# Location Types for Safe Distributed Object-Oriented Programming*

Yannick Welsch and Jan Schäfer

University of Kaiserslautern, Germany
{welsch,jschaefer}@cs.uni-kl.de

**Abstract.** In distributed object-oriented systems, objects belong to different *locations*. For example, in Java RMI, objects can be distributed over different JVM instances. Accessing a reference in RMI has crucial different semantics depending on whether the referred object is local or remote. Nevertheless, such references are not statically distinguished by the type system.

This paper presents *location types*, which statically distinguish *far* from *near* references. We present a formal type system for a minimal core language. In addition, we present a type inference system that gives optimal solutions. We implemented location types as a pluggable type system for the ABS language, an object-oriented language with a concurrency model based on *concurrent object groups*. An important contribution of this paper is the combination of the type system with the flexible inference system and a novel integration into an Eclipse-based IDE by presenting the inference results as overlays. This drastically reduces the annotation overhead while providing full static type information to the user. The IDE integration is a general approach of its own and can be applied to many other type system extensions.

## 1 Introduction

In distributed object-oriented systems, objects belong to different *locations*. A location in this paper is regarded to be an abstract concept, but in practice it may, for example, refer to a physical computation node, some process (like a JVM instance in RMI [19]), or can even be a concept of a programming language. For example, in object-languages with concurrency models based on communicating groups of objects such as E [18], AmbientTalk/2 [24], JCoBox [22], or ABS [15], the location of an object can be considered as the group it belongs to. In these scenarios it often makes a difference whether a reference points to an object at the current location, i.e., the location of the current executing object (in the following called a *near* reference), or to an object at a different location (a *far* reference). For example, in the E programming language [18], a *far* reference can only be used for *eventual sends*, but not for *immediate* method calls. In Java RMI accessing a remote reference may throw a RemoteException, where

* This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models.

J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705, pp. 194–210, 2011.

accessing a normal reference cannot throw such an exception. It is thus desirable to be able to statically distinguish these two kinds of references. This is useful for documentation purposes, to reason about the code, and to statically prevent runtime errors.

We present *location types* which statically distinguish far from near references. Location types can be considered as a lightweight form of ownership types [4, 21] with the following two characteristics. The first is that location types only describe a *flat set* of locations instead of a *hierarchy* of ownership contexts. The second is that ownership types typically define the ownership context of an object in a precise way. Location types abstract from these precise locations by only stating whether an object belongs to the current location or some other location. These two simplifications make location types very lightweight and easy to use, while still being expressive enough to guarantee their desired properties. Location types are *not* used to enforce encapsulation, which is the main goal of many ownership type systems.

As with any type system extension, writing down the extended types can become tiresome for programmers. Furthermore, these annotations may clutter up the code and reduce readability, especially when several of such pluggable type systems [2, 9] are used together. This reduces the acceptance of pluggable type systems in practice. The first issue can be solved by automatically inferring the type annotations and inserting them into the code. But this results again in cluttered code with potentially many annotations. Our solution is to leverage the power of an IDE and present the inferred types to the programmer by using unobtrusive *overlays*. They give the programmer full static type information without cluttering the code with annotations nor reducing readability. The overlays can be turned on and off according to the programmer's need. Type annotations are only needed to make the type checking and inference modular, where the degree of modularity just depends on the interfaces where type annotations appear. This way of integrating type inference into the IDE drastically simplifies the usage of the proposed type system and is applicable to similar type system extensions.

*Contributions.* The three main contributions of this paper are the following. (1) We give the formalization of a type system for location types in a core object-oriented language. (2) We describe a type inference system that gives optimal solutions and helpful error messages. (3) We present an implementation of the type and inference system for the ABS language and show how to integrate such a system into an IDE by using a novel way of visualizing inferred type information.

*Outline.* The remainder of this paper is structured as follows. In Sect. 2 we give an informal introduction to location types and illustrate their usage by an example. Section 3 presents the formalization of location types for a core object-oriented language and the inference system. In Sect. 4 we explain how we implemented and integrated location types into an IDE, and provide a short evaluation. Section 5 discusses location types in the context of related work. Section 6 concludes.

## 2 Location Types at Work

Location types statically distinguish *far* from *near* references. To do so, standard types are extended with additional type annotations, namely *location types.* There are three different location types: Near, Far, and Somewhere. Location types are always interpreted *relatively* to the current object. A variable typed as Near means that it may only refer to objects that belong to the *same* location as the current object. Accordingly, a Far typed variable may only refer to objects that belong to a *different* location than the current object. Somewhere is the super-type of Far and Near and means that the referred object may either be Near or Far. Important to note is that only Near precisely describes a certain location. A Far annotation only states that the location of the referred object is *not* Near. This means that a Far typed variable may over time refer to different locations which are not further defined, except that they are not the location of the current object. What a location actually means is irrelevant to the type system. So whether the location of an object refers to a JVM instance or has some other form of object grouping does not matter. It is only important that an object belongs to a unique location for its entire lifetime.

We illustrate the location type system by applying it to a small implementation of a chat application. For the description we use the abstract behavioral specification language (ABS) [15], which we explain hand-in-hand with the example.

ABS is an object-oriented language with a Java-like syntax. It has a concurrency model that is based on so-called concurrent object groups (COGs). COGs can be regarded as the unit of concurrency and distribution in ABS. Every object in ABS belongs to exactly one unique COG for its entire lifetime. This is similar to the Java RMI setting where objects belong to certain JVM instances, which may run distributed on different machines. At creation time of an object it is specified whether the object is created in the current COG (using the standard **new** expression) or is created in a fresh COG (using the **new cog** expression). Communication in ABS between different COGs happen via *asynchronous method calls* which are indicated by an exclamation mark (!). A reference in ABS is *far* when it targets an object of a different COG, otherwise it is a *near* reference. Similar to the E programming language [18], ABS has the restriction that synchronous method calls (indicated by the standard dot notation) are only allowed on near references. Using a far reference for a synchronous method call results in a runtime exception. Our location type system can be used to statically guarantee the absence of these runtime exceptions.

