Location Types for Safe Programming with Near and Far References*

Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter

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

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

This chapter presents *location types*, which statically distinguish *far* from *near* references. We present a formal type system for a minimal core language and develop a type inference system that gives maximally precise solutions satisfying further desirable properties. We prove soundness of the type system as well as soundness and correctness of the inference system. We have implemented location types as a pluggable type system for the ABS language, an object-oriented language with a concurrency and distribution model based on *concurrent object groups*. To facilitate programming with location types, we provide a tight integration of the type and inference system with an Eclipse-based integrated development environment (IDE) that presents inference results as overlays to the source code. The IDE drastically reduces the annotation overhead while providing full static type information to the programmer.

1 Introduction

In the puristic view of object-oriented programming, objects live in an *unstructured* space (or heap) and communicate via messages. This view is elegant and simple, but fails to address important software engineering principles like decomposition and encapsulation. That is why many researchers work on structuring techniques for object-oriented programming. Structuring the object space provides clear boundaries between different parts of the system. The resulting partitioning can be used to formulate and check important properties, e.g., that an object in one part does not reference an object in another part. Many ownership techniques [1,2,3] realize a hierarchical structuring of the objects into so-called ownership contexts and control access to a context from the surrounding context. The goal of our work is to support a partitioning of the object space as in distributed object-oriented programming where each object belongs to exactly one location. Thus, we can analyze whether two objects are at the same location or at different locations.

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

D. Clarke et al. (Eds.): Aliasing in Object-Oriented Programming, LNCS 7850, pp. 471–500, 2013.

A *location* as formalized in this chapter can take many different forms; it may refer to a physical computation node, some process, or it can be a concept supported by a programming language. This versatile notion of locality is not only useful for distributed programming, but also for programs running on a single computer. For example, in object-oriented languages with concurrency models based on communicating groups of objects such as E [4], AmbientTalk/2 [5], JCoBox [6], or ABS [7], 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 [4], a far reference can only be used for *asynchronous* method calls (named *eventual sends* in E), but not for *synchronous* method calls. In Java Remote Method Invocation (RMI) [8] accessing a remote reference may throw a RemoteException, where accessing a normal reference cannot throw such an exception. It is thus desirable to be able to statically distinguish these two kinds of references. In particular, this distinction 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 [2,3] with the following 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 support different roles of objects. Location types only classify objects as belonging to the current location or some other location. Furthermore, 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, type annotations may clutter the code and reduce its readability, especially when several of such pluggable type systems [9,10] 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 integrated development environment (IDE) and present the inferred types to the programmer by using visual *overlays*. The overlays give the programmer full static type information without cluttering the code with annotations nor reducing readability. Furthermore, the overlays can be turned on and off according to the programmer's need. Type annotations can still be used 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 simplifies the usage of the proposed type system and is applicable to similar type system extensions.

Notice. This chapter is a revised version of a paper that appeared at TOOLS 2011 [11]. Not having to worry about strict page limits, we present the material in more detail. In particular, we provide more depth to the setting and the examples, present additional related work, give a proof of the type soundness theorem as well as a

proof of the soundness and completeness of the inference system, and provide an updated version of the case studies.

Outline. The remainder of this chapter is structured as follows. In Section 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 Section 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

In this section, we illustrate the use of location types. After a short introduction to the type system, we explain the language setting based on concurrent object groups for which we developed our implementation and show the benefits of location types using an example.

Location types. 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 it means that the referred object may either be Near or Far. Note 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.[1] What a location actually means is irrelevant to the type system. So whether the location of an object represents a specific Java Virtual Machine (JVM) on which the object is running or some other form of object grouping does not matter. Note, however, that the type system relies on the assumption that the location of an object does not change over time.

Concurrent object groups. We use location types to distinguish near and far references in languages with a concurrency model based on groups of objects. Concurrent object groups (COGs) follow the actor paradigm [12] and were developed to avoid data races and the complexity of multithreading and to simplify reasoning about concurrent programs. The concurrency model of COGs is used in the abstract behavioral specification language ABS [7] and in JCoBox [6], a Java-based realization of COGs. Groups are created dynamically (cf. [13]) and form the units of concurrency and distribution. Execution within a single group is sequential but groups are running concurrently with other groups. Communication between groups is

[1] In Section 3.2, we present a refined type system that allows to distinguish far locations.

