Tracking Linear and Affine Resources with Java(X)

Markus Degen, Peter Thiemann, and Stefan Wehr

Institut für Informatik, Universität Freiburg
{degen,thiemann,wehr}@informatik.uni-freiburg.de

Abstract. Java(X) is a framework for type refinement. It extends Java's type language with annotations drawn from an algebra X and structural subtyping in terms of the annotations. Each instantiation of X yields a different refinement type system with guaranteed soundness. The paper presents some applications, formalizes a core language, states a generic type soundness result, and sketches the extensions required for the full Java language (without generics).
The main technical innovation of Java(X) is its concept of activity annotations paired with the notion of droppability. An activity annotation is a capability which can grant exclusive write permission for a field in an object and thus facilitates a typestate change (strong update). Propagation of capabilities is either linear or affine (if they are droppable). Thus, Java(X) can perform protocol checking as well as refinement typing. Aliasing is addressed with a novel splitting relation on types.

1 Introduction

A programming language with a static type system eliminates common programming errors right from the start. For instance, the type system may guarantee that no operation receives an illegal argument. Each type system introduces abstraction to make types statically checkable. Thus, there are always programs that would run without errors but which are nevertheless rejected by the type system.

However, the information provided by the type system is not always sufficient to avoid a run-time error. For example, taking the head of a list may lead to a run-time error if the list is empty but this information is not represented in the list type. While there are refinement type systems capturing such information [15, 30], they are not widely used in production programming languages.

A related problem arises with non-trivial object life cycles [24]. Many objects progress through distinct states during their lifetime with state changes caused by method calls. In each state certain methods are disabled and calling them causes a run-time error. A standard type system cannot avoid such run-time errors because it is not aware of the evolving object states. Enhancing a type system to track these state changes is not straightforward because it requires assigning the same variable different types at different places in the program. Such a typestate change causes problems in the presence of aliases that keep obsolete type assumptions. The main challenge here is to keep track of aliasing to the extent that the change of typing is possible.

A type system with additional structure can supply the information needed for such applications. The first application is a type refinement setting that restricts the semantics of programs by incorporating explicit tests for predicates that refine the underlying types. The type soundness property for the extended system becomes more expressive because it guarantees that these predicates are always satisfied. The second kind requires extending refinement typing with accurate state tracking as provided, *e.g.*, by a linear type system [28]. Unfortunately, most systems do not provide a seamless integration, let alone migration, between standard types and linear types.

Our framework JAVA(X) addresses these issues with a family of annotated type systems and an automatic promotion of standard properties to linear ones. An annotated type system extends the type language of some existing system with value annotations. A value annotation restricts the meaning of the type it is attached to and thus enables the type soundness proof to express additional properties.[1] As refinements are domain specific, they are not hardwired into the system. JAVA(X) is parametrized over a partially ordered set (*poset*) of value annotations X, which a programmer can change and extend them easily.

An alternative approach might rely on pre-/post-conditions and invariants, which are stated as logical formulas, but annotations place less burden on the programmer. For our system, a *refinement designer* chooses a set of predicates on objects and abstracts them to a value annotation poset X. This poset can be tailored to the needs of a particular application domain. Thus, the annotations correspond to domain-specific, shrink-wrapped combinations of predicates that are lightweight and ready to use for the programmer, who has to understand the annotations but does not have to be an expert in logics.

In addition to the value annotations, JAVA(X) has a built-in notion of capabilities that can promote a value annotation to a linear or affine annotation. Capabilities are independent of the chosen annotation poset X. They are attached to individual field references via the *activity annotations* on a type. If a variable has an object type with an *active* capability for a field (and everything reachable from it), then the program may update the field with a new value through this variable.

Active capabilities are propagated in a linear manner, that is, at each time and for each field of a reachable object, there exists one access path (starting with a variable), the type of which has the active capability for this field, through which the field can be updated. Any other access path to the same field may only read the field but not update it. The JAVA(X) type system maintains the invariant that only one access path has the update privilege for an active field.

Beyond the update privilege, an active capability carries the most accurate value annotation for the current contents of the field. Hence, active capabilities are well suited for typestate changes. An update only changes the field type for the access path with the active capability. The types of the other access paths

[1] Thus, JAVA(X) performs type-based program analysis [22] in some sense.

(the aliases) do not have to change because they have sufficiently less accurate information.

Thus, active capabilities enable the linear handling of resources such as objects that change their state in reaction to method invocations. However, as described up to now, the system does not seem to allow us to discard such state changing objects because it insists on the invariant that there is always one access path with an active capability for the object. For this reason, JAVA(X) includes the notion of *droppability*. The analysis designer can declare certain states (that is, subsets of annotations) as droppable. If an object is in a droppable state, then its reference can be discarded regardless of its capabilities. In effect, an object in a droppable state is handled in an affine manner: its state is tracked accurately, its active capability cannot be duplicated, but the object may be discarded at any time. An object may switch between droppable and nondroppable states during its life time, just think of a file handle that must not be discarded as long as it is open (see Section 2.2).

Contributions. JAVA(X) is an extension of Java 1.4 with a parametrized annotated type system. Its annotations are drawn from a poset X of value annotations. There is a parametrized type soundness proof for a fully formalized subset MINIJAVA(X). Once a refinement designer supplies a new annotation poset X, a programmer can immediately take advantage of the new invariants guaranteed through it.

We have built a proof-of-concept implementation of a type checker for MINIJAVA(X).[2] The type checker processes all examples of Section 2.

The main novelty of JAVA(X) is the concept of an activity annotation as a capability for updating a field in an object. Activity annotations enable the promotion of the properties described by X to linear and affine properties, which can be tracked accurately and facilitate typestate change. The main technical innovation is the handling of aliasing via a splitting relation. This relation splits the capability for a resource between different access paths to it on a per-field basis.

Overview. Section 2 introduces JAVA(X) with two examples. Section 3 defines the essential core of the language JAVA(X) and its type system formally. Section 4 sketches the type soundness proof. Section 5 explains the extensions needed for the full Java system, and Section 6 discusses related work. Finally, Section 7 concludes.

2 Examples

We introduce our framework with two examples. The first defines an affine instance of the framework providing a refined typing discipline for an XML-processing library. The second is a linear instance tracking operations on files. We defer the formal definition of an instance of the framework to Section 3.2.

[2] http://proglang.informatik.uni-freiburg.de/projects/access-control/

2.1 JDOM Type Analysis

JDOM[3] is a popular Java API for manipulating XML. It views an XML document as a tree composed of nodes of types like `Element` and `Attribute`. Each node (except the root) has a parent field `p` indicating the element that it is attached to. JDOM's `Element` type provides a number of operations for manipulating the tree structure. The method `Element setAttribute(Attribute attr)`, which attaches an attribute node to an element node, serves as a typical example.

JDOM informally imposes a number of invariants on its XML representation. One of them is that "JDOM nodes may not be shared". JDOM enforces this invariant dynamically by checking a *detachment property*: If the attribute node has a non-null parent field then the `setAttribute` method throws an `IllegalAddException`. This exception occurs in the last line of the following example because it attempts to attach the node `attr` a second time.

```
Element p1     = new Element("a");
Element p2     = new Element("a");
Attribute attr = new Attribute("href", "http://www.jdom.org");
p1.setAttribute(attr);          // consumes attr; now attached
p2.setAttribute(attr);          // raises IllegalAddException
```

We now describe an instance of Java(X) which statically tracks the detachment property and rejects uses of `setAttribute(attr)` unless it is clear that `attr` is detached. The instance raises a type error for the example just shown.

In earlier work [27], one of the authors has proposed a type system for DOM. While the earlier system covers properties other than detachment, the present system obtains significantly stronger guarantees for detachment (see Section 6).

Detached Nodes. Static checking of the detachment property requires annotations to the `Attribute` type, which abstract over the state of the parent field `p` as in $\texttt{Attribute}\{\texttt{p} : \langle aa, \texttt{Element}\rangle\}$.[4] The type shows that placing the annotations requires expanding the types to (potentially recursive) record types. The activity annotation aa ranges over the set $\{\triangle(va), \triangledown, \Diamond\}$ where va is drawn from a *value annotation* poset $X_{\texttt{Element}} = (\mathcal{P}(\{\mathbf{N}, \mathbf{D}\}), \subseteq)$ with $\mathcal{P}$ denoting the power set. We abbreviate $\{\mathbf{N}\}$ to $\mathbf{N}$, $\{\mathbf{D}\}$ to $\mathbf{D}$, and $\{\mathbf{N}, \mathbf{D}\}$ to $\mathbf{ND}$. The elements of the poset abstract from the possible states of an `Element` reference. In $X_{\texttt{Element}}$, $\mathbf{N}$ stands for "is null" and $\mathbf{D}$ for "defined" (is not null).

The activity annotation aa provides the access capability. If an `Attribute` reference has its `p` field typed with an *active* annotation $\triangle(va)$, then the parent

[3] `http://www.jdom.org`

[4] In the full type, both the `Attribute` and the `Element` type carry an additional value annotation and there is a record describing the fields of the `Element`, too: $\langle va_A, \texttt{Attribute}\{\texttt{p} : \langle aa, \langle va_E, \texttt{Element}\{\dots\}\rangle\rangle\}\rangle$. In what follows, we concentrate on `Attribute` and generally omit the extra value annotations and the field types of `Element` for readability.

field may be modified through this reference and this modification changes the value annotation *va* in the type of the reference. If the annotation is *inactive* $\triangledown$, then the field is read-only through this reference and there is no extra value annotation. The *semi-active* annotation $\Diamond$ allows for unrestricted assignment, but does not provide any information through a value annotation. We ignore $\Diamond$ for a moment and come back to it on page 6.

