Pedigree Types

Yu David Liu Scott F. Smith

Department of Computer Science
The Johns Hopkins University
{yliu, scott}@cs.jhu.edu

Abstract

Pedigree Types are an intuitive ownership type system requiring minimal programmer annotations. Reusing the vocabulary of human genealogy, Pedigree Types programmers can qualify any object reference with a pedigree – a **child**, **sibling**, **parent**, **grandparent**, *etc* – to indicate what relationship the object being referred to has with the referant on the standard ownership tree, following the owners-as-dominators convention. Such a qualifier serves as a heap shape constraint that must hold at run time and is enforced statically. Pedigree **child** captures the intention of encapsulation, *i.e.* ownership: the modified object reference is ensured not to escape the boundary of its parent. Among existing ownership type systems, Pedigree Types are closest to *Universe Types*. The former can be viewed as extending the latter with a more general form of pedigree modifiers, so that the relationship between any pair of objects on the aforementioned ownership tree can be named and – more importantly – inferred. We use a constraint-based type system which is proved sound via subject reduction. Other technical originalities include a polymorphic treatment of pedigrees not explicitly specified by programmers, and use of linear diophantine equations in type constraints to enforce the hierarchy.

1. Introduction

In this paper we develop Pedigree Types, a static ownership type system with a novel vocabulary for declaring how one heap node is related to another on the ownership tree. The most common relationships between nodes in a tree are already well-named in human genealogy: parents, children, uncles, siblings, great grand-uncles, *etc*. Our type system borrows this vocabulary and gives an object the ability to say, "this reference must point to my **child**" or "that reference must point to my **sibling**". For the entire heap, all invariants associated with all objects are ensured to be consistent globally, so that a dynamic hierarchy can be statically enforced. From the view of genealogy, the global consistency of object pedigrees can be viewed as the case where "no incest exists": one's grandfather cannot also be the grandfather of one's parent. The standard property of the ownership tree – aliases to an object cannot be obtained by objects outside the boundary of its parent on the tree – is also enforced by outlawing encapsulation-breaking pedigrees such as **nephew**.

The majority of existing ownership type systems – such as [NPV98, CPN98, Cla01] – achieve the same goal, but via a very different approach. They often make use of an explicit form of parametric polymorphism by using parameterized classes. For instance, a `Dialog` class can be defined as:

```
class Dialog <t1, t2, t3> {
    Data<t2, t3> d;
    …
}
```

Here, an ownership type, such as `Data<t2, t3>`, is formed by annotating the object type with a sequence of *context parameters*, where the first parameter `t2` denotes the owner of the object `d`, and the rest (`t3` here) are parameters to be forwarded down into the implementation of `Data`, a parameterized class in the same fashion as `Dialog`.

Pedigree Types aim to minimize the need for annotation overhead by programmers. First there is no requirement to conceive and declare these context parameter lists above and the associated parameter forwarding. In real-world programs, the number of context parameters can be quite large, and errors can arise if even a small mistake is made in this forwarding process; and, the programmer is distracted from the primary programming task at hand due to her need to focus on these type system minutae. Furthermore, all pedigree declarations such as **child** and **sibling** in Pedigree Types are optional: programmers only need to declare a pedigree when they care about the invariant; the other object types are treated as having polymorphic pedigree, and are inferred and constrained based on usage. The human pedigree analogy of non-qualified types is "this person is related to me, but I do not care how, as long as it does not involve incest." We believe a principle of minimal declarations is important to the practical viability of ownership type systems: if we attempt to go beyond toy programs and deal with programs containing hundreds or thousands of objects, having inference of pedigree as the default mode frees the programmer to add declarations only where they really matter.

Pedigree types also make the ownership hierarchy more explicitly visible to programmers. Hierarchical decomposition is a fundamental principle for controlling software complexity. With ever larger-scale systems becoming ever more common, the importance of decomposing systems into well-structured hierarchies to manage complexity cannot be overstated. There is a long history of programming language designs to conquer complexity in a hierarchical fashion, primarily the development of module systems. While module systems effectively form a codebase hierarchy, they are less effective at making hierarchical distinctions in the run-time heap structure. This hierarchical heap structure nonetheless pervasively exists: a `Main` object may intuitively be "composed" of a `View` object, a `Model` object and a `Controller` object via the MVC pattern [KP88], and the `View` object may be composed of several `Dialog` objects, and each `Dialog` object composed of several `Button`'s. Intuitively, this compositional structure can be viewed as a tree reference backbone "overlay" on the general heap reference graph. Giving programmers the power to explicitly define and reason about dynamic heap hierarchies helps promote program understanding: it already exists in their conception of the software, and bringing it out in the code itself will help refine and clarify the compositional structure of the heap. For existing ownership type systems using explicit parametric polymorphism, the structure of the ownership tree depends on how context parameters are forwarded, and it is much less visible to programmers when com-

pared with Pedigree Types. For instance, we cannot tell from the code snippet above the relationships between the `Dialog` object and the `Data` object on the tree – that depends on how `t1` and `t2` are instantiated and forwarded.

With the combination of the two advantages above – minimal annotation overhead and explicit shaping of hierarchical decomposition – Pedigree Types have the potential to allow the original hierarchical decomposition intention of the design, lost during the Java implementation stage, to be rediscovered. This is possible because, when all annotations are left out, Pedigree Types *de facto* are transformed into an annotation-free inference system for programs in a Java-like language. We will discuss more about this topic in Sec. 4.

A major inspiration for Pedigree Types comes from Universe Types [MPH01]. Two keywords for modifying object references, `peer` and `rep`, are provided to Universe Types programmers, analogous to what we call a **sibling** and a **child** on the pedigree tree. From this perspective, Pedigree Types aim to extend the philosophy of Universe Types to a more general form for representing pedigrees beyond `peer` and `rep`, so that any pedigree on the tree can be easily referred to. Additionally we statically infer pedigrees and thus avoid the run-time overhead of casting some unknown pedigree to either `peer` or `rep` as is done in Universe Types.

In the rest of the paper, Sec. 2 gives an informal description of the basic ideas of Pedigree Types, which will be formalized in Sec. 3. Sec. 4 describes a number of extensions and Sec. 5 discusses related work. We summarize our contributions in Sec. 6.

2. Informal Overview

We now informally introduce the key innovations of our type system. Pedigree Types are a general static type system which can be built on top of any programming language with arbitrary heap reference structures. This paper describes how they may be built on top of Java-like languages; see [Liu07] for how they may be built upon an alternative object model, *Classages* [LS05].

Fig. 1 gives a code snippet involving a `Dialog` object with two `Button`'s, followed by the UML object diagram for a possible run-time snapshot. The `Dialog` logs important actions in two `Logger` objects, one of which (`publog`) is shared with the rest of the application, logging important actions such as when an `OK` button is pressed, while the other (`privlog`) is only used to log actions with limited impact during the lifecycle of the `Dialog`. A `Ctrl` object controls the behaviors of the GUI objects. In the diagram, labels on object associations are pedigree relationships. Only object associations pertinent to the discussion are shown.

2.1 Hierarchy Shaping with Pedigree Types

The General Form A pedigree type is a regular object type prefixed with a pedigree qualifier, specifying where on the hierarchy an object should sit relative to the current instance. Informally, when a field or a variable has a pedigree type whose pedigree qualifier is `X`, we also say the field or the variable has pedigree `X`.

In fact, qualifiers like **sibling** and **child** are just sugared syntax for special cases of a more general form:

$$\overbrace{(\textbf{parent})\ldots(\textbf{parent})}^{\mathtt{w}}\overbrace{(\textbf{child})\ldots(\textbf{child})}^{\mathtt{z}}$$

where $\mathtt{w} \in \{0, 1, 2, \ldots\}$ and $\mathtt{z} \in \{0, 1\}$. The invariant such a qualifier enforces is that, on the hierarchy the qualified object must be the $\overbrace{\textbf{parent}\text{'s }\textbf{parent}\text{'s}\ldots\textbf{parent}\text{'s}}^{\mathtt{w}}\ \overbrace{\textbf{child}\text{'s}\ldots\textbf{child}}^{\mathtt{z}}$ of the current object. We call `w` the *positive level* of the object being qualified and `z` the *negative level* of the object being qualified. We use the abbreviation $(\textbf{parent})^{\mathtt{w}}(\textbf{child})^{\mathtt{z}}$ for the above definition.

```
class Main {
  void main () {
    EventSource es = new sibling EventSource();
    Ctrl ctrl = new child Ctrl();
    Dialog d = new child Dialog(es, ctrl);
    d.init();
    ...
  }
}
class Dialog {
  EventSource es; Ctrl ctrl;
  sibling Logger publog; child Logger privlog;
  Dialog(EventSource es, Ctrl ctrl)
    { this.es = es; this.ctrl = ctrl; }
  void init() {
    Button ok = new child Button("OK", es, ctrl);
    Button cncl = new child Button("NO", es, ctrl);
    publog = new Logger("Shared Logger");
    privlog = new Logger("Private Logger");
    ok.init(publog, this);
    cncl.init(privlog, this);
  }
  void refresh() { ...}
}
class Button {
  Logger logger; EventSource es; String name; Ctrl ctrl;
  parent Dialog container;
  Button(String name, EventSource es, Ctrl ctrl)
    { this.name = name; this.es = es; this.ctrl = ctrl;}
  void init(Logger logger, Dialog container)
    {this.logger = logger; this.container = container;}
  void log() { ...}
  void refresh() { ...container.refresh(); }
  void oops() { // ctrl = new child Ctrl(); }
  void oops2() { // ctrl.oops3(logger); }
}
class Logger { ...}
class EventSource { ...}
class Ctrl { void oops3(Logger log) { ...} ...
}
```

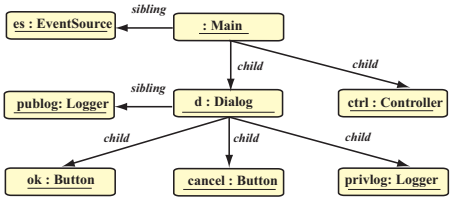

Figure 1. An Example

When the negative level is zero, we can abbreviate the pedigree as simply $(\textbf{parent})^{\mathtt{w}}$.