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

Fig. 1. The annotated interfaces of the chat application

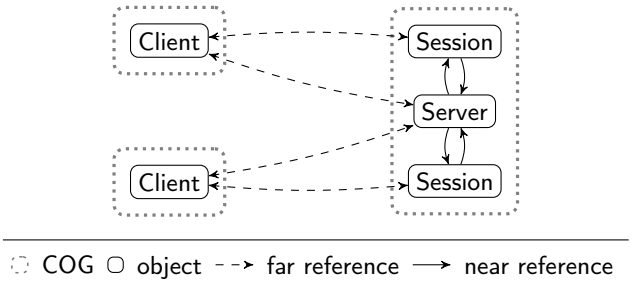


Fig. 2. Runtime structure of the chat application

asynchronous. In a program using COGs, each object belongs to a group for its entire lifetime. This is similar to the Java RMI setting where objects belong to certain JVMs, which may run distributed on different machines. In the following presentation, we use ABS as the language to illustrate location types along with a detailed example. Note, however, that location types are not restricted to ABS. We selected ABS to demonstrate how location types can be used beyond traditional distributed programming.

ABS is an object-oriented language with a Java-like syntax. In ABS, the creation of COGs is related to object creation. The creation expression specifies 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 [4], ABS has the restriction that *synchronous method calls* (indicated by the standard dot notation) are only allowed on near references. Using a synchronous method call on a far reference results in a runtime exception. Our location type system can be used to statically guarantee the absence of these runtime exceptions.

```
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;
6       }
7  }
```

Fig. 3. Fully annotated implementation of the ClientImpl class

Using location types. As an example, we model an IRC-like chat 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 Figure 1. Note that only Server, Client, and Session are reference types, the types Unit, ClientMsg, and ServerMsg are *data types* and represent immutable data.

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 Figure 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. On 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 Figure 1). The crucial fact is that the type system now has to apply a *viewpoint adaptation* [14]; As the target of the call (server) has location type Far from the viewpoint of the caller ClientImpl, the return type of connect (which is Near from the viewpoint of Server) is adapted to the viewpoint of ClientImpl, namely to Far. Furthermore, as the call is an asynchronous one, a *future* is returned, i.e., a placeholder for the value to be computed. The ABS type system uses the built-in polymorphic data type Fut to type futures. The type parameter of Fut is instantiated with the type of the value that it is a placeholder for. The variable f on line 4 is thus of type Fut<[Far] Session>. On line 5, the client waits for the future to be resolved and stores the value in the session field. The built-in **get** operator is used to retrieve the value of the future, blocking if necessary until the value is ready.

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.

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

Fig. 4. Fully annotated implementation of the ServerImpl class

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 (Connected(name) yields the corresponding message), and returns a reference to the session object. In the publish method at line 14, the send method is synchronously called. Here, the location type system guarantees that s always refers to a near object so that the synchronous call does not cause a runtime exception.

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 Section 3.1 we introduce the basic type system for location types as-well-as its soundness properties. In Section 3.2 we improve the precision of the basic type system by introducing named Far types. In Section 3.3 we present the location type inference system.

Notation. We use the overbar notation $\overline{x}$ to denote a list. The empty list is denoted by $\bullet$ and the concatenation of the list $\overline{x}$ and the list $\overline{y}$ is denoted by $\overline{x} \cdot \overline{y}$. Single elements are implicitly treated as lists when needed. The notation $\mathcal{M}[x \mapsto y]$ 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 $\mathcal{M}$.

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

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).

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

Fig. 6. Runtime entities of LocJ. ι ranges over object identifiers and l over locations.

Abstract syntax. LocJ models a core sequential object-oriented Java-like language, formalized in a similar fashion to Welterweight Java [15]. The abstract syntax is shown in Figure 5. The new aspect about LocJ is that objects in LocJ can be created at different *locations*. We do not introduce locations as first-class citizens as they can be encoded using objects. For this, the object creation expression new is augmented with 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. Note that 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). To keep the presentation short, LocJ does not include inheritance and subtyping. However, the formalization can be extended in the usual way to support these features (requiring parameter types of overriding methods to be contravariant and return types to be covariant).

Class-table. Throughout the formalization, we use a class table $\mathcal{CT}$ to look up definitions of classes, fields and methods. We assume that class names are globally unique and that field and method names are unique for each class. We then write $\mathcal{CT}(c)$ to denote the definition C of the class named c. We also write $\mathcal{CT}(c, f)$ to denote the definition V of the field named f in class c and we write $\mathcal{CT}(c, m)$ to denote the definition M of the method named m in class c.