The chat application is a simple IRC-like application, which consists of a single server and multiple clients. For simplicity, there is only a single chat room, so all clients actually broadcast their messages to all other clients. The basic interfaces of the chat application in the ABS language are given in Fig. 1. Note that only

```
interface Server {
   [Near] Session connect(
      [Far] Client c, String name); }
interface Session {
   Unit receive(ClientMsg m);
   Unit close(); }
interface Client {
   Unit connectTo([Far] Server s);
   Unit receive(ServerMsg m); }
```

**Fig. 1.** The annotated interfaces of the chat application

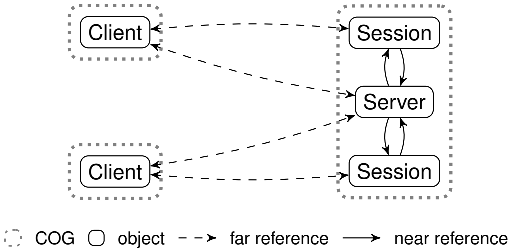


**Fig. 2.** Runtime structure of the chat application

```
1  class ClientImpl(String name) implements Client {
2     [Far] Session session; ...
3     Unit connectTo([Far] Server server) {
4        Fut<[Far] Session> f = server!connect(this, name);
5        session = f.get; } }
```

**Fig. 3.** Fully annotated implementation of the ClientImpl class

Server, Client, and Session are actually reference types, the types Unit, ClientMsg, and ServerMsg are *data types* and represent immutable data and not objects.

Figure 2 shows a possible runtime structure of the chat application. As the clients and the server run independently of each other, they live in their own COGs. This means that all references between clients and the server are far references. The Session objects that handle the different connections with the clients live in the same COG as the Server object. This means that references between Session and Server are near references. In a typical scenario, the client calls the connect method of the server and passes a reference to itself and a user name as arguments. The server then returns a reference to a Session object, which is used by the client to send messages to the server. The interfaces of Fig. 1 are annotated accordingly, e.g., the connect method of the server returns a reference to a Session object that is Near to the server.

Figure 3 shows the ClientImpl class, an implementation of the Client interface. It has a field session which stores a reference to the Session object which is obtained by the client when it connects to the server. Lines 3-5 show the connectTo method. As specified in the interface, the Server parameter has type Far. In Line 4, the client asynchronously (using the ! operator) calls the connect method of the server. The declared result type of the connect method is [Near] Session (see Fig. 1). The crucial fact is that the type system now has to apply a *viewpoint adaptation* [7]. As the target of the call (server) has location type Far, the return type of connect (which is Near) is adapted to Far. Furthermore, as the call is an asynchronous one, the value is not directly returned, but a future instead (i.e. a placeholder for the value). In Line 5, the client waits for the future to be resolved.

```
1  class ServerImpl implements Server {
2    List<[Near] Session> sessions = Nil;
3    [Near] Session connect(
4      [Far] Client c, String name) {
5      [Near] Session s =
6         new SessionImpl(this, c, name);
7      sessions = Cons(s,sessions);
8      this.publish(Connected(name));
9      return s; }
10   Unit publish(ServerMsg m) {
11     List<[Near] Session> sess =
12        sessions;
13     while (~isEmpty(sess)) {
14       [Near] Session s = head(sess);
15       sess = tail(sess);
16       s.send(m);
17   } } ...
18 }
```

**Fig. 4.** Fully annotated implementation of the ServerImpl class

Figure 4 shows the ServerImpl class, an implementation of the Server interface. It has an internal field sessions to hold the sessions of the connected clients. List is a polymorphic data type in ABS whose type parameter is instantiated with [Near] Session, which means that it holds a list of near references to Session objects. When a client connects to the server using the connect method, the server creates a new SessionImpl object in its current COG (using the standard **new** expression), which means that it is statically clear that this object is Near. It then stores the reference in its internal list, publishes that a new client has connected, and returns a reference to the session object. In the publish method at Line 16, the send method is synchronously called. As ABS requires that synchronous calls are only done on near objects, the type system guarantees that s always refers to a near object.

## 3 Formalization

This section presents the formalization of the location type system in a core calculus called LocJ. We first present the abstract syntax of the language and its dynamic semantics. In Sect. 3.1 we introduce the basic type system for location types as-well-as its soundness properties. In Sect. 3.2 we improve the precision of the basic type system by introducing named Far types. In Sect. 3.3 we present the location type inference system.

*Notations.* We use the overbar notation $\overline{x}$ to denote a list. The empty list is denoted by $\bullet$ and the concatenation of list $\overline{x}$ and $\overline{y}$ is denoted by $\overline{x} \cdot \overline{y}$. Single elements are implicitly treated as lists when needed. $\mathcal{M}[x \mapsto y]$ yields the map $\mathcal{M}$ where the entry with key $x$ is updated with the value $y$, or, if no such key exists, the entry is added. The empty map is denoted by $[]$ and $\mathsf{dom}(\mathcal{M})$ and $\mathsf{rng}(\mathcal{M})$ denote the domain and range of the map $\mathcal{M}$.

*Abstract Syntax.* LocJ models a core sequential object-oriented Java-like language, formalized in a similar fashion to Welterweight Java [20]. The abstract syntax is shown in Fig. 5. The main difference is that objects in LocJ can be

$$\begin{array}{rcl@{\qquad}rcl}
P & ::= & \overline{C} & E & ::= & \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} \\
C & ::= & \mathsf{class}\ c\ \{\ \overline{V}\ \overline{M}\ \} & & & \mid \mathsf{new}\ c\ \mathsf{in}\ x \mid x \\
V & ::= & T\ x & & & \mid x.m(\overline{y}) \mid x.f \\
M & ::= & T\ m(\overline{V})\ \{\ \overline{V}\ \overline{S}\ \} & T & ::= & c \\
S & ::= & x \leftarrow E \mid x.f \leftarrow y & & &
\end{array}$$

**Fig. 5.** Abstract syntax of LocJ. $c$ ranges over class names, $m$ over method names and $x, y, z, f$ over field and variable names (including this and result)

$$\begin{array}{rcll}
\zeta & ::= & \overline{\mathcal{F}}, \mathcal{H} & \text{runtime config.} \\
\mathcal{H} & ::= & \iota \mapsto (l, c, \mathcal{D}) & \text{heap} \\
\mathcal{F} & ::= & (\overline{S}, \mathcal{D})^{c,m} & \text{stack frame} \\
\mathcal{D} & ::= & x \mapsto v & \text{variable-value map} \\
v & ::= & \iota \mid \mathsf{null} & \text{value}
\end{array}$$

**Fig. 6.** Runtime entities of LocJ. $\iota$ ranges over object identifiers and $l$ over locations

created at different *locations*. For this, the new-expression has an additional argument, given by the in part, that specifies the target location. The target can either be fresh to create the object in a new (fresh) location, or a variable $x$ to create the object in the same location as the object that is referenced by $x$[1]. We do not introduce locations as first class citizens as they can be encoded using objects, i.e., objects can be simply used to denote locations. To keep the presentation short, LocJ does not include inheritance and subtyping. However, the formalization can be straightforwardly extended to support these features.