Why This General Form? The general form elegantly covers the more familiar cases via the following sugar:

$$\begin{aligned}
\textbf{child} &\stackrel{\text{def}}{=} (\textbf{parent})^0(\textbf{child})^1\\
\textbf{self} &\stackrel{\text{def}}{=} (\textbf{parent})^0(\textbf{child})^0\\
\textbf{sibling} &\stackrel{\text{def}}{=} (\textbf{parent})^1(\textbf{child})^1\\
\textbf{parent} &\stackrel{\text{def}}{=} (\textbf{parent})^1(\textbf{child})^0\\
\textbf{grandparent} &\stackrel{\text{def}}{=} (\textbf{parent})^2(\textbf{child})^0\\
\textbf{aunt} = \textbf{uncle} &\stackrel{\text{def}}{=} (\textbf{parent})^2(\textbf{child})^1
\end{aligned}$$

Declaring an object to have a **child** pedigree aligns with the programmers' intention that the object is encapsulated. We will elaborate on this case, and why in the general case `z` ranges over $\{0, 1\}$ in Sec. 2.2.

Pedigree **parent** provides a strong enforcement of nesting: the `container` field of the `Button` object is declared to have pedigree **parent**. Thus, if at any time the `Button` object's `refresh` method is invoked, which in turn needs to refresh its container `Dialog` object (a common practice in GUI programming), our type system can guarantee that the refreshed `Dialog` is indeed the one containing the `Button` object itself. Observe that a type with **parent** pedigree is a singleton type, since each object can only have one **parent**. All pedigree types with negative level 0 are singleton types, such as **grandparent**. Existing ownership type systems that we know of do not have the equivalent of singleton pedigrees to precisely declare a reference pointing to a strict ancestor on the ownership tree.

Pedigree **self** provides a precise way to type self-references (Java's **this**). This qualifier is also a singleton type qualifier. The importance of typing **this** in a more precise way lies in the consequences of its propagation: when an object passes its **this** to others – a common programming idiom – the receiver side has the opportunity to precisely type the argument with singleton type qualifiers as well. For instance in the example, the `Button` object can have its `container` field hold a value of **parent** singleton pedigree: it is passed by `Dialog`'s method `init`, where **this** is passed. (The reason that a **parent** pedigree in the `Button` object matches the **self** pedigree in the `Dialog` object will be explained shortly.) Another pleasant consequence of precisely typing Java's **this** is the call-back constraints of object-oriented programming can be captured. In Java, programmers typically rely on passing **this** to another object to implement callbacks. The Java type system however cannot ensure the value being passed is indeed **this** so the callback constraint is not enforceable.

Property of Pedigree Relativization A key property of pedigrees is that they can be relativized from the perspective of one object to that of another. Let us revisit the example we brought up to explain **self**. The `Button` object's `init` method expects an object of **parent** as its third argument `dialog`, while the object being passed in is **this**, which is of pedigree **self**. These qualifiers are not the same, but pedigree types are always declared *relative to the current instance*, and the `Button` object is a **child** of the `Dialog` object. From the `Dialog` object's perspective, it easily knows the **parent** from its **child**'s perspective is a **self** from its own perspective. The line `ok.init(publog,this)` thus typechecks.

Pedigree Subsumption Intuitively, oneself is a special case of her parent's children, *i.e.* a pedigree $(\textbf{parent})^0(\textbf{child})^0$ is a special case of $(\textbf{parent})^1(\textbf{child})^1$. Similarly, a parent is a special case of a grandparent's children.

In the general case, a pedigree $(\textbf{parent})^{\mathtt{w}}(\textbf{child})^0$ represents a singleton set of objects which is a subset of those objects represented by the $(\textbf{parent})^{\mathtt{w}+1}(\textbf{child})^1$ pedigree. This genealogical fact is captured by the subsumption relation on pedigrees.

Inter-procedural Pedigree Inference In our calculus, programmers only need to declare a pedigree qualifier on an object type when they care about the pedigree of that object. For all other occurrences with no qualifications, our type system can infer them. The inference algorithm is also able to track pedigree information inter-procedurally. Consider the `oops` method at the end of the `Button` class, which if included would be a type error. If we only look at the code of `Button` itself, the code is perfectly legal. However, note that the field `ctrl` of the `Button` object is set in the constructor to be a `Ctrl` object held by the `Dialog` object, which is in turn held by `Main`. That `Ctrl` object is a **child** of the `Main` object, so its pedigree from the perspective of `Button` is definitely not **child** – the `Ctrl` is its **uncle**.

Our type system is constraint-based. The novel aspect is that constraints are on (positive and negative) levels, which range over (a subset of) the natural numbers. Thus, finding whether conflict exists in the constraint set is reduced to solving a system of linear diophantine equations over natural numbers, a well-studied problem [Sch98] with decidable and efficient solutions.

Polymorphic Pedigrees and Parametric Polymorphism Inferred pedigrees are treated polymorphically, so that the references referred to in a class can have different pedigrees for different instances of the same class, aligning with the "I don't care" intention of programmers. For instance, the `ok Button` and the `cncl Button` can in fact have loggers of different pedigrees. We sometimes call object types without explicit pedigree qualification *polymorphic pedigree types*. The parametric polymorphism used here does not lie far from well-known type theoretic principles, where each class is viewed as being defined via a polymorphic `let` *a la* ML and polymorphic type variables are assigned for levels. We also support cyclic class definitions – the norm of object-oriented programming – which is not possible with pure `let`-polymorphism.

Different objects of the same class can obviously have different pedigrees: one `Logger` is instantiated as the **sibling** of the `Dialog`, and the other as the **child** of the `Dialog`. Objects can also be instantiated with an unspecified pedigree, as is the case for the instantiation of the two `Logger` instances, expressing "I don't care what pedigree it is instantiated with." In that case, it is the object usage which decides whether there is a satisfiable pedigree.

2.2 Alias Protection with Pedigree Types

The general form of pedigree types is consistent with the requirement of ownership encapsulation. Intuitively, an object should only refer to its direct children ($\mathtt{w} = 0, \mathtt{z} = 1$), itself ($\mathtt{w} = 0, \mathtt{z} = 0$), direct and indirect ancestors ($\mathtt{w} > 0, \mathtt{z} = 0$) or direct children of its direct and indirect ancestors ($\mathtt{w} > 0, \mathtt{z} = 1$). For some concrete examples, it means **sibling**s can refer to each other freely; an object can always refer to its ancestors, captured by the notion of all cases where $\mathtt{w} > 1$, but not *vice versa*: $\textbf{nephew} = (\textbf{parent})^1(\textbf{child})^2$ violates the notion of encapsulation as it it is a reference to the **child** of your **sibling** on the hierarchy of encapsulation. What is referable by pedigree types is identical to what is commonly believed to be referable on the standard ownership tree that enforces the property of owners-as-dominators [NPV98, CPN98].

At first glance, protecting owned objects is as simple as disallowing references with a **child** pedigree qualifier from being given out to any object with non-**child** pedigree. However, such a type system would not prevent *indirect leakage*. For instance, consider the pedigree relationship illustrated in Fig. 1. First of all, there is nothing wrong with the `Dialog` object passing its **child** object named `privlog` to its **child** object named `cncl`, as is found in the example when the constructor of the `cncl Button` is invoked. If our type system only checked whether **child** references were passed out, it would be happy to allow the `cncl Button` object to pass the reference to the `privlog Logger` object – not a **child** but a **sibling** from the perspective of the `cncl` object – to the `Ctrl` object (the one in the `ctrl` field). This however would violate encapsulation, as the `privlog` object is an internal representation of the `Dialog` object and should not be exposed to a **sibling** of the `Dialog` object.

Our type system is able to detect this indirect leakage. The key is that it always makes sure that both the sender and the receiver only handle references that can be associated with well-formed pedigree types. Had the passing of the private `Logger` object held by the `cncl Button` to the `Ctrl` object been allowed, relativization would imply the `Logger` object is the `Ctrl` object's **sibling**'s **child**, *i.e.* $(\textbf{parent})(\textbf{child})^2$. This is not a well-formed pedigree qualifier and would result in a type error.