The enclosed value annotation *va* approximates the status of the attribute's parent reference at run time. It is flow-sensitive and may be different for different uses of the same attribute. Between uses, the system propagates the information whether a node is detached in an affine manner: At most one reference to an attribute may carry definitive, active information about the node's parent field. Any other, aliasing reference must have an inactive type for its parent field.

Writing a signature for a method such as `setAttribute` of class `Element` requires one more ingredient. The signature must specify the effect of the method on the state of the object. This effect change the activity annotations only, the underlying Java type does not change:[5]

$$\texttt{Element setAttribute(Attribute}\{\texttt{p} : \langle\triangle(\mathbf{N} \leadsto \mathbf{D}), \texttt{Element}\rangle\}\ \texttt{attr}).$$

The $\mathbf{N} \leadsto \mathbf{D}$ annotation states that the `p` of the `attr` argument must be null ($\mathbf{N}$) before the method call and is not null ($\mathbf{D}$) afterwards. Thus, $\mathbf{N} \leadsto \mathbf{D}$ describes the effect of a method call like the pre- and post-condition of a specification. Effects only apply to active annotations because modifications are only allowed through active references.

Type checking the example from the beginning of this section with the `setAttribute` signature just given leads to a type error. The typing assumes that `new Attribute(...)` creates an attribute node without a parent, *i.e.*, its `p` field has annotation $\triangle(\mathbf{N})$. The comments indicate the typing after execution of the respective statement.

```
Element p1 = ...;
Element p2 = ...;
Attribute attr = new Attribute(...);  // attr : Attribute{p : ⟨△(N), Element⟩}
p1.setAttribute(attr);  // attr : Attribute{p : ⟨△(D), Element⟩}
p2.setAttribute(attr);  // type error: N required, D given
```

Aliasing. Let us now abstract over the pattern. Suppose there is a method `set2` that accepts two `Attribute`s and attaches each to its own element.

```
void set2 (Element p1, Element p2,
           Attribute{p : ⟨△(N) ⇝ △(D), Element⟩} a1,
           Attribute{p : ⟨△(N) ⇝ △(D), Element⟩} a2)
  { p1.setAttribute(a1); p2.setAttribute(a2); }
```

[5] Again, we take the liberty of abbreviating the full syntax, which defines the effect as a change of the type. The full argument type duplicates the whole structure: $\texttt{Attribute}\{\texttt{p} : \langle\triangle(\mathbf{N}), \texttt{Element}\rangle\} \leadsto \texttt{Attribute}\{\texttt{p} : \langle\triangle(\mathbf{D}), \texttt{Element}\rangle\}$.

It is not possible to invoke `set2` with two aliases of the same attribute:

```
Attribute attr = new Attribute(...);  // attr : Attribute{p : ⟨△(N), Element⟩}
set2(p1, p2, attr, attr);  // type error!
```

The type error occurs because Java(X) *splits* the type of `attr` at every point of use such that no active annotation is duplicated. Splitting is driven by a ternary relation $\cdot \succeq \cdot \mid \cdot$ on activity annotations. For the active annotation it holds that $\triangle(va) \succeq \triangle(va) \mid \triangledown$ and $\triangle(va) \succeq \triangledown \mid \triangle(va)$ so that $\triangle$ can only be split into itself and the inactive annotation $\triangledown$; the inactive annotation can only split into itself. Hence, the initial type of `attr` is split into the two types

$$\begin{aligned}&\texttt{Attribute}\{\texttt{p} : \langle \triangle(\mathbf{N}), \texttt{Element}\rangle\} \succeq \\ &\quad \texttt{Attribute}\{\texttt{p} : \langle \triangle(\mathbf{N}), \texttt{Element}\rangle\} \mid \texttt{Attribute}\{\texttt{p} : \langle \triangledown, \texttt{Element}\rangle\}\end{aligned}$$

one of which is assigned to each of the two occurrences of `attr` in the argument list of `set2`. Thus, one occurrence has a suitable argument type for this method, the other one has a mismatch between the required $\triangle(\mathbf{N})$ and the provided $\triangledown$.

JDOM also has API methods that introduce aliasing. For example, the `detach()` method removes an attribute from the element it is attached to (if any) and leaves it in a detached state. The method modifies its receiver object and returns it, too. One possible type signature is

$$\begin{aligned}&\texttt{Attribute}\{\texttt{p} : \langle \triangle(\mathbf{N}), \texttt{Element}\rangle\} \\ &\quad [\texttt{Attribute}\{\texttt{p} : \langle \triangle(\mathbf{ND}) \rightsquigarrow \triangledown, \texttt{Element}\rangle\}]\ \texttt{detach}()\end{aligned}$$

where the type change in the square brackets specifies the effect of a method invocation on the receiver type. Before calling `detach`, the receiver object must have an active parent field in arbitrary state, that is, the receiver may be detached or attached. (We have $\mathbf{N} \subseteq \mathbf{ND}$ and $\mathbf{D} \subseteq \mathbf{ND}$ in our annotation poset $X_{\texttt{Element}}$.) After the call, the receiver's parent field type is inactive. The method returns a detached active reference.

This type is not the only possible choice. We could just as well leave the receiver active and make the return type inactive. Each choice fixes a particular usage pattern, but there is no reason to prefer one over the other. Section 5 introduces annotation polymorphism which allows to defer this choice.

In summary, there are two invariants that guarantee soundness in the presence of aliases. If there is a reference to an object carrying an active annotation for some field, then all aliases have a type with an inactive annotation for this field. Updates are only possible for fields with an active annotation. Such an update also changes the active value annotation of the field.

Unrestricted Assignment. The active and inactive annotations that we have seen so far do not allow a field to be updated through multiple references. As realistic programs contain unrestricted assignments, we need the semi-active annotation $\Diamond$. This annotation neither imposes nor grants access restrictions; like $\triangledown$, it does not track value annotations exactly. Splitting does not affect semi-activity, *i.e.*, $\Diamond \succeq \Diamond \mid \Diamond$. If an alias for an object has a semi-active field, no other alias can have an active annotation for this field. Semi-active fields behave like ordinary instance variables in Java or C#. Hence, semi-active is the default annotation for fields.

Semi-active annotations enable the incremental transition to refined types. For example, the JDOM method `Element setAttribute(Attribute attr)` may initially receive the signature

$$\texttt{Element setAttribute}(\texttt{Attribute}\{\texttt{p} : \langle\Diamond, \texttt{Element}\rangle\}\ \texttt{attr}).$$

Once we decide to track detachment, we switch to the active annotation discussed in the previous subsection.

2.2 File Access

The detachment property of the preceding section is affine because we can always drop an attribute node. We now give an example of a linear property where only values carrying a distinguished annotation may be dropped.

The problem statement is as follows. Opening a file creates an open file handle on which the program can perform read operations until the file handle is closed. No further operation can be performed on a closed file handle. Furthermore, file handles must not be discarded while they are open.

We use the value annotation poset $X_{\texttt{FStat}} = \mathcal{P}(\{\mathbf{O}, \mathbf{C}\})$ for the file access example, where $\mathbf{O}$ stands for "open" and $\mathbf{C}$ stands for "closed". As before, we write $\mathbf{O}$, $\mathbf{C}$, and $\mathbf{OC}$ for the evident elements of $X_{\texttt{FStat}}$. Droppability of files is defined in terms of a droppability predicate, $\rho_{\texttt{FStat}} \subseteq X_{\texttt{FStat}}$. Because an open file must not be discarded, we define $\rho_{\texttt{FStat}}$ as $\{\emptyset, \mathbf{C}\}$.

To be able to change the status of a file, we do not attach these value annotations directly to the `File` class but to a private instance field `FStat status`.There are two distinguished `FStat` objects, namely $\texttt{open} : \langle\mathbf{O}, \texttt{FStat}\{\}\rangle$, and $\texttt{closed} : \langle\mathbf{C}, \texttt{FStat}\{\}\rangle$. The outermost value annotation of a type, which we have ignored until now, describes a persistent property of the values inhabiting the type. By assigning one of these two values to the `status` field, the implementation of the `File` class communicates its internal status to the outside world. The operations provided by `File` are as follows:

```
File{status : ⟨△(O), FStat⟩}(String name)    // constructor
int  [File{status : ⟨△(O), FStat⟩}]          read()
void [File{status : ⟨△(O) ⇝ △(C), FStat⟩}] close()
```

These method types implement exactly the specification given at the beginning of this subsection: `read()` is only possible in state $\mathbf{O}$ and `close()` changes the state to $\mathbf{C}$. An open file handle cannot be dropped because $\mathbf{O} \notin \rho_{\texttt{FStat}}$. A closed file handle can be dropped because $\mathbf{C} \in \rho_{\texttt{FStat}}$.