*Dynamic Semantics.* The dynamic semantics of our language is defined as a small-step operational semantics. The main difference to standard object-oriented languages is that we explicitly model locations to partition the heap. The runtime entities are shown in Fig. 6. Runtime configurations $\zeta$ consist of a stack, which is a list of stack frames, and a heap. The heap maps object identifiers to object states $(l, c, \mathcal{D})$, consisting of a location $l$, a class name $c$, and a mapping from field names to values $\mathcal{D}$. A stack frame consists of a list of statements and a mapping from local variable names to values. Furthermore the stack frame records with which class $c$ and method $m$ it is associated, which we sometimes omit for brevity.

The reduction rules are shown in Fig. 7. They are of the form $\zeta \rightsquigarrow \zeta'$ and reduce runtime configurations. The rules use the helper functions initO and initF to initialize objects and stack frames. The function $\mathsf{initO}(l, c)$ creates a new heap entry $(l, c, \mathcal{D})$ where $\mathcal{D} = [][\overline{f} \mapsto \overline{\mathsf{null}}]$ and $\overline{f}$ are the field names of class $c$. Similarly, $\mathsf{initF}(m, c, \iota, \overline{v})$ creates a new stack frame $(\overline{S}, \mathcal{D})^{c,m}$ where $\overline{S}$ are the statements in the method body of method $m$ in class $c$ and $\mathcal{D} = [][\mathsf{this} \mapsto \iota][\mathsf{result} \mapsto \mathsf{null}][\overline{x} \mapsto \overline{v}][\overline{y} \mapsto \overline{\mathsf{null}}]$ and $\overline{x}$ are the variable names of the formal parameters of method $m$ in class $c$ and $\overline{y}$ are the local variable names.

### 3.1 Basic Location Type System

In this subsection, we present the basic location type system and its soundness properties. To incorporate location types into LocJ programs, we extend types

[1] In ABS, **new cog** C() creates a new location (i.e., corresponds to "new $c$ in fresh" in LocJ) whereas **new** C() creates a new object in the same location as the current object (i.e., corresponds to "new $c$ in this" in LocJ).

$$\frac{\begin{array}{c}\iota \notin \mathsf{dom}(\mathcal{H}) \qquad l \text{ is fresh}\\ \mathcal{H}' = \mathcal{H}[\iota \mapsto \mathsf{initO}(l,c)] \qquad \mathcal{D}' = \mathcal{D}[x \mapsto \iota]\end{array}}{(x \leftarrow \mathsf{new}\ c\ \mathsf{in\ fresh} \cdot \overline{S},\mathcal{D}) \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow (\overline{S},\mathcal{D}') \cdot \overline{\mathcal{F}},\mathcal{H}'}$$

$$\frac{\begin{array}{c}\iota \notin \mathsf{dom}(\mathcal{H}) \qquad (l,\_,\_) = \mathcal{H}(\mathcal{D}(y))\\ \mathcal{H}' = \mathcal{H}[\iota \mapsto \mathsf{initO}(l,c)] \qquad \mathcal{D}' = \mathcal{D}[x \mapsto \iota]\end{array}}{(x \leftarrow \mathsf{new}\ c\ \mathsf{in}\ y \cdot \overline{S},\mathcal{D}) \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow (\overline{S},\mathcal{D}') \cdot \overline{\mathcal{F}},\mathcal{H}'}$$

$$\frac{\begin{array}{c}\iota = \mathcal{D}(x) \qquad (l,c,\mathcal{D}') = \mathcal{H}(\iota)\\ \mathcal{D}'' = \mathcal{D}'[f \mapsto \mathcal{D}(y)] \qquad \mathcal{H}' = \mathcal{H}[\iota \mapsto (l,c,\mathcal{D}'')]\end{array}}{(x.f \leftarrow y \cdot \overline{S},\mathcal{D}) \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow (\overline{S},\mathcal{D}) \cdot \overline{\mathcal{F}},\mathcal{H}'}$$

$$\frac{(\_,\_,\mathcal{D}'') = \mathcal{H}(\mathcal{D}(y)) \qquad \mathcal{D}' = \mathcal{D}[x \mapsto \mathcal{D}''(f)]}{(x \leftarrow y.f \cdot \overline{S},\mathcal{D}) \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow (\overline{S},\mathcal{D}') \cdot \overline{\mathcal{F}},\mathcal{H}}$$

$$\frac{\mathcal{D}' = \mathcal{D}[x \mapsto \mathcal{D}(y)]}{(x \leftarrow y \cdot \overline{S},\mathcal{D}) \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow (\overline{S},\mathcal{D}') \cdot \overline{\mathcal{F}},\mathcal{H}}$$

$$\frac{\begin{array}{c}\mathcal{F} = (x \leftarrow y.m(\overline{z}) \cdot \overline{S},\mathcal{D})\\ (\_,c,\_) = \mathcal{H}(\mathcal{D}(y))\\ \mathcal{F}' = \mathsf{initF}(c,m,\mathcal{D}(y),\mathcal{D}(\overline{z}))\end{array}}{\mathcal{F} \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow \mathcal{F}' \cdot \mathcal{F} \cdot \overline{\mathcal{F}},\mathcal{H}}$$

$$\frac{\begin{array}{c}\mathcal{F} = (x \leftarrow y.m(\overline{z}) \cdot \overline{S},\mathcal{D}')\\ \mathcal{D}'' = \mathcal{D}'[x \mapsto \mathcal{D}(\mathsf{result})]\end{array}}{(\bullet,\mathcal{D}) \cdot \mathcal{F} \cdot \overline{\mathcal{F}},\mathcal{H} \rightsquigarrow (\overline{S},\mathcal{D}'') \cdot \overline{\mathcal{F}},\mathcal{H}}$$

