Heap Decomposition Inference with Linear Programming

Haitao Steve Zhu and Yu David Liu

SUNY Binghamton
Binghamton NY 13902, USA
{hzhu1,davidL}@cs.binghamton.edu

Abstract. Hierarchical decomposition is a fundamental principle that encourages the organization of program elements into nested scopes of access, instead of treating all as "global." This paper offers a foundational study of *heap decomposition inference*, the problem of statically extracting the decomposition hierarchy latent in the runtimes of object-oriented programs, henceforth revealing the compositional nature of the heap. The centerpiece of the paper is Cypress, a sound, precise, and scalable constraint-based ownership type inference coupled with a novel application of linear programming over integers. All constraints in Cypress are linear, and the precision of decomposition – placing objects to scopes as non-global as possible – can be reduced to a linear programming problem. Cypress has been implemented as an open-source tool that can decompose real-world Java applications of more than 100K LOC and up to 6000 statically distinct instantiations.

1 Introduction

The principle of hierarchical decomposition (HD) is a divide-and-conquer strategy to compartmentalize program element accesses into potentially nested scopes. Language support for HD over static program elements – such as lexical scoping – has a history as long as programming language design itself. HD over program runtimes are a direction particularly successful through language design such as ownership type systems [29,6,5,2,3,11].

This paper explores a program analysis approach to HD, addressing how the heap of the standard Java-like object model can be decomposed into a hierarchy that reflects the principle of HD. In other words, even though the heap of Java-like languages is a *global* structure where every object may potentially be referred to by all others, few practical programs take advantage of that potential. Instead, it is more natural to consider every heap object to be implicitly associated with a scope – the same concept as in HD – and view the heap as a *compositional* tree structure where each object serves as the scope of access for its children on the tree. The goal of the inference is to compute a static approximation of the transformation from a global heap to a compositional heap, informing programmers of what their heap "could have been."

Our concrete proposal is Cypress, a sound, precise, and scalable constraint-based type inference. As a form of ownership inference [14,28,34,26,10,17], Cypress is distinct with the following features:

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 104–128, 2013.
© Springer-Verlag Berlin Heidelberg 2013

- *Linear constraints for scalability*. Cypress constraints for computing the compositional heap are *linear*, a scalable solution outsourceable to mature linear solvers.
- *Linear programming for precision*. The precision of decomposition – placing objects to scopes as non-global as possible – is guided by a novel *Cypress Principle* [1], which says a "tall and skinny" view of the compositional heap is favored, and this preference can be reduced to a linear programming problem with a judicious setting of the objective function.
- *Heap-wide scope characterization*. Cypress computes scopes for all heap objects in an algorithm that does not require iteration over each object. The result of the analysis is a tree including all (static) heap objects, organized in a way vividly reflecting how the heap is decomposed according to the principle of HD.
- *Fully automated static inference*: Cypress is a sound type inference that does not require programmer annotation or interactive user assistance.

Beyond the technical contribution of proposing a new ownership inference algorithm, the paper is further aimed at gaining a fundamental understanding of HD as a principle. It provides a unified framework where commonly discussed HD mechanisms – lexical decomposition (lexical scoping), dynamic heap decomposition (dynamic ownership), and static heap decomposition (static ownership) – are formally related, and the HD principle is captured as an invariant independent of the mechanisms that realize it.

Overall, heap decomposition inference addresses a fundamental goal in programming languages – promoting local reasoning – with a broad range of benefits in optimizing, debugging, verifying, securing, and understanding programs. Thread locality – a property crucial for optimizing multithreaded programs (*e.g.* [1,13]) – can be viewed as decomposing heap objects into scopes defined by threads. Objects in the same scope may signify the likelihood of co-use, and co-allocating them may lead to reduced cache misses, a boon for performance [15] and energy efficiency [33]. Localizing the use of objects can also aid bug localization [31] and analyze the impact of change [4]. Progress in separation logic (*e.g.*[12]) demonstrates the usefulness of locality information in verification. In security, the scope of capabilities [9] – object references in object-oriented programs – is crucial in constructing access control and authority isolation [27,23]. The compositional view of the heap is also beneficial in reverse engineering [16].

This paper makes the following contributions:

- a unified framework to reason about HD and its properties (Sec. 2).
- a novel constraint-based type inference to abstract object access in the presence of the compositional heap as linear constraints (Sec. 4).
- a novel application of linear programming to improve inference precision, guided by the Cypress Principle (Sec. 5).
- a prototyped implementation that analyzes real-world programs (Sec. 6).

2 Hierarchical Decomposition

In this section, we define a unified framework for HD, with three common forms of HD – lexical decomposition, dynamic heap decomposition, static heap decomposition –

[1] A cypress – especially its subspecies known as Mediterranean Cypress – is a tree usually "tall and skinny" in shape. It is a common image, such as in Vincent van Gogh's *Starry Night*.

definition \ form	lexical decomposition ($\ell = \mathtt{x}$)	dynamic heap decomposition ($\ell = \mathtt{d}$)	static heap decomposition ($\ell = \mathtt{s}$)
reasoning about	program	run-time state	program (and its relationship with run-time states)
accessors ($a^\ell \in A^\ell$)	code blocks	objects	abstract objects
accessees ($b^\ell \in B^\ell$)	variable names	objects	abstract objects
elements ($c^\ell \in C^\ell$)	$A^\ell \cup B^\ell$		
access relation $a^\ell \hookrightarrow^\ell b^\ell$	a^ℓ immediately encloses b^ℓ use	heap stores/stack frames of a^ℓ refers to b^ℓ	fields/methods of a^ℓ refers to b^ℓ
scope relation $a^\ell \blacktriangleright^\ell c^\ell$	a^ℓ immediately encloses c^ℓ definition	inferred in standard object model (or induced via language design)	
$\mathtt{ehd}^\ell(\hookrightarrow^\ell, \blacktriangleright^\ell)$ for soundness	necessary and sufficient		necessary

Fig. 1. From Lexical Decomposition to Heap Decomposition

summarized in Fig. 1. Metavariables introduced in this section routinely use superscript labels $\mathtt{x}$, $\mathtt{d}$, $\mathtt{s}$ to differentiate the three forms.

Throughout the paper, we say relation $\mathit{Rel} : \mathit{SetA} \times \mathit{SetB}$ is a *rooted tree relation* if *Rel* is injective, surjective, irreflexive, acyclic, and contains one unique element *elmtA* such that $\mathit{elmtA} \in \mathit{SetA}$ and $\mathit{elmtA} \notin \mathit{SetB}$ and $(\mathit{elmtA}, \mathit{elmtB}) \in \mathit{Rel}$ for some *elmtB*. We further say *elmtA* is the *root* of *Rel*. The rooted tree relation intuitively corresponds to the edge set of graph-theoretic rooted directed trees.

2.1 The Essence of Hierarchical Decomposition (though Lexical Decomposition)

Let us start with the most well-understood HD mechanism, block-based lexical scoping. Given a program P, we abstract it as a 5-tuple $\langle A^{\mathtt{x}}; B^{\mathtt{x}}; a^{\mathtt{x}}_{\mathtt{G}}; \hookrightarrow^{\mathtt{x}}; \blacktriangleright^{\mathtt{x}} \rangle$ configuration where

- Accessor/Scope set $A^{\mathtt{x}}$: the set of code blocks (IDs) in P
- Accessee set $B^{\mathtt{x}}$: the set of variables (names) in P
- Global scope $a^{\mathtt{x}}_{\mathtt{G}}$: the code block that implicitly encloses the entire program
- Access relation $\hookrightarrow^{\mathtt{x}}$: $A^{\mathtt{x}} \cup \{a^{\mathtt{x}}_{\mathtt{G}}\} \times B^{\mathtt{x}}$: $a^{\mathtt{x}} \hookrightarrow^{\mathtt{x}} b^{\mathtt{x}}$ says $a^{\mathtt{x}}$ accesses $b^{\mathtt{x}}$. Concretely, it is defined as code block $a^{\mathtt{x}}$ immediately encloses the use of variable $b^{\mathtt{x}}$.

– Scope relation $\blacktriangleright^{\mathrm{x}}$: $A^{\mathrm{x}} \cup \{a_{\mathtt{G}}^{\mathrm{x}}\} \times A^{\mathrm{x}} \cup B^{\mathrm{x}}$: a rooted tree relation with root $a_{\mathtt{G}}^{\mathrm{x}}$ and $a^{\mathrm{x}} \blacktriangleright^{\mathrm{x}} c^{\mathrm{x}}$ says a^{x} is the *lexical scope* of c^{x}. Concretely, it is defined as block a^{x} immediately encloses c^{x} definition, as a variable declaration or a nested block.

It is important to realize that for any program subject to lexical scoping, a latent invariant must hold between $\hookrightarrow^{\mathrm{x}}$ and $\blacktriangleright^{\mathrm{x}}$. Indeed, this invariant is a concrete instance of a more general invariant that epitomizes hierarchical decomposition:

Definition 1 (The Essence of Hierarchical Decomposition). *HD mechanisms maintain the following invariant between the access relation $\hookrightarrow^{\ell}$ and scope relation $\blacktriangleright^{\ell}$:*

$$\mathtt{ehd}^{\ell}(\hookrightarrow^{\ell}, \blacktriangleright^{\ell}) \stackrel{\text{def}}{=} a_1^{\ell} \blacktriangleright^{\ell} b^{\ell} \wedge a_2^{\ell} \hookrightarrow^{\ell} b^{\ell} \implies a_1^{\ell} \blacktriangleright_*^{\ell} a_2^{\ell}$$

where ℓ is the identifier of the HD mechanism itself (such as $\mathtt{x}$), and $\blacktriangleright_*^{\ell}$ is the reflexive and transitive closure of $\blacktriangleright^{\ell}$. The invariant can be interpreted in two equivalent ways:

Accessors from Inner Scopes. If a_1^{ℓ} is the scope of b^{ℓ}, then any access to b^{ℓ} must come from an *inner scope* of a_1^{ℓ}, *i.e.* a scope that can be reached from a^{ℓ} according to $\blacktriangleright^{\ell}$.

Accessees in Outer Scopes. If a_2^{ℓ} accesses b^{ℓ}, then the scope of b^{ℓ} must be an *outer scope* of a_2^{ℓ}, *i.e.* a scope that can reach a_2^{ℓ} according to $\blacktriangleright^{\ell}$.

The **Accessors from Inner Scopes** interpretation demonstrates the key benefit of HD: promoting local reasoning. In Sec. 4, the **Accessees in Outer Scopes** interpretation will help us encode the essence of HD in a type inference algorithm. With this formulation, the common notion of lexical scoping is:

Definition 2 (Lexical Decomposition). *Configuration $\langle A^{\mathrm{x}}; B^{\mathrm{x}}; \mathrm{a}_{\mathtt{G}}^{\mathrm{x}}; \hookrightarrow^{\mathrm{x}}; \blacktriangleright^{\mathrm{x}}\rangle$ conforms to lexical decomposition iff* $\mathtt{ehd}^{\mathrm{x}}(\hookrightarrow^{\mathrm{x}}, \blacktriangleright^{\mathrm{x}})$.