With these signatures, the following statements result in a type error.

```
File{status : ⟨△(O), FStat⟩} f = new File("/etc/passwd");
```

Syntax:

$$\begin{array}{rcl@{\qquad}rcll}
P & ::= & \overline{defn}\ e & s & ::= & \langle aa, t\rangle & \text{(coinductively)}\\
defn & ::= & \texttt{class}\ c\ \{\overline{c\,f};\ \overline{meth}\} & t & ::= & \langle va, u\rangle & \text{(coinductively)}\\
meth & ::= & t\,[t \rightsquigarrow t]\ m(\overline{t \rightsquigarrow t\ x})\ \{\, e\,\} & u & ::= & c\{\overline{f : s[\varsigma]}\} & \text{(coinductively)}\\
v & ::= & x \mid \texttt{null} & aa & ::= & \triangle(va) \mid \Diamond \mid \nabla & \\
e & ::= & v \mid \texttt{new}^{\ell}\ c(\overline{v}) \mid v.m(\overline{v}) & \varsigma & ::= & \text{R} \mid \text{W} \mid \text{B} & \\
 & \mid & \texttt{let}\, x = v.f\ \texttt{in}\, e \mid \texttt{set}\, v.f = v\ \texttt{in}\, e & A & ::= & \emptyset \mid A, x : t & \\
 & \mid & \texttt{let}\, x = e\ \texttt{in}\, e \mid \texttt{if}\, e\ \texttt{then}\, e\ \texttt{else}\, e & & & & \\
 & \mid & \texttt{join}\, v = v.f\ \texttt{from}\, e & & & &
\end{array}$$

$c \in \mathit{ClassName}, f \in \mathit{FieldName}, m \in \mathit{MethodName}, x \in \mathit{VarName}, va \in X_c, \ell \in \mathit{Label}$

Lookup functions:

$$\frac{\texttt{class}\ c\ \{\overline{c\,f};\ \overline{meth}\} \in P}{\mathit{fields}_P(c) = \overline{c\,f}} \qquad \frac{\begin{array}{c}\texttt{class}\ c\ \{\overline{c\,f};\ \overline{meth}\} \in P\\ t\,[t_0 \rightsquigarrow t_0']\ m(\overline{t_i \rightsquigarrow t_i'\ x_i})\ \{\, e\,\} \in meth\end{array}}{\mathit{mbody}_P(c, m) = \overline{x_i}\{e\}}$$

$$\frac{\texttt{class}\ c\ \{\overline{c\,f};\ \overline{meth}\} \in P \qquad t\,[t_1 \rightsquigarrow t_1']\ m(\overline{t_i \rightsquigarrow t_i'\ x_i}^{i\in\{2,\ldots,n\}})\ \{\, e\,\} \in meth}{\mathit{mtype}_P(\langle va, c\{\overline{f : s[\varsigma]}\}\rangle, m) = \overline{t_i \rightsquigarrow t_i'} \to t}$$

Fig. 1: Syntax and lookup functions.

```
f.read();
f.close(); // f now has type File{status : ⟨△(C), FStat⟩}
f.read(); // type error: O expected, C given
```

3 The Language MiniJava(X)

The language MiniJava(X) is an object-oriented language with classes and methods but without inheritance, interfaces, casts, and abstract methods. Its formalization is inspired by ClassicJava [13]. Section 5 discusses the extensions needed for all of Java 1.4.

3.1 Syntax

Figure 1 defines the syntax of MiniJava(X) and some auxiliary functions for accessing pieces of syntax. The notation $\overline{z_i}$ stands for $z_1, \ldots, z_n$, where z is a syntactic entity. The index i can be omitted if no ambiguity arises. We write $\overline{z_i}^{i\in M}$ and $\overline{z_i}^{i\neq j}$ to constrain the index set.

A program consists of a list of class definitions and a main expression. A class definition contains definitions for fields and methods. A method definition $t\,[t' \rightsquigarrow t'']\ m(\overline{t_i \rightsquigarrow t_i'\ x})\ \{\, e\,\}$ specifies the type t' of its receiver in the square brackets $[t' \rightsquigarrow t'']$. Calling the method changes the receiver type from t' to t'' and the argument types from t_i to t_i', respectively. The type change only refers to a change of the annotations, the underlying class type remains unchanged. The type syntax has three levels:

- A *simple type*, u, packages a class name c with a field environment; the field environment records for every field f its field type s and its variance ς, which specifies if the field is only read ($\varsigma = \mathrm{R}$), only written ($\varsigma = \mathrm{W}$), or both ($\varsigma = \mathrm{B}$).
- An *annotated type*, t, attaches a value annotation va to a simple type u to describe a persistent property of the objects of type u in a summary approximation. The annotation va is drawn from a type-specific annotation poset $(X_c, \cdot \leq \cdot)$ where c is the class of u. The instantiation of the framework determines X.
- A *field type*, s, attaches an activity annotation aa to an annotated type t. If a reference to an object has the field type $f : s[\varsigma]$ and s carries the active annotation $\vartriangle(va)$, that is, $s = \langle \vartriangle(va), va', u\rangle := \langle \vartriangle(va), \langle va', u\rangle\rangle$, then the field f may be updated *through this reference* and va describes the current field value. The annotation va is at least as precise as the summary approximation va' because assignments change va but leave va' constant. The well-formedness predicate on types (see Figure 3) ensures that $va \leq va'$.

  The activity annotation may also be semi-active $\Diamond$, which allows updates, or inactive $\triangledown$, which indicates that a field is read-only. In both cases, the system maintains only a summary approximation for the field value.

  An activity annotation acts locally on a single field. It does not affect sibling fields nor descendants: their annotations are completely independent. The activity annotation is also reference specific: each alias for the same object may have a different (but compatible) activity annotation on its type. For instance, compatibility enforces that only one alias may have an active annotation for a certain field of an object.

As customary for modeling object types [6], a type may be recursive through its field environment. The syntax does not have explicit operators to introduce or eliminate such recursive types. Instead, the rules of the type grammar have a coinductive interpretation.

A type environment A binds variables x to annotated types t. When writing $A, x : t$ we assume that A does not already bind x.

Expressions e are in a particular restricted form (which resembles A-normal form [12]) to maximize the amount of information that typing can extract and to simplify the soundness proof. In this form, all essential computations only take values v as operands (that is, a variable or `null`) and sequencing is made explicit using `let` (and field access/modification). Any expression in, say, Java syntax can be easily transformed into this form without changing its meaning.

The expression language comprises values, object creation, method invocation, field access, field modification, let expression, a conditional which tests for `null`, and an intermediate join expression `join` $v = v.f$ `from` e which does not occur in programs but which arises during execution.

Every `new` expression carries a unique label ℓ, so that the initial value annotation of an object may depend on the place of the `new` expression in the source program.

Field access $\texttt{let}\,x = v.f\,\texttt{in}\,e$ is combined with variable binding to increase the precision of the system. The idea is that the binding of x "lends" capabilities from $v.f$ while evaluating e. Afterwards, the lent capabilities are joined back to $v.f$'s using a join expression.

Field update $\texttt{set}\,v.f = v'\,\texttt{in}\,e$ first sets the field and then evaluates e. It does not return a result because doing so would create an alias for v', which would further complicate its typing rule.

3.2 Instances of MiniJava(X).

An instance of MiniJava(X) specifies, for each class c,

– a partially ordered set $(X_c, \leq)$ with least element for the value annotations;
– a non-empty predicate $\rho_c \subseteq X_c$ of droppable annotations such that $b \in \rho_c$ if $a \in \rho_c$ and $b \leq a$ (for all $a, b \in X_c$);
– predicates $R^{\texttt{new}}_{\ell,c}, R^{\texttt{null}}_c \subseteq X_c$, for each label ℓ, such that $b \in R^{\texttt{new}}_{\ell,c}$ ($b \in R^{\texttt{null}}_c$) if $a \in R^{\texttt{new}}_{\ell,c}$ ($a \in R^{\texttt{null}}_c$) and $a \leq b$ (for all $a, b \in X_c$).

We assume that $c \neq c'$ implies $X_c \cap X_{c'} = \emptyset$ and set $\rho := \cup_c \rho_c$.

The predicates $R^{\texttt{null}}_c$ and $R^{\texttt{new}}_{\ell,c}$ provide the persistent annotations for the `null` reference and for objects created at program location ℓ, respectively. Indeed, the motivation for including ℓ in the formal presentation at all is the ability to define predicates that depend on the creation location. Otherwise, the system would only be able to capture the nullness property. Several instances of value annotations may easily be combined using the Cartesian product.

Examples. The nullness analysis required for the JDOM detachment property works on the poset $X_{\texttt{Element}} = (\mathcal{P}(\{\mathbf{N}, \mathbf{D}\}), \subseteq)$ with $\rho_{\texttt{Element}} = X_{\texttt{Element}}$ and the two predicates $R^{\texttt{null}}_{\texttt{Element}}(va) \Leftrightarrow \{\mathbf{N}\} \leq va$ and $R^{\texttt{new}}_{\ell,\texttt{Element}}(va) \Leftrightarrow \{\mathbf{D}\} \leq va$. That is, every object is droppable regardless of whether it has a parent object. Further, the value annotation for `null` must contain $\mathbf{N}$, and the annotation of a newly created object must contain $\mathbf{D}$.

The file access example uses the poset $X_{\texttt{FStat}} = (\mathcal{P}(\{\mathbf{O}, \mathbf{C}\}), \subseteq)$ with $\rho_{\texttt{FStat}} = \{\{\mathbf{C}\}, \emptyset\}$. The two predicates are defined as $R^{\texttt{null}}_{\texttt{FStat}}(va) \Leftrightarrow \mathsf{False}$ and $R^{\texttt{new}}_{\ell,\texttt{FStat}}(va) \Leftrightarrow (\ell = \ell^o \Rightarrow \{\mathbf{O}\} \leq va) \wedge (\ell = \ell^c \Rightarrow \{\mathbf{C}\} \leq va)$ where ℓ^o and ℓ^c are the program locations where the `FStat` object `open` and `closed` are defined, respectively. That is, a file handle is droppable as long as its status cannot be open. The value annotation of a file status object must contain $\mathbf{O}$ if it was created at location ℓ^o and analogously for `C` and ℓ^c.

3.3 Dynamic Semantics.

Figure 2 defines the dynamic semantics of MiniJava(X) as a small-step operational semantics. Its judgment $P \vdash \langle e, \mathcal{S}\rangle \hookrightarrow \langle e', \mathcal{S}'\rangle$ describes a single evaluation step of an expression e under store $\mathcal{S}$ governed by program P. The evaluation step produces a new expression e', and a new store $\mathcal{S}'$.

