IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

A Calculus for Boxes and Traits in a Java-Like Setting

This is a pre print version of the following article:

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/79357 since

Publisher:

Springer

Published version:

DOI:10.1007/978-3-642-13414-2_4

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

18 November 2026

A Calculus for Boxes and Traits in a Java-like Setting

Lorenzo Bettini[1], Ferruccio Damiani[1], Marco De Luca[1], Kathrin Geilmann[2], and Jan Schäfer[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Department of Computer Science, University of Kaiserslautern

Abstract. The box model is a component model for the object-oriented paradigm, that defines components (the boxes) with clear encapsulation boundaries. Having well-defined boundaries is crucial in component-based software development, because it enables to argue about the interference and interaction between a component and its context. In general, boxes contain several objects and inner boxes, of which some are local to the box and cannot be accessed from other boxes and some can be accessible by other boxes. A trait is a set of methods divorced from any class hierarchy. Traits can be composed together to form classes or other traits. We present a calculus for boxes and traits. Traits are units of fine-grained reuse, whereas boxes can be seen as units of coarse-grained reuse. The calculus is equipped with an ownership type system and allows us to combine coarse- and fine-grained reuse of code by maintaining encapsulation of components.

1 Introduction

In the development of software systems the reuse of code is vital. Most mainstream programming languages are object-oriented, based on the concept of classes. The granularity level of classes is often not appropriate for reuse. Since classes play the primary role of generators of instances, they must provide a complete set of features describing an object. Therefore, classes are often too coarse-grained to provide a minimal set of sensible reusable features [11] and too fine-grained to provide a software fragment that can be deployed independently [26].

Component-based software systems are built by linking components together or by building new components from existing ones. A key issue in component-based development is to have clear encapsulation boundaries for components. Knowing the component's boundaries enables modular analysis and the reuse of components in different program contexts without being bothered by unexpected interference between the component and its context. Typical OOLs do not have a language-level component concept other than single objects. In practice, however, objects often rely on other objects to hold additional state and realize additional behavior, forming implicit components at runtime. The box model [20, 19, 21] is a component model, which makes these components explicit. It defines components both on a syntactic level (by a set of classes) and as a runtime entity (as a set of objects). Components are instantiated and similar to objects: they have an identity and a local state. A component instance is called a *box*. In general, a box may contain several objects and can have nested boxes. To guarantee

encapsulation of certain objects of a box an ownership type system [8, 6, 15, 9] is used. It defines two ownership domains [1, 24, 19] for each box, namely a *local* and a *boundary* domain. Each object of a box and each inner box is located in one of these domains. Objects (and inner boxes) in the local domain are considered to be private to the component and are only accessible by objects of the same box and of inner boxes, whereas objects (boxes) in the boundary domain can be accessed by objects of other boxes. This is called the encapsulation property of the type system. Additionally, the box model defines a restriction to casts: it forbids downcasts to types which are subtypes of the type declared in the interface of a box. The cast restriction and the encapsulation property of the type system together give a clear encapsulation boundary to boxes, therefore it is possible to use boxes as a *unit for coarse-grained reuse*.

Traits have been designed to play the role of *units for fine-grained reuse* in order to counter the problems of class-based inheritance with respect to code reuse. A trait is a set of methods, completely independent from any class hierarchy. The common behavior (i.e., the common methods) of a set of classes can be factored into a trait. Traits were introduced and implemented in the dynamically-typed class-based language SQUEAK/SMALLTALK [11][3]. Various formulations of traits in a JAVA-like setting can be found in the literature (see, e.g., [25, 17, 3, 23, 4, 14]). Also two recent languages, SCALA and FORTRESS, incorporate forms of the trait construct.

In this paper, we present a JAVA-like core calculus for boxes and traits. The calculus supports interface-based polymorphism and uses traits and boxes as units of fine-grained and coarse-grained reuse, respectively. Namely, we consider: *Interfaces*, as object types, defining only method signatures. *Box interfaces*, as box types, defining only method signatures. *Traits*, as units of behavior reuse, defining only methods. *Classes*, as generators of objects, implementing interfaces by using traits. *Box classes*, as generators of boxes, implementing box interfaces by using traits. At the best of our knowledge, this is the first attempt to combine boxes and traits. In order to focus on the interactions between traits and boxes, we do not consider class-based inheritance in our calculus (along the lines and design choices of [3] and of the FORTRESS language).

Controlling the access to specific objects in specific contexts (boxes) is crucial to coordinate the access to sensible data and to keep consistency in an application. Our approach scales to a concurrent and distributed setting, since the synchronization of the access to data is orthogonal to our ownership type system. This type system complements concurrency mechanisms providing guarantees that the access of specific resources is allowed only to the desired components.

The combination of traits and boxes allows us to use a fine-grained reuse mechanism for the implementation of components (that is, coarse-grained units that can be deployed independently). Namely, it allows us to share code among components, without giving up the encapsulation properties. As we will point out at the end of Section 2, due to the ownership type system that usually comes with boxes in order achieve encapsulation, supporting this combination is quite challenging.

Organization of the Paper. In Section 2, we introduce boxes and traits by an example. Syntax, operational semantics, type system, and properties of the calculus are presented in Section 3. We conclude by discussing some related and further work.

[3] A related reuse construct called "trait" was introduced in [27], for the language SELF.

```
interface IObserver { void update(String o); }
interface ISubject {
  void register(IObserver o);
  void unregister(IObserver o);
}
trait TSubject is {
  List<IObserver> obs; // required field
  String getTopic(); // required method
  void register(IObserver o) { obs.add(o); }
  void unregister(IObserver o) { obs.remove(o); }
  void notify() { /* iterate over observers */ }
}
trait TTopicHolder {
  String topic; // required field
  void notify(); // required method
  void setTopic(String t) { topic = t; notify(); }
  String getTopic() { return topic; }
}
```

```
interface ICachedSubject extends ISubject {
  String getPrevTopic();
}
trait TCachedSubject is
  TSubject[notify aliasAs oldNotify]
          [exclude notify] + {
  String prevTopic; // required field
  void oldNotify(); // required method
  String getPrevTopic() { return prevTopic; }
  void notify() {
    prevTopic = getTopic(); oldNotify();
  }
}
class CSubject implements ICachedSubject
    by TCachedSubject, TTopicHolder {
  String topic; String prevTopic;
  List<IObserver> obs;
}
```

Fig. 1. Implementation of a Subject with interfaces and traits

2 Motivating Example

In this section we present our language and its core features from the programmer's point of view. We describe traits, boxes, ownership annotations and their combination.