**Fig. 7.** Operational semantics of LocJ

$$\begin{array}{lll} T & ::= \cdots \mid L\ c & \text{annotated type}\\ L & ::= \mathsf{Near} \mid \mathsf{Far} \mid \mathsf{Somewhere} & \text{location type}\end{array}$$

**Fig. 8.** Basic location types

$T$ with location types $L$ (see Fig. 8), where a location type can either be Near, Far, or Somewhere. We assume that a given program is already well-typed using a standard Java-like type system and we only provide the typing rules for typing the location type extension. The typing rules are shown in Fig. 9. Statements and expressions are typed under a type environment $\overline{V}$, which defines the types of local variables. The typing judgment for expressions is of the form $\overline{V} \vdash e : L$ to denote that expression $e$ has location type $L$. The helper functions $\mathsf{anno}(c, f)$ and $\mathsf{anno}(c, m, x)$ return the declared location type of field $f$ or variable $x$ of method $m$ in class $c$ and $\mathsf{params}(c, m)$ returns the formal parameter variables of method $m$ in class $c$.

The crucial parts of the type system are the subtyping ($L <: L'$) and the viewpoint adaptation ($L \triangleright_K L'$) relations which are shown in Fig. 10. The location types Near and Far are both subtypes of Somewhere but are unrelated otherwise. Viewpoint adaption is always applied when a type is used in a different context. There are two different directions ($K \in \{\mathsf{From}, \mathsf{To}\}$) to consider. (1) Adapting a type $L$ *from another viewpoint* $L'$ to the current viewpoint, written as $L \triangleright_{\mathsf{From}} L'$. (2) Adapting a type $L$ from the current viewpoint *to another viewpoint* $L'$, written as $L \triangleright_{\mathsf{To}} L'$.[2] In typing rule WF-FieldGet we adapt the type of the field *from* the viewpoint of $y$ to the current viewpoint, whereas in rule WF-FieldSet we adapt the type of $y$ from the current viewpoint to the viewpoint of $x$.

[2] Whereas in other ownership type systems (e.g. [7]), only one direction is considered, we chose to explicitly state the direction in order to achieve a simple and intuitive encoding.

As an example for the viewpoint adaptation, assume a method is called on a $\mathsf{Far}$ target and the argument is of type $\mathsf{Near}$. Then the adapted type is $\mathsf{Far}$, because the parameter is $\mathsf{Near}$ in relation to the caller, but from the perspective of the callee, it is actually $\mathsf{Far}$ in that case. Important is also the case where we pass a $\mathsf{Far}$ typed variable $x$ to a $\mathsf{Far}$ target. In that case we have to take $\mathsf{Somewhere}$ as the adapted type, because it is not statically clear whether the object referred to by $x$ is in a location that is different from the location of the target object.

$$\textsc{(wf-P)}\quad \frac{P = \overline{C} \qquad \vdash C_i}{\vdash P}
\qquad
\textsc{(wf-C)}\quad \frac{c \vdash M_i}{\vdash \mathsf{class}\ c\ \{\ \overline{V}\ \overline{M}\ \}}
\qquad
\textsc{(wf-NewFresh)}\quad \frac{}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in\ fresh} : \mathsf{Far}}$$

$$\textsc{(wf-M)}\quad \frac{\mathsf{Near}\ c\ \mathsf{this} \cdot T\ \mathsf{result} \cdot \overline{V} \cdot \overline{V'} \vdash S_i}{c \vdash T\ m(\overline{V})\ \{\ \overline{V'}\ \overline{S}\ \}}
\qquad
\textsc{(wf-NewSame)}\quad \frac{L\ \_\ x \in \overline{V}}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ x : L}
\qquad
\textsc{(wf-Var)}\quad \frac{L\ \_\ x \in \overline{V}}{\overline{V} \vdash x : L}$$

$$\textsc{(wf-Assign)}\quad \frac{\overline{V} \vdash E : L \qquad L'\ \_\ x \in \overline{V} \qquad L <: L'}{\overline{V} \vdash x \leftarrow E}
\qquad
\textsc{(wf-FieldGet)}\quad \frac{L\ c\ y \in \overline{V} \qquad L' = \mathsf{anno}(c, f)}{\overline{V} \vdash y.f : L' \rhd_{\mathsf{From}} L}$$

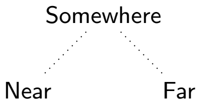
$$\textsc{(wf-FieldSet)}\quad \frac{\begin{array}{cc} L\ c\ x \in \overline{V} & L' = \mathsf{anno}(c, f) \\ L''\ \_\ y \in \overline{V} & (L'' \rhd_{\mathsf{To}} L) <: L' \end{array}}{\overline{V} \vdash x.f \leftarrow y}
\qquad
\textsc{(wf-Call)}\quad \frac{\begin{array}{c} L\ c\ y \in \overline{V} \\ L_i\ \_\ z_i \in \overline{V} \qquad \overline{x} = \mathsf{params}(c, m) \\ (L_i \rhd_{\mathsf{To}} L) <: \mathsf{anno}(c, m, x_i) \end{array}}{\overline{V} \vdash y.m(\overline{z}) : \mathsf{anno}(c, m, \mathsf{result}) \rhd_{\mathsf{From}} L}$$

**Fig. 9.** Typing rules of LocJ. Note that indices are implicitly all-quantified

*Type Soundness.* The location type system guarantees that variables of type $\mathsf{Near}$ only reference objects that are in the same location as the current object and that variables of type $\mathsf{Far}$ only reference objects that are in a different location to the current object. We formalize this by defining a well-formed runtime configuration. As helper functions, we define the location of a heap entry as $\mathsf{loc}((l, c, \mathcal{D})) = l$ and the dynamically computed location type as $\mathsf{dtype}(l, l') = \mathsf{Near}$ if $l = l'$, and $\mathsf{Far}$ otherwise.