Definitions:

$Value \ni w ::= l \mid \texttt{null}$

$\mathcal{E} ::= [\,] \mid \texttt{let}\, x = \mathcal{E}\, \texttt{in}\, e \mid \texttt{if}\, \mathcal{E}\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{join}\, w = l.f\, \texttt{from}\, \mathcal{E}$

$l \in Loc \subseteq VarName,\ \mathcal{S} \in Store = Loc \to ClassName \times Label \times FieldMap,$

$\mathcal{F} \in FieldMap = FieldName \to Value$

Reduction rules:

$P \vdash \langle \mathcal{E}[\texttt{new}^{\ell}\ c(\overline{w_i})]; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[l]; \mathcal{S}, l \mapsto \langle c, \ell, \overline{f_i \mapsto w_i}\rangle\rangle$
  if $fields_P(c) = \overline{c_i\, f_i}$

$P \vdash \langle \mathcal{E}[\texttt{let}\, x = l.f\, \texttt{in}\, e]; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[\texttt{join}\, w = l.f\, \texttt{from}\, [w/x]e]; \mathcal{S}\rangle$
  if $\mathcal{S}(l) = \langle c, \ell, \mathcal{F}\rangle$ **and** $\mathcal{F}(f) = w$

$P \vdash \langle \mathcal{E}[\texttt{join}\, w = l.f\, \texttt{from}\, w']; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[w']; \mathcal{S}\rangle$

$P \vdash \langle \mathcal{E}[\texttt{set}\, l.f = w\, \texttt{in}\, e]; \mathcal{S}, l \mapsto \langle c, \ell, \mathcal{F}\rangle\rangle \hookrightarrow \langle \mathcal{E}[e]; \mathcal{S}, l \mapsto \langle c, \ell, \mathcal{F}[f \mapsto w]\rangle\rangle$

$P \vdash \langle \mathcal{E}[l.m(\overline{w_i})]; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[\texttt{let}\, this, \overline{x_i} = l, \overline{w_i}\, \texttt{in}\, e]; \mathcal{S}\rangle$
  if $\mathcal{S}(l) = \langle c, \ell, \mathcal{F}\rangle$ **and** $mbody_P(c, m) = \overline{x_i}\{e\}$

$P \vdash \langle \mathcal{E}[\texttt{let}\, x = w\, \texttt{in}\, e]; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[[w/x]e]; \mathcal{S}\rangle$

$P \vdash \langle \mathcal{E}[\texttt{if}\, l\, \texttt{then}\, e_1\, \texttt{else}\, e_2]; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[e_1]; \mathcal{S}\rangle$

$P \vdash \langle \mathcal{E}[\texttt{if}\, \texttt{null}\, \texttt{then}\, e_1\, \texttt{else}\, e_2]; \mathcal{S}\rangle \hookrightarrow \langle \mathcal{E}[e_2]; \mathcal{S}\rangle$

$P \vdash \langle \mathcal{E}[\texttt{let}\, x = \texttt{null}.f\, \texttt{in}\, e]; \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}; \mathcal{S}\rangle$

$P \vdash \langle \mathcal{E}[\texttt{set}\, \texttt{null}.f = w\, \texttt{in}\, e]; \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}; \mathcal{S}\rangle$

$P \vdash \langle \mathcal{E}[\texttt{let}\, x = \texttt{null}.m(\overline{w})\, \texttt{in}\, e]; \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}; \mathcal{S}\rangle$

Fig. 2: Dynamic semantics.

A store $\mathcal{S}$ is a mapping from locations l to objects $\langle c, \ell, \mathcal{F}\rangle$ where c is the class of the object, ℓ is the place where the object was created, and the field map $\mathcal{F}$ records the values w of its instance fields. The notation $\mathcal{S}, l \mapsto \langle c, \ell, \mathcal{F}\rangle$ assumes that $\mathcal{S}$ does not bind l, whereas $\mathcal{F}[f \mapsto w]$ implies that $\mathcal{F}$ contains a binding for f which is updated to w. The reduction rules for `new`, `let`, and `if` are standard.

The reductions for $\texttt{let}\, x = v.f\, \texttt{in}\, e$ and $\texttt{join}\, w = l.f\, \texttt{from}\, w'$ belong together. They implement the aforementioned lending of the field's capabilities to x. Reducing the `let` leaves behind a `join` expression that remembers the lending for the duration of e's evaluation. Once the body of the `let/join` is reduced to a value, the `join` reduces. Thus, the join expression has no operational significance, it's just there to make the type system happy.

The reduction for `set` is standard but it is sequenced with the evaluation of another expression to avoid returning a value from `set`.

A method invocation reduces to the corresponding method body wrapped in `let` expressions that bind the formal parameters to the actual ones. Operationally, this wrapping is not necessary but it simplifies the soundness proof by separating concerns.

Beyond the explicit errors, an expression becomes stuck if it tries to access a non-existent field of an object or invoke a non-existent method. The latter errors are already captured by the underlying standard type system.

3.4 Static Semantics.

This section specifies the static semantics of MiniJava(X). Figure 3 defines various relations on types, annotations, and environments. Because types are defined coinductively, all rules involving types have a coinductive interpretation. Figure 4 defines the typing rules for expressions and some auxiliary judgments, Figure 5 lists additional typing rules for intermediate expressions that only arise during the evaluation of a program. Figure 6 contains the remaining rules for programs. Boxed premises in the rules serve as extension points provided by an instance of the framework.

Droppability. A program can only discard a reference if its type is droppable. This policy ensures that the program keeps at least one reference to each "precious" resource, which it recognizes by an active annotation with a non-droppable value annotation. Technically, an object is droppable if all its fields have droppable types. An active field type is droppable if its annotation is. A field which is semi-active or inactive can always be dropped: A semi-active field can only have droppable annotations (by well-formedness of types), and an object is never responsible for the contents of its inactive fields.

Splitting. If a program uses the same variable multiple times, then each use of the variable receives a different type where the activity annotations on the original type of the variable are split among all uses. If field type s splits into s' and s'' ($s \succeq s' \mid s''$), then s, s', and s'' are structurally equivalent and differ only in their activity annotations. So it is sufficient to define splitting on the activity annotations. Splitting of $\triangledown$ and $\Diamond$ is trivial. An active annotation splits into one active and one inactive annotation: both $\triangle(va) \succeq \triangle(va) \mid \triangledown$ and $\triangle(va) \succeq \triangledown \mid \triangle(va)$ are acceptable. Splitting ensures that at most one type for a field reference receives an active annotation.

Well-formedness. The well-formedness relation ensures that the value part of an active annotation of a field type is not weaker then the summary approximation for that field; that a semi-active field type is droppable; that a value annotation is taken from the appropriate annotation poset; and that a field environment is correct with respect to the field declarations of the corresponding class.

Subtyping. Subtyping is structural and derived from the annotation orderings. Moreover, an active field type can be treated as semi-active or inactive if it is droppable. The subtyping of field environments takes the variance ς into account: if a field is only read ($\varsigma = \text{R}$), then it can be treated covariantly; if it is only written ($\varsigma = \text{W}$), then contravariantly; if it is read and written ($\varsigma = \text{B}$), then it must be treated invariantly (Pierce [23, Chapter 15.5] attributes this technique to Reynolds).

Effect application. The effect application relation $\vdash_A A := A' \downarrow \overline{v_i : t_i \rightsquigarrow t'_i}$ is used in the rules for method application and for a restricted version of the

Droppability:

$$\frac{\boxed{va \in \rho} \qquad \vdash_t \rho(t)}{\vdash_s \rho(\langle \Delta(va), t\rangle)} \qquad \vdash_s \rho(\langle \Diamond, t\rangle) \qquad \frac{\vdash_t \rho(t)}{\vdash_s \rho(\langle \nabla, t\rangle)} \qquad \frac{(\forall i)\ \vdash_s \rho(s_i)}{\vdash_t \rho(\langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle)}$$

Splitting:

$$\frac{\vdash_{aa} aa \succeq aa' \mid aa'' \qquad \vdash_u u \succeq u' \mid u''}{\vdash_s \langle aa, va, u\rangle \succeq \langle aa', va, u'\rangle \mid \langle aa'', va, u''\rangle} \quad \frac{(\forall i)\ \vdash_s s_i \succeq s_i' \mid s_i''}{\vdash_u c\{\overline{f_i : s_i[\varsigma_i]}\} \succeq c\{\overline{f_i : s_i'[\varsigma_i]}\} \mid c\{\overline{f_i : s_i''[\varsigma_i]}\}}$$

$$\vdash_{aa} \Delta(va) \succeq \Delta(va) \mid \nabla \qquad \vdash_{aa} \Delta(va) \succeq \nabla \mid \Delta(va) \qquad \vdash_{aa} \Diamond \succeq \Diamond \mid \Diamond \qquad \vdash_{aa} \nabla \succeq \nabla \mid \nabla$$

Well-formedness:

$$\frac{\boxed{va \leq va'} \qquad P \vdash_t wf(\langle va', u\rangle)}{P \vdash_s wf(\langle \Delta(va), va', u\rangle)} \qquad \frac{\vdash_t \rho(t) \qquad P \vdash_t wf(t)}{P \vdash_s wf(\langle \Diamond, t\rangle)} \qquad \frac{P \vdash_t wf(t)}{P \vdash_s wf(\langle \nabla, t\rangle)}$$

$$\frac{va \in X_c \qquad \mathit{fields}_P(c) = \overline{c_i\, f_i} \qquad (\forall i)\ s_i = \langle aa_i, va_i, c_i\{\overline{f_{i_j} : s_{i_j}[\varsigma_{i_j}]}\}\rangle \quad P \vdash_s wf(s_i)}{P \vdash_t wf(\langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle)}$$