Dynamic semantics. The dynamic semantics of our language is defined as a small-step operational semantics. The main difference with standard object-oriented languages is that we explicitly model locations to partition the heap. The runtime entities are shown in Figure 6. Runtime configurations ζ consist of a stack $\overline{\mathcal{F}}$, which is a list of stack frames, and a heap $\mathcal{H}$. A stack frame $\mathcal{F}$ consists of a list of statements $\overline{S}$ and a mapping $\mathcal{D}$ 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 heap maps object identifiers to object states $(l, c, \mathcal{D})$, consisting of a location l, a class name c, and a mapping $\mathcal{D}$ from field names to values.

The reduction rules are shown in Figure 7. They are of the form $\zeta \rightsquigarrow \zeta'$ and reduce runtime configurations. The rules use the helper functions initO and initF defined in Figure 8 to initialize objects and stack frames.

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

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

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

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

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

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

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

Fig. 7. Operational semantics of LocJ

3.1 Basic Location Types

In this subsection, we present the basic location type system and its soundness properties. To incorporate location types into LocJ programs, we extend types T with location types L (see Figure 9), 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 Figure 10. Judgments with indices i are implicitly all-quantified. For example, $c \vdash M_i$ means that for all $M_i \in \overline{M}$ the previous judgment holds. 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 $\mathsf{anno}(c, m, x)$, defined in Figure 8, 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 location subtyping ($L <: L'$) and the viewpoint adaptation ($L \triangleright_K L'$) relations which are shown in Figure 11. 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 the Universe type system [14] only one direction is considered, we chose to explicitly state the direction in order to achieve a simple and intuitive encoding.

$$
\begin{aligned}
\mathsf{initO}(l,c) &= (l,c,\mathcal{D}) \text{ if } \mathcal{D} = [][\overline{f} \mapsto \overline{\mathsf{null}}] \text{ and } \mathcal{CT}(c) = \mathsf{class}\ c\ \{\ \overline{T\ f}\ \overline{M}\ \}\\
\mathsf{initF}(m,c,\iota,\overline{v}) &= \begin{array}{l}(\overline{S},\mathcal{D})^{c,m} \text{ if } \mathcal{CT}(c,m) = T\ m(\overline{T\ x})\ \{\ \overline{T'\ y}\ \overline{S}\ \} \text{ and}\\ \quad \mathcal{D} = [][\mathsf{this} \mapsto \iota][\mathsf{result} \mapsto \mathsf{null}][\overline{x} \mapsto \overline{v}][\overline{y} \mapsto \overline{\mathsf{null}}]\end{array}\\
\mathsf{anno}(c,f) &= L \text{ if } \mathcal{CT}(c,f) = L\ c\ f\\
\mathsf{anno}(c,m,x) &= L \text{ if } \mathcal{CT}(c,m) = T\ m(\overline{V})\ \{\ \overline{V'}\ \overline{S}\ \} \text{ and } L\ c\ x \in \overline{V} \cdot \overline{V'}\\
\mathsf{params}(c,m) &= \overline{V} \text{ if } \mathcal{CT}(c,m) = T\ m(\overline{V})\ \{\ \overline{V'}\ \overline{S}\ \}\\
\mathsf{loc}((l,c,\mathcal{D})) &= l\\
\mathsf{dtype}(l,l') &= \begin{cases}\mathsf{Near} & \text{if } l = l'\\ \mathsf{Far} & \text{otherwise}\end{cases}\\
\mathsf{abs}(L) &= \begin{cases}\mathsf{Far} & \text{if } L = \mathsf{Far}(n) \text{ for some } n\\ L & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 8. Helper definitions

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

Fig. 9. Basic location types

As an example for the viewpoint adaptation, assume a method is called on a Far target and the argument is of type Near. Then the adapted type is Far, because the parameter is Near in relation to the caller, but from the perspective of the callee, it is actually Far in that case. Important is also the case where we pass a Far typed variable x to a Far target. In that case we have to take 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.

Type soundness. The location type system guarantees that variables of type Near only reference objects that are in the same location as the current object and that variables of type Far only reference objects that are in a different location to the current object. We formalize this under the notion of well-formed runtime configurations. We give a few helper functions in Figure 8. We define a function loc() to extract the location of a heap entry and the dynamic location type function dtype(l,l') that compares whether two locations l and l' are near or far to each other. A configuration is well-formed if heap and stack are well-formed; more precisely:

Definition 1 (Well-formed runtime configuration). *Let $\zeta = \overline{\mathcal{F}}, \mathcal{H}$ be a runtime configuration. ζ 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 (e.g., no dangling references, well-typed heap and stack, ...)*