C	::=	$\overrightarrow{\mathsf{a} \mapsto \langle M; F\rangle}$	*classes*
M	::=	$\overrightarrow{\mathsf{m} \mapsto \lambda x.e}$	*methods*
F	::=	$\overrightarrow{\mathsf{f}}$	*fields*
e	::=	() | x | *const* | $e; e'$ | **this**	*expressions*
	|	**new** *Ped* a	
	|	$e.\mathsf{m}(e')$	
	|	f | f:=e	
x			*variable*
const	$\in$	{..., -1, 0, 1, ...}	*integer*
a			*class name*
m			*method name*
f			*field name*

Ct	::=	$\overrightarrow{\mathsf{a} \mapsto sig}$	*class signatures*
sig	::=	$\langle mod; Mt; Ft\rangle$	*class signature*
Mt	::=	$\overrightarrow{\mathsf{m} \mapsto (st \to st')}$	*method signatures*
Ft	::=	$\overrightarrow{\mathsf{f} \mapsto st}$	*field signatures*
mod	::=	**default** | **strict**	*modularity mode*
st	::=	**unit** | **int** | *Ped* a	*declared types*
Ped	::=	$(\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}$ | ϵ	*pedigree qualifier*
w	$\in$	{0, 1, ...}	*positive level*
z	$\in$	{0, 1}	*negative level*

Figure 2. Abstract Syntax

3. The Formal System

3.1 Abstract Syntax

We first define some basic notation. $\overline{x_n}$ denotes a set $\{x_1, \ldots, x_n\}$. $\overrightarrow{x_n \mapsto y_n}$ is used to denote a mapping sequence (also called a mapping) $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$. Given $M = \overrightarrow{x_n \mapsto y_n}$, $\mathrm{Dom}(M)$ denotes the domain of M, and it is defined as $\{x_1, \ldots x_n\}$. We also write $M(x_1) = y_1, \ldots M(x_n) = y_n$. A sequence, denoted $\overrightarrow{x_n}$, is defined as a degenerate form of mapping sequence where all elements in the domain map to **null**. When no confusion arises, we also drop the subscript n for sets and mapping sequences and simply use $\overline{x}$ and $\overrightarrow{x \mapsto y}$. Notation $\propto$ is used to denote containment, for instance $(x_2 \mapsto y_2) \propto M$. We write $M[x \mapsto y]$ as a mapping update: M and $M[x \mapsto y]$ are identical except that $M[x \mapsto y]$ maps x to y. Updatable mapping concatenation $\rhd$ is defined as $M_1 \rhd M \stackrel{\text{def}}{=} M_1[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$. We use $\emptyset$ to denote empty set and $[\,]$ to denote mapping sequences of length zero.

The abstract syntax of our calculus is defined in Fig. 2. Its presentation is optimized for our formalization, but the coverage of features is largely similar to Featherweight Java [IPW99]. We leave out inheritance (but not subtyping) and dynamic casting from the language core, and briefly cover them in Sec. 4. Other minor differences include 1) we do not model constructors as they are orthogonal to Pedigree Types; 2) We only model private field access and do not model the cross-object one (such as $e.\mathsf{f}$), because the latter can be encoded as a pair of getter/setter methods accessing private fields. 3) The arity of method arguments is simplified to be 1.

We have separated class code and class signatures, and have taken the mapping-based notations for lists. A program is composed of classes (C), with one of them being the bootstrapping class named `Main` with a special method `main`. A class has a unique name (a), followed by a list of methods (M) and local fields (F). Object instantiation is modeled by the expression **new** *Ped* a, which is identical to Featherweight Java's **new** a expression, except that programmers can specify the pedigree for the newly created instance. The "I-don't-care" pedigree is represented as ϵ. In class signatures (Ct), the *mod* modifier gives programmers a choice of how modular the typechecking process is. This topic will be detailed in Sec. 3.2.6. User-declared types (st) include primitive types and pedigree types (*Ped* a).

$$
\begin{array}{rcl}
\Psi(\overrightarrow{\mathsf{a}_n \mapsto sig_n}) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{a}_n \mapsto psig_n} \\
& & \text{if } \forall\, i \in \{1, \ldots, n\}. \\
& & \quad psig_i = \Psi_{\mathtt{param}}(sig_i, \overrightarrow{\mathsf{a}_n \mapsto sig_n}, \emptyset) \\
\Psi_{\mathtt{param}}(sig, Ct, Z) & \stackrel{\text{def}}{=} & \forall\langle\alpha_1; \beta_1\rangle \ldots \forall\langle\alpha_u; \beta_u\rangle.\Psi_{\mathtt{sig}}(sig, Ct, Z) \\
& & \text{if } FP(\Psi_{\mathtt{sig}}(sig, Ct, Z)) = \\
& & \qquad \{\langle\alpha_1; \beta_1\rangle, \ldots, \langle\alpha_u; \beta_u\rangle\} \\
& & \quad sig = \langle mod; Mt; Ft\rangle \\
& & \quad mod = \textbf{strict} \text{ implies } u = 0 \\
\Psi_{\mathtt{sig}}(\langle mod; Mt; Ft\rangle, Ct, Z) & \stackrel{\text{def}}{=} & \langle \Psi_{\mathtt{ml}}(Mt, Ct, Z); \Psi_{\mathtt{fl}}(Ft, Ct, Z)\rangle \\
\Psi_{\mathtt{ml}}(\overrightarrow{\mathsf{m} \mapsto (st \to st')}, Ct, Z) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (\Psi_{\mathtt{st}}(st, Ct, Z) \to \Psi_{\mathtt{st}}(st', Ct, Z))} \\
\Psi_{\mathtt{fl}}(\overrightarrow{\mathsf{f} \mapsto st}, Ct, Z) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{f} \mapsto \Psi_{\mathtt{st}}(st, Ct, Z)} \\
\Psi_{\mathtt{st}}(\textbf{int}, Ct, Z) & \stackrel{\text{def}}{=} & \textbf{int} \\
\Psi_{\mathtt{st}}(\textbf{unit}, Ct, Z) & \stackrel{\text{def}}{=} & \textbf{unit} \\
\Psi_{\mathtt{st}}(Ped\ \mathsf{a}, Ct, Z) & \stackrel{\text{def}}{=} & \begin{cases} \mu\mathsf{a}.\mathcal{M}\ @\rho & \text{if } \mathsf{a} \notin Z \\ \mathsf{a}@\rho & \text{if } \mathsf{a} \in Z \end{cases} \\
& & \text{if } \Psi_{\mathtt{sig}}(Ct(\mathsf{a}), Ct, Z \cup \{\mathsf{a}\}) = \langle\mathcal{M}; \mathcal{F}\rangle \\
& & \rho = \begin{cases} (\textbf{parent})^{\alpha}(\textbf{child})^{\beta} & \text{if } Ped = \epsilon \\ & \quad \alpha, \beta \text{ fresh} \\ Ped & \text{otherwise} \end{cases}
\end{array}
$$

$\mathcal{C}$	::=	$\overrightarrow{\mathsf{a} \mapsto psig}$	*parameterized signatures*
psig	::=	$\forall\langle\alpha; \beta\rangle.\langle\mathcal{M}; \mathcal{F}\rangle$	*parameterized signature*
$\mathcal{M}$	::=	$\overrightarrow{\mathsf{m} \mapsto (s\tau \to s\tau')}$	*processed method signatures*
$\mathcal{F}$	::=	$\overrightarrow{\mathsf{f} \mapsto s\tau}$	*processed field signatures*
τ	::=	$s\tau$ | $\bot$	*types*
$s\tau$	::=	**unit** | **int** | $\mu\mathsf{a}.\mathcal{M}\ @\rho$ | $\mathsf{a}@\rho$	*types on signatures*
ρ	::=	$(\textbf{parent})^{\nu}(\textbf{child})^{\nu'}$	*pedigree expression*
ν	::=	$\nu + \nu'$ | $\nu - \nu'$ | α | w	*level expression*
α, β			*level type variable*

Figure 3. $\Psi(Ct) = \mathcal{C}$: Computing Parameterized Signatures

3.2 The Type System

As we explained in Sec. 2.1, our type system supports parametric polymorphism, but in an implicit form – the programmer syntax in Fig. 2 does not provide explicit parameterization of classes. A straightforward function called Ψ is used to properly parameterize the class signature into the more type-theory-friendly form where all type variables are bound at the beginning. We first introduce the Ψ function in Sec. 3.2.1, and then introduce the main parts of the type system in the rest of the section.

3.2.1 Level Parameterization and Recursive Type Unfolding

In the signature of `Button` in Fig. 1, the `logger` field is not qualified with a pedigree. Intuitively, the "I-don't-care-about-the-pedigree" intention translates as each instance of `Button` having a fresh pair of levels (positive level and negative level). A lengthy but unsurprising Ψ function is defined in Fig. 3 to help capture this notion. It simply goes through the structure of each signature (sig_i), and generates a pair of variables for each unqualified occurrence such as `logger` (*i.e.* $Ped = \epsilon$). These variables are called *level type variables*, with metavariable α ranging over positive pedigree levels, and β ranging over negative pedigree levels. The class signatures produced by Ψ are bound with these variables, with helper function $FP()$ enumerating all pairs of level type variables occurring free in the signature. The Ψ function processes class signatures transitively. This means a class not only provides parametric polymorphism for the levels of direct object references in the class, but also allows these parameters to be forwarded to provide polymor-

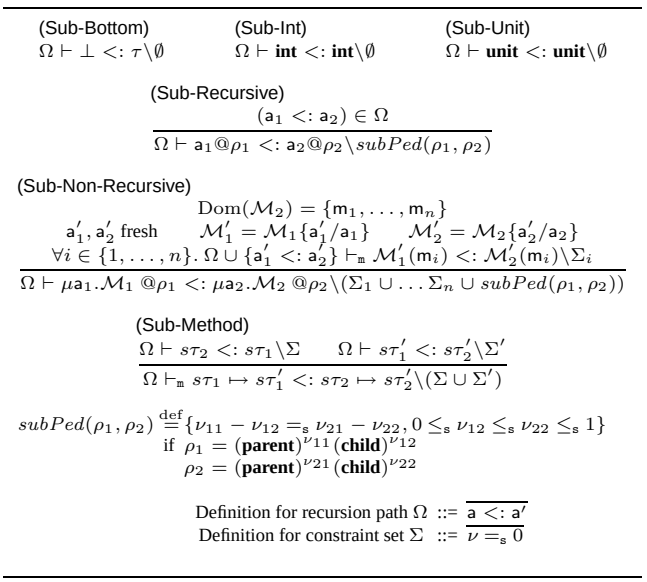

Figure 4. Subtyping Rules

phism for indirect object references. The computed signature list ($\mathcal{C}$) is used in most typing rules of the type system. As defined in the same figure, $\mathcal{C}$ is homomorphic to Ct in programmer syntax. We will postpone the discussion of the modularity mode (*mod*) in Sec. 3.2.6.