Programming with Traits. In the language considered in this paper a trait consists of *methods*, of *required methods*, which parametrize the behavior, and of *required fields* that can be directly accessed in the body of the methods, along the lines of [2, 4]. Traits are building blocks to compose classes and other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state. Therefore a class composed by using traits has to provide the required fields. The trait composition operations considered in our language are as follows: A *basic trait* defines a set of methods and declares the required fields and the required methods. The *symmetric sum* operation, +, merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods) and have compatible requirements (two requirements on the same method/field name are compatible if they are identical). The operation `exclude` forms a new trait by removing a method from an existing trait. The operation `aliasAs` forms a new trait by adding a copy of an existing method with a new name. The original method is still available and, when a recursive method is aliased, its recursive invocation refers to the original method. The operation `renameTo` forms a new trait by renaming all the occurrences of a required field name or of a required/provided method name in an existing trait.

In Figure 1 we show an example of the subject part of a subject-observer pattern (this example will be extended with boxes in the rest of the section) in order to explain the main features of our traits and interfaces. Note that the trait `TCachedSubject` reuses `TSubject` by aliasing the method `notify` and provides another version of `notify` (this can be seen as method overriding in standard class-based inheritance). The

class CSubject implements the derived interface ICachedSubject by using the trait TCachedSubject and by declaring all the fields which are required by the trait. The trait TTopicHolder provides the required method getTopic.

Programming with Boxes. Our component model, the box model, extends the object-oriented programming world of interfaces, classes, objects, references, object-local state and methods with components, which we call boxes. Similar to an object, a box is a runtime entity, which is created dynamically, has an identity and a state. In general, it groups several objects together and its state is composed of the contained object states. During runtime each object belongs to exactly one box, thus defining a clear runtime boundary. A more detailed description including the discussion of design decisions and showing the use of the model for modular specification can be found in [20].

On source level, boxes are described by modules. A module is a set of classes and traits. Exactly one class for each module has to be a *box class*, which has to implement a *box interface*. The box class is the only class that is visible outside the module, all other classes and traits are not. When a box class is instantiated, a new box is created together with the object of the box class. The resulting object has the type of the corresponding box interface because classes cannot be used as types outside of their defining module. The box interface exactly defines which types, i.e., interfaces, the box exposes. Only types which transitively appear in the box interface as parameters or return types of methods can be used outside of a module. In particular, this means that downcasting of these types to not-exposed types is forbidden and checked at runtime. Note that a box is a runtime entity, so the downcasting of objects of another box is restricted even if it is of the same module. Boxes form a tree at runtime, with a special global box at the root. A box is nested in the box that created it. The main expression of the program is always evaluated in the global box.

Figure 2 shows the implementation of the subject-observer example with the use of boxes and ownership annotations. The ownership annotations are explained below. The implementation introduces a module with a box class KSubjectManager, which manages and uses the CSubject class to represent its subject. CSubject implements the ITopicHolder and the ICachedSubject interfaces. These interfaces, however, should not be visible to clients, because they reveal implementation details and allow the modification of the topic of the subject. Instead, clients should only be allowed to use the ISubject interface, which is a super type of ICachedSubject. In order to guarantee this with boxes, the box interface BSubjectManager, which is implemented by the box class, only returns the ISubject interface. This ensures that, outside the subject manager box, downcasts to more specific interfaces are not possible and result in runtime errors. Note that, inside a box, casts to owned objects are not restricted and behave like in standard Java. The cast restriction allows the implementation to be changed without breaking compatibility with clients, even in the presence of downcasting. This also improves the reasoning about the behavior and about properties of a box in a modular way (see [20] for more details).

Figure 3 shows an example client implementation. The subject can be obtained from the subject manager by using the getSubject method. This allows the client to register as an observer, because the ISubject interface can be used by the client. However, it is not possible for the client to downcast to the ICachedSubject interface, for example.

```
interface IObserver<a>{ void update(a String o); }
interface ISubject<a,b> {
  void register(a IObserver<b> o);
  void unregister(a IObserver<b> o);
}
interface ICachedSubject<a,b>
  extends ISubject<a,b> { ... }
interface ITopicHolder<a> {
  void setTopic(a String top);
  a String getTopic();
}
trait TSubject<a,b> is {
  local List<a IObserver<b>> obs;
  b String getTopic();
  ... // similar to Fig. 1
}
trait TCachedSubject<a,b> is
  TSubject<a,b>[notify aliasAs oldNotify]
              [exclude notify] + {
  ... // similar to Fig. 1
}
trait TTopicHolder<a> is ... // similar to Fig. 1
box interface BSubjectManager<a> { void init();
  boundary ISubject<a,global> getSubject(); }
```

```
module {
  box class KSubjectManager<a>
    implements BSubjectManager<a>
    by TSubjectManager<a> {
    boundary CSubject<a,global> subj;
  }
  trait TSubjectManager<a> is {
    boundary CSubject<a,global> subj;
    boundary ISubject<a,boundary> getSubject(){
      return subj; }
    void init() {
      subj = new boundary CSubject<a>(); }
    ... // further code
  }
  class CSubject<a>
    implements ICachedSubject<a,global>,
               ITopicHolder<global>
    by TCachedSubject<a,global> +
       TTopicHolder<global> {
    global String topic;
    global String prevTopic;
    local List<a IObserver<global>> obs;
  }
}
```

Fig. 2. Implementation of a Subject Manager

```
box interface BClient { void init(global BSubjectManager<owner> sm); }
module {
  box class KClient<a> implements BClient, IObserver<global> by TClient { }
  trait TClient is { void update(global String o) { ... }
                     void init(global BSubjectManager<owner> sm) {
                       sm.init(); sm.boundary ISubject<owner> s = sm.getSubject(); s.register(this);
                       ((ICachedSubject) s).getPrevTopic(); /* invalid cast */ }
} }
```

Fig. 3. An example usage of the `SubjectManager`-box

This allows the implementation of the `CSubject` class to be changed to use arbitrary other interfaces. In addition, the module can be analyzed in a modular way to guarantee that the topic of the internal subject cannot be changed outside of a box.