Our goal is to use the familiar lexical decomposition to shed light on heap decomposition. As it turns out, lexical decomposition and heap decomposition bear remarkable similarity: they share the same $\mathtt{ehd}^{\ell}$ invariant. For instance, the two interpretations above resonate with deep ownership and owners-as-dominators [5] in ownership types.

2.2 Dynamic Heap Decomposition

The main difference between lexical decomposition and heap decomposition is that the latter is concerned with the access relation and the scope relation *among objects*. Let us first consider the case of dynamic heap decomposition, *i.e.,* HD for the run-time state at a program execution step. For our discussion here, we abstract the run-time state as a configuration CF^{d} in the form of a 4-tuple $\langle O^{\mathrm{d}}; o_{\mathtt{G}}; \hookrightarrow^{\mathrm{d}}; \blacktriangleright^{\mathrm{d}}\rangle$ including:

– Object set (unified accessor/accessee/scope set) O^{d}: the set of objects (IDs)
– Global scope $o_{\mathtt{G}}$: the implicit "global" object that encloses the bootstrapping code
– Access relation $\hookrightarrow^{\mathrm{d}}$: $O^{\mathrm{d}} \cup \{o_{\mathtt{G}}\} \times O^{\mathrm{d}}$, where $o_1 \hookrightarrow^{\mathrm{d}} o_2$ says o_1 accesses o_2. Under concrete dynamic semantics with heaps and stacks, one possible $\hookrightarrow^{\mathrm{d}}$ relation is the object reference relation[2], where $o_1 \hookrightarrow^{\mathrm{d}} o_2$ holds iff the heap store for object o_1 or any stack frame for any method invocation of o_1 contains a reference to o_2.

[2] Defining the access relation as the reference relation is similar to the conventional capability model [9]. There are alternative ways to define the access relation, such as the accessor reads from and writes to a field of the accessee.

– Scope relation $\blacktriangleright^{\mathtt{d}}$: $O^{\mathtt{d}} \cup \{o_{\mathtt{G}}\} \times O^{\mathtt{d}}$, a rooted tree relation where $o_1 \blacktriangleright^{\mathtt{d}} o_2$ says o_1 is the *dynamic object scope* of o_2, and $o_{\mathtt{G}}$ is the root

To strictly mirror the definition we used for lexical decomposition, the 4-tuple above is indeed a degenerate 5-tuple $\langle A^{\mathtt{d}}; B^{\mathtt{d}}; o_{\mathtt{G}}; \hookrightarrow^{\mathtt{d}}; \blacktriangleright^{\mathtt{d}}\rangle$ where $A^{\mathtt{d}} = B^{\mathtt{d}} = O^{\mathtt{d}}$. In other words, objects are both accessors and accessees in dynamic heap decomposition. With this alignment, the essence of dynamic heap decomposition can be analogously defined:

Definition 3 (Dynamic Heap Decomposition). *Configuration* $\langle \mathrm{O}^{\mathtt{d}}; o_{\mathtt{G}}; \hookrightarrow^{\mathtt{d}}; \blacktriangleright^{\mathtt{d}}\rangle$ *is a dynamic heap decomposition iff* $\mathtt{ehd}^{\mathtt{d}}(\hookrightarrow^{\mathtt{d}}, \blacktriangleright^{\mathtt{d}})$.

Even though the invariant behind lexical decomposition and dynamic heap decomposition is in essence identical, there is often a difference on how the invariant is applied. In the former, both $\hookrightarrow^{\mathtt{x}}$ and $\blacktriangleright^{\mathtt{x}}$ are readily available during parsing, so $\mathtt{ehd}^{\mathtt{x}}$ is mostly used to provide a boolean answer: does the program conform (to lexical decomposition)? In dynamic heap decomposition however, only $\hookrightarrow^{\mathtt{d}}$ is available in Java-like run-times, so $\mathtt{ehd}^{\mathtt{d}}$ is more often applied to *dynamic heap decomposition inference*: given a run-time state whose access relation is $\hookrightarrow^{\mathtt{d}}$, find a $\blacktriangleright^{\mathtt{d}}$ such that $\mathtt{ehd}^{\mathtt{d}}(\hookrightarrow^{\mathtt{d}}, \blacktriangleright^{\mathtt{d}})$.

In plain words, dynamic heap decomposition inference is aimed at (figuratively) organizing the objects of a run-time state into a hierarchy that conforms to the principle of HD, based on information of inter-object access at the run-time state. Just as lexical decomposition, dynamic heap decomposition also promotes local reasoning: it attempts to provide a scope to every object instead of assuming all as "global." Existing work on dynamic UML composition inference (*e.g.* [16]) and dynamic ownership inference (*e.g.* [28]) are concrete instances in this category. In those contexts, the object that forms the "dynamic object scope" of another is called a *compositional object* or an *owner*.

2.3 Static Heap Decomposition

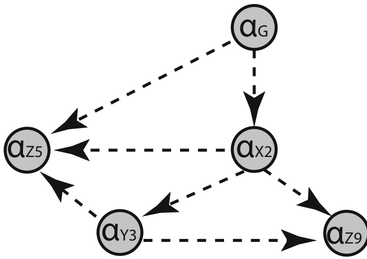
The problem with dynamic heap decomposition is it only applies HD to one particular runtime state of a program's execution (or a finite combination of them). What is more useful is to characterize how heap decomposition can be applied to all run-time states of a program P. This goal is achieved by static heap decomposition, a conservative approximation of dynamic heap decomposition for all run-time states.

Static heap decomposition (checking or inference) systems are constructed over a program abstraction which we represent as configuration $CF^{\mathtt{s}}$, a tuple $\langle O^{\mathtt{s}}; \alpha_{\mathtt{G}}; \hookrightarrow^{\mathtt{s}}; \blacktriangleright^{\mathtt{s}}\rangle$ including:

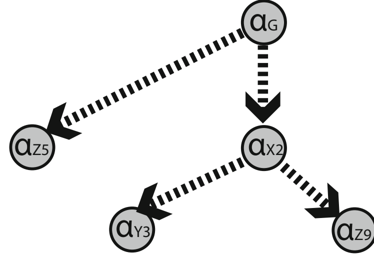
– Abstract object set (unified accessor/accessee/scope set) $O^{\mathtt{s}}$: the set of static approximations of distinct run-time objects of the program
– Global scope $\alpha_{\mathtt{G}}$: the approximation of the implicit "global" object that encloses the bootstrapping code
– Access relation $\hookrightarrow^{\mathtt{s}}$: $O^{\mathtt{s}} \cup \{\alpha_{\mathtt{G}}\} \times O^{\mathtt{s}}$, where $\alpha_1 \hookrightarrow^{\mathtt{s}} \alpha_2$ says α_1 is the static approximation of an object that accesses an object of which α_2 is the static approximation. We also informally say α_1 *statically accesses* α_2.
– Scope relation $\blacktriangleright^{\mathtt{s}}$: $O^{\mathtt{s}} \cup \{\alpha_{\mathtt{G}}\} \times O^{\mathtt{s}}$, a rooted tree relation where $\alpha_1 \blacktriangleright^{\mathtt{s}} \alpha_2$ says α_1 is the *static object scope* of α_2, and $\alpha_{\mathtt{G}}$ is the root

```
 1 class Main {                          16         y2 = y1;
 2   void main() {                       17         z1 = new Z@αZ5();
 3       X@αX1 x;                        18         z2 = y2.mdy(z1);
 4       Z@αZ1 z;                        19         return y2.fdz;
 5       x = new X@αX2();                20     }
 6       z = x.mdx();                    21 }
 7   }                                   22 class Y {
 8 }                                     23   Z@αZ6 fdz;
 9 class X {                             24   Z@αZ7 mdy(Z@αZ8 z) {
10   Z@αZ2 mdx() {                       25     this.fdz = z;
11       Y@αY1 y1;                       26     return new Z@αZ9();
12       Y@αY2 y2;                       27   }
13       Z@αZ3 z1;                       28 }
14       Z@αZ4 z2;                       29 class Z {...}
15       y1 = new Y@αY3();
```

(a) an example



(b) access relation $\hookrightarrow^{s}$



(c) scope relation $\blacktriangleright^{s}$ (HDT)

Legends: abstract object static access relation static scope relation

Fig. 2. Static Heap Decomposition Inference

In the rest of the presentation, we call each object static approximation an *abstract object*, or simply *object* when no confusion can arise. Due to the standard feature of aliasing in object-oriented languages, static approximations may or may not represent distinct run-time objects. We liberally use the term "abstract object" – and its representing metavariable α – to refer to both. For convenience, we also have α subsumes $\alpha_{\mathtt{G}}$.

Following the analogy of dynamic heap decomposition, it would be natural to provide the following definition:

Definition 4 (Static Heap Decomposition). *Configuration* $\langle \mathrm{O}^{s}; \alpha_{\mathtt{G}}; \hookrightarrow^{s}; \blacktriangleright^{s}\rangle$ *is a static heap decomposition iff* $\mathtt{ehd}^{s}(\hookrightarrow^{s}, \blacktriangleright^{s})$.

Let us consider a Java program in Fig. 2(a) to gain some intuition. For convenience, we directly associate each object type declaration with an abstract object ID. For instance, expression **new** $\texttt{Y@}\alpha_{\texttt{Y3}}\texttt{()}$ at Line 15 is the Java expression **new** `Y()` with abstract object $\alpha_{\texttt{Y3}}$. In other words, we treat each program element associated with a type declaration – a local variable, a method parameter, a return value, or an instantiated object – as an abstract object. These abstract objects may or may not lead to "distinct approximations" due to aliasing, but the ones associated with the **new** expressions indeed do, *i.e.*, $O^{\mathsf{s}} = \{\alpha_{\texttt{X2}}, \alpha_{\texttt{Y3}}, \alpha_{\texttt{Z5}}, \alpha_{\texttt{Z9}}\}$. $\hookrightarrow^{\mathsf{s}}$ can be concretely defined as the points-to information, which in the example is illustrated in Fig. 2(b). One possible $\blacktriangleright^{\mathsf{s}}$ is illustrated in Fig. 2(c). It can be easily seen that the two conform to static heap decomposition: every accessor of an object is from inner scopes and every accessee of an object is within outer scopes.

Both checking and inference systems can be constructed for static heap decomposition. A static heap decomposition checker answers whether $\texttt{ehd}^{\mathsf{s}}$ holds given a program and its information on $\hookrightarrow^{\mathsf{s}}$ and $\blacktriangleright^{\mathsf{s}}$ (explicitly or implicitly defined). A *static heap decomposition inference* finds a $\blacktriangleright^{\mathsf{s}}$ that satisfies $\texttt{ehd}^{\mathsf{s}}$ given a program and its information on $\hookrightarrow^{\mathsf{s}}$ (explicitly or implicitly defined). Broadly construed, the typechecking process of ownership type systems [29,6,5] is an instance of static heap decomposition checker. In that context, $\blacktriangleright^{\mathsf{s}}$ is the relationship among ownership type parameters, whereas $\hookrightarrow^{\mathsf{s}}$ and $\texttt{ehd}^{\mathsf{s}}$ are implict in the typechecking rules. As another example, ownership inferences based on points-to analyses [24,22] are instances of static heap decomposition inference. They explicitly compute $\hookrightarrow^{\mathsf{s}}$ and finds the $\blacktriangleright^{\mathsf{s}}$ that conforms to $\texttt{ehd}^{\mathsf{s}}$.