In object-oriented languages, recursive types are the norm. In the Ψ function, we also unfold recursive types explicitly, using a variation of the standard μ-type techniques [AC93]. Set Z is used to keep track of recursion. In the resulting signature computed by Ψ, each occurrence of user-declared pedigree type is transformed into either the $\mu\mathsf{a}.\mathcal{M}\ @\rho$ form or the $\mathsf{a}@\rho$ form. $\mu\mathsf{a}.\mathcal{M}\ @\rho$ in principle still follows the general form of the pedigree type – a combination of pedigree (ρ) and the object type (the rest of it, with name a and structure information in $\mathcal{M}$). $\mathsf{a}@\rho$ is used for the object type already bound by μ in its enclosing class signature. For convenience, we sometimes call the two forms of pedigree types *the μ form* and *the abbreviated form* respectively.

3.2.2 Types and Subtyping

As defined in Fig. 3, expression types τ used by the type system are also similar to the programmer declared types, except that a $\bot$ type is added to typecheck uninitialized fields, and the pedigree types are now either in the μ form or in the abbreviated form. In the general case of expression typing, the positive levels and negative levels of a pedigree type might be linear expressions over level variables. The general form of ρ thus is a type expression, which we call a *pedigree expression*. The linear expressions in this case are called *level expressions*. In the abstract syntax, we omit parentheses for level expressions. They are implicitly added to preserve arithmetic precedence. For instance if $\nu_1 =_{\mathtt{s}} 1$ and $\nu_2 =_{\mathtt{s}} 1 + 3$, then $\nu_1 - \nu_2$ is $1 - (1 + 3)$. When there is no need to distinguish the positive level and the negative level, we also use metavariable w to represent either level constant, and metavariable α to represent either level variable. To avoid confusion, equality/inequality symbols showing up in the constraint sets are denoted as $=_{\mathtt{s}}$, $\geq_{\mathtt{s}}$, $\leq_{\mathtt{s}}$ respectively.

Subtyping is defined in Fig. 4 with judgments of the form $\Omega \vdash \tau <: \tau' \backslash \Sigma$. This judgment reads "$\tau$ is a subtype of τ' under pedigree constraints Σ and assumptions Ω". The main rules are (Sub-Non-Recursive) and (Sub-Recursive), which show the object type part of the pedigree type follows standard structural subtyping. Since types may be recursive, the standard rules for subtyping over recursive types – commonly known as the Amber rules – are used [AC93]. Data structure Ω, which is a partial order with elements of the form $\mathsf{a} <: \mathsf{a}'$, is standard for this purpose, as is the alpha conversion (*i.e.* $\mathcal{M}_1\{\mathsf{a}'_1/\mathsf{a}_1\}$ and $\mathcal{M}_2\{\mathsf{a}'_2/\mathsf{a}_2\}$ in (Sub-Non-Recursive)).

The interesting part of the subtyping rules is perhaps the subtyping of two pedigree expressions, captured by $subPed$. It captures the notion of pedigree subsumption (Sec. 2.1); the basic case of pedigree identity is also included implicitly by this definition. For instance, $subPed(\textbf{self}, \textbf{sibling})$ and $subPed(\textbf{sibling}, \textbf{sibling})$ are both consistent.

3.2.3 The Typechecking Process

All typing rules are presented in Fig.5.

The Rules The whole program is globally typechecked with (T-Global). This rule first typechecks each class via $\vdash_{\mathtt{cls}}$, and then merges the constraints via $\vdash_{\mathtt{cons}}$. When there are no conflicting constraints in the final constraint set Σ, the program typechecks. This section focuses on per-class typechecking. The discussion on constraint merging is deferred later in this section. We also postpone discussion on typechecking the **new** expression and the data structure Π in the rules, as they are also closely related to constraint merging.

Rules (T-Class) and (T-Methods) are straightforward for typing the respective constructs. Expressions are typed via the judgment $\Gamma, \mathcal{C} \vdash e : \tau \backslash \Sigma,\ \Pi$. Typing environment Γ is defined at the bottom of Fig. 5; it maps variables to types, and maps the special keyword **me** to the name of the class enclosing the expression. (T-Self) types **this**, which is given a **self** pedigree. (T-Const), (T-Var), and (T-Continue) are obvious rules to type integer constants, variables and continuations. (T-Sub) bridges with the subtyping rules. (T-Invoke) will be explained shortly.

Pedigree Constraints and Decidable Constraint Solving All constraints in Σ (defined at the bottom of Fig. 5) are constraining pedigree levels (positive and negative), which are natural numbers. The constraint solving task in our type system is to find nonnegative solutions to a system of linear diophantine equations. For positive level type variables, this is obvious: they range over {0, 1, ...}, and finding a satisfiable positive level is equivalent to finding nonnegative solutions to linear equations. Negative level type variables slightly complicate the matter: they only range over {0, 1} and not {0, 1, ...}. Our solution is that when such a variable is used, one constraint is computed by the $twoVals$ function and merged to the main constraint set Σ, and $twoVals(\beta) \stackrel{\text{def}}{=} \{0 \leq_{\mathtt{s}} \beta \leq_{\mathtt{s}} 1\}$. Inequality constraints in the form of $\nu \geq_{\mathtt{s}} 0$ and $\nu \leq_{\mathtt{s}} 1$ can be easily rewritten as equality ones: the former is equivalent to $\nu =_{\mathtt{s}} \alpha$ for fresh α and the latter is equivalent to $1 - \nu \geq_{\mathtt{s}} 0$. Constraint $\nu_1 =_{\mathtt{s}} \nu_2$ is equivalent to $\nu_1 - \nu_2 =_{\mathtt{s}} 0$.

The algorithm for finding whether nonnegative solutions exist to a system of linear diophantine equations is decidable [Sch98]. As a result, the judgment $\vdash_{\mathtt{G}} C : Ct$ is also decidable since the type rules are deterministic modulo choice of fresh variables. So, the question of whether the program C is typeable with program signature Ct is decidable. Efficient ways of solving linear diophantine equations exist. For instance, the equations can be solved incrementally [CD94] to avoid solving a large set of equations all at once.

3.2.4 Method Invocation and Pedigree Relativization

Method invocation is typechecked by (T-Invoke). As pedigrees are always relative to the object they are declared in, pedigree relativization is needed for message passing between two objects. Relativization is captured by the $relativize$ function, defined in Fig. 6. $relativize(\rho_1, \rho_2) = \rho_3, \Sigma$ means a pedigree ρ_1 in one object is

(T-Global)

$$\frac{\Psi(Ct) = \mathcal{C} \qquad \mathrm{Dom}(\mathcal{C}) = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\} \quad \forall i \in \{1, \ldots, n\} \text{ such that } [\mathbf{me} \mapsto \mathsf{a}_i], \mathcal{C} \vdash_{\mathtt{cls}} C(\mathsf{a}_i) : \mathcal{C}(\mathsf{a}_i) \backslash \Sigma_i, \Pi_i \quad \overrightarrow{\mathsf{a}_n \mapsto \langle \Sigma_n, \Pi_n \rangle}, [\,] \vdash_{\mathtt{cons}} \langle \mathtt{main}; [\,]; [\,] \rangle \backslash \Sigma \qquad \Sigma \text{ consistent}}{\vdash_{\mathtt{G}} C : Ct}$$

(T-Class)

$$\frac{\mathrm{Dom}(F) = \mathrm{Dom}(\mathcal{F}) \quad \Gamma, \mathcal{C} \vdash_{\mathtt{m}} M : \mathcal{M} \backslash \Sigma, \Pi \qquad \mathit{psig} = \forall \langle \alpha_1; \beta_1 \rangle, \ldots \forall \langle \alpha_n; \beta_n \rangle . \langle \mathcal{M}; \mathcal{F} \rangle}{\Gamma, \mathcal{C} \vdash_{\mathtt{cls}} \langle M; F \rangle : \mathit{psig} \backslash \Sigma \cup \bigcup_{i \in \{1, \ldots, n\}} \mathit{twoVals}(\beta_i), \Pi}$$

(T-Methods)

$$\frac{\mathrm{Dom}(M) = \mathrm{Dom}(\mathcal{M}) = \{\mathsf{m}_1, \ldots, \mathsf{m}_n\} \quad \forall i \in \{1, \ldots, n\} \qquad M(\mathsf{m}_i) = \lambda x_i . e_i \quad \mathcal{M}(\mathsf{m}_i) = \tau_i \to \tau_i' \qquad \Gamma \rhd (x_i \mapsto \tau_i),\ \mathcal{C} \vdash\ e_i : \tau_i' \backslash \Sigma_i, \Pi_i}{\Gamma,\ \mathcal{C} \vdash_{\mathtt{m}} M : \mathcal{M} \backslash \Sigma_1 \cup \ldots \Sigma_n, \Pi_1 \cup \ldots \Pi_n}$$

(T-Const)
$\Gamma,\ \mathcal{C} \vdash \mathit{cst} : \mathbf{int} \backslash \emptyset, \emptyset$

(T-Unit)
$\Gamma,\ \mathcal{C} \vdash () : \mathbf{unit} \backslash \emptyset, \emptyset$

(T-Var)
$\Gamma,\ \mathcal{C} \vdash x : \Gamma(x) \backslash \emptyset, \emptyset$