Subtyping:

$$\frac{\vdash_t t \leq t' \qquad \boxed{va \leq va'}}{\vdash_s \langle \Delta(va), t\rangle \leq \langle \Delta(va'), t'\rangle} \qquad \frac{\begin{array}{c}\vdash_t t \leq t' \\ aa \in \{\Diamond, \nabla\} \qquad \vdash_t \rho(t)\end{array}}{\vdash_s \langle \Delta(va), t\rangle \leq \langle aa, t'\rangle} \qquad \frac{\begin{array}{c}\vdash_t t \leq t' \\ aa \in \{\Diamond, \nabla\}\end{array}}{\vdash_s \langle \Diamond, t\rangle \leq \langle aa, t'\rangle}$$

$$\frac{\vdash_t t \leq t'}{\vdash_s \langle \nabla, t\rangle \leq \langle \nabla, t'\rangle} \qquad \frac{\boxed{va \leq va'} \qquad (\forall i)\ \varsigma_i' \vdash_s s_i \leq s_i' \quad \vdash_\varsigma \varsigma_i \leq \varsigma_i'}{\vdash_t \langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle \leq \langle va', c\{\overline{f_i : s_i'[\varsigma_i']}\}\rangle}$$

$$\frac{\vdash_s s \leq s' \qquad \vdash_s s' \leq s}{\mathrm{B} \vdash_s s \leq s'} \qquad \frac{\vdash_s s \leq s'}{\mathrm{R} \vdash_s s \leq s'} \qquad \frac{\vdash_s s' \leq s}{\mathrm{W} \vdash_s s \leq s'} \qquad \vdash_\varsigma \varsigma \leq \varsigma \qquad \vdash_\varsigma \mathrm{B} \leq \mathrm{R}$$

$$\vdash_\varsigma \mathrm{B} \leq \mathrm{W} \qquad \vdash_A \emptyset \leq \emptyset \qquad \frac{\vdash_A A \leq A' \qquad \vdash_t t \leq t'}{\vdash_A A, x : t \leq A', x : t'}$$

Effect application:

$$\vdash_A A := A \downarrow \overline{\texttt{null} : t_i \rightsquigarrow t_i'} \qquad \frac{\begin{array}{c} v_j = x \\ \vdash_t t := t' \downarrow t_j \rightsquigarrow t_j' \qquad \vdash_A A := A', x : t \downarrow \overline{v_i : t_i \rightsquigarrow t_i'}^{i \neq j}\end{array}}{\vdash_A A := A', x : t' \downarrow \overline{v_i : t_i \rightsquigarrow t_i'}}$$

$$\frac{\boxed{va''' \leq va} \qquad (\forall i)\ \vdash_\varsigma \varsigma_i''' \leq \varsigma_i \qquad \vdash_s s_i := s_i' \downarrow s_i'' \rightsquigarrow s_i'''}{\vdash_t \langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle := \langle va', c\{\overline{f_i : s_i'[\varsigma_i']}\}\rangle \downarrow \langle va'', c\{\overline{f_i : s_i''[\varsigma_i'']}\}\rangle \rightsquigarrow \langle va''', c\{\overline{f_i : s_i'''[\varsigma_i''']}\}\rangle}$$

$$\frac{\vdash_{aa} aa := aa' \downarrow aa'' \rightsquigarrow aa''' \qquad \vdash_t t := t' \downarrow t'' \rightsquigarrow t'''}{\vdash_s \langle aa, t\rangle := \langle aa', t'\rangle \downarrow \langle aa'', t''\rangle \rightsquigarrow \langle aa''', t'''\rangle}$$

$$\vdash_{aa} aa := aa' \downarrow \Delta(va) \rightsquigarrow aa \qquad \frac{aa' \in \{\Diamond, \nabla\}}{\vdash_{aa} aa := aa \downarrow aa' \rightsquigarrow aa''}$$

Fig. 3: Relations on types, annotations, and environments. (The rules for types have a coinductive interpretation.)

`let` expression. Its purpose is to transfer the type state changes from one alias that goes out of scope to another. For example, the expression $\texttt{let}\,x = y\,\texttt{in}\,e$ introduces a new alias x for y. Inside e, the same object may be updated through both x and y which also changes their types. When leaving the scope of x, the type changes to x are lost with the standard `let` rule, but the effect application in the restricted rule merges the final type of x back into y's type.

Technically, the relation defines how type changes $\overline{t_i \rightsquigarrow t'_i}$ for values $\overline{v_i}$ affect an environment A'. If v_j is `null` then nothing happens. If v_j is a variable x, then the new type for x is t defined as $\vdash_t t := A'(x) \downarrow t_j \rightsquigarrow t'_j$. The effect application relation on types, $\vdash_t t := t' \downarrow t'' \rightsquigarrow t'''$, changes at most the annotations of t' for which the corresponding annotation of t'' is active but leaves the other annotations of t' intact.

Expressions. The judgment for expressions, $P; A \vdash_e e : t \rhd A'$, assigns a type t and an updated environment A' (after e's evaluation) to expression e in the context of program P and environment A (see Figures 4 and 5).

– In the variable rule, each use of a variable splits of the properties needed and passes the remaining properties on to subsequent uses.
– The rule for `null` relies on an auxiliary judgment $P \vdash_{\texttt{null}} t$ which ensures that t is well-formed and carries a suitable annotation.
– The rule for `new` determines an annotation for the newly created object with $R^{\texttt{new}}$. The judgment $P; A \vdash_{\overline{e}} \overline{v} : \overline{t} \rhd A'$ types the constructor arguments.
– The rule for accessing field f performs the already mentioned lending of capabilities. The type of the the dereferenced object lends its capabilities at field f through the type access judgment $t_y = t'_y \mid_f t_x$ to the extracted value. After typing the body expression e with the resulting types it merges the final types back into the type of the reference.
  This rule has a number of related rules in Figure 5. They treat the case that the dereferenced object is `null` and the `join` expression that arises from reducing the field access. There is a special rule for a `join` expression where the extracted value is `null`.
– Field assignment $\texttt{set}\,x.f = v\,\texttt{in}\,e$ changes the type of field f in x's type using the type update judgment $P; u \vdash f \leftarrow t \rhd u'$ which states that field f of an object with type u can be assigned a value of type t while modifying the object's type to u'. Two rules define this judgment:
  • The first rule allows a strong update of f which may change its type. It requires the old type of f to be *entirely active* (judgment $\triangle \vdash_s s$). If there was a semi-active or inactive field, then the field might be updated through an alias thus invalidating the change in the type.
  • The second rule deals with “ordinary” updates. It requires that the old type of f is semi-active (judgment $\Diamond \vdash_s s$) because overwriting an inactive field would result in an invalid typing assumption about a reference carrying an active annotation for this field.
– The rule for method calls uses the effect application relation to propagate the type changes of the method signature to the resulting type environment.

Expression typing:

$$\frac{\vdash_t t \succeq t_1 \mid t_2}{P; A, x : t \vdash_e x : t_1 \rhd A, x : t_2} \qquad \frac{P \vdash_{\texttt{null}} t}{P; A \vdash_e \texttt{null} : t \rhd A}$$

$$\frac{P; A \vdash_{\overline{e}} \overline{v_i} : \overline{t_i} \rhd A' \qquad \boxed{R^{\texttt{new}}_{\ell,c}(va)} \qquad t = \langle va, c\{\overline{f_i : \langle aa_i, t_i\rangle[\varsigma_i]}\}\rangle \qquad P \vdash_t wf(t)}{P; A \vdash_e \texttt{new}^{\ell}\ c(\overline{v_i}) : t \rhd A'}$$

$$\frac{t_y = t'_y \mid_f t_x \qquad P; A, y : t'_y, x : t_x \vdash_e e : t \rhd A', y : t''_y, x : t'_x \qquad t'''_y = t''_y \mid_f t'_x}{P; A, y : t_y \vdash_e \texttt{let}\, x = y.f\, \texttt{in}\, e : t \rhd A', y : t'''_y}$$

$$\frac{P; A \vdash_e v : t \rhd A', x : \langle va, u\rangle \qquad u \vdash f \leftarrow t \rhd u' \qquad P; A', x : \langle va, u'\rangle \vdash_e e : t' \rhd A''}{P; A \vdash_e \texttt{set}\, x.f = v\, \texttt{in}\, e : t' \rhd A''}$$

$$\frac{P; A \vdash_{\overline{e}} \overline{v_i} : \overline{t_i} \rhd A' \qquad mtype_P(t_1, m) = \overline{t_i \rightsquigarrow t'_i} \to t \qquad \vdash_A A'' := A \downarrow \overline{v_i : t_i \rightsquigarrow t'_i}}{P; A \vdash_e v_1.m(\overline{v_i}^{i \in \{2,\ldots n\}}) : t \rhd A''}$$

$$\frac{P; A \vdash_e e_1 : t_1 \rhd A_1 \qquad P; A_1, x : t_1 \vdash_e e_2 : t_2 \rhd A_2, x : t'_1 \qquad \vdash_t \rho(t'_1)}{P; A \vdash_e \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : t_2 \rhd A_2}$$

$$\frac{P; A \vdash_e v : t_1 \rhd A_1 \qquad P; A_1, x : t_1 \vdash_e e : t \rhd A_2, x : t_2 \qquad \vdash_A A_3 := A_2 \downarrow v : t_1 \rightsquigarrow t_2}{P; A \vdash_e \texttt{let}\, x = v\, \texttt{in}\, e : t \rhd A_3}$$

$$\frac{\begin{array}{c} P; A \vdash_e e_1 : t \rhd A' \\ P; A' \vdash_e e_2 : t' \rhd A'' \qquad P; A' \vdash_e e_3 : t' \rhd A'' \end{array}}{P; A \vdash_e \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : t' \rhd A''} \qquad \frac{\begin{array}{c} P; A \vdash_e e : t_1 \rhd A_1 \\ \vdash_t t_1 \leq t_2 \qquad \vdash_A A_1 \leq A_2 \end{array}}{P; A \vdash_e e : t_2 \rhd A_2}$$