A heap-entry is well-formed if its fields annotated by Near or Far only reference objects at the same or at a different location, respectively:

$$\frac{P = \overline{C} \qquad \vdash C_i}{\vdash P}\ (\text{WF-P}) \qquad \frac{c \vdash M_i}{\vdash \mathsf{class}\ c\ \{\ \overline{V}\ \overline{M}\ \}}\ (\text{WF-C}) \qquad \frac{\mathsf{Near}\ c\ \mathsf{this} \cdot T\ \mathsf{result} \cdot \overline{V} \cdot \overline{V'} \vdash S_i}{c \vdash T\ m(\overline{V})\ \{\ \overline{V'}\ \overline{S}\ \}}\ (\text{WF-M})$$

$$\frac{\overline{V} \vdash E : L \qquad L'\ \_\ x \in \overline{V} \qquad L <: L'}{\overline{V} \vdash x \leftarrow E}\ (\text{WF-ASSIGN}) \qquad \frac{L\ c\ y \in \overline{V} \qquad L' = \mathsf{anno}(c, f)}{\overline{V} \vdash y.f : L' \triangleright_{\mathsf{From}} L}\ (\text{WF-FIELDGET})$$

$$\frac{L\ c\ x \in \overline{V} \qquad L' = \mathsf{anno}(c, f) \qquad L''\ \_\ y \in \overline{V} \qquad (L'' \triangleright_{\mathsf{To}} L) <: L'}{\overline{V} \vdash x.f \leftarrow y}\ (\text{WF-FIELDSET})$$

$$\frac{}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} : \mathsf{Far}}\ (\text{WF-NEWFRESH}) \qquad \frac{L\ \_\ x \in \overline{V}}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ x : L}\ (\text{WF-NEWSAME}) \qquad \frac{L\ \_\ x \in \overline{V}}{\overline{V} \vdash x : L}\ (\text{WF-VAR})$$

$$\frac{L\ c\ y \in \overline{V} \qquad L_i\ \_\ z_i \in \overline{V} \qquad \overline{x} = \mathsf{params}(c, m) \qquad (L_i \triangleright_{\mathsf{To}} L) <: \mathsf{anno}(c, m, x_i)}{\overline{V} \vdash y.m(\overline{z}) : \mathsf{anno}(c, m, \mathsf{result}) \triangleright_{\mathsf{From}} L}\ (\text{WF-CALL})$$

Fig. 10. Typing rules of LocJ. Judgments containing indices i are implicitly all-quantified.

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

A stack-entry is well-formed if its local variables annotated by Near or Far only reference objects at the same location or at a different location, respectively:

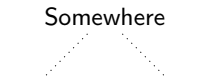
Definition 3 (Well-formed stack frame). *$(\overline{S}, \mathcal{D})^{c,m}$ is well-formed under $\mathcal{H}$ iff for all x with $\mathcal{D}(x) = \iota$, we have* $\mathsf{dtype}(\mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this}))), \mathsf{loc}(\mathcal{H}(\iota))) <: \mathsf{anno}(c, m, x)$.

The soundness of a type system is proven by showing preservation and progress (cf. [16]).

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

As shown at the end of Appendix A, the theorem directly follows from the preservation theorem for refined location types (Theorem 2) that is presented in the next subsection.

As location annotations do not put any additional restrictions on the dynamic semantics of LocJ, the proof of progress remains the same as for Welterweight Java [15]. In ABS however, where the semantics of the synchronous call depends on location information, progress needs to be shown. For the synchronous method call $y.m(\ldots)$, ABS requires that the object ι that y refers to is in the same COG as the current object (referred to by this). In our formalization, this means that



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. 11. 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 Section 3.2.

```
1  [Far] Server server = new cog ServerImpl();
2  [Far] Client client1 = new cog ClientImpl("Alice");
3  [Far] Client client2 = new cog ClientImpl("Bob");
4  client1 ! connectTo(server); // type error
5  client2 ! connectTo(server); // type error
```

Fig. 12. The code of the main block of the chat application, annotated with basic location types

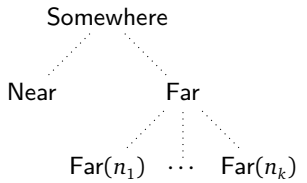
$\mathsf{loc}(\mathcal{H}(\iota)) = \mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this})))$. For the proof, we assume that the ABS program is well-typed; this means in particular that the location type of y is Near. The proof then follows directly from the preservation theorem. We assume that the stack frame that contains the synchronous call $y.m(\overline{z})$ is well-typed. By Definition 3, we get $\mathsf{dtype}(\mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this}))), \mathsf{loc}(\mathcal{H}(\iota))) <: \mathsf{Near}$. The claim $\mathsf{loc}(\mathcal{H}(\iota)) = \mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this})))$ then directly follows from the definitions of $<:$ and $\mathsf{dtype}(,)$ in Figures 11 and 8.