(T-New)

$$\frac{\rho = \begin{cases} (\mathbf{parent})^{\alpha}(\mathbf{child}) & \text{if } \mathit{Ped} = \epsilon \text{ and } \alpha \text{ fresh} \\ \mathit{Ped} & \text{if } \mathit{Ped} = (\mathbf{parent})^{\mathsf{w}}(\mathbf{child}) \end{cases} \quad \mathcal{C}(\mathsf{a}) = \forall \langle \alpha_1; \beta_1 \rangle, \ldots, \forall \langle \alpha_n; \beta_n \rangle . \langle \mathcal{M}; \mathcal{F} \rangle \quad \alpha_1', \ldots, \alpha_n', \beta_1', \ldots, \beta_n' \text{ fresh} \quad \sigma = [\langle \alpha_1; \beta_1 \rangle \mapsto \langle \alpha_1'; \beta_1' \rangle, \ldots, \langle \alpha_n; \beta_n \rangle \mapsto \langle \alpha_n'; \beta_n' \rangle] \quad \Sigma = \bigcup_{i=\{1,\ldots,n\}} \mathit{twoVals}(\beta_i')}{\Gamma,\ \mathcal{C} \vdash \mathbf{new}\ \mathit{Ped}\ \mathsf{a} : (\mu \mathsf{a} . \mathcal{M}\ @\rho)[\sigma] \backslash \Sigma, \{\langle \mathsf{a}; \rho; \sigma \rangle\}}$$

(T-Invoke)

$$\frac{\Gamma,\ \mathcal{C} \vdash e : \mu \mathsf{a} . \mathcal{M}\ @(\mathbf{parent})^{\nu_1}(\mathbf{child})^{\nu_2} \backslash \Sigma, \Pi \quad \mathcal{M}' = \mathcal{M}\{\mathsf{a} \circlearrowleft \mathcal{M}\} \qquad \mathcal{M}'(\mathsf{m}) = \tau'' \to \tau''' \quad \Gamma,\ \mathcal{C} \vdash e' : \tau' \backslash \Sigma', \Pi' \qquad \mathit{convert}(\tau', (\mathbf{parent})^{\nu_2}(\mathbf{child})^{\nu_1}) = \tau_{\mathtt{c1}}, \Sigma_{\mathtt{c1}} \quad \mathit{convert}(\tau'', (\mathbf{parent})^{\nu_1}(\mathbf{child})^{\nu_2}) = \tau_{\mathtt{c2}}, \Sigma_{\mathtt{c2}} \quad \emptyset \vdash \tau_{\mathtt{c1}} <: \tau''' \backslash \Sigma_{\mathtt{s1}} \qquad \emptyset \vdash \tau_{\mathtt{c2}} <: \tau \backslash \Sigma_{\mathtt{s2}}}{\Gamma,\ \mathcal{C} \vdash e.\mathsf{m}(e') : \tau \backslash \Sigma \cup \Sigma' \cup \Sigma_{\mathtt{s1}} \cup \Sigma_{\mathtt{s2}} \cup \Sigma_{\mathtt{c1}} \cup \Sigma_{\mathtt{c2}}, \Pi \cup \Pi'}$$

(T-Read)

$$\frac{\mathcal{C}(\Gamma(\mathbf{me})) = \overline{\forall \langle \alpha; \beta \rangle}.\ \langle \mathcal{M}; \mathcal{F} \rangle}{\Gamma,\ \mathcal{C} \vdash \mathsf{f} : \mathcal{F}(\mathsf{f}) \backslash \emptyset, \emptyset}$$

(T-Write)

$$\frac{\mathcal{C}(\Gamma(\mathbf{me})) = \overline{\forall \langle \alpha; \beta \rangle}.\ \langle \mathcal{M}; \mathcal{F} \rangle \qquad \Gamma,\ \mathcal{C} \vdash e : \mathcal{F}(\mathsf{f}) \backslash \Sigma, \Pi}{\Gamma,\ \mathcal{C} \vdash \mathsf{f} := e : \mathcal{F}(\mathsf{f}) \backslash \Sigma, \Pi}$$

(T-Self)

$$\frac{\Gamma(\mathbf{me}) = \mathsf{a} \qquad \mathcal{C}(\mathsf{a}) = \overline{\forall \langle \alpha; \beta \rangle}.\ \langle \mathcal{M}; \mathcal{F} \rangle}{\Gamma,\ \mathcal{C} \vdash \mathbf{this} : \mu \mathsf{a} . \mathcal{M}\ @\mathbf{self} \backslash \emptyset, \emptyset}$$

(T-Continue)

$$\frac{\Gamma,\ \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi \qquad \Gamma,\ \mathcal{C} \vdash e' : \tau' \backslash \Sigma', \Pi'}{\Gamma,\ \mathcal{C} \vdash e; e' : \tau' \backslash \Sigma \cup \Sigma', \Pi \cup \Pi'}$$

(T-Sub)

$$\frac{\Gamma,\ \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi \qquad \emptyset \vdash \tau <: \tau' \backslash \Sigma'}{\Gamma,\ \mathcal{C} \vdash e : \tau' \backslash (\Sigma \cup \Sigma'), \Pi}$$

Γ	::=	$\overrightarrow{x \mapsto \tau \mid \mathbf{me} \mapsto \mathsf{a}}$	*typing environment*
Π	::=	$\overrightarrow{\langle \mathsf{a}; \rho; \sigma \rangle}$	*instantiation record set*
σ	::=	$\overrightarrow{\langle \alpha; \beta \rangle \mapsto \langle \alpha'; \beta' \rangle}$	*level variable substitution*

Figure 5. Typing Rules and Related Definitions

$$\begin{array}{rcl}
\mathit{convert}(\mathbf{int}, \rho) & \stackrel{\mathrm{def}}{=} & \mathbf{int},\ \emptyset \\
\mathit{convert}(\mathbf{unit}, \rho) & \stackrel{\mathrm{def}}{=} & \mathbf{unit},\ \emptyset \\
\mathit{convert}(\mu \mathsf{a} . \mathcal{M}\ @\rho_1, \rho_2) & \stackrel{\mathrm{def}}{=} & \mu \mathsf{a} . \mathcal{M}\ @\rho_3,\ \Sigma \\
 & \text{if} & \mathit{relativize}(\rho_1, \rho_2) = \rho_3, \Sigma \\
 & & \\
\mathit{relativize}(\rho_1, \rho_2) & \stackrel{\mathrm{def}}{=} & (\mathbf{parent})^{\nu_{21} + \nu_{11} - \nu_{22}}(\mathbf{child})^{\nu_{12}}, \\
 & & \{\nu_{11} - \nu_{22} \geq_{\mathtt{s}} 0\} \\
 & \text{if} & \rho_1 = (\mathbf{parent})^{\nu_{11}}(\mathbf{child})^{\nu_{12}} \\
 & & \rho_2 = (\mathbf{parent})^{\nu_{21}}(\mathbf{child})^{\nu_{22}} \\
 & & \\
\mathit{relativizem}(\rho, [\,]) & \stackrel{\mathrm{def}}{=} & \rho, \emptyset \\
\mathit{relativizem}(\rho, [\rho_1, \ldots \rho_n]) & \stackrel{\mathrm{def}}{=} & \rho_n', \Sigma_1 \cup \ldots, \Sigma_n \\
 & \text{if} & n \geq 1 \\
 & & \mathit{relativize}(\rho, \rho_1) = \rho_1', \Sigma_1 \\
 & & \mathit{relativize}(\rho_1', \rho_2) = \rho_2', \Sigma_2 \\
 & & \ldots \\
 & & \mathit{relativize}(\rho_{n-1}', \rho_n) = \rho_n', \Sigma_n
\end{array}$$

Figure 6. Pedigree Relativization

(T-Merge)

$$\frac{P = [\langle \mathsf{a}_1; \rho_1; \sigma_1 \rangle, \ldots, \langle \mathsf{a}_m; \rho_m; \sigma_m \rangle] \qquad \forall i \in \{1, \ldots, m\}.\ \mathsf{a}_i \neq \mathsf{a} \quad G(\mathsf{a}) = \langle \Sigma; \Pi \rangle \qquad \Pi = \{\langle \mathsf{a}_1'; \rho_1'; \sigma_1' \rangle, \ldots, \langle \mathsf{a}_n'; \rho_n'; \sigma_n' \rangle\} \quad \forall j \in \{1, \ldots, n\}.\ G, P \rhd \langle \mathsf{a}; \rho; \sigma \rangle \vdash \langle \mathsf{a}_j'; \rho_j'; \sigma_j' \rangle \backslash \Sigma_j}{G, P \vdash_{\mathtt{cons}} \langle \mathsf{a}; \rho; \sigma \rangle \backslash \Sigma[\sigma] \cup \Sigma_1 \cup \cdots \cup \Sigma_n}$$

(T-Merge-Recursive)

$$\frac{P = [\langle \mathsf{a}_1; \rho_1; \sigma_1 \rangle, \ldots, \langle \mathsf{a}_m; \rho_m; \sigma_m \rangle] \qquad \mathsf{a} = \mathsf{a}_p, p \in \{1, \ldots, m\} \quad \sigma_p = [\langle \alpha_1; \beta_1 \rangle \mapsto \langle \alpha_1'; \beta_1' \rangle, \ldots, \langle \alpha_u; \beta_u \rangle \mapsto \langle \alpha_u'; \beta_u' \rangle] \quad \sigma = [\langle \alpha_1; \beta_1 \rangle \mapsto \langle \alpha_1''; \beta_1'' \rangle, \ldots, \langle \alpha_u; \beta_u \rangle \mapsto \langle \alpha_u''; \beta_u'' \rangle] \quad \forall i \in \{1, \ldots, u\} \quad \rho_i'' = (\mathbf{parent})^{\alpha_i''}(\mathbf{child})^{\beta_i''} \qquad \rho_i''' = (\mathbf{parent})^{\alpha_i'''}(\mathbf{child})^{\beta_i'''} \quad \mathit{relativizem}(\rho_i'', [\rho_m, \rho_{m-1}, \ldots, \rho_{p+1}]) = \rho_i''', \Sigma_i}{G, P \vdash_{\mathtt{cons}} \langle \mathsf{a}; \rho; \sigma \rangle \backslash \bigcup_{i \in \{1 \ldots, u\}} \{\alpha_i' =_{\mathtt{s}} \alpha_i'''\} \cup \{\beta_i' =_{\mathtt{s}} \beta_i'''\} \cup \Sigma_i}$$