Type access:

$$\frac{\vdash_\varsigma \varsigma_j \leq \mathrm{R} \qquad s_j = \langle aa, t\rangle \qquad \vdash_t t \succeq t_1 \mid t_2 \qquad s'_j = \langle aa, t_1\rangle}{\langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle = \langle va, c\{f_j : s'_j[\varsigma_j]; \overline{f_i : s_i[\varsigma_i]}\}\rangle \mid_{f_j} t_2}$$

Type update:

$$\frac{\begin{array}{c} \triangle \vdash_s s_j \qquad \vdash_t \rho(\langle va_j, u_j\rangle) \\ \vdash_\varsigma \varsigma_j \leq \mathrm{W} \qquad s_j = \langle aa_j, va_j, u_j\rangle \qquad s'_j = \langle \triangle(va), va_j, u\rangle \qquad \vdash_s wf(s'_j) \end{array}}{c\{\overline{f_i : s_i[\varsigma_i]}\} \vdash f_j \leftarrow \langle va, u\rangle \rhd c\{f_j : s'_j[\varsigma_j]; \overline{f_i : s_i[\varsigma_i]}^{i \neq j}\}}$$

$$\frac{\Diamond \vdash_s s_j \qquad s_j = \langle aa, t'\rangle \qquad \vdash_t t \leq t'}{c\{\overline{f_i : s_i[\varsigma_i]}\} \vdash f_j \leftarrow t \rhd c\{\overline{f_i : s_i[\varsigma_i]}\}}$$

Fully active types:

$$\frac{(\forall i)\ \triangle \vdash_s s_i}{\triangle \vdash_s \langle \triangle(va), va', c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle} \qquad \frac{(\forall i)\ \Diamond \vdash_s s_i}{\Diamond \vdash_s \langle \Diamond, va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle}$$

Auxiliaries:

$$\frac{t = \langle va, c\{\ldots\}\rangle \qquad \boxed{R^{\texttt{null}}_c(va)} \qquad P \vdash_t wf(t)}{P \vdash_{\texttt{null}} t} \qquad \frac{(\forall i)\ P; A_{i-1} \vdash_e v_i : t_i \rhd A_i}{P; A_0 \vdash_{\overline{e}} \overline{v_i} : \overline{t_i} \rhd A_n}$$

Fig. 4: Typing rules for expressions.

$$\frac{P; A, x : t_x \vdash_e e : t \rhd A', x : t'_x \qquad P \vdash_t \mathit{wf}(t_x)}{P; A \vdash_e \texttt{let}\, x = \texttt{null}.f\ \texttt{in}\, e : t \rhd A'}$$

$$\frac{t_y = t'_y \mid_f t_x \qquad t'''_y = t''_y \mid_f t'_x \qquad P; A, y : t'_y, l : t_x \vdash_e e : t \rhd A', y : t''_y, l : t'_x}{P; A, y : t_y \vdash_e \texttt{join}\, l = y.f\ \texttt{from}\, e : t \rhd A', y : t'''_y}$$

$$\frac{t_y = t'_y \mid_f t_x \qquad t'''_y = t''_y \mid_f t'_x \qquad P; A, y : t'_y \vdash_e e : t \rhd A', y : t''_y}{P; A, y : t_y \vdash_e \texttt{join}\, \texttt{null} = y.f\ \texttt{from}\, e : t \rhd A', y : t'''_y}$$

$$\frac{P; A \vdash_e v : t' \rhd A' \qquad P; A' \vdash_e e : t \rhd A''}{P; A \vdash_e \texttt{set}\, \texttt{null}.f = v\ \texttt{in}\, e : t \rhd A''}$$

Fig. 5: Typing rules for intermediate expressions.

$$\frac{P = \overline{\mathit{defn}_i}\ e \qquad (\forall i)\ P \vdash \mathit{defn}_i \qquad \text{class names in } \overline{\mathit{defn}_i} \text{ disjoint} \qquad P, \emptyset \vdash_e e : t \rhd \emptyset}{\vdash P}$$

$$\frac{\begin{array}{c}(\forall i)\ \texttt{class}\, c_i\, \{\overline{c'\, f'};\ \overline{\mathit{meth}'}\} \in P \qquad (\forall j)\ P; c \vdash \mathit{meth}_j \\ \text{field names } \overline{f_i} \text{ disjoint} \qquad \text{method names in } \overline{\mathit{meth}_j} \text{ disjoint}\end{array}}{P \vdash \texttt{class}\, c\, \{\overline{c_i\, f_i};\ \overline{\mathit{meth}_j}\}}$$

$$\frac{\begin{array}{c} t_0 = \langle va, c\{\overline{f : s[\varsigma]}\}\rangle \\ (\forall i \in \{0, \dots, n\})\ P \vdash_t \mathit{wf}(t_i) \qquad P; \mathit{this} : t_0, \overline{x_i : t_i} \vdash_e e : t \rhd \mathit{this} : t'_0, \overline{x_i : t'_i}\end{array}}{P; c \vdash t\, [t_0 \rightsquigarrow t'_0]\, m(\overline{t_i \rightsquigarrow t'_i\ x_i})\, \{\, e\, \}}$$

Fig. 6: Typing rules for programs.

– There are two rules for `let` expressions. The standard one ensures that the type of the let-bound variable is droppable after evaluating the body of the let expression. The restricted one requires a value in its header, so that a restricted let creates an alias of a variable. In this case, the rule implements lending of capabilities just like described for the field access rule. It uses effect application to merge the changes of the alias back into the type of the original reference.

4 Soundness

We prove type soundness using the standard syntactic technique [29]. To apply it, we first have to extend typing to configurations $\langle e; \mathcal{S} \rangle$. To this end, we have to introduce an Urtype assumption $\mathcal{A}$. This assumption assigns to each location/object its activity annotated type before it is used in the program. Every activity annotated type of a use of a particular location in the program must be split off the Urtype for the location. The Urtype assumption changes during evaluation to reflect changes of the field values of an object with active fields. The most important point about the Urtype assumption is that it guarantees consistent use and distribution of the activity annotations throughout the uses of the locations in the program.

We start by introducing a function $R_{L,\mathcal{S}} \in \mathcal{P}(\mathit{Loc}) \to \mathcal{P}(\mathit{Loc})$ so that its smallest fixed point $\mu R_{L,\mathcal{S}}$ is the set of locations reachable from $L \subseteq \mathit{Loc}$ through $\mathcal{S}$. The predicate $\mathit{drop\text{-}ok}(e, A, \mathcal{S})$ indicates whether all locations with a non-droppable type are reachable from e.

$$\begin{gathered} R_{L,\mathcal{S}}(M) = (L \cup \bigcup\{\mathit{ran}(\mathcal{F}) \mid l' \in M, \mathcal{S}(l') = \langle c, \ell, \mathcal{F} \rangle\}) \cap \mathit{dom}(\mathcal{S}) \\ \mathit{drop\text{-}ok}(e, A, \mathcal{S}) = \forall l \in \mathit{dom}(\mathcal{S}).\ \vdash_t \rho(A(l)) \vee l \in \mu R_{\mathit{fv}(e),\mathcal{S}} \end{gathered}$$

Let Path range over finite access paths $\overline{f}$ of field names. The notation $f \oplus \overline{f'}$ attaches f to the front of path $\overline{f'}$. The predicate

$$\mathit{aliases\text{-}ok}(l, \mathcal{A}, A, \mathcal{S}) \Leftrightarrow \mathcal{A}(l), \mathcal{S}(l) \precsim \{\!\{ A(l_i).\overline{f_i} \mid \mathcal{S}(l_i).\overline{f_i} \}\!\}$$

relates all type assumptions about a single location l with an Urtype assumption $\mathcal{A}$. Every active annotation in the typing must be sanctioned by an active annotation in the Urtype assumption. The Urtype assumption for a location is responsible (1) for the local activity annotation of fields that refer to defined locations and (2) for the full type of fields that contain `null`. The definition collects relevant types in a multiset (indicated by $\{\!\{ \dots \}\!\}$) because each occurrence of a type contributes to the activity. Thus, the *aliases-ok* predicate ensures that there is at most one active annotation in all type assumptions about l.