Programming with Ownership Annotations. The purpose of the box model is to define a precise boundary of object-oriented runtime components. In addition, the box model conceptually structures the heap into hierarchical components. One important aspect of components is encapsulation. Encapsulation is partly achieved by the cast restriction, which restricts what a client can do with an exposed object of a box, but it does not ensure that certain objects are never exposed by a box. To guarantee this aspect of encapsulation we add an ownership type system to our language. The basic idea is to group the objects of a box into distinct domains – a *local* and a *boundary* domain. Local objects are encapsulated in the box and cannot be referenced from the outside, boundary objects are accessible from the outside. The owner object of a box, i.e., the instance of the box class, is always accessible by the outside. It does not belong to the boundary domain of the box, instead it belongs to some domain of its surrounding box. In general the accessibility between objects follows three rules, called the *accessibility invariant*:

(*i*) objects in the same box can access each other, (*ii*) when an object can access a box, it can access the boundary objects of the box, (*iii*) objects can access any object of a surrounding box.

To guarantee the accessibility invariant, we extend the type system of our language with ownership annotations. A type `T` is annotated with a domain annotation `d` by writing `d T`. A domain annotation can be `local`, `boundary`, `owner`, or `global`. In addition, types can have domain parameters to express genericity in domain annotations. A domain annotation is always relative to a certain box. By default, this is the current box, i.e., the box of the `this`-object. For example, the type `local I` means that all instances of that type belong to the local domain of the current box. A box can also be explicitly specified by using a local variable referencing a box owner, i.e., an object of a box class. For example, the domain annotation `x.boundary` refers to the boundary domain of the box of the object referred by `x`. Note that in our language local variables are always `final`. The `owner` domain annotation refers to the domain in which the current `this`-object belongs to, the `global` domain represents the global domain. Types are only assignable if they have compatible domain annotations. Domain annotations are compatible if at runtime they always refer to the same domain. The domain annotations restrict the usage of types to guarantee the encapsulation of objects. In particular, it is guaranteed that all access paths from an object outside of a box to an object of a local domain of a box must go through either the owner of the box, or a boundary object. This is ensured by the type system by the restricting of assignments as described above and the prevention of certain domain annotations like `x.local`, for example.

In the subject manager example (cf. Figure 2) the `CSubject` object should be accessible outside the box. Thus it is put into the boundary domain. The `String` objects, which represent the topics of the subject are put into the global domain, as they should be accessible from everywhere. The `List` object of the subject, which holds the list of observers, however, should not be directly accessible from outside the box. It is thus put into the local domain. Note that the domain of the elements of the list, namely the observer objects, is left open as a parameter and is thus generic. To lower the annotation overhead for such a type system, type inference can be used (see [19]). This allows most annotations of classes and traits to be inferred and mainly requires to annotate interfaces. The client code in Figure 3 shows how the subject manager is used. The `KSubjectManager` instance is stored in a local variable `sm`. The domain of the subject object from the subject manager is then typed as `sm.boundary`. Figure 4 shows the runtime structure of the system after the method `init` is executed and the observer has been notified about some changes on the topic. The `KSubjectManager` object is the owner of its box. The `CSubject` object is in the boundary domain, the `List` object in the local domain of the box. The `KClient` can access the topic `String` objects, which live in the global domain, as well as the `CSubject` object, as it lives in the boundary domain of the `KSubjectManager` box. As the client has access to the box owner, it also has access to its boundary domain. The cast restriction of the box only allows the client to use the `ISubject` interface. The ownership type system guarantees that the client can never obtain a reference to the `List` object as it lives in the local domain of the `KSubjectManager` box.

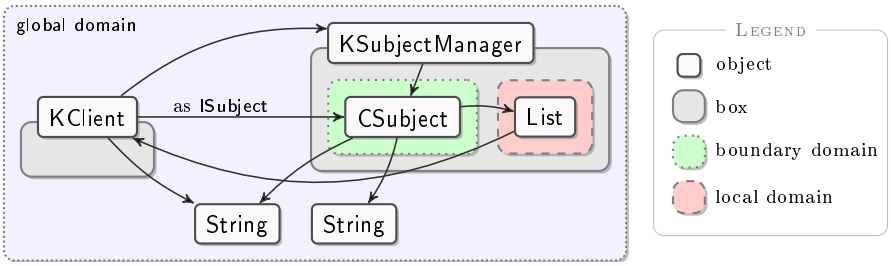

Fig. 4. Runtime view to the subject-observer example

Programming with Traits, Boxes and Ownership Annotations. A desirable feature of a programming language is the ability to support both fine- and coarse-grained reuse. The combination of traits and boxes provides this ability. Consider for instance the code in Figure 2. In order to better support encapsulation and ownership type-checking, classes are only visible inside the module where they are declared. The possibility of declaring traits at the top level (that is, outside of any module), like the traits `TSubject`, `TCatchedSubject` and `TTopicHolder`, makes it possible to reuse code across different modules. Type-checking a trait in isolation from the classes that use it is quite challenging. The problem is due the fact that, while type-checking the body of the method `m`, in order to be able to perform the ownership type-checks it is needed to know both the class `C` that contains the method `m` and the box class associated to the module containing the declaration of `C`. In this paper we address this problem by introducing a minimal core calculus for boxes and traits and by equipping it with a constraint-based ownership type system.

3 A Calculus for Boxes and Traits

Syntax. The syntax of our calculus, WELTERWEIGHT BOX TRAIT JAVA (WBTJ), is presented in Fig. 5. We use similar notations as FJ [12]. For instance: "$\bar{\mathtt{e}}$" denotes the possibly empty sequence "$\mathtt{e}_1, \ldots, \mathtt{e}_n$" and the pair "$\bar{\mathtt{N}}\ \bar{\mathtt{f}}$;" stands for "$\mathtt{N}_1\ \mathtt{f}_1; \ldots \mathtt{N}_n\ \mathtt{f}_n$;". The empty sequence is denoted by "$\bullet$" and the length of a sequence $\bar{\mathtt{e}}$ is denoted by $|\bar{\mathtt{e}}|$.