**Definition 1 (Well-formed runtime configuration).** *Let $\zeta = \overline{\mathcal{F}}, \mathcal{H}$ be a runtime configuration. $\zeta$ is well-formed iff all heap entries $(l, c, \mathcal{D}) \in \mathsf{rng}(\mathcal{H})$ and all stack frames $\mathcal{F} \in \overline{\mathcal{F}}$ are well-formed under $\mathcal{H}$ and the configuration satisfies all the standard conditions of a class-based language.*

**Definition 2 (Well-formed heap entry).** *$(l, \_, \mathcal{D})$ is well-formed under $\mathcal{H}$ iff for all $f$ with $\mathcal{D}(f) = \iota$ and $(l', c, \_) = \mathcal{H}(\iota)$, we have $\mathsf{dtype}(l, l') <: \mathsf{anno}(c, f)$.*



| *Original* | $\triangleright_K$ | *Viewpoint* | = | *Adapted* |
|---|---|---|---|---|
| $L$ | $\triangleright_K$ | Near | = | $L$ |
| Near | $\triangleright_K$ | Far | = | Far |
| Far | $\triangleright_K$ | Far | = | Somewhere |
| Somewhere | $\triangleright_K$ | Far | = | Somewhere |
| $L$ | $\triangleright_K$ | Somewhere | = | Somewhere |

**Fig. 10.** Subtyping and viewpoint adaptation (where $K \in \{\mathsf{From}, \mathsf{To}\}$). Note that the direction $K$ does not influence basic location types, but is important for our extension in Sect. 3.2

```
[Far] Server server = new cog ServerImpl();
[Far] Client client1 = new cog ClientImpl("Alice");
[Far] Client client2 = new cog ClientImpl("Bob");
client1 ! connectTo(server);
client2 ! connectTo(server);
```

**Fig. 11.** The code of the main block of the chat application, annotated with location types

**Definition 3 (Well-formed stack frame).** $(\overline{S}, \mathcal{D})^{c,m}$ *is well-formed under* $\mathcal{H}$ *iff for all* $x$ *with* $\mathcal{D}(x) = \iota$*, we have* $\mathsf{dtype}(\mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this}))), \mathsf{loc}(\mathcal{H}(\iota))) <: \mathsf{anno}(c, m, x)$.

**Theorem 1 (Preservation for location types).** *Let* $\zeta$ *be a well-formed runtime configuration. If* $\zeta \rightsquigarrow \zeta'$*, then* $\zeta'$ *is well-formed as well.*

*Proof.* The proof proceeds by a standard case analysis on the reduction rule used and is available in the accompanying report [25].

### 3.2 Named Far Location Types

The location type system so far can only distinguish near from far references. The type system knows that a near reference always points to a different location than a far reference. But whether two far references point to the same location or different ones is not statically known. This makes the type system often too weak in practice. As an example, let us consider the *main block*[3] of the ABS chat application in Fig. 11, annotated with location types. The server and both clients are created by using the **new cog** expression. This means that all these objects live in their own, fresh COG and thus they can be typed to Far, because these locations are different to the current COG (the *Main* COG). However, for the method call client1!connectTo(server) to successfully type-check, the formal parameter of the connectTo method would need to be typed as Somewhere because the actual (adapted) parameter type is of type Somewhere ($= \mathsf{Far} \triangleright_{\mathsf{To}} \mathsf{Far}$). This issue arises because the type system cannot distinguish that client1 and server point to different locations. The example shows that in its basic form, the location type system often has to conservatively use the Somewhere type to

[3] A main block in ABS corresponds to a main method in Java.

remain sound, which in fact means that the type system cannot say anything about the location.

To improve the precision of the location type system we introduce *named* far types:

$$L ::= \cdots \mid \mathsf{Far}(i)$$

A named far type is a far type parametrized with an arbitrary name[4]. Far types with different names represent disjoint sets of far locations and are incompatible to each other. The following typing rule WF-NEWFRESHP is added, which allows new locations to be more precisely described.

$$\frac{(\textsc{wf-NewFreshP})}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in\ fresh} : \mathsf{Far}(i)}$$

The subtyping and viewpoint adaptation relations are extended accordingly in Fig. 12. Adapting a $\mathsf{Far}(i)$ to a $\mathsf{Far}(j)$ for $i \neq j$ yields a $\mathsf{Far}(i)$, as they denote different sets of locations. Adapting a $\mathsf{Far}(i)$ to a $\mathsf{Far}(i)$ does not yield Near, however, as two variables with the same $\mathsf{Far}(i)$ type can refer to objects of different locations.

In practice the user does not explicitly provide the names. Instead the inference system automatically infers them when possible. These refined far types are then used to improve the viewpoint adaptation. In the chat example our type system is now able to infer that the server and the client variables actually refer to different far locations. This means that the argument of the connectTo method call can be typed to Far instead of Somewhere.

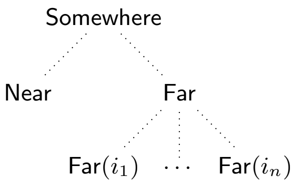
Our experience with case studies shows that this extension is expressive enough for our purposes (cf. Sect. 4). However, other extensions to improve the expressiveness and precision of the location type system are imaginable, e.g. location type polymorphism similar to owner polymorphism in ownership type systems [4, 3, 17].

*Type Soundness.* Similar as for Thm. 1, a proof of type soundness for the named far location type system extension is available in the accompanying report [25].

### 3.3 Location Type Inference

The type system presented in the previous section requires the programmer to annotate all type occurrences with location types. In this subsection we present an inference system for location types. We first present a sound and complete inference system, which makes it possible to use the location type system without writing any type annotations and only use type annotations for achieving modular type checking. The second part then presents an inference system that can deal with type-incorrect programs and that finds not only *some* solution but an *optimal* solution.

[4] Note that these are *not* object identifiers.



| *Original* | $\rhd_K$ | *Viewpoint* | = *Adapted* | |
|---|---|---|---|---|
| ... (extension of Fig. 10) | | | | |
| Near | $\rhd_{\mathsf{To}}$ | Far($i$) | = Far | |
| Near | $\rhd_{\mathsf{From}}$ | Far($i$) | = Far($i$) | |
| Far | $\rhd_K$ | Far($i$) | = Somewhere | |
| Somewhere | $\rhd_K$ | Far($i$) | = Somewhere | |
| Far($i$) | $\rhd_{\mathsf{To}}$ | Far($j$) | = Far($i$) | if $i \neq j$ |
| Far($i$) | $\rhd_{\mathsf{From}}$ | Far($j$) | = Somewhere | if $i \neq j$ |
| Far($i$) | $\rhd_K$ | Far($i$) | = Somewhere | |
| Far($i$) | $\rhd_K$ | Far | = Somewhere | |

**Fig. 12.** Subtyping and viewpoint adaptation for extended location types

$$\begin{array}{rll} \mathcal{Q} ::= & \alpha \rhd_K \beta = \gamma & \text{adaptation constraint} \\ | & \alpha <: \beta & \text{subtype constraint} \\ | & \alpha = L \mid \alpha \neq L & \text{constant constraint} \end{array}$$

**Fig. 13.** Location type constraints

**Sound and Complete Inference.** The formal model for inferring location types follows the formalization of other type system extensions [8]. The idea is to introduce location type variables at places in the program where location types occur in our typing rules. Type inference then consists of two steps. First, generating constraints for the location type variables. Second, checking whether a substitution for the location type variables exists such that all constraints are satisfied.