3.2 Refined Location Types

The location type system presented in the last section can only distinguish near and far: Near references point to a location that is different from locations pointed to by Far references. But whether two far references point to the same location or different ones is statically not known. This makes the type system often too weak in practice. As an example, let us consider the *main block* of the ABS chat application in Figure 12, annotated with location types. Note that a main block in ABS corresponds to a main method in Java. The server and both client objects are created in their own, fresh COG, and thus they can be typed as Far, because these locations are different from the current COG (the *Main* COG). However, the method call client1!connectTo(server) does not type-check in the basic location type system. According to rule WF-CALL in Figure 10, the adaptation of the actual parameter type of the connectTo method to the caller has to be a subtype of the formal parameter type, but the adaptation $\mathsf{Far} \triangleright_{\mathsf{To}} \mathsf{Far}$ yields Somewhere, which is not a subtype of the formal parameter type Far. This problem 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

```
1 [Far(s)] Server server = new cog ServerImpl();
2 [Far(c1)] Client client1 = new cog ClientImpl("Alice");
3 [Far(c2)] Client client2 = new cog ClientImpl("Bob");
4 client1 ! connectTo(server);
5 client2 ! connectTo(server);
```

Fig. 13. The code of the main block of the chat application, annotated with refined location types



Original	$\triangleright_K$	*Viewpoint* = *Adapted*		
… (extension of Figure 11)				
Near	$\triangleright_{\mathsf{To}}$	Far(n)	= Far	
Near	$\triangleright_{\mathsf{From}}$	Far(n)	= Far(n)	
Far	$\triangleright_K$	Far(n)	= Somewhere	
Somewhere	$\triangleright_K$	Far(n)	= Somewhere	
Far(n_1)	$\triangleright_{\mathsf{To}}$	Far(n_2)	= Far(n_1)	if $n_1 \neq n_2$
Far(n_1)	$\triangleright_{\mathsf{From}}$	Far(n_2)	= Somewhere	if $n_1 \neq n_2$
Far(n)	$\triangleright_K$	Far(n)	= Somewhere	
Far(n)	$\triangleright_K$	Far	= Somewhere	

Fig. 14. Subtyping and viewpoint adaptation for extended location types

conservatively use the Somewhere type to remain sound, which is often too weak to type practical programs.

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

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

A named far type is a far type parametrized with an arbitrary name. We let n range over far names, a fresh syntactic category. Far types with different names represent disjoint sets of far locations and are incompatible with each other. Figure 13 illustrates the application of named far location types. The locations of server, client1, and client2 are distinguished by using different names for typing different variables. Using this technique, the programmer can distinguish finitely many Far types in a program. This is similar to static analysis techniques that use source code positions to distinguish abstract program entities [17].

The following typing rule WF-NEWFRESHP is added to the basic type system. It allows to type object creations at fresh locations by an arbitrarily named Far type:

$$\frac{}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} : \mathsf{Far}(n)}\ (\text{WF-NEWFRESHP})$$

The subtyping and viewpoint adaptation relations are extended accordingly in Figure 14. Adapting a Far(n_1) to a Far(n_2) for $n_1 \neq n_2$ yields a Far(n_1), as they denote different sets of locations. Adapting a Far(n) to a Far(n) does not yield Near, however, as two variables with the same Far(n) type can refer to objects of different locations. Thus, in the refined type system, the chat example in Figure 13 is

type correct: The adaptation of the actual parameter type Far(s) of the connectTo method in Line 4 to the caller type Far(c1) yields Far(s), which is a subtype of the formal parameter type Far.

In practice, the programmer does not need to provide the names. Instead the inference system (see Sect. 3.3) automatically infers the named far types. Thus, the programmer is not confronted with more complex typing. Using the refined location type system, we were able to fully type the case studies presented in Section 4. However, other refinements or 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 [2,18,19].

Type soundness. We adapt the well-formedness definitions from Section 3.1 to the refined location type system. The definitions for well-formed heap entries and stack frames are similar to the ones in the previous subsection except that we abstract away the named far types using the helper function $\mathsf{abs}(L)$ defined in Figure 8:

Definition 4 (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* $\mathsf{dtype}(l, l') <: \mathsf{abs}(\mathsf{anno}(c, f))$.