A program consists of interfaces, box interfaces, traits, modules and an expression, which represents the main method of the program. Every module contains one box class implementing a box interface, normal classes implementing normal interfaces and traits. The traits declared inside a module can be used only in the module itself, while the traits defined at the top level (that is, outside of any module) can be used by different modules. For simplicity, we assume that each class (and each box class) has a companion interface (resp. box interface) that it implements. Interfaces and box interfaces list the public methods of a class. The language has no constructors; new objects are created by setting all fields to `null`. Note that constructors can be simulated by ordinary method calls. Each class or box class declares fields and defines methods through a trait expression.

Classes are only visible inside their module, therefore within a module `MODULE` only objects of classes declared in `MODULE` can be created. In our simple language,

ID	::= [box] interface I$\langle\overline{\alpha}\rangle$ extends $\overline{\text{I}\langle\bar{\text{d}}\rangle}$ { $\bar{\text{S}}$; }	interfaces
S	::= N m ($\bar{\text{N}}$ $\bar{\text{x}}$)	method headers
N	::= I$\langle\bar{\text{d}}\rangle$	source types
G	::= N | C$\langle\bar{\text{d}}\rangle$	types
d	::= α | b.c | global	domain annotations
b	::= box | x | null | ?	domain owners
c	::= local | boundary	domain kinds
TD	::= trait T$\langle\overline{\alpha}\rangle$ is TE	traits
TE	::= { $\bar{\text{F}}$; $\bar{\text{S}}$; $\bar{\text{M}}$ } | T$\langle\bar{\text{d}}\rangle$ | TE + TE | TE[exclude m] | TE[m aliasAs m] | TE[m renameTo m] | TE[f renameTo f]	trait expressions
F	::= N f	fields
M	::= S { return e; }	methods
e	::= x | null | this.f | this.f = e | e.m($\bar{\text{e}}$) | new C$\langle\bar{\text{d}}\rangle$ | (N)e | let N x = e in e	expressions
CD	::= [box] class C$\langle\overline{\alpha}\rangle$ implements I$\langle\bar{\text{d}}\rangle$ by TE { $\bar{\text{F}}$; }	classes
MODULE	::= module { $\overline{\text{CD}}$ $\overline{\text{TD}}$ }	modules
PROGRAM	::= $\overline{\text{ID}}$ $\overline{\text{TD}}$ $\overline{\text{MODULE}}$ e	programs

Fig. 5. WBTJ: Syntax (I $\in$ interface names, T $\in$ trait names, C $\in$ class names, m $\in$ method names, f $\in$ field names, $\alpha, \beta \in$ domain parameters)

interface names and box interface names are the only source level types. It would be straightforward to extend the language to allow the programmer to use class names as source level types inside their module.

The set of expressions is quite standard. Just observe that, since interface names and box interface names are the only source level types, fields can be selected only on `this`.

For conciseness of the formalization, we have streamlined the notation of ownership annotations used in Sect. 2. Instead of writing the owning domain in front of the type, we now write it as the first parameter of the type. This also means that there is no `owner` keyword, because the first domain parameter always represents the owning domain. A domain annotation can either be a domain parameter α, the global domain, or is of the form *b.c*, where the first part defines the owner of the domain, and the second part defines the domain kind, that is, whether it is the boundary or local domain. The keyword `box` denotes the owner of the current box. The name of a local variable x is used for objects of box classes and denotes the box owned by x. For example, x.`local` denotes the local domain of the box owned by x. Owners `null` and ? do not belong to the user syntax (indicated by an underline), but can appear during reduction. ? as owner represents an invalid domain annotation, and `null` is the owner of the global domain. In fact, all occurrences of global are treated as `null.local`.

Flattening. The *flattening principle* has been introduced in the original formulation of traits in SQUEAK/SMALLTALK [11] in order to provide a canonical semantics to traits. Flattening states that the semantics of a method introduced in a class through a trait should be identical to the semantics of the same method defined directly within a class. This makes it possible to reason about the properties of a language with traits by relying on the semantics of the subset of the language without traits.

$$
\begin{array}{rcl}
[\![[\texttt{box}]\ \texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\bar{\texttt{d}}\rangle\ \texttt{by}\ \texttt{TE}\ \{\ \bar{\texttt{F}};\ \}]\!] & \stackrel{\text{def}}{=} & [\texttt{box}]\ \texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\bar{\texttt{d}}\rangle\ \texttt{by}\ \{\bar{\texttt{F}};\ \bullet;\ [\![\texttt{TE}]\!]\}\ \{\ \bar{\texttt{F}};\ \} \\
[\![\{\bar{\texttt{F}};\ \bar{\texttt{S}};\ \bar{\texttt{M}}\}]\!] & \stackrel{\text{def}}{=} & \bar{\texttt{M}} \\
[\![\texttt{T}\langle\bar{\texttt{d}}\rangle]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}[\bar{\texttt{d}}/\overline{\alpha}]]\!] \qquad \textbf{if}\ \texttt{trait}\ \texttt{T}\langle\overline{\alpha}\rangle\ \texttt{is}\ \texttt{TE} \\
[\![\texttt{TE}_1 + \texttt{TE}_2]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}_1]\!] \cdot [\![\texttt{TE}_2]\!] \\
[\![\texttt{TE}\ [\texttt{exclude}\ \texttt{m}]]\!] & \stackrel{\text{def}}{=} & \bar{\texttt{M}}' \cdot \bar{\texttt{M}}'' \qquad \textbf{if}\ [\![\texttt{TE}]\!] = \bar{\texttt{M}}' \cdot \texttt{M} \cdot \bar{\texttt{M}}''\ \textbf{and}\ \texttt{M} = \cdots\texttt{m}(\cdots)\{\cdots\} \\
[\![\texttt{TE}\ [\texttt{m}\ \texttt{aliasAs}\ \texttt{m}']]\!] & \stackrel{\text{def}}{=} & \bar{\texttt{M}} \cdot (\texttt{N}\ \texttt{m}'(\bar{\texttt{N}}\ \bar{\texttt{x}})\{\texttt{return}\ \texttt{e};\}) \\
 & & \qquad \textbf{if}\ [\![\texttt{TE}]\!] = \bar{\texttt{M}}\ \textbf{and}\ \texttt{N}\ \texttt{m}(\bar{\texttt{N}}\ \bar{\texttt{x}})\{\texttt{return}\ \texttt{e};\} \in \bar{\texttt{M}} \\
[\![\texttt{TE}[\texttt{f}\ \texttt{renameTo}\ \texttt{f}']]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}]\!][\texttt{f}'/\texttt{f}] \\
[\![\texttt{TE}[\texttt{m}\ \texttt{renameTo}\ \texttt{m}']]\!] & \stackrel{\text{def}}{=} & \mathit{mR}([\![\texttt{TE}]\!], \texttt{m}, \texttt{m}') \\
\mathit{mR}(\texttt{N}\ \texttt{n}(\bar{\texttt{N}}\ \bar{\texttt{x}})\{\texttt{return}\ \texttt{e};\}, \texttt{m}, \texttt{m}') & \stackrel{\text{def}}{=} & \texttt{N}\ \texttt{n}[\texttt{m}'/\texttt{m}](\bar{\texttt{N}}\ \bar{\texttt{x}})\{\texttt{return}\ \texttt{e}[\texttt{this.m}'/\texttt{this.m}];\} \\
\mathit{mR}(\texttt{M}_1 \cdot \ldots \cdot \texttt{M}_n, \texttt{m}, \texttt{m}') & \stackrel{\text{def}}{=} & (\mathit{mR}(\texttt{M}_1, \texttt{m}, \texttt{m}')) \cdot \ldots \cdot (\mathit{mR}(\texttt{M}_n, \texttt{m}, \texttt{m}'))
\end{array}
$$

