982)
7. Damas, L.: Type Assignment in Programming Languages. Ph.D. thesis, University of Edinburgh (1984)
8. Eifrig, J., Smith, S.F., Trifonov, V.: Type inference for recursively constrained types and its application to oop. ENTCS 1, 132–153 (1995)
9. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in Haskell. ACM TOPLAS 18(2), 109–138 (Mar 1996)
10. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of AMS 146, 29–60 (1969)
11. Leroy, X., Weis, P.: Polymorphic type inference and assignment. In: Proc. of POPL. pp. 291–302 (1991)
12. Lhoták, O., Hendren, L.J.: Scaling java points-to analysis using spark. In: Proc. of CC. pp. 153–169 (2003)
13. Liang, D., Pennings, M., Harrold, M.J.: Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In: Proc. of PASTE. pp. 73–79 (2001)

14. Matsumoto, Y.: The Ruby programming language. www.ruby-lang.org (1996–2013)
15. McAdam, B.J.: Graphs for recording type information. Tech. rep., UNIVERSITY OF EDINBURGH (1999)
16. McDirmid, S.: Escaping the maze of twisty classes. In: Proc. of SPLASH Onward! pp. 127–138 (Oct 2012)
17. Meyer, B.: Beware of polymorphic catcalls. www.eiffel.com/doc/manuals/technology/typing/cat.html (1995)
18. Milner, R.: A theory of type polymorphism in programming. JCSS 17, 348–375 (1978)
19. Moon, D.A.: Object-oriented programming with Flavors. In: Proc. of OOPSLA. pp. 1–8 (1986)
20. Nathan, A.: Windows Presentation Foundation Unleashed (WPF) (Unleashed). Sams (2006)
21. Odersky, M., Zenger, C., Zenger, M.: Colored local type inference. In: Proc. of POPL. pp. 41–53 (2001)
22. Odersky, M., Zenger, M.: Scalable component abstractions. In: Proc. of OOPSLA. pp. 41–57 (2005)
23. Ohori, A., Buneman, P.: Static type inference for parametric classes. In: Proc. of OOPSLA. pp. 445–456 (1989)
24. Oxhøj, N., Palsberg, J., Schwartzbach, M.I.: Making type inference practical. In: Proc. of ECOOP. pp. 329–349 (1992)
25. Palsberg, J.: Efficient inference of object types. In: Proc. of LICS. pp. 186–195 (1994)
26. Palsberg, J., O'Keefe, P.: A type system equivalent to flow analysis. ACM TOPLAS 17(4), 576–599 (Jul 1995)
27. Palsberg, J., Wand, M., O'Keefe, P.: Type inference with non-structural subtyping. Formal Asp. Comput. 9(1), 49–67 (1997)
28. Palsberg, J., Zhao, T., Jim, T.: Automatic discovery of covariant read-only fields. ACM TOPLAS 27(1), 126–162 (Jan 2005)
29. Plevyak, J., Chien, A.A.: Precise concrete type inference for object-oriented languages. In: Proc. of OOPSLA. pp. 324–340 (1994)
30. van Rossum, G.: The Python programming language manual. www.python.org (1990–2013)
31. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for java using annotated constraints. In: Proc. of OOPSLA. pp. 43–55 (2001)
32. Spoon, S.A., Shivers, O.: Dynamic data polyvariance using source-tagged classes. In: Proc. of DLS. pp. 35–48 (2005)
33. Suzuki, N.: Inferring types in Smalltalk. In: Proc. of POPL. pp. 187–199 (1981)
34. Tofte, M.: Type inference for polymorphic references. Inf. Comput. 89(1), 1–34 (Sep 1990)
35. Wang, T., Smith, S.F.: Precise constraint-based type inference for java. In: Proc. of ECOOP (2001)
36. Whaley, J., Lam, M.S.: An efficient inclusion-based points-to analysis for strictly-typed languages. In: Proc. of SAS. pp. 180–195 (2002)
37. Zhao, T.: Polymorphic type inference for scripting languages with object extensions. In: Proc. of DLS. pp. 37–50 (2011)