Definition 5 (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* $\mathsf{dtype}(\mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this}))), \mathsf{loc}(\mathcal{H}(\iota))) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$.

To formalize that far types with different names represent disjoint sets of far locations, we introduce a relation $\mathsf{reach}(\zeta)$ that denotes for a runtime configuration ζ what the types of the variables are that point to a certain location:

Definition 6 (Reachability of l under ζ). *Let $\zeta = \overline{\mathcal{F}}, \mathcal{H}$; We write* $\mathsf{reach}(\zeta)$ *to denote the smallest relation that satisfies the following conditions:*

- *If $(\overline{S}, \mathcal{D})^{c,m} \in \overline{\mathcal{F}}$ and $\mathcal{D}(x) = \iota$ and $l_x = \mathsf{loc}(\mathcal{H}(\iota))$, then* $(\mathsf{anno}(c, m, x), l_x) \in \mathsf{reach}(\zeta)$.
- *If $(l, c, \mathcal{D}) \in \mathsf{rng}(\mathcal{H})$ and $\mathcal{D}(f) = \iota$ and $l_f = \mathsf{loc}(\mathcal{H}(\iota))$, then* $(\mathsf{anno}(c, f), l_f) \in \mathsf{reach}(\zeta)$.

Using $\mathsf{reach}(\zeta)$, we can state that in well-formed configurations variables with different named far types refer to objects in different locations:

Definition 7 (Well-formed named Far locations). *ζ is well-formed wrt. named* Far *locations iff $\forall(\mathsf{Far}(n_1), l_1), (\mathsf{Far}(n_2), l_2) \in \mathsf{reach}(\zeta)$ with $n_1 \neq n_2$, we have $l_1 \neq l_2$.*

In summary, we get the following revised definition of well-formed runtime configurations:

Definition 8 (Well-formed runtime configuration). *Let $\zeta = \overline{\mathcal{F}}, \mathcal{H}$ be a runtime configuration. ζ 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 ζ is well-formed wrt. named* Far *locations and the configuration satisfies all the standard conditions of a class-based language.*

As explained in the last section, it suffices to show the preservation property for type soundness of the refined location type system:

Theorem 2 (Preservation for refined location types). *Let ζ be a well-formed runtime configuration. If $\zeta \rightsquigarrow \zeta'$, then ζ' is well-formed as well.*

The proof proceeds by a case analysis on the reduction rules. It is presented in Appendix A.

3.3 Location Type Inference

Without further support, the type system presented in the previous section requires the programmer to annotate all type occurrences with location types. To avoid this tedious work and make the approach practical, we developed 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 to achieve modular type checking. A type inference system is *sound* if it yields only correct typings. It is *complete*, if every correct typing can be inferred. In the second part of this section, we improve the inference system such that it can deal with type-incorrect programs with the purpose of generating meaningful error messages. We also make the inference system configurable such that it finds not only any possible solution, but good solutions satisfying further desirable properties.

Sound and Complete Inference. The formal model for inferring location types follows the formalization of other type system extensions [20]. For a broader overview of type inference, we refer to the excellent book by Pierce [21] from which we borrow the notation. 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, P denotes programs that are fully annotated with pairwise distinct location type variables. This means that all type occurrences in P are of the form $\alpha\, c$. The constraints which are generated by the inference system are shown in Figure 15. We use the judgment $\vdash P : \overline{\mathcal{Q}}$ to denote the generation of the constraints $\overline{\mathcal{Q}}$ from program P and similar judgments for classes and statements. The judgment for expressions E has the form $\overline{V} \vdash E : \alpha, \overline{\mathcal{Q}}$ with the following meaning: Assuming the bindings $\overline{V}$, the location type of E is (the solution for) α, and $\overline{\mathcal{Q}}$ are the inferred constraints. The judgments are defined in Figure 16. Note that additional *fresh* location type variables are introduced during the constraint generation and that the constraints generated for a program are unique modulo the renaming of the fresh location type variables. Using an appropriate naming convention, we can assume without loss of generality that there is a unique constraint set for a program P, denoted by $\overline{\mathcal{Q}}_P$.