To introduce location type variables into programs we extend the syntax of location types accordingly:

$$L ::= \cdots \mid \alpha \qquad \text{location type variables (also } \beta, \gamma, \text{ and } \delta)$$

In the following we consider $P$ as a program which is fully annotated with pairwise distinct location type variables. The constraints which are generated by the inference system are shown in Fig. 13. We use the judgment $\vdash P : \overline{\mathcal{Q}}$, defined in Fig. 14, to denote the generation of the constraints $\overline{\mathcal{Q}}$ from program $P$. Note that additional *fresh* location type variables are introduced during the constraint generation.

*Soundness and Completeness.* Let $\sigma$ be a mapping function from location type variables to location types, i.e., $\alpha$ to $\{\mathsf{Near}, \mathsf{Far}, \mathsf{Somewhere}, \mathsf{Far}(i_1), ..., \mathsf{Far}(i_n)\}$. Then $\sigma \models \overline{\mathcal{Q}}$ if the constraints $\overline{\mathcal{Q}}$ are satisfiable under $\sigma$. We write $\sigma P$ to denote that all location type variables in $P$ have been replaced by location types according to the substitution function $\sigma$.

*Conjecture 1 (Soundness and Completeness of the Inference).* The inference is sound and complete in the sense that every typing inferred can be successfully type-checked and every typing which type-checks can also be inferred.

$$\frac{P = \overline{C} \qquad \vdash C_i : \overline{\mathcal{Q}_i}}{\vdash P : \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_n}}
\qquad
\frac{\delta\ c\ \mathsf{this} \cdot T\ \mathsf{result} \cdot \overline{V} \cdot \overline{V'} \vdash S_i : \overline{\mathcal{Q}_i} \qquad \delta \text{ is fresh}}{c \vdash T\ m(\overline{V})\ \{\ \overline{V'}\ \overline{S}\ \} : \delta = \mathsf{Near} \cdot \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_n}}$$

$$\frac{c \vdash M_i : \overline{\mathcal{Q}_i}}{\vdash \mathsf{class}\ c\ \{\ \overline{V}\ \overline{M}\ \} : \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_n}}
\qquad
\frac{\begin{array}{c}\alpha\ c\ x \in \overline{V} \\ \beta = \mathsf{anno}(c, f) \qquad \gamma\ \_\ y \in \overline{V} \qquad \delta \text{ is fresh}\end{array}}{\overline{V} \vdash x.f \leftarrow y : \delta <: \beta \cdot \gamma \rhd_{\mathsf{To}} \alpha = \delta}$$

$$\frac{\overline{V} \vdash E : \beta, \overline{\mathcal{Q}} \qquad \alpha\ \_\ x \in \overline{V}}{\overline{V} \vdash x \leftarrow E : \beta <: \alpha \cdot \overline{\mathcal{Q}}}
\qquad
\frac{\alpha\ c\ y \in \overline{V} \qquad \beta = \mathsf{anno}(c, f) \qquad \gamma \text{ is fresh}}{\overline{V} \vdash y.f : \gamma, \beta \rhd_{\mathsf{From}} \alpha = \gamma}$$

$$\frac{\delta \text{ is fresh}}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} : \delta, \delta \neq \mathsf{Near}}
\qquad
\frac{\begin{array}{c}\alpha\ c\ y \in \overline{V} \qquad \alpha_i\ \_\ z_i \in \overline{V} \qquad \overline{x} = \mathsf{params}(c, m) \\ \beta_i = \mathsf{anno}(c, m, x_i) \qquad \beta = \mathsf{anno}(c, m, \mathsf{result}) \\ \mathcal{Q}_i = \alpha_i \rhd_{\mathsf{To}} \alpha = \gamma_i \cdot \gamma_i <: \beta_i \\ \gamma_i \text{ is fresh} \qquad \gamma \text{ is fresh}\end{array}}{\overline{V} \vdash y.m(\overline{z}) : \gamma, \beta \rhd_{\mathsf{From}} \alpha = \gamma \cdot \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_n}}$$

$$\frac{\alpha\ \_\ y \in \overline{V}}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ y : \alpha, \bullet}
\qquad
\frac{\alpha\ \_\ x \in \overline{V}}{\overline{V} \vdash x : \alpha, \bullet}$$

**Fig. 14.** Constraint generation rules

- *Soundness*: If $\vdash P : \overline{\mathcal{Q}}$ and $\sigma \models \overline{\mathcal{Q}}$, then $\vdash \sigma P$.
- *Completeness*: If $\vdash \sigma P$ for some minimal $\sigma$, then $\exists \overline{\mathcal{Q}}$ such that $\vdash P : \overline{\mathcal{Q}}$ and $\exists \sigma'$ such that $\sigma'$ is an extension of $\sigma$ and $\sigma \models \overline{\mathcal{Q}}$. Note that $\sigma'$ is an extension of $\sigma$ iff $\sigma'(\alpha) = \sigma(\alpha)$ for all $\alpha \in \mathsf{dom}(\sigma)$.

**Optimal and Partial Inference.** Whereas soundness and completeness is important, it is not sufficient for an inference system to be usable in practice. Two additional properties are required, namely:

1. If multiple inference solutions exist, an *optimal* solution should be taken. This is important, because the user in general wants to have the most precise solution, i.e., with the least amount of Somewhere annotations.
2. If no typable solution can be inferred, at least a partially typable solution should be provided. It is otherwise nearly impossible to use the inference system if one only gets a *"No solution can be found"* result. In addition, this partially typable solution should lead to the least amount of type errors.