Fig. 6. Flattening WBTJ to FWBTJ

In order to formalize flattening for WBTJ we consider a subset of the calculus that we call FWBTJ (FLAT WBTJ), where there are no trait declarations and the syntax of trait expressions is simplified as follows: $\texttt{TE} ::= \{\bar{\texttt{F}};\ \bullet;\ \bar{\texttt{M}}\}$. A FWBTJ class $\texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\bar{\texttt{d}}\rangle\ \texttt{by}\ \{\ \bar{\texttt{F}};\ \bullet;\ \bar{\texttt{M}}\ \}\ \{\ \bar{\texttt{F}};\}$ can be understood (modulo the domain annotations) as the standard JAVA class $\texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\bar{\texttt{d}}\rangle\ \{\ \bar{\texttt{F}};\ \bar{\texttt{M}}\ \}$. Similarly for box classes. Therefore, the canonical (static and dynamic) semantics for WBTJ can be specified by providing: (*i*) a semantics for FWBTJ, and (*ii*) a flattening translation that maps a WBTJ program into a FWBTJ program.

The flattening translation is specified through the function $[\![\cdot]\!]$, given in Figure 6, that maps each WBTJ class or box class declaration to a FWBTJ class or box class declaration, respectively, and maps a trait expression to a sequence of method declarations. We write $[\![\texttt{PROGRAM}]\!]$ to denote the program obtained from PROGRAM by dropping all the trait declarations and by translating all the class and box class declarations. The clauses in Figure 6 should be self-explanatory. Note that the flattening clause for field renaming is simpler than the flattening clause for method renaming (which uses the auxiliary function *mR*); this is due to the fact that fields can be accessed only on `this`.

Flattening aims only to provide a canonical semantics to traits, it is not an especially effective implementation technique (see, e.g., [17, 13]).

Typing. A desirable property of a formulation of traits within a statically typed programming language is to conform to the *trait type-checking in isolation principle* (that guided, for instance, the design of the pioneering experimental language CHAI$_2$ [25]). It states that typing must support the type-checking of traits in isolation from the classes or traits that use them. So that it is possible to type-check a method defined in a trait only once (instead of having to reinspect its code whenever that trait is used by a class or by another trait). In order to conform to the above principle a type system must be able to type-check a method definition without knowing the classes that will use it. This can be done by exploiting the constraint-base typing technology.

Ownership Typing for FWBTJ. Before introducing the constraint-based ownership type system for WBTJ, we introduce an ownership type system for FWBTJ. The ownership type system for FWBTJ, which is similar to the ownership type system described in [19], represents the specification for the constraint-based ownership type system for WBTJ. In the following we will use H to range over (box) class and (box) interface names. The ownership type rules are of the form $\Gamma;\mathtt{G};\mathtt{b} \vdash_o \ldots$, where Γ is type environment of local variables, G is the type of the current context class or interface, and b refers to the current box owner.

The subtyping relation of our language is quite standard, it is the transitive, reflexive closure of the `extends`- and `implements`-relation between interfaces and classes, except that box interfaces and box classes are subtypes of box interfaces only (analogous for normal classes and interfaces). This leads to two disjoint type hierarchies, which are needed to define the accessibility between domains (see below).

The typing rules for the ownership type system are mostly standard. The interesting rules are those for classes and for expressions. The rule for classes is:

(T-CLASS)

$$\frac{\emptyset;\mathtt{C}\langle\bar{\alpha}\rangle;\mathtt{box} \vdash_o \bar{\mathtt{N}} \qquad \mathtt{this} : \mathtt{C}\langle\bar{\alpha}\rangle;\mathtt{C}\langle\bar{\alpha}\rangle;\mathtt{box} \vdash_o \bar{\mathtt{M}} \qquad \mathit{isBoxType}(\mathtt{C}) \Leftrightarrow \mathit{isBoxType}(\mathtt{I}) \quad \text{All methods of interface I are implemented in } \bar{\mathtt{M}}}{\vdash_o \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{I}\langle\bar{\mathtt{d}}\rangle\ \mathtt{by}\ \{\bar{\mathtt{F}};\ \bullet;\ \bar{\mathtt{M}}\}\ \ \{\ \bar{\mathtt{N}}\ \bar{\mathtt{f}};\ \}}$$