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

Fig. 15. Location type constraints

$$\frac{P = \overline{C} \qquad \vdash C_i : \overline{\mathcal{Q}_i}}{\vdash P : \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_k}}$$

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

$$\frac{c \vdash M_i : \overline{\mathcal{Q}_i}}{\vdash \mathsf{class}\ c\ \{\ \overline{V}\ \overline{M}\ \} : \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_k}}$$

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

$$\frac{\overline{V} \vdash E : \beta, \overline{\mathcal{Q}} \qquad \alpha\ \_\ x \in \overline{V}}{\overline{V} \vdash x \leftarrow E : \beta <: \alpha \cdot \overline{\mathcal{Q}}}$$

$$\frac{\alpha\ c\ y \in \overline{V} \qquad \beta = \mathsf{anno}(c, f) \qquad \gamma \text{ is fresh}}{\overline{V} \vdash y.f : \gamma, \beta \triangleright_{\mathsf{From}} \alpha = \gamma}$$

$$\frac{\delta \text{ is fresh} \qquad \gamma \text{ is fresh}}{\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} : \delta, \delta <: \gamma \cdot \gamma = \mathsf{Far}}$$

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

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

Fig. 16. Constraint generation rules. Judgments with indices i are implicitly all-quantified.

Let us denote the location variables of P by $\mathsf{locv}(P)$ and those occurring in $\overline{\mathcal{Q}_P}$ by $\mathsf{locv}(\overline{\mathcal{Q}_P})$; obviously, $\mathsf{locv}(P) \subseteq \mathsf{locv}(\overline{\mathcal{Q}_P})$. Let σ be a variable substitution from $\mathsf{locv}(P)$ (or $\mathsf{locv}(\overline{\mathcal{Q}_P})$) to location types $\{\mathsf{Near}, \mathsf{Far}, \mathsf{Somewhere}, \mathsf{Far}(n_1), ..., \mathsf{Far}(n_k)\}$. We write σP to denote the program that is obtained from P by replacing all location type variables according to σ. We call σ a *solution* of $\overline{\mathcal{Q}_P}$, written as $\sigma \vDash \overline{\mathcal{Q}_P}$, if the constraints $\overline{\mathcal{Q}_P}$ are satisfied under σ. Type inference is *sound*, if every solution leads to a correct typing of P. It is *complete*, if each typing of P can be inferred.

Theorem 3 (Soundness and Completeness of the Inference). *The described inference system is sound and complete:*

- Sound*: If* $\sigma \vDash \overline{\mathcal{Q}_P}$*, then* $\vdash \sigma P$.
- Complete*: If* $\vdash \sigma P$ *with* $\mathsf{dom}(\sigma) = \mathsf{locv}(P)$*, then there is a solution* σ' *of* $\overline{\mathcal{Q}_P}$ *such that* $\sigma' \mid_{\mathsf{dom}(\sigma)} = \sigma$.

The proof of this theorem is presented in Appendix B.

Partial and Tunable Inference. Soundness and completeness guarantee that all inferred solutions lead to correct typings and all typings can be inferred. From a practical point of view, an inference system should meet further requirements, in particular:

1. If no typable solution can be inferred, at least a partially typable solution should be provided (a message "No solution" is not really helpful to correct the program). In addition, this partially typable solution should lead to the least amount of type errors.
2. If multiple solutions exist, a "good" solution should be selected. Users do not want any solution, but a solution that satisfies further properties. For example, a "good" solution could provide more precise types than other solutions. An inference system that allows to specify such additional properties is called *tunable* [20].

To infer partial solutions that satisfy the first requirement, we extend our formal model in the following way. We introduce two constraint categories: *must-have* and *should-have*. The *must-have* constraints must always be satisfied. These are for example in Figure 16 the adaptation constraints ($\alpha \triangleright_K \beta = \gamma$) and the constant constraints ($\alpha = L$), characterizing the types of subexpressions. They also encompass the constant constraints which result from user annotations (not considered in the formalization of Figure 16, 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 Figure 16, should always be satisfied in order to get a valid typing, but can be unsatisfied for partially correct solutions.

As an example for partial inference, consider the ServerImpl class in Figure 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 *typeOf*(s) <: *typeOf*(result) which is generated at the last line of the connect method (*typeOf* yields the corresponding type variable). 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).

The second requirement, namely inferring "good" solutions, can be realized by adding the additional category of *nice-to-have* constraints. The *nice-to-have* constraints are those that are used to specify further desirable properties, e.g., least amount of Somewhere annotations or Far types at the places where the precision of Far(n) types is not needed.

Inferring a partial and "good" 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. Prioritizing must-have and should-have constraints ensures that the inference system remains sound for the cases where a typable solution exists. 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 (see Section 4).