Cypress is a static heap decomposition inference. To highlight the central role of the $\blacktriangleright^{\mathsf{s}}$ relation in this context – it is the output of the inference – we reinstate it with the following definition (note that $\blacktriangleright^{\mathsf{s}}$ by definition is a tree relation):

Definition 5 (Heap Decomposition Tree (HDT)). *We call* $\blacktriangleright^{\mathsf{s}}$ *a* heap decomposition tree *for* P *if configuration* $\langle \mathrm{O}^{\mathsf{s}}; \alpha_{\texttt{G}}; \hookrightarrow^{\mathsf{s}}; \blacktriangleright^{\mathsf{s}} \rangle$ *is a static heap decomposition for* P *for some* O^{s}, $\alpha_{\texttt{G}}$, *and* $\hookrightarrow^{\mathsf{s}}$.

In the rest of the discussion, we will liberally interpret HDT in a graph-theoretic manner. For instance, the *nodes* of an HDT $\blacktriangleright^{\mathsf{s}}$ are the union of the domain and the range of $\blacktriangleright^{\mathsf{s}}$, and the *edges* are the set interpretation of $\blacktriangleright^{\mathsf{s}}$ itself.

2.4 Challenges

Challenge I: Soundness. Is every static heap decomposition according to Def. 4 a "good" one? At the beginning of Sec. 2.3, we explained a key motivation of constructing static heap decomposition is to "characterize how heap decomposition can be applied to all run-time states of a program." Def. 4 unfortunately does not correspond with the run-time states. For this latter goal, let us first introduce *correspondence relation* $CF^{\mathsf{d}} \xrightarrow{\mathcal{X}} CF^{\mathsf{s}}$, which says CF^{s} is a static approximation for run-time configuration CF^{d} over *abstraction mapping* $\mathcal{X}$. An abstraction mapping is a simple mapping from o's to α's, where $\mathcal{X}(o)$ is the abstract object approximating o. A correspondence relation $\langle O^{\mathsf{d}}; o_{\texttt{G}}; \hookrightarrow^{\mathsf{d}}; \blacktriangleright^{\mathsf{d}} \rangle \xrightarrow{\mathcal{X}} \langle O^{\mathsf{s}}; \alpha_{\texttt{G}}; \hookrightarrow^{\mathsf{s}}; \blacktriangleright^{\mathsf{s}} \rangle$ is well-formed iff

- $O^{\mathsf{d}} \subseteq \mathtt{dom}(\mathcal{X})$
- $O^{\mathsf{s}} = \mathtt{ran}(\mathcal{X})$
- $\mathcal{X}(o_{\mathtt{G}}) = \alpha_{\mathtt{G}}$
- $\mathcal{X}(o) \hookrightarrow^{\mathsf{s}} \mathcal{X}(o)$ for any $o \hookrightarrow^{\mathsf{d}} o'$
- $\mathcal{X}(o) \blacktriangleright^{\mathsf{s}} \mathcal{X}(o)$ for any $o \blacktriangleright^{\mathsf{d}} o'$

where $\mathtt{dom}(\mathcal{X})$ and $\mathtt{ran}(\mathcal{X})$ operators compute the domain and range of $\mathcal{X}$ respectively.

Definition 6 (Soundness). *Given a program P, its static heap decomposition* CF^{s} *is sound with respect to $\mathcal{X}$ iff there exists a* CF^{d} *for any run-time state of P, such that* $\mathrm{CF}^{\mathsf{d}} \xrightarrow{\mathcal{X}} \mathrm{CF}^{\mathsf{s}}$ *and* CF^{d} *is a dynamic heap decomposition.*

In practice, the $\hookrightarrow^{\mathsf{s}}$ relation in CF^{s} represents the points-to information, and the $\hookrightarrow^{\mathsf{d}}$ relation in CF^{d} represents the reference relation at a particular run-time state. The two are determined once the concrete abstractions of the program and the runtime are given. Consequently, the soundness definition here mainly reveals the relationship between the static scope relation in CF^{s} and the dynamic scope relation in CF^{d}. In other words, a static scope relation $\blacktriangleright^{\mathsf{s}}$ in a sound static heap decomposition must serve as a "template" for at least one dynamic scope relation $\blacktriangleright^{\mathsf{d}}$ at every run time state such that (1) $\blacktriangleright^{\mathsf{d}}$ is indeed "instantiated" from $\blacktriangleright^{\mathsf{s}}$ (according to the last well-formedness condition of the correspondence relation); and (2) $\blacktriangleright^{\mathsf{d}}$ offers hierarchical decomposition for the run-time state (a condition of the soundness definition).

We are able to show Cypress is a sound static decomposition inference in that any HDT (*i.e.* $\blacktriangleright^{\mathsf{s}}$) it infers forms a sound static decomposition against a static semantics with standard points-to abstraction for $\hookrightarrow^{\mathsf{s}}$ and a dynamic semantics with standard definitions of $\mathcal{X}$ and $\hookrightarrow^{\mathsf{d}}$.

Challenge II: Efficient Heap-Wide Characterization. Cypress computes a surjective relation $\blacktriangleright^{\mathsf{s}}$ over the abstract object set (O^{s} in CF^{s}) of a program. In other words, the scope of every abstract object is determined. The output of Cypress is thus more expressive than decision procedures that answer "given objects α and α', is α in the scope of α'?" or "what is the scope of object α?" or "what objects does scope α' include?"

It is possible to achieve heap-wide characterization by mechanical pairwise applications of the decision procedure "is α in the scope of α'?" In contrast, Cypress computes the entire $\blacktriangleright^{\mathsf{s}}$ relation through a single instance of linear programming.

Challenge III: Precision. First, a well-formed program – *i.e.* a well-typed program according to Java typechecking – at least has one HDT:

Lemma 1 (HDT Existence). *Given a well-formed program whose abstract object set is* O^{s}, $\{(\alpha_{\mathtt{G}}, \alpha) \mid \alpha \in \mathrm{O}^{\mathsf{s}}\}$ *is an HDT.*

In other words, this HDT is a trivial "egalitarian" tree where $\alpha_{\mathtt{G}}$ is the scope of all abstract objects. The bad news is that this HDT is the least useful for characterizing heap decomposition: it defaults to the "global heap" view.

HDTs may not be unique for a program. For example, Fig. 2(c) is an HDT for program Fig. 2(a), but so is the egalitarian tree we just described. Now that multiple HDTs

may exist, the more interesting question is the preference over them. If we take the graph-theoretic view of the HDTs, observe the "egalitarian" tree above is the "short and broad" whereas the tree in Fig. 2(c) is relatively "tall and skinny." Indeed, the shorter and broader a tree is, the closer it is to the default view of the global heap, and the less it promotes local reasoning.

Cypress is designed with the *Cypress Principle* as guidance. Intuitively, it says whenever a static heap decomposition inference algorithm is faced with a choice between constructing the HDT broader or taller, it should favor the latter. Formally, let us define the depth of α – denoted as $\mathtt{depth}(\alpha, \blacktriangleright^{\mathsf{s}})$ – as the length of the path from $\alpha_{\mathtt{G}}$ to α when $\blacktriangleright^{\mathsf{s}}$ is interpreted graph-theoretically. Thus:

Definition 7 (Cypress Principle). *Given P with two sound static heap decompositions $\langle \mathrm{O}^{\mathsf{s}}; \alpha_{\mathtt{G}}; \hookrightarrow^{\mathsf{s}}; \blacktriangleright^{\mathsf{s}}_1 \rangle$ and $\langle \mathrm{O}^{\mathsf{s}}; \alpha_{\mathtt{G}}; \hookrightarrow^{\mathsf{s}}; \blacktriangleright^{\mathsf{s}}_2 \rangle$, and $\sum_{\alpha \in \mathrm{O}^{\mathsf{s}}} \mathtt{depth}(\alpha, \blacktriangleright^{\mathsf{s}}_1) < \sum_{\alpha \in \mathrm{O}^{\mathsf{s}}} \mathtt{depth}(\alpha, \blacktriangleright^{\mathsf{s}}_2)$, the Cypress Principle favors the latter.*

The ideal is to find an optimal HDT whose aggregated depth (*i.e.* the sum of depth of α's in the abstract object set) is maximal. Indeed, this is strictly the opposite of the egalitarian tree whose aggregated depth is the minimal. In Sec. 5, we shall see such an optimal HDT can be effectively computed by Cypress through linear programming.

Challenge IV: Scalability. Def. 4 appears to indicate that to compute $\blacktriangleright^{\mathsf{s}}$, one needs full knowledge of $\hookrightarrow^{\mathsf{s}}$. This route is followed by some related work [24,22,26]. Cypress demonstrates a type inference approach can avoid the explicit computation of $\hookrightarrow^{\mathsf{s}}$, hence decoupling the scalability of our algorithm from that of points-to analyses. In addition, Cypress yields linear constraints, further promoting scalability.

3 Abstract Syntax

We develop our inference over a small language whose abstract syntax is defined in Fig. 3. The language is similar to Featherweight Java (FJ) [18], where notation $\overline{\bullet}$ represents a sequence of $\bullet$'s. The most noticeable difference here is expressions are A-Normal, a form commonly used for specifying alias analyses over Java programs (*e.g.*, [25]). In this form, (single) expressions include assignment $\mathtt{x} = \mathtt{y}$, field read $\mathtt{x} = \mathtt{y.fd}$, field update $\mathtt{x.fd} = \mathtt{y}$, method invocation $\mathtt{x} = \mathtt{y.md}(\mathtt{z})$ and object instantiation $\mathtt{x} =$ **new** τ. The use of the A-normal form requires us to explicitly declare local variables in the method definition (see M). Such local variable declarations are represented by a type environment Γ that maps variable names to types. Pre-defined variable **this** is Java's self reference. Pre-defined class name `Object` is the root class. A program P is formed by a sequence of classes $\overline{CL}$, followed by local variable declarations Γ, and the bootstrapping expression e.