A class is well-typed if all fields have valid types in the class C and the current box, and all methods are well-typed in the context of the class C. This rule (implicitly parametrized by the module in which C is defined) instantiates the context used in the rules for expressions and to test validity of types and accessibility of domains. For the typecheck on method calls and field selection expressions, the declared types of parameters and fields have to be translated to the domain in which they are used, i.e. the declared domains have to be replaced by actual ones. A domain parameter α is replaced by the actual domain, the `box` keyword is replaced depending whether the type to translate is a boxtype or not. For normal types, `box` is replaced by the owner of the type, for boxtypes `box` is translated to the owner if the expression in which the translation occurs denotes a valid owner. The translation function can introduce domain annotations with `?` as owner. These domains signify failed translations and, in the typing rules, the check for validity of types with these annotations will fail. The rule for field access is:

(T-FIELD)

$$\frac{\Gamma;\mathtt{G};\mathtt{b} \vdash_o \mathtt{this} : \mathtt{C}\langle\bar{\mathtt{d}}\rangle \qquad \mathtt{N} = \mathit{trans}(\mathtt{C}\langle\bar{\mathtt{d}}\rangle, \mathtt{this}, \mathit{ftype}(\mathtt{C},\mathtt{f})) \qquad \Gamma;\mathtt{G};\mathtt{b} \vdash_o \mathtt{N}}{\Gamma;\mathtt{G};\mathtt{b} \vdash_o \mathtt{this.f} : \mathtt{N}}$$

To type `this.f`, we have to find the type for `this`, lookup the declared type of the field `f` (done by *ftype*) and then translate the declared type into the current context. Then, the translated type has to be type correct as well. The rule for calls look similar, but translates the declared parameter types.

The first key element of the type system is the *accessibility relation* on domains presented in Figure 7. Judgments of the form $\Gamma;\mathtt{T_{cb}};\mathtt{b} \vdash_o \mathtt{d}_1 \rightarrow \mathtt{d}_2$ tell us that domain $\mathtt{d}_1$ can access domain $\mathtt{d}_2$ in the given context meaning that all objects in $\mathtt{d}_1$ can access all objects in domain $\mathtt{d}_2$. The relation formalizes the accessibility between domains depending on the ownership hierarchy of boxes. A domain has access to itself (A-REFL)

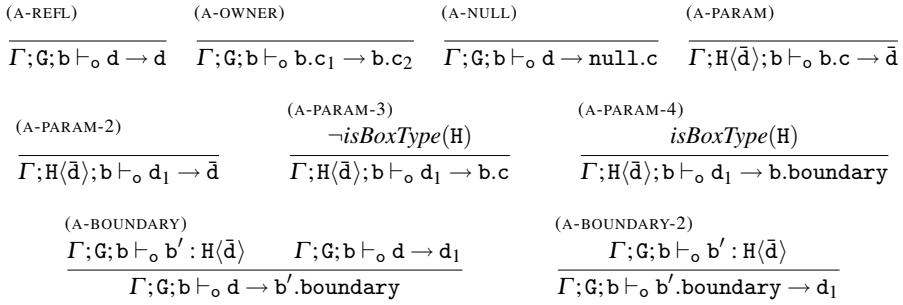

Fig. 7. Accessibility relation

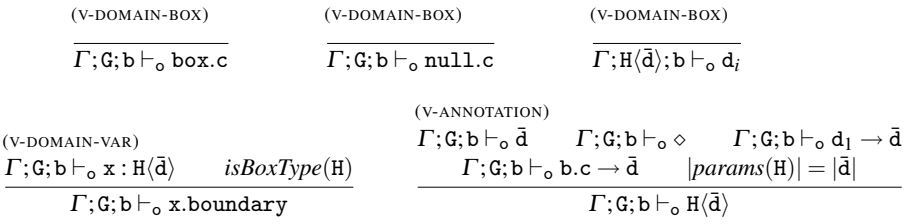

Fig. 8. Valid domains and types

and to the global domain (A-NULL). The domains with the same owner, i.e. belonging to the same box, can access each other (A-OWNER). The rules (A-PARAM) to (A-PARAM4) relate the domains of the current context type to the domains of the current box. (A-PARAM) states that the domains of the current box can access the domains of the context type and (A-PARAM2) says that the owning domain, i.e. d_1, has access to all parameter domains. These two rules guarantee that it is impossible to pass domains through other domains without following the box hierarchy. The owning domain of the context type, i.e. the domain of the `this` object, has access to the domains of the current box if it is not a box type (A-PARAM3). If the context type is a boxtype, its owning domain has only access to the boundary domain of the current box (A-PARAM4). This rule applies if the current box is an inner box of the box containing the context type and local domains of inner boxes are protected against the access from surrounding boxes. A domain can always access the boundary domain of a box, which it can access (A-BOUNDARY) and the boundary domain has always access to the owning domain of the box (A-BOUNDARY2).

The second key element of the ownership type system is the definition of *valid domains and types* shown in Figure 8. The notion of validity is strongly related to the accessibility relation. The most important rule is (V-ANNOTATION), which guarantees that breaking encapsulation by passing domains as parameters, which are not accessible by the first domain parameter, i.e. the domain of `this`, is impossible.

$$
\begin{array}{lll}
\psi & ::= & \mathit{isValid}(\mathtt{O}) \mid \mathit{isValid}(\mathtt{T}\langle\bar{\mathtt{d}}\rangle) \mid \mathtt{O} <: \mathtt{O}' \mid \mathtt{O} \not<: \mathtt{O}' \mid \mathtt{O} \neq \mathtt{O}' & \text{constraints} \\
 & \mid & \overline{\mathtt{X}} = \mathit{trans}(\mathtt{O}, \mathtt{e}, \mathtt{O}') \mid (\mathtt{O} \neq \chi)\ ?\ (\mathtt{O} <: \mathtt{O}')\ :\ (\mathit{req}(\mathtt{O}')) \mid \mathit{req}(\mathtt{N}) & \\
\mathtt{O} & ::= & \mathtt{G} \mid \mathtt{X} \mid \chi & \text{open types}
\end{array}
$$