To support these two properties, we extend our formal model in the following way. We introduce three constraint categories: *must-have*, *should-have*, and *nice-to-have*. The *must-have* constraints must always be satisfied. These are for example in Fig. 14 the adaptation constraints ($\alpha \rhd_K \beta = \gamma$) and the constant constraints ($\alpha = L$, $\alpha \neq L$), characterizing the types of subexpressions. They also encompass the constant constraints which result from user annotations (not considered in the formalization of Fig. 14, but present in the implementation). Note that there is always a solution to these constraints in our inference system as they are based on freshly allocated location type variables. The *should-have* constraints, e.g. the subtype constraints ($\alpha <: \beta$) in Fig. 14, should always be satisfied in order to get a valid typing, but can be unsatisfied for partially correct solutions. The *nice-to-have* constraints are those that give us a *nice* (optimal) solution, i.e., with the least amount of Somewhere annotations or with Far types at the places where the precision of Far($i$) types is not needed.

Inferring an optimal solution consists of solving the following problem. First, all *must-have* constraints, then the most amount of *should-have* constraints, and finally the most amount of *nice-to-have* constraints should be satisfied. The problem can be encoded as a partially weighted MaxSAT problem by assigning appropriate weights to the constraints. This means that *must-have* constraints are hard clauses (maximum weight) and *should-have* constraints correspond to soft clauses whose weight is greater than the sum of all weighted *nice-to-have* clauses. Solving such a problem can be efficiently done using specialized SAT solvers.

As an example for partial inference, consider the ServerImpl class in Fig. 4. Assume that there are no annotations on the signature and the body of the connect method except for the return type which has been wrongly annotated by the programmer as Far. The inference system then still gives a solution where all constraints are satisfied except one *should-have* constraint, namely s <: result which is generated at the last line of the connect method. The inference system assigns the type Near to variable s because if it were to assign Far to s, more *should-have* constraints would be unsatisfied (i.e. those resulting from lines 5 to 7).

## 4 Implementation and IDE Integration

We have implemented the type and inference system for location types, including named far location types and optimal and partial inference, as an extension of the ABS compiler suite. The type and inference system is integrated into an Eclipse-based IDE, but can also be used from the command line.

*Inference System.* The inference system internally uses the Max-SAT solver SAT4J [16] to solve the generated inference constraints. As the inference system may return a solution that is not fully typable, we use the type checker for location types to give user-friendly error messages.

The alias analysis for named Far locations (cf. Sect. 3.2) can be configured to use scopes of different granularity: basic (no alias analysis), method-local, class-local, module-local, and global analysis. This allows the user to choose the best tradeoff between precision and modularity. For the inference, an upper bound on the number of possible named Far($i$) locations is needed. This is calculated based on the number of new $c$ in fresh expressions in the current scope.

*IDE Integration.* ABS features an Eclipse-based IDE[5] for developing ABS projects. The interesting part of the IDE for this paper is that we have incorporated visual overlays which display the location type inference results. For

[5] http://tools.hats-project.eu/eclipseplugin/installation.html

each location type there is a small overlay symbol, e.g., N for Near and F for Far, which are shown as superscripts of the type name. For example, a Far Client appears as Client$^{F}$. Whenever the user saves a changed program, the inference is triggered and the overlays are updated. They give the user complete location type information of all reference types, without cluttering the code. In addition, the overlays can easily be toggled on or off. It is also possible to write the inference results back as annotations into the source code, with user-specified levels of granularity, e.g., method signatures in interfaces.

*Evaluation.* We evaluated the location type system by applying it to three case studies. The Trading System (1164 LOC, 150 types to annotate) and Replication System (702 LOC, 62 types to annotate) case studies are ABS programs developed as parts of the case studies in the HATS project. The Chat Application (251 LOC, 55 types to annotate) is an extended version of the one presented in Sect. 2.

The evaluation results are presented in Fig. 15. They show how precise the case studies can be typed and how fast the inference works. We also restricted the alias analysis by various scopes to see the impact on performance and precision. First of all, all case studies can be fully typed using our type system. The chart on the left shows the precision (percentage of near and far annotations) of the type inference. As can be seen, the basic type system already has a good precision ($> 60\%$) in all three case studies. As expected, the precision increased with a broader analysis scope. Using a global aliasing analysis, the inference achieved a precision of 100% in the Chat as well as the Trading System case studies. In the Replication Server case study, the best precision was already achieved with a method-local scope.

The chart on the right shows the performance results of the inference. It shows that the performance of the inference is fast enough for the inference system to be used interactively. It also shows that the performance depends on the chosen scope for the aliasing analysis. Note that the examples where completely unannotated, so that all types had to be inferred. In practice, programs are often partially annotated, which additionally improves the performance of the type inference. Our implementation of the inference focused more on correctness than performance, which means that many improvements in the encoding, and thus in the solving time are still possible.

## 5 Discussion and Related Work

Location types are a variant of ownership types that concentrate on *flat* ownership contexts. We presented the type system in the context of distributed object-oriented systems, but it can be applied to any context where flat ownership contexts are sufficient. Ownership types [4, 3, 17] and similar type systems [1, 6] typically describe a hierarchical heap structure. On one hand this makes these systems more general than location types, because ownership types could be used for the same purpose as location types; on the other hand this makes

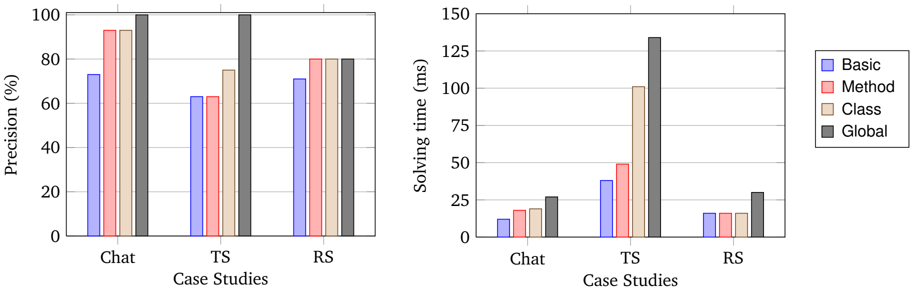


**Fig. 15.** Precision and solving time of the location type inference for the three case studies, using four different scopes for the aliasing analysis. The measurements where done on a MacBook Pro laptop (Intel Core 2 Duo T7400 2.16GHz CPU, 2GB RAM, Ubuntu 10.04, Sun JDK 1.6.16). We used the `-Xms1024` parameter to avoid garbage collection. As working in an IDE usually consists of an edit-compile-run cycle, we provide the performance results (the mean of 20 complete runs) after warming up the JVM with 5 dry runs. We measured the time that the SAT-solver required for finding a solution using the `System.nanoTime()` method.