FJ function $\mathtt{fields}(\mathtt{X})$ computes the sequence of fields for class $\mathtt{X}$, in the form of F. FJ function $\mathtt{mtype}(\mathtt{md}, \mathtt{X})$ computes the signature of method $\mathtt{md}$ for class $\mathtt{X}$, in the form of $\tau \rightarrow \tau'$ where τ and τ' are the argument/return types respectively. FJ-like function $\mathtt{mbody}(\mathtt{md}, \mathtt{X})$ computes the method body of $\mathtt{md}$ for class $\mathtt{X}$, in the form of $\mathtt{x}.\mathtt{y}.\Gamma.e$ where $\mathtt{x}$ is the formal argument name, Γ is the local variable declarations,

P	$::= \langle \overline{CL}; \Gamma; e \rangle$	*program*
CL	$::=$ **class** $\mathtt{X}$ **extends** $\mathtt{Y}$ $\{F\ M\}$	*class*
F	$::= \overline{\tau\ \mathtt{fd}}$	*fields*
M	$::= \overline{\tau\ \mathtt{md}(\tau\ \mathtt{x})\{\Gamma\ e\}}$	*methods*
e	$::= s \mid s; e$	*expression*
s	$::= \mathtt{x} = \mathtt{y} \mid \mathtt{x} = \mathtt{y.fd} \mid \mathtt{x.fd} = \mathtt{y}$	*single expression*
	$\mid \mathtt{x} = \mathtt{y.md(z)} \mid \mathtt{x} =$ **new** τ	
Γ	$::= \overline{\mathtt{x} \mapsto \tau}$	*type environment*
$\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{U}, \mathtt{V}$	$\in \mathbb{CN} \cup \{\mathtt{Object}\}$	*class name*
$\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{u}, \mathtt{v}$	$\in \mathbb{VAR} \cup \{$**this**$\}$	*variable name*
$\mathtt{fd}$	$\in \mathbb{FN}$	*field name*
$\mathtt{md}$	$\in \mathbb{MN}$	*method name*
τ		*type (see Sec. 4)*

Fig. 3. Abstract Syntax

and e is the expression that constitutes the method body. For convenience, we have an explicitly bound variable $\mathtt{y}$ to store the return value. For a method $\mathtt{Y}\ \mathtt{md}(\mathtt{X}\ \mathtt{x})\{\mathtt{Z}\ \mathtt{z}; \mathtt{z} = \mathtt{x};$ **return** $\mathtt{z}; \}$ in Java syntax, the method body computed by $\mathtt{mbody}$ is $\mathtt{x.u.}(\mathtt{z} \mapsto \tau).(\mathtt{z} = \mathtt{x}; \mathtt{u} = \mathtt{z})$ where τ is the type form in our system for $\mathtt{Z}$. The definitions of $\mathtt{fields}(\mathtt{X})$, $\mathtt{mtype}(\mathtt{md}, \mathtt{X})$, $\mathtt{mbody}(\mathtt{md}, \mathtt{X})$ are identical to their FJ counterparts, except that $\mathtt{mbody}$ computes a 4-component method body as we just described. In addition, we reuse $\mathtt{X} <: \mathtt{Y}$ to denote $\mathtt{X}$ is a (nominal) subclass of $\mathtt{Y}$. Relation $<:$ is a partial order, whose definition is identical to that of FJ. We defer the concrete definition of type τ to the next section.

4 Type Inference with Linear Constraints over Walk Indices

This section defines a constraint-based type inference to encode the essence of heap decomposition: given a program P, every solution to the constraints generated for P represents an HDT for P.

Walk Index on HDT. Recall the essence of HD is the invariant between the access relation and the scope relation. **Accessees in Outer Scopes** says that every referred object must belong to an outer scope (including the current one) of the referring object. Fig. 4 illustrates 8 possible examples, where the access relation and the scope relation overlay on each figure, and the root of the scope relation is at the top. It is not difficult to observe that the first 6 examples conform to **Accessees in Outer Scopes**, but the last 2 do not.

Graph theoretically, a rooted tree such as the HDT enjoys very strong properties in expressing the relative position of tree nodes. The relative position of α_2 to α_1 can be expressed by the shortest path between them: first "going up" (*i.e.* root-bound) for n_1 edges ($n_1 \geq 0$) from α_1, and then "going down" (*i.e.* leaf-bound) for n_2 ($n_2 \geq 0$) edges until reaching α_2. In short, the relative position can be encoded by a pair of integer values, n_1 and n_2. We say the *walk index* of α_2 from the perspective of α_1 is $\nearrow^{n_1}\searrow^{n_2}$, or

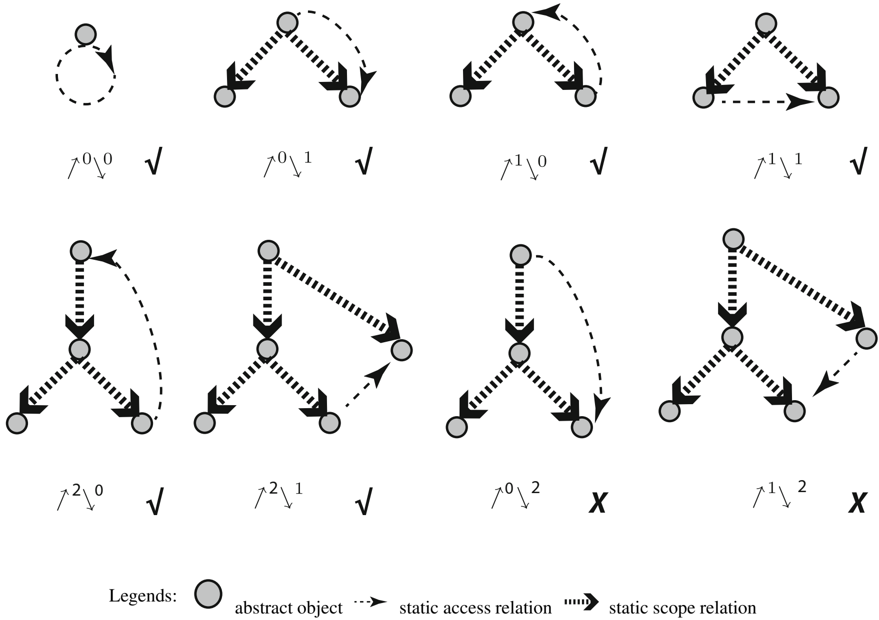


Fig. 4. Access, Scope, and Walk Index

informally say α_1 can walk to α_2 via $\nearrow^{n_1}\searrow^{n_2}$. Encoding "outer/inner scopes" as a pair of integers was first investigated in a previous work by Liu and Smith [21], where the integer pair in a type system design setting was called pedigrees. Broadly, the algorithm introduced in this section can be viewed as a type inference of the type system in that work, an extreme case where every type is inferred.

With the walk index encoding, the essence of HD can be succinctly captured as a restriction on the walk index: the referring object must be able to walk to every of its referred object via walk index $\nearrow^{n_1}\searrow^{n_2}$, *where* $n_2 \leq 1$. Fig. 4 also shows each walk index of the accessee from the perspective of the accessor. The last two examples do not satisfy the requirement of $n_2 \leq 1$, and they violate the principle of HD as well.

Types. In Java-like languages, every referred object is represented by an object reference held by the referring object, and such object reference should have a type in a well-typed Java program. Recall our goal here is to guarantee every referred object to have a walk index (from the perspective of the referring object) that satisfies some restriction ($n_2 \leq 1$ above). Combining the two, we extend the Java type system by building the walk index into the object type, as τ in Fig. 5. Metavariable ω represents a form similar to the walk index we introduced earlier, except that it is associated with a pair of type variables, *p* and *q*, which we call an *up-step type variable* and a *down-step type variable* respectively. The type inference algorithm attempts to solve them with integers (*i.e.* the n_1 and n_2 earlier). We put *p* and *q* into different syntatical categories, so that the constraint solver (Sec. 5) will find non-negative solutions for *p*, whereas 0-1

τ	$::=$	$\mathtt{X}@\alpha[\omega]$	*type*
ω	$::=$	$\nearrow^p\searrow^q$	*walk index with type variables*
p	$\in$	$\mathbb{PVAR}$	*up-step type variable (solution over* $\{0, 1, \ldots\}$*)*
q	$\in$	$\mathbb{QVAR}$	*down-step type variable (solution over* $\{0, 1\}$ *)*
α	$\in$	$\mathbb{LAB} \cup \{\alpha_{\mathtt{G}}\}$	*abstract object ID*
$\mathcal{K}$	$::=$	$\overline{\kappa}$	*constraint set*
κ	$::=$	$\theta = n \mid \theta \geq n$	*linear constraint*
θ	$::=$	$\theta + t \mid \theta - t$	*linear expression*
t	$::=$	$p \mid q$	*type variable*
n	$\in$	$\{0, 1, \ldots\}$	*non-negative integer*

Fig. 5. Types and Constraints

binary solutions for q. As we discussed earlier, the latter reflects the principle of HD. When no confusion can arise, we also call ω a walk index.

Overall, a type takes the form of $\mathtt{X}@\alpha[\omega]$, where $\mathtt{X}$ is the Java nominal type. For the purpose of presentation, we retain the abstract object ID α in the type. It was used in the example of Fig. 2, where every distinct type declaration is associated with a distinct α. We keep the same convention, except in the case of overriding, we follow the standard FJ requirement of equating the signatures – and hence every component of the types – of the overriding methods and overridden methods. For each program, we define $\mathcal{PV}$ as the smallest mapping that includes (α, p) where $\mathtt{X}@\alpha[\nearrow^p\searrow^q]$ occurs in the program, and $\mathcal{QV}$ as the smallest mapping that includes (α, q) where $\mathtt{X}@\alpha[\nearrow^p\searrow^q]$ occurs in the program. We further require $\mathcal{PV}$ and $\mathcal{QV}$ be bijective. In other words, object references with distinct IDs are associated with distinct up-step/down-step type variables.

Let us reiterate the relative nature of the walk index. By definition, it characterizes the referred object from the perspective of the referring object. For instance, if a class $\mathtt{X}$ has a field which is associated with type $\mathtt{Y}@\alpha[\omega]$, it says the walk index of any object stored in that field is ω from the perspective of the $\mathtt{X}$ object with the field.

Inference Rules. The most critical constraint for achieving HD is that q variables must be binary, which can be succinctly expressed by linear solvers. For a program, these q variables are clearly dependent. We define a type inference to relate type variables (both p and q variables) through linear constraints. Type inference rules are defined in Fig. 6, where $\Gamma \vdash e \backslash \mathcal{K}$ says that expression e yields constraints $\mathcal{K}$ under typing environment Γ. The definition of Γ is given in Fig. 3. Γ, Γ' is defined as the smallest Γ'' such that $\Gamma''(\mathtt{x}) = \Gamma'(\mathtt{x})$ if $\mathtt{x} \in \mathtt{dom}(\Gamma')$ or $\Gamma''(\mathtt{x}) = \Gamma(\mathtt{x})$ if $\mathtt{x} \notin \mathtt{dom}(\Gamma')$ and $\mathtt{x} \in \mathtt{dom}(\Gamma)$. Constraints are defined in Fig. 5. Note that all constraints are linear. We informally say "ω fresh" to mean p and q are fresh type variables where $\omega = \nearrow^p\searrow^q$.