G	::=	$\overrightarrow{\mathsf{a} \mapsto \langle \Sigma; \Pi \rangle}$	*class summaries*
P	::=	$\overrightarrow{\langle \mathsf{a}; \rho; \sigma \rangle}$	*instantiation path*

Figure 7. Constraint Merging

pedigree ρ_3 in the other object if the first object is the ρ_2 of the second one, with the constraints of Σ. We also define a function $\mathit{relativizem}(\rho, [\rho_1, \ldots \rho_n])$, which is a composition of the original *relativize* function. It relativizes pedigree ρ to an object of pedigree ρ_1, and then from there relativizes the resulting pedigree to an object of its pedigree ρ_2, and so on. With function *relativize* defined, type relativization – modeled by the *convert* function – is pedigree relativization for pedigree types and isomorphic transformation otherwise. To facilitate the soundness proof, type relativization is always defined in the direction of value passing, *i.e.* we consistently convert a type being the contravariant of the invoker and covariant of the invokee. The rule also implicitly uses the intuitive fact that if one object is $(\mathbf{parent})^{\nu_1}(\mathbf{child})^{\nu_2}$ of the other, the second object is $(\mathbf{parent})^{\nu_2}(\mathbf{child})^{\nu_1}$ of the first.

Definition $\mathcal{M}\{\mathsf{a} \circlearrowleft \mathcal{M}'\}$ in (T-Invoke) unfolds recursive types: all pedigree type occurrences of the abbreviated form $\mathsf{a}@\rho$ within $\mathcal{M}$ are unfolded to the μ form $\mu \mathsf{a} . \mathcal{M}'\ @\rho$. This definition is omitted for this short presentation. Unfolding recursive types to equate the abbreviated form and the μ form is standard.

3.2.5 Polymorphic Instantiation and Constraint Merging

(T-New) says that if programmers do not give a pedigree to the instantiated object, it will freshly generate a positive level for it. Note that it is not allowed to have a newly instantiated object with a pedigree of negative level being 0. These pedigrees are singleton pedigrees, such as **parent**. The singleton property would not be preserved if programmers could freely instantiate objects of such pedigrees.

Since all type variables show up in pairs in our calculus, a general form of substitution mapping σ is thus of form $[\langle\alpha_1;\beta_1\rangle \mapsto \langle\alpha'_1;\beta'_1\rangle, \ldots, \langle\alpha_n;\beta_n\rangle \mapsto \langle\alpha'_n;\beta'_n\rangle]$. Substitution of α_1 with α'_1, $\ldots$, α_n with α'_n, β_1 with β'_1, $\ldots$, β_n with β'_n is then denoted $-[\sigma]$ for "$-$" being either a type, a class signature, or a constraint set.

The rest of the rule deals with type variable instantiation. The related technique – polymorphic type inference for object-oriented languages – is a well-studied area; relevant approaches include [EST95, Age96, WS01]. The technique being used here is closest to [EST95], where let-polymorphism is used in combination with type inference of object types. A perfect alignment with let-polymorphism would have been drawn, if it were not for the presence of mutually recursive classes. The latter greatly complicates the typing rules. Consider the following program:

```
classA = { ... y = new sibling classB; ... }
classB = { ... x = new child classA;  ... }
```

If standard constraint-based techniques for let-polymorphism were used, typechecking `classA` would involve merging the (refreshed) constraints associated with `classB`, say Σ_B, which are not known at that point. Our polymorphic type inference strategy can be viewed as extending the general idea of let-polymorphism to recursive programs. Inference of polymorphic recursion has been extensively studied, see *e.g.* [Hen93]; we present one particular approach which works well in the context of mutually recursive classes.

As we explained earlier in this section, typechecking occurs in two passes. We now explain the two passes with regard to polymorphic instantiations. The first pass is the per-class typechecking defined by $\vdash_{\texttt{cls}}$. When expression **new sibling** `classB` is encountered by (T-New), all level type variables of `classB` are chosen fresh as in let-polymorphism, and the program is typechecked in the analogous way. However, the constraints Σ_B are not merged in since they are not yet known. Our system processes Σ_B lazily: the type rule (T-New) adds an entry to the *instantiation record set* (Π in the type rules, defined in Fig. 5). For the example we described above, an entry $\langle$`classB`; **sibling**; $\sigma_B\rangle$ is added into Π, where σ_B maps the parameter list of the signature for `classB` to the new fresh type variables. After the first pass, each class is typechecked, with pedigree constraints collected in Σ, and Π indicating the constraints that need to be merged lazily. The information is represented by data structure called *summaries* (G in Fig. 7).

In the second pass, defined by $\vdash_{\texttt{cons}}$ in Fig. 7 and used by (T-Global), the typechecker starts from the bootstrapping class `Main`, and checks its Π to lazily merge constraints, with substitution performed. Judgment $G, P \vdash_{\texttt{cons}} \langle \mathsf{a};\rho;\sigma\rangle\backslash\Sigma$ means Σ is the merged constraints of per-class constraints of a, together with those that need to be merged lazily. This process is propagated through the dependency chain of Π and is defined by the (T-Merge) rule of Fig. 7. When there is a cycle in the Π dependency chain, the constraints must be merged via rule (T-Merge-Recursive) in the figure to avoid infinite looping. Data structure P in the rules is used to track the path on the dependency chain of Π from that of `Main` to the node representing the constraint set to be merged. For every step that the constraints of a particular class are to be merged, the rules check whether that class has already shown up in P. If so, a cyclic Π dependency chain has formed. Rather than further merging the constraints, (T-Merge-Recursive) simply makes sure the fresh type variables generated for typing the recurrent class instantiation is related to those generated for typing the previous class instantiation. This is achieved through pedigree relativization via $relativizem$.

Δ	::=	$\langle H; R\rangle$	*configuration*
H	::=	$\overrightarrow{\mathtt{o} \mapsto \langle \mathsf{a};\sigma;S\rangle}$	*heap*
R	::=	$\overrightarrow{\langle \mathtt{o}_1;\mathtt{o}_2\rangle \mapsto (\textbf{parent})^{\theta}(\textbf{child})}$	*pedigree relation store*
θ	::=	$\alpha \mid \mathsf{w}$	*store positive pedigree*
S	::=	$\overrightarrow{\mathsf{f} \mapsto v}$	*field store*
v	::=	$() \mid$ *const* $\mid \mathtt{o} \mid$ **null**	*value*
exd	::=	$e \mid v \mid$ **exception** $\mid$ **in**$(\mathtt{o}, exd)$	*extended expression*
$\mathtt{o}$	$\in$	$\mathbb{RID}$	*object reference*

Figure 8. Runtime Data Structure Definitions

3.2.6 Modularity and the default/strict Modes

In our calculus, programmers can optionally declare a class to be **strict** (as shown by the syntax in Fig. 2), and all other classes are in the **default** mode. What the two modes differ is *how modular* the type-checking process is for the class they are modifying. The **default** mode follows the bulk of our previous discussions, where programmers can liberally leave out pedigree declarations. In **strict** mode however, all pedigree declarations will be present. This is enforced by the Ψ definition in Fig. 3. There, for a **strict** class, its signature can not contain level type variables (hence $u = 0$ in $\Psi_{\texttt{param}}$ definition). Intuitively, it says all pedigrees contained by the signatures must be given by programmers.

What the **strict** mode guarantees here is the class typechecked via $\vdash_{\texttt{cls}}$ will have no pedigree discrepancies arising at constraint merge time for these classes. On the high level, it guarantees that, if a class typechecks, no pedigree discrepancy can happen no matter what other classes are, making their typing completely modular.

Even if all classes are declared to be of the **default** mode, our system still retains a degree of modularity. Our system typechecks each class modularly via the $\vdash_{\texttt{cls}}$ definition, and collects constraints on pedigree variables. All type errors excepting inter-class pedigree discrepancies can be immediately reported. Inter-class pedigree consistency is guaranteed by merging constraint sets obtained from each modular class typechecking, via the $\vdash_{\texttt{cons}}$ definition. The type system never re-typechecks a class.

These two modes represent a trade-off between modular systems requiring programmer declarations (the **strict** mode), and more flexible inference systems which can significantly improve programing productivity (the **default** mode), but which are less strongly modular. We believe the **default** mode is the best choice for most software development environments, allowing users to declare many "I-don't-care" intentions, but cases where more strictly modular interfaces are needed may still benefit from the advantages of pedigree types through the use of the **strict** mode.

3.3 Operational Semantics

As Pedigree Types are built on a Java-like object model, the dynamic semantics of our calculus is standard. The only exception is we create an auxiliary data structure called *pedigree relation store* (R in Fig. 8) to record the pedigree relations between objects. It is only used for proving the correctness of the type system and hence does not affect reduction. Every time a **new** expression is evaluated, the pedigree of the instantiated object relative to the instantiating object is recorded in R. An entry $\langle \mathtt{o}_1;\mathtt{o}_2\rangle \mapsto \rho$ in R says $\mathtt{o}_1$ is ρ of $\mathtt{o}_2$. Based on R, function $rel(R, \mathtt{o}_1, \mathtt{o}_2)$ produces the pedigree

of $\mathtt{o}_1$ relative to $\mathtt{o}_2$. Intuitively, R forms a directed tree where the nodes are objects (their IDs) and the edges signify how the objects instantiate one another. Each edge can be imagined as being associated with a "weight", indicating the pedigree of the instantiated object relative to the instantiating object. Function $rel(R, \mathtt{o}_1, \mathtt{o}_2)$ simply computes the "weighted distance" between tree node $\mathtt{o}_1$ and $\mathtt{o}_2$. Readers can find this simple yet formally lengthy definition in [Liu07].