these systems more complex. An ownership type system which is close to location types in nature is that by Clarke et al. [5], which applies ownership types to active objects. In their system ownership contexts are also flat, but ownership is used to ensure encapsulation of objects with support for a safe object transfer using unique references and cloning. Haller and Odersky [13] use a capability-based type system to restrict aliasing in concurrent programs and achieve full encapsulation. As these systems are based on encapsulation they do not have the concept of *far* references. Places [12] also partition the heap. However, the set of places is fixed at the time the program is started. Similar, but less expressive than our type system, is Loci [26], which only distinguishes references to be either thread-local or shared. Loci only uses defaults to reduce the annotation overhead. Loci is also realized as an Eclipse plug-in. Regions are also considered in region-based memory management [23], but for another purpose. They give the guarantee that objects inside a region do not refer to objects inside another region to ensure safe deallocation.

Using a Max-SAT solver with weighted constraints was also used in [11] to infer types that prevent data-races and in [8] to find good inference solutions for universe types. A crucial aspect of our work is the integration of type inference results into the IDE by using overlays. To the best of our knowledge there is no comparable approach. A widely used type system extension is the non-null type system [10]. For variations of this type system, there exist built-in inference mechanisms in Eclipse[6] and IntelliJ IDEA[7] as well as additional plug-ins such as [14]. None of these IDE integrations provide type information by using overlays,

[6] `http://wiki.eclipse.org/JDT_Core/Null_Analysis`

[7] `http://www.jetbrains.com/idea/webhelp/inferring-nullity.html`

but only give warnings in cases of type errors, which makes it difficult for the user to find the root of the problem.

## 6 Conclusion and Future Work

We have presented a type system for distributed object-oriented programming languages to distinguish near from far references. We applied the type system to the context of the ABS language to guarantee that far references are not used as targets for synchronous method calls. A complete type inference implementation allows the programmer to make use of the type system without making any annotations. The type inference results are visualized as overlay annotations directly in the development environment. Application of the type system to several case studies shows that the type system is expressive enough to type realistic code. The type inference implementation is fast enough to provide inference results within fractions of a second, so that interactive use of the system is possible.

We see two directions for future work. First, the type system could be applied to other settings where the location of an object is important, e.g., Java RMI [19]. Second, it would be interesting to investigate the visual overlay technique for other (pluggable) type systems, e.g., the nullness type system [14].

**Acknowledgements.** We thank our Master's students Thomas Fischer, Christian Seise, Florian Strauß, and Mathias Weber for implementing most parts of the ABS-Eclipse integration. We also thank Peter Wong for applying the location type system to the Replication System case study, and Arnd Poetzsch-Heffter and Patrick Michel for proof-reading earlier drafts of this paper. Finally, we thank the anonymous reviewers of TOOLS Europe 2011 for their constructive feedback.

## References

1. Aldrich, J.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
2. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. In: Tarr, P.L., Cook, W.R. (eds.) OOPSLA, pp. 57–74. ACM Press, New York (2006)
3. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, pp. 213–223. ACM Press, New York (2003)
4. Clarke, D., Potter, J., Noble, J.: Ownership Types for Flexible Alias Protection. In: OOPSLA, pp. 48–64. ACM Press, New York (1998)
5. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal Ownership for Active Objects. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 139–154. Springer, Heidelberg (2008)
6. Dietl, W.: Universe Types: Topology, Encapsulation, Genericity, and Tools. PhD thesis. ETH Zurich, Switzerland (2009)

7. Dietl, W., Gairing, M., Müller, P.: Generic Universe Types. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)
8. Dietl, W., Ernst, M., Müller, P.: Tunable Universe Type Inference. Tech. rep. 659. Department of Computer Science, ETH Zurich (December 2009)
9. Ernst, M.D.: Type Annotations Specification (JSR 308) and The Checker Framework: Custom pluggable types for Java, `http://types.cs.washington.edu/jsr308/`
10. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA 2003, pp. 302–312. ACM Press, New York (2003)
11. Flanagan, C., Freund, S.N.: Type inference against races. Sci. Comput. Program. 64, 140–165 (2007)
12. Grothoff, C.: Expressive Type Systems for Object-Oriented Languages. PhD thesis. University of California, Los Angeles (2006)
13. Haller, P., Odersky, M.: Capabilities for Uniqueness and Borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010)
14. Hubert, L., Jensen, T., Pichardie, D.: Semantic Foundations and Inference of Non-null Annotations. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 132–149. Springer, Heidelberg (2008)
15. Hähnle, R., et al.: Report on the Core ABS Language and Methodology: Part A. Report. The HATS Consortium (March 2010), `http://www.hats-project.eu/`
16. Le Berre, D., Parrain, A.: The SAT4J library, Release 2.2, System Description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
17. Lu, Y.: On Ownership and Accessibility. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 99–123. Springer, Heidelberg (2006)
18. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency Among Strangers. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 195–229. Springer, Heidelberg (2005)
19. Oracle Corporation: Java SE 6 RMI documentation , `http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html`
20. Östlund, J., Wrigstad, T.: Welterweight Java. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 97–116. Springer, Heidelberg (2010)
21. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic Ownership for Generic Java. In: Tarr, P.L., Cook, W.R. (eds.) OOPSLA, ACM Press, New York (2006)
22. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
23. Tofte, M., Talpin, J.-P.: Region-based Memory Management. Inf. Comput. 2, 109–176 (1997)
24. Van Cutsem, T., Mostinckx, S., Boix, E.G., Dedecker, J., Meuter, W.D.: AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In: SCCC, pp. 3–12. IEEE Computer Society Press, Los Alamitos (2007)
25. Welsch, Y., Schäfer, J.: Location Types for Safe Distributed Object-Oriented Programming. Tech. rep. 383/11. University of Kaiserslautern (April 2011)
26. Wrigstad, T., Pizlo, F., Meawad, F., Zhao, L., Vitek, J.: Loci: Simple Thread-Locality for Java. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 445–469. Springer, Heidelberg (2009)