The key to understanding this set of rules is the four definitions toward the bottom of Fig. 6. The constraint computed by $\mathtt{iC}$ is used by (T-New). Combined, it says that the down-step associated with an instantiation must be non-0. From Fig. 4, observe that a walk index whose down-step is 0 means a reference to its object scope (or scope of the scope, and so on). To have an object o instantiate an object o' that represents its scope

for all rules: $\Gamma(\mathtt{x}) = \mathtt{X}@\alpha_{\mathtt{x}}[\omega_{\mathtt{x}}], \Gamma(\mathtt{y}) = \mathtt{Y}@\alpha_{\mathtt{y}}[\omega_{\mathtt{y}}], \Gamma(\mathtt{z}) = \mathtt{Z}@\alpha_{\mathtt{z}}[\omega_{\mathtt{z}}]$

$$\text{(T-New)}\ \Gamma \vdash \mathtt{x} = \textbf{new}\ \mathtt{U}@\alpha[\omega] \backslash (\omega \Rightarrow \omega_{\mathtt{x}}) \cup \mathtt{iC}(\omega)$$

$$\text{(T-Assign)}\ \Gamma \vdash \mathtt{x} = \mathtt{y} \backslash \omega_{\mathtt{y}} \Rightarrow \omega_{\mathtt{x}}$$

$$\text{(T-Write)}\ \frac{\mathtt{U}@\alpha_{\mathtt{u}}[\omega_{\mathtt{u}}]\ \mathtt{fd} \in \mathtt{fields}(\mathtt{X}) \qquad \omega \text{ fresh}}{\Gamma \vdash \mathtt{x.fd} = \mathtt{y} \backslash (\omega_{\mathtt{y}} \Rightarrow \omega) \cup (\omega_{\mathtt{x}} \curvearrowright \omega_{\mathtt{u}} \circlearrowleft \omega)}$$

$$\text{(T-Read)}\ \frac{\mathtt{U}@\alpha_{\mathtt{u}}[\omega_{\mathtt{u}}]\ \mathtt{fd} \in \mathtt{fields}(\mathtt{Y}) \qquad \omega \text{ fresh}}{\Gamma \vdash \mathtt{x} = \mathtt{y.fd} \backslash (\omega \Rightarrow \omega_{\mathtt{x}}) \cup (\omega_{\mathtt{y}} \curvearrowright \omega_{\mathtt{u}} \circlearrowleft \omega)}$$

$$\text{(T-Msg)}\ \frac{\begin{array}{c} \mathtt{mtype}(\mathtt{md}, \mathtt{Y}) = \tau \rightarrow \tau' \qquad \tau = \mathtt{U}@\alpha_{\mathtt{u}}[\omega_{\mathtt{u}}] \qquad \tau' = \mathtt{V}@\alpha_{\mathtt{v}}[\omega_{\mathtt{v}}] \\ \omega, \omega', \omega'' \text{ fresh} \qquad \mathtt{mbody}(\mathtt{md}, \mathtt{Y}_i) = \mathtt{u}_i.\mathtt{v}_i.\Gamma_i.e_i \text{ for each } \mathtt{Y}_i <: \mathtt{Y} \\ [\textbf{this} \mapsto \mathtt{Y}_i@\alpha_{\mathtt{y}}[\omega], \mathtt{u}_i \mapsto \tau, \mathtt{v}_i \mapsto \tau'], \Gamma_i \vdash e_i \backslash \mathcal{K}_i \\ \mathcal{K}_{\mathtt{pm}} = (\omega_{\mathtt{z}} \Rightarrow \omega') \cup (\omega_{\mathtt{y}} \curvearrowright \omega_{\mathtt{u}} \circlearrowleft \omega') \\ \mathcal{K}_{\mathtt{rt}} = (\omega'' \Rightarrow \omega_{\mathtt{x}}) \cup (\omega_{\mathtt{y}} \curvearrowright \omega_{\mathtt{v}} \circlearrowleft \omega'') \end{array}}{\Gamma \vdash \mathtt{x} = \mathtt{y.md}(\mathtt{z})) \backslash \mathcal{K}_{\mathtt{pm}} \cup \mathcal{K}_{\mathtt{rt}} \cup \mathtt{thisC}(\omega) \cup \overline{\mathcal{K}}}$$

$$\text{(T-Cont)}\ \frac{\Gamma \vdash s \backslash \mathcal{K} \qquad \Gamma \vdash e \backslash \mathcal{K}'}{\Gamma \vdash s; e \backslash \mathcal{K} \cup \mathcal{K}'}$$

$$\begin{array}{rcl} \mathtt{iC}(\nearrow^{p}\searrow^{q}) & \stackrel{\text{def}}{=} & \{q = 1\} \\ \mathtt{thisC}(\nearrow^{p}\searrow^{q}) & \stackrel{\text{def}}{=} & \{p = 0, q = 0\} \\ \nearrow^{p_1}\searrow^{q_1} \Rightarrow \nearrow^{p_2}\searrow^{q_2} & \stackrel{\text{def}}{=} & \{p_1 - q_1 = p_2 - q_2, q_2 \geq q_1\} \\ (\nearrow^{p_1}\searrow^{q_1} \curvearrowright \nearrow^{p_2}\searrow^{q_2} \circlearrowleft \nearrow^{p}\searrow^{q}) & \stackrel{\text{def}}{=} & \{p = p_1 + p_2 - q_1, q = q_2, p_2 \geq q_1\} \end{array}$$

Fig. 6. Inference Rules and Definitions

directly violates the basic requirement of HD, because references to o (hence access) would exist outside its scope. The constraint computed by `thisC` is used in (T-Msg) for typing **this**. Combined, it says that the walk index of **this** should be solved to $\nearrow^{0}\searrow^{0}$. In figurative terms, an object can walk to itself on the HDT with no steps.

Operator $\omega \Rightarrow \omega'$ constrains the data flow from the object with ω to the object with ω'. Take (T-Assign) for example. The variables in $\omega_{\mathtt{y}}$ and $\omega_{\mathtt{x}}$ are related. Given the q variables binary, the constraints of $\omega_y \Rightarrow \omega_x$ allow for two cases: $\omega_{\mathtt{x}}$ and $\omega_{\mathtt{y}}$ are either solved to identical walk indices, or when $\omega_{\mathtt{y}}$ is solved to $\nearrow^{n}\searrow^{0}$, $\omega_{\mathtt{x}}$ can be solved to $\nearrow^{n+1}\searrow^{1}$. The latter is useful in cases of, say, assigning **this** to a variable defined in the same method. As data flows appear in every expression form, $\Rightarrow$ appears in every rule.

Ternary operator $\omega \curvearrowright \omega' \circlearrowleft \omega''$ computes the constraints for *walk index composition*. Let us imagine we have three objects α, α', α'' where α can walk to α' via walk index ω, α' can walk to α'' via ω', and α can walk to α'' via ω''. Operator $\omega \curvearrowright \omega' \circlearrowleft \omega''$ computes the necessary constraints to keep ω, ω', and ω'' consistent: after all, they are all defined over one HDT. For example in (T-Read), α above is the object enclosing the field read expression. α' is the object represented by `y`, and α'' is the object stored in field `fd` of `y`. The same definition is needed for (T-Write) and (T-Msg). For the latter, imagine one can view parameter passing as a write (to a stack variable) on the message receiver, and value returning as a read (from a stack variable) on the message receiver. Defining $\omega \curvearrowright \omega' \circlearrowleft \omega''$ is purely a graph-theoretic matter on encoding tree path composition, whiich we explain more in the Appendix.

Definition 8 (Whole-Program Constraints). *We use* $[\![P]\!]$ *to denote the constraints for program* P*. It is defined as* $\mathcal{K}$ *where* $\Gamma \vdash e \backslash \mathcal{K}$ *and* $P = \langle \overline{CL}; \Gamma; e \rangle$*.*

Optimizations. We do not formalize several standard optimization techniques implemented by our compiler. First, the formal system only requires α (and hence its associated *p* and *q*) be distinct for distinct type declarations in the source code. This is known as a 0CFA formulation in polymorphic type inference, or context-insensitive formulation in program analysis. Our compiler refines α as a pair including both the context label and the program point (for the type declaration), and two α's are distinct when either their context labels or program points are different. Second, in (T-Msg), we conservatively consider the method body of every subclass of the message receiver's declared class. In our implementation, a concrete class analysis is in place, so that only the classes whose instances may flow into the message receiver are inspected.

5 Computing and Grooming Heap Decomposition Tree

We now study the solutions to linear constraints in $[\![P]\!]$. We describe how these solutions can be used to compute the scope relation (*i.e.* construct an HDT), its soundness, and how the Cypress Principle can "groom" HDTs to a shape that reflects the principle of HD with optimality. The notion of "grooming" is achieved by preferring an HDT through linear programming. As we shall see, linear programming not only brings efficiency, but also allows us to express HD-related goals as objective functions.

We first introduce some basic notations on constraint solving. We use *placement* $\zeta : \mathbb{PVAR} \rightharpoonup \{0, 1, \dots\} \cup \mathbb{QVAR} \rightharpoonup \{0, 1\}$ to refer to the solution (sub-)vector of a linear constraint set. Recall in Sec. 4, we discussed that the solution to the up-step variable must be a non-negative integer, whereas the solution to the down-step variable must be 0-1 binary. We use predicate $\zeta \downarrow \mathcal{K}$ to denote ζ is a solution to $\mathcal{K}$, defined as every $\kappa \in \mathcal{K}[\zeta]$ is an arithmetic tautology, where $\mathcal{K}[\zeta]$ is standard constraint set substitution. Selection operator $\zeta \uparrow^n$ computes the set of up-step variables mapped to n in ζ. Formally, $\zeta \uparrow^n \stackrel{\text{def}}{=} \{p \mid p \mapsto n \in \zeta\}$.

5.1 From Linear Constraints to HDT

Intuitively, solving the constraints of a program P subsumes resolving all *p* and *q* variables associated with the **new** expressions in the program – such as $\alpha_{\texttt{X2}}$, $\alpha_{\texttt{Y3}}$, $\alpha_{\texttt{Z5}}$, $\alpha_{\texttt{Z9}}$

in Fig. 2(a) – to integers. With that, the relative position between the object instantiated by a **new** expression and the object whose class lexically encloses the **new** expression is known. Assuming all instantiation points are reachable from the bootstrapping code, an HDT can be constructed with these relative positions. We formalize this intuition in this section. From now on, since we are mainly concerned with instantiated points, we consider the abstract object IDs associated with instantiated points belong to set $\mathbb{ILAB}$, a subset of $\mathbb{LAB}$. We define convenience function $\mathtt{iA}(P)$ to enumerate all α's in program P where $\alpha \in \mathbb{ILAB}$. We further define $\mathtt{iP}(P)$ as $\{\mathcal{PV}(\alpha) \mid \alpha \in \mathtt{iA}(P)\}$.