Fig. 9. Constraints syntax

Constraint-based Ownership Typing for WBTJ. The constraint-based ownership type system for WBTJ can be understood as a refinement of the type system (for a Java-like calculus with traits) proposed in [4], to keep box and ownership annotations into account. The type-checking of each trait definition, in isolation from the classes or traits that use it, is realized by collecting, for each method definition in the trait, the constraints on the use of `this` within the method body. Each method $\mathtt{M} = \mathtt{S}\ \{\ \mathtt{return}\ \mathtt{e};\ \}$ defined within a basic trait expression $\{\ \bar{\mathtt{F}};\ \bar{\mathtt{S}};\ \bar{\mathtt{M}}\ \}$ is type-checked by assuming for `this` the structural type $\langle\ \bar{\mathtt{F}}\ \vdash\ \bar{\mathtt{S}}'\ \rangle$, where $\bar{\mathtt{F}}$ and $\bar{\mathtt{S}}'$ are the required fields and the headers of the required/provided methods of the basic trait expression, respectively. The typing judgment for method definitions has the form: $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \vdash\ \bar{\mathtt{S}}'\ \rangle \vdash_{\mathtt{co}} \mathtt{M} : \mathtt{S}\ \vdash\ \langle\ \bar{\mathtt{F}}'\ \vdash\ \bar{\mathtt{S}}''\ \vdash\ \bar{\mathtt{N}}\ \rangle \vdash \Phi$, where `S` is the header of the method; the triple $\langle\ \bar{\mathtt{F}}'\ \vdash\ \bar{\mathtt{S}}''\ \vdash\ \bar{\mathtt{N}}\ \rangle$ specifies that the body of the method (the expression `e`) selects the fields $\bar{\mathtt{F}}'$ ($\subseteq \bar{\mathtt{F}}$) and the methods with headers $\bar{\mathtt{S}}''$ ($\subseteq \bar{\mathtt{S}}'$) on `this`, requires that `this` has the types $\bar{\mathtt{N}}$; and Φ is the set of the validity ownership type-checks that can be performed only when the class or box class that is using the method is known. In particular: (i) $\bar{\mathtt{N}}$ contains the types of the formal parameters of methods to which `this` is passed as argument and, when the method returns `this`, also the return type of the method; (ii) the elements of Φ, ranged over by ψ, represent the ownership type-checks that would be performed by the ownership type system for FWBTJ when type-checking a class or box class containing the method definition `M`.

Let μ range over the the types assigned to method definitions. The typing judgment for trait expressions has the form $\vdash_{\mathtt{co}} \mathtt{TE} : \mu_1 ... \mu_n$, where $\mu_1, ..., \mu_n$ (with $n \geq 0$) are the types of the n methods defined by the trait expression `TE`.

The typing rule for class definitions $\mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{I}\langle\bar{\mathtt{d}}\rangle\ \mathtt{by}\ \mathtt{TE}\ \{\ \bar{\mathtt{F}};\}$ checks that the constraints in the types of the methods provided by `TE` are satisfied, that is: class `C` provides all fields and methods required by the trait expression `TE`; the trait expression `TE` has no required methods and provides all the methods of the interfaces implemented by `C`; each interface required by `TE` is a superinterface of some interface implemented by `C`; and all the constraints inferred for the methods provided by `TE` are satisfied within class `C`. The typing rule for box class definitions is similar.

The syntax of the constraints ψ representing the ownership type-checks is given in Figure 9, where `X` ranges over type variables that will be instantiated to types and χ is a distinguished type variable that will be instantiated to the type of `this`. A constraint can be: a type validity check ($\mathit{isValid}(\mathtt{O})$); an annotated trait name validity check ($\mathit{isValid}(\mathtt{T}\langle\bar{\mathtt{d}}\rangle)$); a subtype or type equality check; the binding of type variable to types generated by the *trans* function; a conditional constraint, depending on the type of `this`; or a required type for `this` ($\mathit{req}(\mathtt{N})$). Given a context $\Gamma; \mathtt{T}_{\mathtt{cb}}; \mathtt{b}$, all the type variables occurring in a set of constraints Φ are univocally bound to types. We will write $\Gamma; \mathtt{T}_{\mathtt{cb}}; \mathtt{b} \vdash_{\mathtt{co}} \Phi$ to mean that all the constraints in Φ are satisfied in the context $\Gamma; \mathtt{T}_{\mathtt{cb}}; \mathtt{b}$ according to the subtyping relation (not given for space reasons) and to the relation in

Figure 8. Note that rule (V-ANNOTATION) in Figure 8 has to be used also for checking annotated trait names, i.e., the metavariable H occurring in the rule can be also a trait name T. The rule for constraint-based type-checking of field access is the following:

$$\frac{\begin{array}{l}\text{(OTC-FIELD)}\\ \Gamma \vdash_{\mathtt{co}} \mathtt{this} : \langle\, \bar{\mathtt{F}} \mid \bar{\mathtt{S}} \,\rangle \mid \langle\, \bullet \mid \bullet \mid \bullet \,\rangle \mid \{\} \qquad N\mathtt{f} \in \bar{\mathtt{F}} \\ \mathtt{X} \text{ fresh} \qquad \Phi = \{\, \mathtt{X} = \mathit{trans}(\chi, \mathtt{this}, N),\ \mathit{isValid}(\mathtt{X}) \,\}\end{array}}{\Gamma \vdash_{\mathtt{co}} \mathtt{this.f} : \mathtt{X} \mid \langle\, \mathtt{Xf} \mid \bullet \mid \bullet \,\rangle \mid \Phi}$$

Note that the two constraints in the set Φ represent the validity check on the translation of the type of f, which cannot be performed until the class of this will be known.

The type τ of a closed expression e is of the form $\mathtt{H}\langle\bar{\mathtt{d}}\rangle$ where H is an interface, box interface, class or box class name. In fact, although class names and box class names cannot be used as types in the program, they are used by the type system to type object/box creation expressions. We say that a WBTJ program $\mathtt{PROGRAM} = \overline{\mathtt{ID}}\ \overline{\mathtt{ID}}\ \overline{\mathtt{TD}}\ \overline{\mathtt{MODULE}}\ \mathtt{e}$ is *well typed* with type τ to mean that all the interfaces, box interfaces, traits, classes, box classes in the program are well typed and that the expression e has type τ.