Some auxiliary notation is needed to define the action of access paths on types and stores:

$$t.\varepsilon = t \qquad \frac{t = \langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle \qquad s_j = \langle aa, t_j\rangle}{t.(f_j \oplus \overline{f_{j_i}}) = t_j.\overline{f_{j_i}}}$$

$$\mathcal{S}(w).\varepsilon = w \qquad \frac{\mathcal{S}(l) = \langle c, \ell, \overline{f_i \mapsto w_i}\rangle}{\mathcal{S}(l).(f_j \oplus \overline{f_{j_i}}) = \mathcal{S}(l_j).\overline{f_{j_i}}}$$

It remains to define the "sanctions" relation between an entry in an Urtype assumption (an annotated type), an entry in a store, and a multiset of annotated types. Its first stage projects out, for each field, the corresponding field type, the stored value, and the multiset of field types.

$$\frac{(\forall i)\ s_i, w_i \precsim \{\!\{ s_i^\iota \mid \iota \in J \}\!\}}{\langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle, \langle c, \ell, \overline{f_i \mapsto w_i}\rangle \precsim \{\!\{ \langle va^\iota, c\{\overline{f_i : s_i^\iota[\varsigma_i^\iota]}\}\rangle \mid \iota \in J \}\!\}}$$

Its second stage states that the annotation from the Urtype assumption splits into the multiset of the activity annotations. For each `null` value, the multiset of types is also split from the type in the Urtype assumption.

$$\frac{aa \succeq \{\!\{ aa^\iota \mid \iota \in J \}\!\} \qquad w = \mathtt{null} \Rightarrow t \succeq \{\!\{ t^\iota \mid \iota \in J \}\!\}}{\langle aa, t\rangle, w \precsim \{\!\{ \langle aa^\iota, t^\iota\rangle \mid \iota \in J \}\!\}}$$

The typing judgment $P; \mathcal{A}; A \vdash_c \langle e; L; \mathcal{S}\rangle : t \rhd A'$ for configurations $\langle e; \mathcal{S}\rangle$ in context L (a multiset of locations) formalizes the main invariant of the preservation lemma. It holds if the store is consistently typed, the expression is well typed, the program is well-formed, the locations occurring in the expression are all defined in the store, every location which is not reachable from L and the locations in the expression must have a droppable type, the locations in L are all typed and use up enough capabilities of the final assumptions A' so that all types in the remaining assumption A'' are droppable.

$$\frac{\begin{array}{c} P; \mathcal{A} \vdash_{\mathcal{S}} \mathcal{S} : A \qquad P; A \vdash_e e : t \rhd A' \qquad \vdash P \qquad \mathit{fv}(e) \subseteq \mathit{dom}(S) \\ \mathit{drop\text{-}ok}(\mathit{fv}(e) + L, A, \mathcal{S}) \qquad L \subseteq \mathit{dom}(A') \qquad P; A' \vdash_{\overline{e}} L : \overline{t} \rhd A'' \qquad \rho(A'') \end{array}}{P; \mathcal{A}; A \vdash_c \langle e; L; \mathcal{S}\rangle : t \rhd A'}$$

$$\frac{\begin{array}{c} \mathit{dom}(\mathcal{S}) \subseteq \mathit{dom}(A) \\ \mathit{dom}(A) = \mathit{dom}(\mathcal{A}) \qquad (\forall l \in \mathit{dom}(\mathcal{S}))\ P; \mathcal{A}; A; \mathcal{S} \vdash\!\!\urcorner\ l : A(l) \end{array}}{P; \mathcal{A} \vdash_{\mathcal{S}} \mathcal{S} : A}$$

$$\frac{\begin{array}{c} \mathcal{S}(l) = \langle c, \ell, \mathcal{F}\rangle \qquad \boxed{R^{\mathtt{new}}_{\ell,c}(va)} \qquad \mathit{ran}(\mathcal{F}) \subseteq \mathit{dom}(\mathcal{S}) \cup \{\mathtt{null}\} \\ (\forall i)\ \mathcal{F}(f_i) = \mathtt{null} \Rightarrow P \vdash_{\mathtt{null}} s_i \qquad (\forall i)\ P \vdash_s \mathit{wf}(s_i) \qquad \mathit{aliases\text{-}ok}(l, \mathcal{A}, A, \mathcal{S}) \end{array}}{P; \mathcal{A}; A; \mathcal{S} \vdash\!\!\urcorner\ l : \langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle}$$

The judgment $\vdash_{\mathcal{S}}$, which states the consistency of the assumptions about the store, has a standard inductive reading despite the presence of cyclic structures in the store. All potentially cyclic references are broken by the explicit use of the type environment A.

The preservation lemma uses an extension relation $\precapprox$ between Urtype assumptions, which holds between successive configurations. It basically states that active capabilities cannot be created from nothing.

$$\frac{dom\mathcal{A}_1 \subseteq dom\mathcal{A}_2 \quad (\forall l \in dom\mathcal{A}_1)\ \mathcal{A}_1(l) \precapprox \mathcal{A}_2(l)}{\mathcal{A}_1 \precapprox \mathcal{A}_2}$$

$$\frac{(\forall i)\ s_i \precapprox s_i'}{\langle va, c\{\overline{f_i : s_i[\varsigma_i]}\}\rangle \precapprox \langle va', c\{\overline{f_i : s_i'[\varsigma_i]}\}\rangle} \qquad \frac{(aa = \triangle(va) \vee aa = aa') \quad t \precapprox t'}{\langle aa, t\rangle \precapprox \langle aa', t'\rangle}$$

The type preservation lemma states that reducing an expression does not change its type. The notation $A\restriction_M$ denotes the environment obtained by restricting A to the variables in M.

Lemma 1 (Preservation). *Suppose* $P; \mathcal{A}_1; A_1 \vdash_c \langle e_1; L; \mathcal{S}_1\rangle : t \rhd A_1'$ *and* $P \vdash \langle e_1; \mathcal{S}_1\rangle \hookrightarrow \langle e_2; \mathcal{S}_2\rangle$. *Then there exist* $\mathcal{A}_2$, A_2, *and* A_2' *with* $\mathcal{A}_1 \precapprox \mathcal{A}_2$, $dom(A_1') \subseteq dom(A_2')$, *and* $\vdash_A A_2'\restriction_{dom(A_1')} \leq A_1'$ *such that* $P; \mathcal{A}_2; A_2 \vdash_c \langle e_2; L; \mathcal{S}_2\rangle : t \rhd A_2'$.

Proof. By induction on the definition of $\hookrightarrow$.

The progress lemma ensures that a well-typed expression is not stuck.

Lemma 2 (Progress). *Suppose* $P; \mathcal{A}; A \vdash_c \langle e; L; \mathcal{S}\rangle : t \rhd A'$. *Then either* e *is a value, or there exists* $\langle e'; \mathcal{S}'\rangle$ *such that* $P \vdash \langle e; \mathcal{S}\rangle \hookrightarrow \langle e'; \mathcal{S}'\rangle$, *or* $P \vdash \langle e; \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\!:\ dereferenced\ null}, \mathcal{S}\rangle$.

Proof. By structural induction on e.

5 From MiniJava(X) to Java(X)

The formalization of MiniJava(X) covers the core expression language of Java 1.4 and imperative field update. This section discusses the extensions necessary for the full system Java(X) with inheritance, subtyping, and with constrained parametric polymorphism over annotations in the style of HM(X) [21].

We have refrained from formally specifying the extensions in this paper because they add technical complication and obscure the simplicity of the approach by cluttering the presentation.

5.1 Polymorphism

The extension to polymorphism essentially adds annotation variables to the type language and allows constrained abstraction over them. The splitting, droppability, and subtyping relations become constraints, which can be abstracted over. In fact, the addition of polymorphism to a monomorphic type-based program analysis is a schematic, but tedious effort. Our extension is modeled after the HM(X)

$$\frac{P; A \vdash_e e : \langle va, c'\{\overline{f : s[\varsigma]}\}\rangle \rhd A'}{P; A \vdash_e (c)e : \langle va, c\{\overline{f : s[\varsigma]}\}\rangle \rhd A'}$$

Fig. 7: Type cast rule for Java(X).

framework [21] which provides a parameterized extension of Hindley-Milner typing (including type inference) by suitable constraint theories and subtyping.

The resulting constrained polymorphism adds technical complication, but it greatly increases the expressiveness. As an example, we revisit the typing of the `detach()` method of the JDOM API. In Section 2.1, we had to decide on one particular usage pattern for `detach()`. Either the typing made the method return the active reference or it modified the active receiver object. With annotation polymorphism, the system can postpone the decision by abstracting over the annotations and making the required splitting into a constraint. Here is the resulting type abstracting over the activity annotation variables ψ' and ψ'':

$$\begin{array}{l} \forall\psi'\psi''.\ \triangle(\mathbf{N}) \succeq \psi' \mid \psi'' \Rightarrow \\ \qquad \texttt{Attribute}\{\texttt{p} : \langle\psi'', \texttt{Element}\rangle\} \\ \qquad\quad [\texttt{Attribute}\{\texttt{p} : \langle\triangle(\mathbf{ND}) \rightsquigarrow \psi', \texttt{Element}\rangle\}]\ \texttt{detach}() \end{array}$$

The splitting constraint $\triangle(\mathbf{N}) \succeq \psi' \mid \psi''$ fixes the relationship between ψ' and ψ''. The two type signatures for `detach()` suggested in Section 2.1 are the only instances of the above parameterized type.

5.2 Inheritance

Inheritance and interfaces can be treated with a minor—but important—extension as in RAJA [16]. In MiniJava(X), the type of an object includes only the descriptions of the fields belonging to the object's class. In Java(X), with subtyping and a cast operation, the type of an object includes descriptions of all fields of all classes and a cast changes the class type but leaves the field environment untouched. Figure 7 contains the rule for a cast; the subsumption rule changes so that it can also raise the class type (as well as the annotations as shown in Figure 4). Interface types can be treated in the same way. Their addition just affects Java's subtyping relation.

The expanded class type is required for type checking cast operations in a meaningful way. Suppose that class `A` is a subclass of class `B`:

```
class B {}
class A extends B {
  Object mystate;
  public A (Object state) {...}
}
```

and the following use of an `A` object:

```
B b = new A (init);
A a = (A) b;
```

Suppose the newly created `A` object has type $\mathtt{A}\{\mathtt{mystate} : \langle \triangle(\mathtt{init}), \ldots \rangle\}$. If each class type only had the fields of its own class, then the subsumption to `B` would strip away the information about the `mystate` field. This information would be lost forever and the subsequent upcast back to `A` would have to invent some information about `mystate`.

With our choice, a cast or subsumption never changes the field map but only changes the static class name associated with it. Thus, information is neither lost nor reinvented.

Another issue is method consistency. If a subclass overrides a method of a superclass, then the annotated type in the superclass must subsume the one in the subclass as in method specialization [19].