For a program P, we define a simple *instantiation relation* – denoted as $\searrow_P$, or $\searrow$ for short – where $\alpha \searrow \alpha'$ says the instantiation point of α' is lexically enclosed by methods of α. It is defined as the smallest relation over $\mathbb{ILAB} \cup \{\alpha_{\mathtt{G}}\}$, s.t. (1) $\alpha_{\mathtt{G}} \searrow \alpha$ for every α appearing in the bootstrapping code of P; (2) $\alpha \searrow \alpha'$ for every α' whose instantiation expression appears in methods of abstract object α. We next define a relation that serves as a "candidate" HDT, as follows:

Definition 9 (Scope Candidate Relation). *Given a program P and $\zeta \downarrow [\![P]\!]$, a scope candidate relation of program P with respect to ζ is a smallest relation $\blacktriangleright^{\mathtt{sc}}$: $\mathtt{iA}(P) \cup \{\alpha_{\mathtt{G}}\} \times \mathtt{iA}(P)$ satisfying:*

$$\alpha \searrow_P \alpha' \textit{ and } \zeta(\mathcal{PV}(\alpha')) = n \textit{ and } \alpha_0 \blacktriangleright^{\mathtt{sc}} \alpha_1 \textit{ and } \alpha_1 \blacktriangleright^{\mathtt{sc}} \alpha_2 \textit{ and } \\ \ldots, \alpha_{n-1} \blacktriangleright^{\mathtt{sc}} \alpha \implies \alpha_0 \blacktriangleright^{\mathtt{sc}} \alpha'$$

Intuitively, the definition describes an algorithm to "construct" an HDT monotonically. Using informal terms, if ζ resolves a particular walk index to $\nearrow^3\searrow^1$, the definition here attempts to "put" 3 + 1 elements into $\blacktriangleright^{\mathtt{sc}}$, each of which (hopefully) represents an edge in HDT.

Is a scope candidate relation a (static) scope relation? We first demonstrate:

Lemma 2 ($\blacktriangleright^{\mathtt{sc}}$ as Rooted Tree Relation). *If $\blacktriangleright^{\mathtt{sc}}$ is a surjective scope candidate relation of program P with respect to ζ, then $\blacktriangleright^{\mathtt{sc}}$ is a rooted tree relation with root $\alpha_{\mathtt{G}}$.*

Here we are only interested in surjective scope candidate relations. Imagine a degenerate program where $\mathtt{x} =$ **new** $\mathtt{X}@\alpha[\nearrow^p\searrow^q]$ is the only expression in the bootstrapping code. The solution to p is unbound (*i.e.* any non-negative integer). In this case, a surjective scope candidate relation can only be constructed if $\zeta(p)$ is 0. One can view any non-surjective scope candidate relation as the result of attempting to place objects outside the global scope $\alpha_{\mathtt{G}}$. (As we shall see, this morbid case does not have relevance in practice, because they can be avoided by judicious settings of objective functions.)

Lemma 3 ($\blacktriangleright^{\mathtt{sc}}$ Uniqueness). *If $\blacktriangleright^{\mathtt{sc}}_1$ and $\blacktriangleright^{\mathtt{sc}}_2$ are both surjective scope candidate relations of program P with respect to ζ, $\blacktriangleright^{\mathtt{sc}}_1 = \blacktriangleright^{\mathtt{sc}}_2$.*

We provide a conventional definition for $\hookrightarrow^{\mathtt{s}}$ [8] where $\alpha \hookrightarrow^{\mathtt{s}} \alpha'$ is the standard points-to abstraction. With Lem. 2, we are able to demonstrate:

Lemma 4 ($\blacktriangleright^{\mathtt{sc}}$ as Scope Relation/HDT). *Given a program P and a surjective scope candidate relation $\blacktriangleright^{\mathtt{sc}}$ of P with respect to ζ, configuration $\langle \mathtt{iB}(P); \alpha_{\mathtt{G}}; \hookrightarrow^{\mathtt{s}}; \blacktriangleright^{\mathtt{sc}} \rangle$ is a static heap decomposition where $\hookrightarrow^{\mathtt{s}}$ is static access relation of P.*

Thus, a surjective scope candidate relation $\blacktriangleright^{\mathsf{sc}}$ is a *bona fide* scope relation, an HDT, and is the only HDT according to Lem. 3. From now on, we denote this unique scope candidate relation of program P w.r.t. ζ as $\mathtt{HDT}(P,\zeta)$. It is undefined if the scope candidate relation is not surjective.

5.2 Soundness of Cypress

We define a conventional small-step operational semantics [8] where $\hookrightarrow^{\mathsf{d}}$ is the standard reference relation at run time – as summarized in Fig. 1 – and $\mathcal{X}$ is the predictable mapping from objects and their abstract object IDs. With that, we can state the main soundness theorem of Cypress:

Theorem 1 (Soundness). *Given a program P and any ζ such that $\mathtt{HDT}(P,\zeta)$ if defined, $\mathtt{HDT}(P,\zeta)$ is sound.*

This important theorem tells us the HDT computed here – a data structure purely computed statically – *can* characterize dynamic heap decomposition. Also note that we never need to explicitly compute $\hookrightarrow^{\mathsf{s}}$ – the points-to information is implicit in the linear constraints except for the sake of stating Lem. 4 and constructing the proof for the theorem here – providing a solution to address Challenge IV.

5.3 Compositional Objects

In Java-like languages, evaluating the **new** expression yields a reference to the instantiated object, say o. This implies the reference to o can at least be obtained by the object o' whose method lexically encloses the **new** expression. We call o' the "scope ground zero" of o, *i.e.* the scope of o must either be o' or an outer scope of o'. Thus, the problem of determining whether an object o has o' as its scope can be converted as determining whether $[\![P]\!] \cup \{p = 0\}$ has solutions, where the instantiation expression is **new** $\mathtt{X}@\alpha[\nearrow^{p}\searrow^{q}]$. This is a standard problem for linear solvers, or if we phrase it in a slightly different way, a linear programming problem to minimize p over constraint set $[\![P]\!]$ and check whether the solution is 0.

Formally, we define objective function θ as a linear expression in the form of $p_1 + p_2 \cdots + p_k$ for some $k \geq 1$, which can be abbreviated as $\bigoplus_{j=1..k} p_j$. We represent an instance of linear programming to minimize an objective function θ over constraints $\mathcal{K}$ as $\min_{\mathcal{K}} \theta$. If ζ is the solution of the linear programming instance, *i.e.* $\zeta \downarrow \mathcal{K}$, then $\min_{\mathcal{K}} \theta$ is defined as the restriction of ζ to the domain formed by the variables that appear in θ. Thus, the following set computes the *compositional objects* in program P, *i.e.* the object that has its "scope ground zero" as the scope:

$$\{\alpha \mid \mathcal{PV}(\alpha) \in \biguplus_{p \in \mathtt{iP}(P)} (\min_{[\![P]\!]} p \uparrow^{0})\}$$

This strategy however implies we need to apply linear programming $|\mathtt{iP}(P)|$ times, clearly inefficient when the set of $[\![P]\!]$ is large. Cypress instead only performs linear programming *once*, through:

$$\{\alpha \mid \mathcal{PV}(\alpha) \in \min_{[\![P]\!]} \bigoplus_{p \in \mathtt{ip}(P)} p \Big|^{\uparrow 0} \}$$

This form is clearly more efficient, but does it produce the same result as the former? We answer it affirmatively, and the root reason is the *compositionality* of placements:

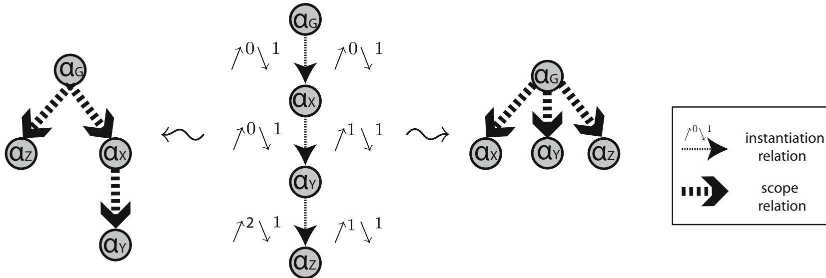
Lemma 5 (Local Placement Compositionality). $\min_{[\![P]\!]} \mathrm{p}_i = [\mathrm{p}_i \mapsto 0]$ *for* $i = 1, 2$ *iff* $\min_{[\![P]\!]} \mathrm{p}_1 + \mathrm{p}_2 = [\mathrm{p}_1 \mapsto 0, \mathrm{p}_2 \mapsto 0]$ *for any* P *and* $\mathrm{p}_1, \mathrm{p}_2 \in \mathtt{iP}(P)$.

In other words, the second strategy is an optimization of the first.

5.4 Cypress Grooming

The previous section is a primitive use of the Cypress Principle – it favors 0 solutions for the up-step variables of objects, *de facto* placing the object further away from the root. If an object indeed escapes from its scope ground zero, what should be its "minimal escape"? The Cypress Principle guides linear programming for this task, and we term the resulting preferred HDT a *cypress*.

One possible solution is to design a "minimizing all" approach through linear programming, *i.e.* $\min_{[\![P]\!]} \bigoplus_{p \in \mathtt{ip}(P)} p$. This however may not yield the expected "tall and skinny" cypress. As a counterexample, consider a simple scenario below, where the graph in the center denotes a $\searrow$ relation for a program P, and the label on each edge is the walk index of the head from the perspective of the tail. Observe there might be two different solutions of $[\![P]\!]$ but both satisfy $p + p' + p'' = 2$:



The Cypress Principle – grooming the tree tall and skinny – favors the left tree instead of the right one, but this preference cannot be expressed here. The root of this problem is that the walk index from $\alpha_{\mathtt{x}}$ to $\alpha_{\mathtt{y}}$ and that from $\alpha_{\mathtt{y}}$ to $\alpha_{\mathtt{z}}$ are dependent: a lesser up-step value for an object closer to the root not only helps place itself further away from the root on the HDT, but also helps place the objects it can reach through $\searrow$. Building on this insight, we minimize the sum of the up-steps for each object *relative to the root*. For the example above, note that the (combined) up-steps for $\alpha_{\mathtt{x}}$ relative to $\alpha_{\mathtt{G}}$ is p, and the combined up-steps for $\alpha_{\mathtt{y}}$ relative to $\alpha_{\mathtt{G}}$ is $p + p'$, and the combined up-steps for $\alpha_{\mathtt{z}}$ relative to $\alpha_{\mathtt{G}}$ is $p + p' + p''$. Thus, the objective function to minimize is $(p) + (p + p') + (p + p' + p'')$. Formally, let $\Delta_{P,\alpha}$ denote a set $\{p_1, \ldots, p_n\}$ where

$\alpha_{\texttt{G}} \searrow \alpha_1, \ldots, \alpha_{n-1} \searrow \alpha_n$ and $\mathcal{PV}(\alpha_i) = p_i$ for $i = 1, \ldots, n$. We can now compute the cypress as follows:

Definition 10 (Cypress). *The cypress of program P, denoted as* $\texttt{cypress}(P)$*, is defined as* $\texttt{HDT}(P, \zeta)$*, where*

$$\zeta = \underset{[\![P]\!]}{\textbf{\textit{min}}} \bigoplus_{\alpha \in \texttt{iA}(P)} \bigoplus_{\mathrm{p} \in \Delta_{P,\alpha}} \mathrm{p}$$

It would not be difficult to show the scope candidate relation w.r.t. ζ is surjective, and hence $\texttt{HDT}(P, \zeta)$ is defined. We next state an important theorem that says that the cypress we produce is a "tall and skinny" HDT whose aggregated depth is the greatest:

Theorem 2 ("Tall and Skinny"). *Given program P and some ζ such that* $\texttt{HDT}(P, \zeta)$ *is defined,* $\sum_{\alpha \in \texttt{iA}(P)} \texttt{depth}(\alpha, \texttt{HDT}(P, \zeta)) \leq \sum_{\alpha \in \texttt{iA}(P)} \texttt{depth}(\alpha, \texttt{cypress}(P))$.