Properties. In order to define the properties of our type systems, we need an operational semantics. Thanks to flattening, in order to specify the semantics of WBTJ, it is enough to specify a semantics for FWBTJ. Our operational semantics is mainly a standard semantics with two extensions. First the semantics explicitly represents boxes as runtime entities, which are created whenever an instance of a box class is created. In addition, each object is mapped to the box, which it belongs to. The second extension is the restriction of casts. At runtime casts are checked to prevent illegal downcasts, i.e., casts to types, which are not defined in a box boundary. For more details we refer to [18]. Based on the operational semantics, we have the following central properties for the WBTJ type systems. The first property is *subject reduction*, which is a prerequisite to prove that objects during runtime can only access other objects according to their declared static domains.

Theorem 1 (Subject Reduction for $\vdash_{\mathtt{o}}$). *If an expression e is $\vdash_{\mathtt{o}}$-typable, then e can be reduced to e' and the type of e' is a subtype of the type of e.*

The proof is by the standard preservation and progress lemmas, which can be proved by standard structural induction. The main property of the ownership type system is the *accessibility invariant*. It states, which objects can access each other.

Theorem 2 (Accessibility Invariant for $\vdash_{\mathtt{o}}$-typable programs). *Objects can only access other objects which are: (i) in the same box, (ii) in the boundary domain of a box which they can access, (iii) in a surrounding box of their own box.*

The proof is by structural induction with the use of the subject reduction theorem and the accessibility and validity definitions.

Corollary 1 (Encapsulation Invariant for $\vdash_{\mathtt{o}}$-typable programs). *Objects in the local domain of a box cannot be accessed by objects of surrounding boxes.*

The following theorems (that can be proved by structural induction on typing derivations) state that the constraint-based ownership type system for WBTJ ($\vdash_{\mathtt{co}}$) satisfies the specification provided by the ownership type system for FWBTJ ($\vdash_{\mathtt{o}}$) and the conformance of the constraint-based ownership type system to the flattening principle, respectively. Therefore, $\vdash_{\mathtt{co}}$-typable programs enjoy the encapsulation invariant.

Theorem 3 (Equivalence of $\vdash_{\mathtt{co}}$-typability and $\vdash_{\mathtt{o}}$-typability on FWBTJ programs). *For every* FWBTJ *program* PROGRAM *it holds that* PROGRAM *is well typed with type* τ *in* $\vdash_{\mathtt{co}}$ *if and only if* PROGRAM *is well typed with type* τ *in* $\vdash_{\mathtt{o}}$.

Theorem 4 (Flattening preserves $\vdash_{\mathtt{co}}$-typing). *If the* WBTJ *program* PROGRAM *is well typed with type* τ *then the* FWBTJ *program* $[\![\mathtt{PROGRAM}]\!]$ *is well typed with type* τ.

4 Conclusion, Related and Future Work

The literature on traits and boxes has been partially quoted throughout the paper. Here we briefly discuss the relation with ownership type systems. The basic idea of the box component model, namely to hierarchically structure the heap into dynamically created regions, originated from ownership disciplines. They were originally developed to check confinement properties by type systems (see [7] for an introduction and overview; [5] for a system to check concurrency properties; [10, 22] for generic ownership type systems). Several different variants of ownership types exist, with varying expressiveness and flexibility. The original ownership type system by Clarke et al. [8] and similar systems [6, 9] enforces a so-called *owners-as-dominators* property, which states that all accesses from external objects to owned objects must go through the owner object. This property does not allow for multiple objects at the boundary of a component. The Universe type system [16, 10] is more flexible by permitting read-only references to owned objects. The ownership type system of Lu et al. [15] generalize this by using an additional accessibility modifier. Most closest to the ownership system presented in this paper is the general *Ownership Domains* [1] approach. The ownership system of this paper without traits together with an inference algorithm is presented in [19].

Currently we are working on the implementation of the programming language based on the WBTJ calculus (both on the constraint-based type system as well as on an runtime environment supporting boxes). Moreover, we are planning to extend the type system to deal with generics.

Acknowledgment. We thank the anonymous referees and Susan Eisenbach for comments and suggestions for improving the paper. The authors of this work have been partially supported by the German-Italian University Centre (Vigoni program).

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, LNCS 3086, pages 1–25. Springer, 2004.
2. L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *SAC*, pages 2096–2102. ACM, 2010.

3. V. Bono, F. Damiani, and E. Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *FTfJP*, 2007. (www.cs.ru.nl/ftfjp/).
4. V. Bono, F. Damiani, and E. Giachino. On Traits and Types in a Java-like setting. In *TCS (Track B)*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.
5. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, Feb. 2004.
6. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223. ACM Press, 2003.
7. D. Clarke. *Object Ownership and Containment*. PhD thesis, Univ. New South Wales, 2001.
8. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM Press, 1998.
9. D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe types for topology and encapsulation. In *FMCO*, LNCS 5382, pages 72–112. Springer, 2008.
10. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, LNCS 4609, pages 28–53. Springer, 2007.
11. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.
12. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
13. G. Lagorio, M. Servetto, and E. Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL*, 2009. (www.cs.hmc.edu/~stone/FOOL/).
14. L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM TOPLAS*, 30(2):1–32, 2008.
15. Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP*, LNCS 4067, pages 99–123. Springer, 2006.
16. P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *FTfJP*, 2000. (www.cs.ru.nl/ftfjp/).
17. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT*, 5(4):129–148, 2006.
18. A. Poetzsch-Heffter, J.-M. Gaillourdet, and J. Schäfer. Towards a fully abstract semantics for object-oriented program components. http://softech.cs.uni-kl.de/pub?id=129, July 2008.
19. A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Infering ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, LNCS 4444. Springer, 2007.
20. A. Poetzsch-Heffter and J. Schäfer. Modular specification of encapuslated object-oriented components. In *FMCO*, LNCS 4111, pages 313–341. Springer, 2006.
21. A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In *FMOODS*, LNCS 4468, pages 157–173. Springer, 2007.
22. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *OOPSLA*, pages 311–324. ACM Press, 2006.
23. J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP*, LNCS 4609, pages 373–398. Springer, 2007.
24. J. Schäfer and A. Poetzsch-Heffter. A parameterized type system for simple loose ownership domains. *Journal of Object Technology (JOT)*, 5(6):71–100, June 2007.
25. C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP*, LNCS 3586, pages 453–478. Springer, 2005.
26. C. Szyperski, D. Gruntz, and S. Murer. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
27. D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.