Other run-time data structures are shown in Fig. 8. Runtime configuration Δ records the mutable state of the execution. It is composed of a standard heap (H), and the aforementioned R. An entry $\mathtt{o} \mapsto \langle \mathsf{a}; \sigma; S \rangle$ in the heap says that an object $\mathtt{o}$ is instantiated from a class named a, and has the mutable states recorded in S. Auxiliary structure σ is only used by the proof, and it keeps track of the type variable instantiation for each object.

Values (of set $\mathbb{V}$) are either an object reference (with ID $\mathtt{o}$), the standard **null** value, or primitive data. Expressions are extended to include values, **exception**, and a closure expression only used by the reduction system to model method invocation: $\mathbf{in}(\mathtt{o}, exd)$ means exd is to be evaluated with regard to object $\mathtt{o}$.

The reduction relation is $\Delta, exd \xrightarrow{C,\,\mathtt{o}} \Delta', exd'$, meaning exd is one-step reduced to exd' in object $\mathtt{o}$, the runtime configuration changes from Δ to Δ', and C, $\mathtt{o}$ remain unchanged over reductions. Multi-step reduction $\Delta, exd \xrightarrow{C,\,\mathtt{o}}_* \Delta', exd'$ is the transitive closure of one-step reduction. $\langle \Delta; exd \rangle \Uparrow_{C,\mathtt{o}}$ means the reduction diverges. The bootstrapping process is modeled by reduction $C \xrightarrow{\mathtt{init}} \langle \Delta; \mathtt{o}; exd \rangle$, which prepares the initial values for these data structures. A configuration $\langle \Delta; \mathtt{o}; exd \rangle$ is said to be *attainable* from program C, denoted as $C \xrightarrow{\clubsuit} \langle \Delta; \mathtt{o}; exd \rangle$ iff $C \xrightarrow{\mathtt{init}} \langle \Delta_0; \mathtt{o}; exd_0 \rangle$ and $\Delta_0, exd_0 \xrightarrow{C,\,\mathtt{o}}_* \Delta, exd$.

3.4 Theoretical Properties

We now establish the main properties of our calculus. These properties are rigorously proved in [Liu07] based on a somewhat more refined object model. The two models share the same notion of object reference, and so the underlying proof structure is largely the same.

Theorem 1 (Type Soundness)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\mathtt{init}} \langle \Delta; \mathtt{o}; exd \rangle$*, then either* $\langle \Delta; exd \rangle \Uparrow_{C,\mathtt{o}}$*, or* $\Delta, exd \xrightarrow{C,\,\mathtt{o}}_* \Delta', v$ *for some* $\Delta', v \in \mathbb{V}$*, or* $\Delta, exd \xrightarrow{C,\,\mathtt{o}}_* \Delta',$ **exception**.

This theorem states that the execution of statically typed programs either diverges, computes to a value, or throws exceptions. In this calculus, exceptions are thrown when an uninitialized field is accessed. The theorem is established via lemmas of subject reduction, progress, and the bootstrapping process leads into a well-typed initial configuration.

We now state several important theorems about how pedigree types enforce pedigrees at runtime. Theorem 2 states that if a programmer puts a pedigree qualifier, say **sibling**, on a field of a class, then at run time that field will always hold a reference which is a **sibling** (or a pedigree subsumed by **sibling**, *i.e.* **self**) of the current object. Theorem 3 says any object reference on the heap must refer to some object which can be named by a well-formed pedigree qualifier. No object can hold a reference to the internal representation of another object, since that relationship cannot be expressed by a well-formed pedigree type.

Theorem 2 (Shape Enforcement)**.** *If* $\vdash_{\mathtt{G}} C : Ct$*, for all* $\mathsf{a} \in \mathrm{Dom}(Ct)$, $Ct(\mathsf{a}) = \langle mod; Mt, Ft \rangle$*, for all* $(\mathsf{f} \mapsto t) \propto Ft$

$$t = (\mathbf{parent})^{\mathsf{w}}(\mathbf{child})^{\mathsf{z}}\,\mathsf{a}' \implies (rel(R, S(\mathsf{f}), \mathtt{o}) = \rho) \wedge (subPed(\rho, (\mathbf{parent})^{\mathsf{w}}(\mathbf{child})^{\mathsf{z}})\ \textit{consistent})$$

for all $\mathtt{o} \in \mathrm{Dom}(H)$, $\Delta = \langle H; R \rangle$, $H(\mathtt{o}) = \langle \mathsf{a}; \sigma; S \rangle$*, and* $C \xrightarrow{\clubsuit} \langle \Delta; \mathtt{o}'; exd \rangle$,

Theorem 3 (Alias Protection)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\clubsuit} \langle \Delta; \mathtt{o}'; exd \rangle$*, and* $\Delta = \langle H; R \rangle$*, then for all* $\mathtt{o} \in \mathrm{Dom}(H)$, $H(\mathtt{o}) = \langle \mathsf{a}; \sigma; S \rangle$*, and for all* $(\mathsf{f} \mapsto v) \propto S$ *and* $v \in \mathbb{RID}$*, we know* $rel(R, v, \mathtt{o}) = \rho$ *and there exists some* ν_1 *and* ν_2 *such that* $subPed(\rho, (\mathbf{parent})^{\nu_1}(\mathbf{child})^{\nu_2}) \cup \{\nu_1 \geq_{\mathsf{s}} 0\} \cup twoVals(\nu_2)$ *is consistent.*

The previous property is analogous to the notion of deep ownership [Cla01] in ownership type systems.

4. Extensions

We now demonstrate how a number of extensions can be built on top of the compact core calculus of Pedigree Types. Several of these are only sketched here, and their fleshing out constitutes interesting future research.

Selective Exposure The intention of encapsulation sometimes does not align well with the intention of hierarchical decomposition. When that happens, there is a need for selectively exposing an otherwise encapsulated object to objects that normally would not have gained access. Selective exposure is useful for programming idioms such as iterators and has generated significant interest for existing ownership type systems [AC04, LP06].

In the context of Pedigree Types, what selective exposure means is an object, say X, might occasionally accessed by some object, say Y, which is its **grandparent**. (Note that this is not allowed in the core model since X is Y's $(\mathbf{parent})^0(\mathbf{child})^2$, resulting a negative level not ranging over $\{0, 1\}$). Our model can be extended with selective exposure by allowing each object to be associated with an access policy stating which pedigrees higher up on the hierarchy can access the methods of an object. For instance, a policy could be $[\mathbf{grandparent} \mapsto \mathsf{m}_1; \mathbf{uncle} \mapsto \mathsf{m}_1, \mathsf{m}_2]$ which states that the **grandparent** of the current object can access its method m_1, while its **uncle** can access m_1 and m_2. Since **grandparent** and **uncle** otherwise would not be granted access, the strategy here *de facto* achieves selective exposure. We are confident that a sound type system of this fashion can be built based on our previous experience with a similar system [SS02]. This solution also does not lie far in spirit from existing selective exposure solutions such as Ownership Domains [AC04] and Variant Ownership [LP06].

An alternative solution to this problem is to make changes to the object model itself. We have worked out a full solution along this direction in [Liu07].

Opt-Out Pedigrees for Large-Scale Hierarchy Inference When all programmer pedigree annotations are omitted, our type system is *de facto* a hierarchy inference system where each constraint solution provides a possible "hierarchy layout" for heap objects. The high-level meaning of doing this is to rediscover the hierarchical decomposition intention of programmers.

To make such a technique practical for large-scale software systems, there is one hurdle to cross: when the application grows large, the general reference structure is commonly a graph, not a hierarchy. Blindly using our type system would lead to untypable programs when a strict hierarchy cannot be realized by constraint solving. The solution to this problem is simply to allow programmers to declare "opt-out" references that do not follow the hierarchy. The high-level intuition behind this is to allow programmers to count some references as "back-edges" on top of the hierarchy "backbone". Such an extension does not add any technical difficulties to the type system itself: for an opt-out reference, we would simply not constrain the pedigree associated with it.

The access policy solution outlined above for selective exposure is a more precise form of opt-out pedigree: rather than completely freeing the reference from obeying any hierarchy constraints, it allows back-edges but constrains to whom an object may have a back-edge.

Java-Style Inheritance In principle, Java-style inheritance captures two independent issues of object design: code composition and subtyping. The issue of code composition is orthogonal to type system design. For subtyping, Java inheritance enforces a combination of nominal subtyping (explicitly declared by **extends**) and a restricted form of structural subtyping (the subtype can have more methods but overridden methods must have the same signatures). Supporting nominal subtyping involves minimal work: the only change would be add one extra check at (Sub-Recursive) and (Sub-Non-Recursive), making sure a_1 and a_2 in the two rules also conform to the nominal constraints declared by **extends**. For structural subtyping, our type system in fact is more flexible than Java's as it supports unrestricted width and depth subtyping.