6 Implementation and Evaluation

A prototype implementation of Cypress has been built on top of the Polyglot framework 2.4 [30]. The compiler supports inheritance, dynamic dispatch, super calls, method/constructor overloading, static fields, static methods, multi-dimensional arrays, inner classes, generics, reflection, autoboxing/unboxing, and enum types. The implementation of Java 1.5 features is modified from a Polyglot extension at UCLA[3]. We choose an open-source linear solver `lpsolve`[4] for constraint solving. Our compiler produces the tree data of the cypress in XML, subsequently rendered by Prefuse[5].

Cypress implements a (field-sensitive) k-object context-sensitive algorithm [25]. To create a stress test for scalability, we choose an expensive form: k in our algorithm is not fixed to a (usually small) constant. The entire chain of instantiation site labels is preserved to represent distinct contexts, except that when recursion happens, we only use the chain of labels between two occurrences of the reappearing label. This technique was used in some compilers previously developed by us [19,7].

Our benchmarks are selected from diverse sources: `puzzle` (PU) [19] solves a famous 4x4 sliding puzzle; `montecarlo` (MO) is from the JavaGrande suite [32]; `jspider` (JS)[6] is an open-source web robot; `messadmin` (ME)[7] is a Java HTTP session monitor; `lusearch` (LU) is from DaCapo[8] suites; `cypress+polyglot` (CY) is our compiler itself. The last benchmark is a meta-circular effort to help us validate the correctness of our implementation. We find this approach very useful for discovering implementation bugs, as we are familiar with the minute details of our own software, and whether the details of its cypress "makes sense" can be quickly examined.

[3] `http://www.cs.ucla.edu/~todd/research/polyglot5.html`
[4] `http://lpsolve.sourceforge.net/`
[5] `http://prefuse.org/`
[6] `http://j-spider.sourceforge.net/`
[7] `http://messadmin.sourceforge.net/`
[8] `http://dacapobench.org/`

name	#LOC	#RM	#I	#F	#V	#EQ	GT(s)	IT (s)	FT (s)
PU	882	38+1000	314	42+37	572	765	0.49	0.69	0.45
MO	3,128	131+984	294	33+19	206	278	0.48	0.59	0.40
JS	13,986	1434+4329	1761	1557+173	35,134	54,598	45	28.79	24.65
ME	65,356	2144+1973	903	556+191	16,168	22,789	4	6.07	4.01
LU	112,649	6077+2958	1718	657+1086	38,842	72,731	65	39.37	34.84
CY	118,309	4935+14009	6406	4646+1398	107,172	120,185	470	205.98	188.98

Fig. 7. Benchmarking Results (#*LOC*: program LOC; #*RM*: the number of reachable methods (two parts: application methods + library methods); #*I*: the number of distinct instantiation points; #*F*: the number of accessed fields (two parts: non-**private** fields + **private** fields); #*V*: the number of type variables generated by Cypress; #*EQ*: the number of linear equations generated by Cypress; *GT*/*IT*/*FT*: time, see text for details)

All experiments are conducted on Intel Core Duo 2.53GHz with 4GB RAM, with data reported in Fig. 7. All #*RM*, #*I*, and #*F* data are reported context-sensitively. To see the effect of the stress test we created, note that `cypress+polyglot` reports 6406 instantiation points, significant among the state of the art of program analysis. In comparison, the largest #*I* counts are 1261 in [24] and 4152 in [22]. We construct two experiments for validation, one for generating cypresses and the other for finding compositional fields. (The first task subsumes the goal of finding compositional objects.) The last three columns report the time (in seconds) for the two experiments. *GT* reports the time used for all compilation steps other than constraint solving, shared by both experiments. *IT* and *FT* are constraint solving time for the two experiments respectively.

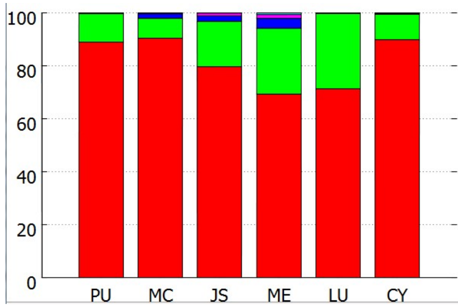


Fig. 8. Distance to "ground zero" (Red: compositional; Green: escape by 1 scope; colors upwards: escape by 2/3/4 scopes)

Encapsulated Objects vs. Escaped Objects. Fig. 8 provides a normalized distribution of the number of distinct instantiation points escaping from "ground zero," computed via Def. 10. Across all benchmarks, the vast majority of the objects (70%-80%) stay "encapsulated" or "owned" within "ground zero." More rigorously, this means any object in this category is never accessed in any outer scope (Def. 1) of the object whose methods or field initializers include the instantiation expression of the former. When objects do not fall into this category, which we intuitively say they are "escaped," it is rare that they escape by more than one scope. This clearly does not result from the lack of scopes in the cypresses – the cypress heights for the benchmarks are 6, 6, 11, 10, 6, 10, respectively. We believe it demonstrates the compositional nature of the heap.

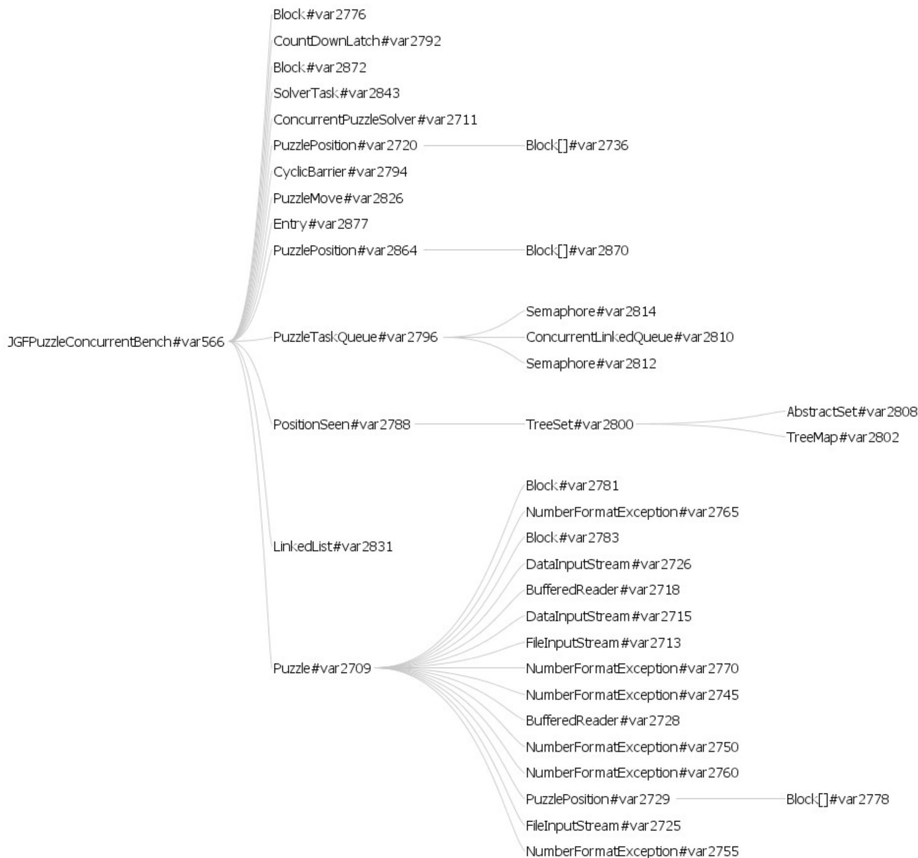


Fig. 9. (Partial) Cypress for PU Benchmark (Best View: Scale to 200%)

Cypress Graphs for Program Understanding. The cypresses of all benchmarks have been rendered in a graphic tree form. Here we include a partial cypress of the smallest benchmark in Fig. 9, showing only the subtree that focuses on application objects (*i.e.* non-library objects). The cypress is rendered with the root on the left, with each edge intuitively says the left-hand node is the scope of the object of right-hand node. More rigorously, there is an edge with α on the left and α' on the right iff $\alpha \blacktriangleright^{s} \alpha'$.

Each node on the cypress is labeled in the form of C#v, where C is the class name of the represented object, and v is the abstract object ID to uniquely identify the object. Observe that in a context-sensitive algorithm, it is insufficient to identify an object by class names (or the instantiation program points for that matter). Not presented here, the Cypress tool further accompanies the rendered cypress with a table to associate each v with context labels – a chain of instantiation program points that ultimately lead to the instantiation of the object represented by v.

The cypress offers a vivid representation of the heap decomposition structure latent in object-oriented programs: how the heap is decomposed from the root, level by

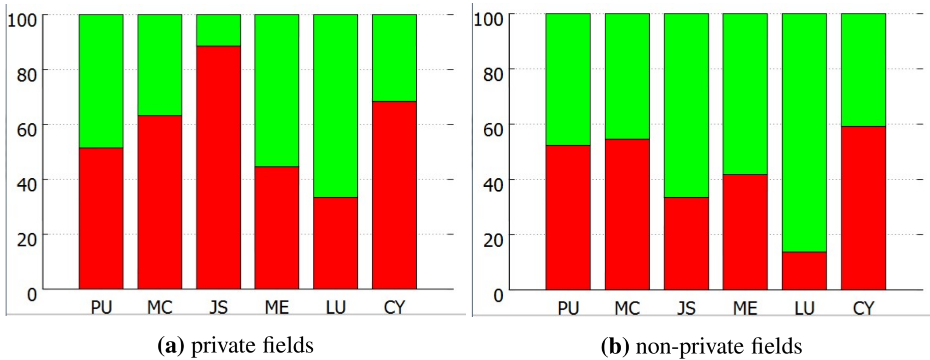
level, as we move from left to right. Observe that the `PuzzleTaskQueue#var2796` object successfully encapsulates its internal linked list representation, an object labeled `ConcurrentLinkedQueue#var2810`. As another example, a `Puzzle#var2709` object encapsulates the `PuzzlePosition#var2729` object, which in turn encapsulates the `Block[]#var2778` array.

The cypresses of all benchmarks are online for review [8]. As the $\#I$ numbers suggest, some of the trees are large.

Compositional Fields. The problem of *compositional fields* is to determine whether a field `fd` of an object α in program P always refers to objects that are never accessed outside the scope formed by α. This is particularly relevant for **private** fields, as a non-compositional **private** field may be counter-intuitive to programmers [22,24,35].