6 Related Work

There are two closely related lines of work in type systems: refinement and ownership. Refinement types add extra information to an existing type system to check additional properties at compile time. Ownership types enforce access restrictions by providing extra structure on reference types.

Freeman and Pfenning [15] have proposed refinement types as an extension to ML with union and intersection types. Their approach attaches a property lattice to each type as we do, but they do not distinguish linear and non-linear resources. Their ideas have been further refined in various directions. For example, indexed types can express invariants of data types [30]. Type state checking [26, 25] is a precursor of refinement typing using similar techniques but for a more restricted first-order imperative language.

Another direction is the development of a logical system to model properties on top of the type system, as in the work of Mandelbaum et al. [18]. They graft a fragment of intuitionistic linear logic on top of the ML type system adapted for use with the monadic metalanguage. While this approach is highly expressive, it requires a lot of program modifications. Our work encodes the logical properties in annotations and has a built-in mechanism (activity annotations) to transform standard properties to linear properties.

Type qualifications are similar to type annotations. A typical work on type qualifications is the paper by Foster et al. [14] which enables the flow-sensitive checking of atomic properties that refine standard types. They present an efficient inference algorithm for their system. The goal of their work is similar to ours, however, our work combines flow-sensitive and flow-insensitive aspects.

Semantic type qualifiers [7] share some concerns with our work. They allow the specification of a type qualifier together with a logical formula defining its meaning in terms of the program state. They automatically discharge the resulting proof obligation and thus obtain a correct system automatically. However, their properties only correspond to our value annotations and they do not support the notion of strong update.

A number of works solve specific problems with ad-hoc constructed type systems and may be viewed as specializations of one of the above frameworks, in particular exploiting flow-sensitivity. Examples are the work on atomicity and race detection [10, 11], the work on Vault [8], and many others.

JavaCOP [1] is a tool for implementing certain annotated type systems for Java. It provides a language for defining predicates on typed abstract syntax trees for Java. JavaCOP is integrated with a Java compiler that checks the defined predicates before generating code. While JavaCOP provides a flexible and convenient framework for implementing such systems, it is a purely syntactic tool: it neither provides any soundness guarantees nor does it have a notion of flow dependency which would be necessary to track linear resource use. JAVA(X) provides both. Both JavaCOP and JAVA(X) make the important distinction between analysis designers (who define predicate/design annotation posets) and programmer (who work in terms of annotations).

The Fugue system [9] implements typestate assertions and checking for C♯. It structures the state of an object in different frames corresponding to the nesting of subclasses. For each frame, the programmer can state formulae in first-order predicate logic. In comparison to JAVA(X), Fugue has a different approach of handling aliases, it introduces extra program constructs to expose typestate, it requires the programmer to writer formulae instead of predefined abstract values, and there is no soundness proof.

Another related system is the Hob system [17]. It basically allows the specification of pre- and postconditions using an abstract specification language based on sets. However, the underlying interpretation of this language is configurable to different logical systems and there is an aspect-oriented mechanism to simplify authoring of specifications. JAVA(X) manages abstraction the other way round. An analysis designer carves out domain-specific abstract values from predicates, thus hiding some complexity from the programmer.

The goal of an ownership type system is to improve modularity by partitioning the state of a system in a hierarchical manner. Such a system restricts inter-object accesses to those that are compatible with the hierarchy. Although JAVA(X) was not conceived with ownership in mind, it turns out that notions like unique and borrowed references are closely related to our notion of active and inactive references.

There is a lot of work on ownership types and related notions [2, 31, 20] but we focus only on the most closely related work by Boyland and others. In a series of articles culminating in 2005 [5], Boyland and others have established a notion of permissions which are attached to an object type along with an effect system to abstract the state dependencies of a method call. The permissions govern whether a reference is readable or writable. In earlier work, Boyland [3] has proposed splitting of permissions in fractions where only the full permission "1" allows full read/write access and proper fractions only allow read access. This kind of permission seems to be related to our notions of active, inactive, and semi-active. Effects are also present in our system, albeit in the form of explicit state transitions on the argument and receiver types of a method.

It is also instructive to compare the notions of active, semi-active, and inactive with similar notions in the realm of ownership type systems as categorized by Boyland and others [4]. Their categorization includes the permissions R (read), W (write), $\bar{R}$ (exclusive read), and $\bar{W}$ (exclusive write: no other alias may write)[6]. Active corresponds to $RW\bar{W}$ (read, write, and exclusive write permission), semi-active to RW, and inactive to R (transposed to a per-field setting; their original work categorizes variables).

In previous work [27], we have proposed an annotated type system for a Java subset without inheritance that provides improved types for the DOM interface. The previous work has inspired the example in Section 2.1 but it is limited in several respects: It is tied to one particular amalgamation of annotation and activity, it can only keep track of one affine state of a resource (either the resource is in this state or nothing is known about it; there is no notion of droppability), it does not support type state change, and it does not treat inheritance. The present work overcomes all these weaknesses.

Hofmann and Jost [16] have defined a type-based analysis to predict the consumption of heap space by Java methods. Their system RAJA is inspired by amortized complexity analysis. The underlying design ideas of their type system are similar to ours, however, the details are different and our work has been developed independently. For example, splitting works very differently and JAVA(X)'s annotations of arguments may change through method calls whereas RAJA's annotations are simply used up because they denote a potential passed to a method invocation through the parameters.

7 Conclusion

JAVA(X) extends the type system of Java 1.4 with an annotation framework for tracking value-based properties as well as affine and linear properties. A transient property of an object is always tied to the value of a particular field of the object. The system only requires the specification of posets for the properties of field values and adds the tracking of linear and affine uses of references in a generic way. Linear and affine uses of references improve the accuracy of the properties because they are subject to type state change.

Our first experiences with the type checker are encouraging. Future work includes extending the type checker to full JAVA(X) and implementing some form of type inference. We also would like to connect some notion of semantics to our purely syntactic annotations and to investigate further variations of the activity annotations.

Acknowledgments. We thank Martin Hofmann for extensive discussions and the anonymous reviewers for their helpful comments.

[6] Their remaining permissions O, I, and $\bar{I}$ are not important here.

References

1. C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. 21th ACM Conf. OOPSLA*, pages 57–74, Portland, OR, USA, 2006. ACM Press, New York.
2. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In G. Morrisett, editor, *Proc. 30th ACM Symp. POPL*, pages 213–223, New Orleans, LA, USA, Jan. 2003. ACM Press. ACM SIGPLAN Notices (38)1.
3. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Proc. Intl. Static Analysis Symposium, SAS'03*, number 2694 in LNCS, pages 55–72, San Diego, CA, USA, June 2003. Springer.
4. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 2–27, London, UK, 2001. Springer-Verlag.
5. J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In M. Abadi, editor, *Proc. 32nd ACM Symp. POPL*, pages 283–295, Long Beach, CA, USA, Jan. 2005. ACM Press.
6. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.
7. B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proc. 2005 ACM Conf. PLDI*, pages 85–95, New York, NY, USA, June 2005. ACM Press.
8. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 PLDI*, pages 59–69, Snowbird, UT, United States, June 2001. ACM Press, New York, USA.
9. R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *18th ECOOP*, volume 3086 of *LNCS*, pages 465–490, Oslo, Norway, June 2004. Springer.
10. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, Vancouver, British Columbia, Canada, June 2000. ACM Press. Volume 35(5) of SIGPLAN Notices.
11. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. 2003 PLDI*, pages 338–349, New York, NY, USA, May 2003. ACM Press.
12. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. 1993 PLDI*, pages 237–247, Albuquerque, New Mexico, June 1993.
13. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 241–269. Springer, 1999.
14. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. 2002 PLDI*, pages 1–12, Berlin, Germany, June 2002. ACM Press.
15. T. Freeman and F. Pfenning. Refinement types for ML. In *Proc. PLDI '91*, pages 268–277, Toronto, Canada, June 1991. ACM.
16. M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *Proc. 15th European Symp. Programming*, number 3924 in LNCS, Vienna, Austria, Mar. 2006. Springer.
17. P. Lam, V. Kuncak, and M. Rinard. Crosscutting techniques in program specification and analysis. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 169–180, New York, NY, USA, 2005. ACM Press.

18. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In O. Shivers, editor, *Proc. Intl. Conf. Functional Programming 2003*, pages 213–225, Uppsala, Sweden, Aug. 2003. ACM Press, New York.
19. J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. POPL*, pages 109–124, San Francisco, CA, Jan. 1990. ACM Press.
20. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP*, number 1445 in LNCS, pages 158–185, Brussels, Belgium, July 1998. Springer.
21. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
22. J. Palsberg. Type-based analysis and applications. In ACM, editor, *ACM SIGPLAN – SIGSOFT Workshop on Program Analysis for Software Tools and Engineering: PASTE'01*, pages 20–27, New York, NY, June 09 2001. ACM Press.
23. B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.
24. M. Schrefl and M. Stumptner. Behavior-consistent specialization of object life cycles. *ACM Trans. Software Engineering and Methodology*, 11(1):92–148, 2002.
25. R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Softw. Eng.*, 19(5):478–485, 1993.
26. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
27. P. Thiemann. A type safe DOM API. In G. Bierman and C. Koch, editors, *Tenth International Symposium on Database Programming Languages (DBPL'05)*, number 3774 in LNCS, Trondheim, Norway, Aug. 2005. Springer.
28. D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1. MIT Press, 2005.
29. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
30. H. Xi and F. Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Proc. 26th ACM Symp. POPL*, pages 214–227, San Antonio, Texas, USA, Jan. 1999. ACM Press.
31. T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for Featherweight Java. In *Proc. 18th ACM Conf. OOPSLA*, pages 135–148, Anaheim, CA, USA, 2003. ACM Press, New York.