Dynamic Class Loading In Sec. 3.2.6, we explained that whether a class is declared to be **default** or **strict**, the code of each class is only type-checked once in a modular fashion. This is good news for dynamic class loading, since we do not need all classes to be available at the onset of program execution. When a class is declared **default**, inter-class consistency checking is performed at constraint merging time. What this implies in the context of dynamic class loading is the constraint merging will happen incrementally at run time. This dynamic constraint merging in fact is already commonplace; for example, the JVM maintains a set of "loader constraints" on types [LB98] which are dynamically updated and checked at each class load. The incremental algorithm for solving linear diophantine equations [CD94] is well-suited to incremental constraint merging.

Top Pedigree The core calculus does not have a "top pedigree" which subsumes all pedigrees (just as how Java's `Object` subsumes all object types). This feature would be useful when one wanted to create a heterogeneous `List` with objects each with a different pedigree. Supporting a top pedigree is simple: the only change is to add one subtyping rule to indicate all pedigrees are subsumed by the top pedigree.

Dynamic Casting Dynamic casting of pedigree types involves two dimensions: casting the pedigrees and casting the underlying object types. The second part is a standard Java operator with a standard solution. The non-trivial issue to address here is the casting of pedigrees.

Compared with Universe Types, Pedigree Types programmers should encounter very few scenarios where there is a need for casting pedigrees. In Universe Types, if a reference, say `r1`, of `peer` pedigree is assigned to a reference, say `r2`, without a pedigree (sometimes named `any` pedigree), the pedigree information is lost. If `r2` is then subsequently assigned to another reference, say `r3`, of `peer` pedigree, the second assignment would fail unless dynamic casting is used (especially when such assignments go beyond object boundaries, such as via method parameter passing). In Pedigree Types, since all variables without a pedigree declaration are treated with a polymorphic pedigree, assignment `r3 = r2` will succeed without any cast.

One situation where dynamic casting would be useful in Pedigree Types is when an object declared with a top pedigree is downcast to a more precise pedigree, *e.g.* when retrieving elements from a heterogeneous `List`. The standard solution for dynamic casting should also work for Pedigree Types: pedigree information can be maintained at run time, and some constraint solving also deferred to run time.

Programmer-Defined Pedigree Constraints Our core calculus does not allow programmers to express "two references are on the same (unknown) level of the ownership tree", a demand that can be easily supported by existing ownership type systems using explicit parametric polymorphism. A simple extension however can achieve this. All that needs to be added is extra syntax allowing programmers to indicate such an intention. The type system requires no change: it simply entails the addition of two constraints – one equating the positive levels of the two pedigrees and one equating the negative ones – into Σ in (T-Class). Similarly, inequality constraints such as " the two references are on unknown levels, but one must be higher than the other" and even arithmetic constraints such as "the two references are on unknown levels but one must be 1 level higher than the other" can be supported without difficulty.

5. Related Work

Explicitly Parametric Ownership Type Systems Numerous ownership type systems – such as [NPV98, CPN98, Cla01] and many later extensions and variations – use explicit parametric polymorphism (EPP) to enforce ownership. Pedigree Types also incorporate parametric polymorphism behind the scenes, and for this reason the aforecited EPP systems and Pedigree Types are generally on an equal footing in terms of expressiveness for writing meaningful programs, excepting some minor differences which we now cover. For instance, EPP systems allow users to specify that two references have the "same owner" on an unknown level, while Pedigree Types cannot (a similar notion can in fact be easily supported by a simple extension to Pedigree Types, see the last part of Sec. 4). On the flip side, Pedigree Types have several expressiveness gains over EPP systems as well. They can express singleton pedigree constraints, such as a reference pointing to a **parent** to indicate a reference to the directly enclosing object on the ownership tree. This feature can be useful for identifying containers widely used in Container-Component-based frameworks such as Enterprise Javabeans. In addition, EPP systems do not have an equivalent for our **self**; its positive impact on capturing the call-back constraints of object-oriented programming was discussed in Sec. 2.1.

The main advantages of Pedigree Types over EPP systems, as explained in Sec. 1, are 1) minimal annotation overhead and 2) a natural and intuitive programming interface to explicitly support the intention of hierarchical decomposition.

EPP system modularity is analogous to our **strict** mode: each class is strictly modular at the expense of the need for a large type annotation overhead. As discussed in Sec. 3.2.6, we prefer the reverse emphasis via our **default** mode, but support both modes to let the programmer make the ultimate decision.

Solutions to reduce annotation overhead in EPP systems are limited. SafeJava [Boy04] allows intra-procedural inference. A calculus by Wren [Wre03] infers context parameters of EPP systems based on a notion of Extended Object Graph. AliasJava [AKC02] describes an inter-procedural algorithm. It is formally undecidable, but the implementation shows reasonable results in practical situations. In Uno [MF07], program analysis techniques such as points-to analysis are used for ownership inference. Some properties in Uno overlap with our work, including their predicate OWNFIELD which is analogous to inferring a **child** pedigree in our system.

Within the past decade, many advanced features have been designed on top of EPP systems, which as of now are not covered by Pedigree Types. Examples include how to add variant types [LP06], how to unify Java generics and ownership types [PNCB06], how to support multiple owners [CDNS07].

Ownership Type Systems Without EPP Universe Types have a syntax similar to Pedigree Types; we previously explained their `peer` and `rep` modifiers in Sec. 1. All non-`peer` non-`rep` ref-

erences are treated as `any`[1], which in our terminology is a top pedigree and not a polymorphic pedigree. Dynamic casting thus is often needed – see the discussion of this topic in Sec. 4. In addition, Pedigree Types have a more general form of pedigrees (**parent**)^w(**child**)^z, whereas in Universe Types no pedigrees can be given to objects which are neither a `peer` nor a `rep` on the ownership tree. This is not necessarily a weakness of Universe Types, because their design is optimized for program verification where object referential structures generally have shallow hierarchies of 2 or 3 levels. By supporting a general form of pedigree to refer to arbitrary hierarchy levels and particularly allowing for level inference, Pedigree Types are a more general solution to the problem of expressing and rediscovering the decomposition hierarchy. No equivalents of **self** and **parent** exist in Universe Types.

Instead of preserving the owners-as-dominators property, Universe Types do allow an object to be accessed by objects outside the owner, as long as the access is read-only (the property is known as owners-as-modifers). Pedigree Types can adapt to this property easily: in that case, $twoVals$ constraints (used in various places of the type system) are only added when they are constraining pedigrees of non-read-only references. Recently, a run-time inference algorithm was designed for Universe Types [DM07]. Ownership transfer is now supported by Universe Types [MR07] as well, an interesting topic Pedigree Types do not yet address.

Several other non-EPP systems have a domain target and make simplifications based on need, such as Confined Types [VB99] for security, and Scoped Types [ACG$^+$06] for real-time memory management. These systems achieve their domain goals, illustrating the value of object encapsulation in these domains.

6. Conclusion

This paper presents an intuitive and powerful type system to shape the heap into a hierarchy of decomposition and encapsulation. The main contributions of this paper can be summarized as follows:

- The intuitive nomenclature of human genealogy is applied to a type system to navigate hierarchies. Familiar genealogical pedigrees such as **parent**, **child**, **sibling** are unified into one compact general form. Inherent aspects of human genealogy trees are captured, including pedigree relativization and pedigree subsumption.
- Implicit parametric polymorphism is introduced to capture the intuitive notion of an "I don't care"-kind of polymorphic pedigree. The resulting system retains the expressiveness of many existing ownership type systems, at the same time presenting a simple and intuitive programming interface to end users.
- A novel constraint-based type system is constructed to enforce the consistency of pedigree invariants, where constraints are linear systems of integer arithmetic equations, and constraint solving is finding a solution to these linear equations.
- Formal properties of type soundness, pedigree invariant enforcement, and alias protection are established.

In conclusion, Pedigree Types put the power of an explicit formalism for defining and constraining the heap hierarchy into the hands of programmers, and give programmers a greater awareness of, and thus control of, the heap reference structure.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.

[AC04] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP'04*, pages 1–25, 2004.

[ACG$^+$06] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java. In *ECOOP'06*, pages 124–147, 2006.

[Age96] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, Stanford, CA, USA, 1996.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA'02*, pages 311–330, 2002.

[Boy04] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[CD94] Evelyne Contejean and Herve Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.

[CDNS07] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *OOPSLA'07*, pages 441–460, 2007.

[Cla01] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[CPN98] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98*, pages 48–64, 1998.

[DM07] W. Dietl and P. Müller. Runtime universe type inference. In *IWACO'07*, 2007.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA'95*, pages 169–184, 1995.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM TOPLAS*, 15(2):253–289, 1993.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *OOPSLA'99*, volume 34(10), pages 132–146, N. Y., 1999.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA'98*, pages 36–44, 1998.

[Liu07] Yu David Liu. *Interaction-Oriented Programming (online at `http://www.cs.jhu.edu/~yliu/thesis/`)*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2007.

[LP06] Yi Lu and John Potter. On ownership and accessibility. In *ECOOP'06*, pages 99–123, 2006.

[LS05] Yu David Liu and Scott F. Smith. Interaction-based Programming with Classages. In *OOPSLA'05*, pages 191–209, 2005.

[MF07] Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA'07*, pages 423–440, 2007.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[MR07] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *OOPSLA'07*, pages 461–478, 2007.

[NPV98] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *ECOOP'98*, Brussels, Belgium, July 1998.

[PNCB06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA'06*, pages 311–324, 2006.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 1998.

[SS02] Christian Skalka and Scott Smith. Static use-based object confinement. In *Proceedings of the Foundations of Computer Security Workshop (FCS '02)*, Copenhagen, Denmark, July 2002.

[VB99] Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA'99*, pages 82–96, 1999.

[Wre03] Alisdair Wren. Ownership type inference. Master's thesis, Imperial College, 2003.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01*, pages 99–117, 2001.

[1] In some versions of Universe Types, the omitted pedigrees are `peer`.