Cypress determines field compositionality as follows. First, observe that each field is declared with a type, which is also associated with a walk index. Let us denote the set of all the up-step variables of such walk indices as $\mathtt{fp}(P)$ for program P. If the object stored in the field is local to the scope formed by α, the walk index from α to the stored object must resolve to $\nearrow^0\searrow^1$ or $\nearrow^0\searrow^0$. Following a similar formulation as compositional objects (Sec. 5.3), the set of all compositional fields for program P can be computed by only one instance of linear programming:

$$\{\alpha \mid \mathcal{PV}(\alpha) \in \min_{[\![P]\!]} \bigoplus_{p \in \mathtt{fp}(P)} p \,\Big|^{\nearrow 0} \}$$



(a) private fields **(b)** non-private fields

Fig. 10. Field Compositionality (Compositional? Red / Green : Yes/No)

We next show some experimental results on compositional fields. In Fig. 7, $\#F$ is the number of accessed fields, divided by non-**private** ones and **private** ones. *FT* is the time (in seconds) for finding all composition fields through linear programming. Fig. 10 shows the normalized distribution of field compositionality. Benchmarks are placed from left to right in the same order as in the table earlier. Fig. 10 (a) shows a

phenomenon consistent with previous findings [24,22] – a **private** declaration does not always guarantee compositionality. Our system reports a slightly higher percentage of compositional fields (a difference of 5%-10% on average). Other than the possibility of Cypress being a more precise algorithm, we speculate this may result from our precise settings of context sensitivity thanks to the scalability of linear programming. We also constructed experiments for non-**private** fields, with results in Fig. 10 (b). As expected, the overall percentage of compositional fields has decreased, but interestingly, we found a significant percentage of them are compositional. This might result from the fact that the default modifier choice of Java is not **private**. Programmers uninterested in visibility protection are in fact declaring non-**private** fields.

JATO Support. We have extended Cypress to support language features important for multi-threaded programs – such as representing each thread as a node on the cypress, and differentiating read/write access to fields – and apply the extended tool to assist JATO [20], a system for atomicity enforcement across the JNI boundary. In that extension, any descendent object of the thread node on the cypress is guaranteed to be thread-local according to Java-side analysis. If such an object crosses the boundaries of JNI and its access on the C side falls into some simple patterns (such as only being read/written), the object is statically guaranteed to be thread-local; no locks are needed for their atomicity enforcement. This application of Cypress is a concrete example of a well-established direction: reasoning about thread locality with ownership (*e.g.*[35]).

7 Related Work

The most advanced direction related to our work is ownership type inference, which can be achieved either through dynamic analysis [28,34] or static analysis [10,26,17]. Dynamic approaches are sound only w.r.t. the (finite number of) executions the analysis is performed over, but on the positive side, they can often offer insight into complex structures of programs beyond ownership (an example would be a "butterfly" in [28]). Dietl et. al. designed a tunable static analysis [10] for inferring the modifiers of Universe Types (UT) [11]. One feature of their system is that the inference is reduced to an SAT satisfiability problem, where practical solvers exist. The high-level vision – reducing a non-standard problem into a standard problem – is shared by Cypress, as we reduce our problem to linear programming. As an ownership type system, UT is known to have some distinctive features: it does not conform to deep ownership, but considers a form of interaction between immutability and ownership (known as owners-as-modifiers). As a result, the inference built for UT differs with ours in the underlying invariant. Milanova and Vitek designed dominator inference [26], a type inference for modifiers of Ownership Types (OT) [29,6,5]. OT supports deep ownership, so OT inference is closer in goal to Cypress. Their approach assumes a pre-computation of the points-to set. Huang et. al. [17] designed a unified approach to infer modifiers for both UT and OT. Their system allows the preference over different modifiers to be expressed through a ranking, and an iterative inference is constructed to place priority on higher ranked modifiers. None of the cited systems uses linear programming.

Points-to analyses have been designed to address two related problems: UML composition inference [24] and field composition identification [22]. Explained in our terminology, this line of work first assumes the pre-knowledge of $\hookrightarrow^{s}$ – points-to information from standard points-to analyses – and then (re-)analyzes programs with two tasks: satisfying $\mathtt{ehd}^{s}$ and guaranteeing soundness. Cypress demonstrates that a type inference can be constructed for sound static heap decomposition with no need for full knowledge of points-to information. In general, systems in this category may take a very different approach from ours, but they share our goal of constructing a fully automated static analysis.

Language designs for ownership support generally have some inference ability to reduce annotation overhead (*e.g.*[2]), but the goal is not to analyze annotation-free programs. An inference based on Confined Types [14] analyzes unannotated Java code, with a premise that Java packages serve as the scope for objects defined inside the package. Scope Types [3] is an ownership type system lightweight on annotation overhead by design. Our previous work Pedigree Types [21] allows programmers to optionally declare "pedigrees" as type modifiers, a language design incarnation of walk index. Pedigree Types has the ability to infer some type modifiers elided by the programmer. In that light, Cypress considers the extreme case where all pedigree modifiers are ignored – a case Pedigree Types trivially (but unhelpfully) answers "it typechecks" – and demonstrates how linear programming can help precisely recover them. Overall, the route of language design and the route of program analysis for static heap decomposition are complementary.

8 Conclusion

This paper studies heap decomposition, and describes a static ownership type inference that can offer vivid insight into the dynamic nature of software: the compositional view of the runtime heap. In the future, we plan to apply Cypress to two application domains: thread locality for optimization and cache locality for energy efficiency.

Cypress has been implemented as an open-source tool and can be downloaded [8]. The technical report at the same website contains the operational semantics, the proofs, and the cypress graphs of all benchmarks.

Acknowledgments. We thank the reviewers for their thorough and insightful comments, Xiangjin Xu and Ashish Agarwal for their suggestions on linear programming, Scott Smith for earlier discussions on Pedigree Types, Tom Bartenstein for proof reading, and Leena Joseph, Nandan Samant, and Jacob Strohm for developing helper software modules being integrated to our compiler. This work was partially supported by a Google Faculty Award and NSF Award No. CCF-1054515.

References

1. Aldrich, J., Chambers, C., Sirer, E.G., Eggers, S.: Static analyses for eliminating unnecessary synchronization from Java programs. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 19–38. Springer, Heidelberg (1999)

2. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA 2002, pp. 311–330 (2002)
3. Andreae, C., Coady, Y., Gibbs, C., Noble, J., Vitek, J., Zhao, T.: Scoped types and aspects for real-time Java. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 124–147. Springer, Heidelberg (2006)
4. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
5. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (July 2001)
6. Clarke, D.G., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA, pp. 48–64 (1998)
7. Cohen, M., Zhu, H.S., Emgin, S.E., Liu, Y.D.: Energy types. In: OOPSLA 2012 (October 2012)
8. Cypress, `http://www.cs.binghamton.edu/~davidl/cypress/`
9. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM 9(3), 143–155 (1966)
10. Dietl, W., Ernst, M.D., Müller, P.: Tunable static inference for generic universe types. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 333–357. Springer, Heidelberg (2011)
11. Dietl, W., Müller, P.: Universes: Lightweight ownership for jml. Journal of Object Technology 4(8), 5–32 (2005)
12. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
13. Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI 2005, pp. 47–58 (2005)
14. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: OOPSLA 2001, pp. 241–255 (2001)
15. Grunwald, D., Zorn, B., Henderson, R.: Improving the cache locality of memory allocation. In: PLDI 1993, pp. 177–186 (1993)
16. Guéhéneuc, Y.-G., Albin-Amiot, H.: Recovering binary class relationships: putting icing on the uml cake. In: OOPSLA 2004, pp. 301–314 (2004)
17. Huang, W., Dietl, W., Milanova, A., Ernst, M.D.: Inference and checking of object ownership. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 181–206. Springer, Heidelberg (2012)
18. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java - a minimal core calculus for Java and GJ. In: TOPLAS, pp. 132–146 (1999)
19. Kulkarni, A., Liu, Y.D., Smith, S.F.: Task types for pervasive atomicity. In: OOPSLA 2010 (October 2010)
20. Li, S., Liu, Y.D., Tan, G.: JATO: Native code atomicity for Java. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 2–17. Springer, Heidelberg (2012)
21. Liu, Y.D., Smith, S.: Pedigree types. In: IWACO 2008, pp. 63–71 (July 2008)
22. Ma, K.-K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: OOPSLA 2007, pp. 423–440 (2007)
23. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 125–140 (2010)
24. Milanova, A.: Precise identification of composition relationships for uml class diagrams. In: ASE 2005, pp. 76–85 (2005)
25. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. TOSEM 14, 1–41 (2005)

26. Milanova, A., Vitek, J.: Static dominance inference. In: TOOLS (49), pp. 211–227 (2011)
27. Miller, M.S., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 195–229. Springer, Heidelberg (2005)
28. Mitchell, N.: The runtime structure of object ownership. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
29. Noble, J., Potter, J., Vitek, J.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
30. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
31. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39 (2003)
32. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Supercomputing 2001 (2001)
33. Su, C.-L., Despain, A.M.: Cache design trade-offs for power and performance optimization: a case study. In: Proceedings of the 1995 International Symposium on Low Power Design, ISLPED 1995, pp. 63–68. ACM, New York (1995)
34. Wren, A.: Ownership type inference. Master's thesis, Imperial College (2003)
35. Wrigstad, T., Pizlo, F., Meawad, F., Zhao, L., Vitek, J.: Loci: Simple thread-locality for Java. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 445–469. Springer, Heidelberg (2009)

Appendix: Walk Index Composition Constraints

In this section, we offer an explanation to the constraints used to capture walk index composition, as defined by $\omega_1 \curvearrowright \omega_2 \circlearrowleft \omega$ in Fig. 6. The relationship among $\omega_1, \omega_2, \omega$ is the standard problem of tree path composition. Not to lose generality, there are only two subcases, depending on the comparison between the ω_1's down-step (q_1) and ω_2's up-step (p_2):

- (a) If $q_1 \leq p_2$, then $p = p_1 + p_2 - q_1$ and $q = q_2$.
- (b) If $q_1 > p_2$, then $p = p_1$ and $q = q_1 - p_2 + q_2$.

For (b), observe that to satisfy $q_1 > p_2$, any solution ζ must have $\zeta(q_1) = 1$ and $\zeta(p_2) = 0$. In this case, the only way to satisfy $q = q_1 - p_2 + q_2$ is $\zeta(q_2) = 0$. In other words, ω_2 must be solved to $\nearrow^0\searrow^0$, with the only satisfiable object reference being **this**. Note that in Fig. 6, in all uses of $\omega_1 \curvearrowright \omega_2 \circlearrowleft \omega$, ω_2 is a walk index associated with a method formal parameter, a method formal return type, or a field declaration. Allowing them to be inferred with a walk index $\nearrow^0\searrow^0$ is hardly useful for HD inference. The constraints defined for $\omega_1 \curvearrowright \omega_2 \circlearrowleft \omega$ in Fig. 6 only considers (a). (Observe however, **this** can still be passed as a method parameter, method return value, or stored in fields, except that the walk index of the associated formal paramter, return type, or field type would be solved to $\nearrow^1\searrow^1$.)