4 Implementation and IDE Integration

We have implemented the type system and the inference system for location types, including named far location types and partial inference satisfying further heuristics, 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 [22] 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 based on named Far locations (cf. Section 3.2) can be configured to use scopes of different granularity: basic (no alias analysis), method-local, class-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(n) 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[3] 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 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 four case studies:

CS (251 non-commented non-empty lines of code (LOC), 59 types to annotate) is an extended version of the chat application presented in Section 2.
TS (1123 LOC, 152 types to annotate) is an academic case study of a trading system for handling sales in supermarkets.
RS (3698 LOC, 104 types to annotate) models parts of an industrial case study on server-based software systems.
LS (2771 LOC, 301 types to annotate) is an academic case study of a lecture management system.

The evaluation results are presented in Figure 17. They show how precise the subject systems can be typed and how quickly the inference runs. We also restricted the alias analysis by various scopes to see its impact on performance and precision. First

[3] Download link for the tool:
`http://tools.hats-project.eu/eclipseplugin/installation.html`

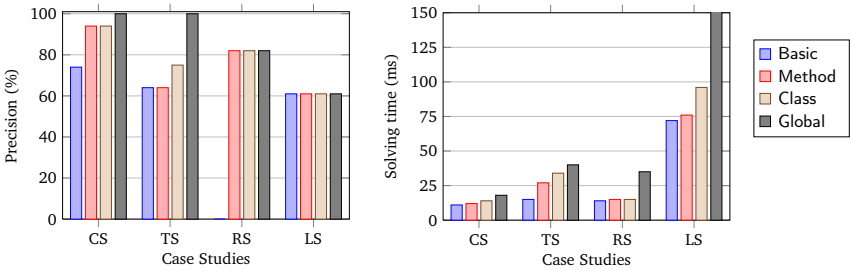


Fig. 17. Precision and solving time of the location type inference for the three case studies, using four different scopes for the alias analysis. The measurements where done on a MacBook Pro laptop (Intel Core 2 Duo 2.8GHz CPU, 4GB RAM, Mac OS X 10.6). We used the `-Xms1024` parameter to avoid garbage 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.

of all, all subject systems 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 cases. As expected, the precision increased with a broader analysis scope. Using a global alias analysis, the inference achieved a precision of 100% for CS and TS. It is also interesting to note that, depending on the case study, extending the scope of the alias analysis can have an effect or not. RS could not be correctly typed with basic precision (but returned a partial solution in 14 ms). For CS and RS, the best precision was already achieved with a method-local scope. For LS, basic scope was sufficient.

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 alias analysis. In the evaluation, we used completely unannotated examples, so that all types had to be inferred. In practice, programs are often partially annotated, which additionally improves the performance of the type inference. We believe the analysis to be scalable, as it is based on a modular type system. The analysis can be applied to parts of a system, and the analysis results can then be reused to infer the types in different parts of the system.

It remains to be investigated whether other constraint encodings and backends will yield better performance. At the moment, the presented solving times strongly rely on the performance of the underlying SAT solver.

5 Discussion and Related Work

Location types are a lightweight variant of ownership types that focus on *flat* ownership contexts. This means that no nesting of locations is permitted. The object space is structured such that we can distinguish between local and non-local objects. We

exploited this information in the context of distributed object-oriented programming to check whether an access or method call is local or remote. Locations can also be used for other purposes, in particular

- as scheduling units in multi-core scenarios [23],
- to raise the level of granularity of memory management and garbage collection from single objects to group of objects [24],
- for encapsulation of objects [25] or other values [13], and
- to support specification and verification techniques [26].

Ownership types [2,18,19] and similar type systems [27,28] 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 these systems more complex. An ownership type system which is close to location types in nature is that by Clarke et al. [29], 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 [30] 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 [31] also partition the heap. However, the set of places is fixed at the time the program is started. Similar in nature, but less expressive than our type system, is Loci [32], 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 [33], 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 by Flanagan and Freund [34] to infer types that prevent data-races and by Dietl, Ernst and Müller [20] 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 [35]. For variations of this type system, there exist built-in inference mechanisms in Eclipse[4] and IntelliJ IDEA[5] as well as additional plug-ins [36]. However, none of these IDE integrations provide the option to visualize the inferred type information in all relevant places directly using overlays, but only provide specific overlays on request.

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

[4] `http://wiki.eclipse.org/JDT_Core/Null_Analysis`
[5] `http://www.jetbrains.com/idea/webhelp/inferring-nullity.html`

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. Our evaluation 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 three 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 [8]. Second, it would be interesting to investigate the visual overlay technique for other (pluggable) type systems, e.g., the nullness type system [36]. Third, it seems worthwhile to weaken the premise that objects stay at a location for their entire lifetime (for a motivation of object migration see Mycroft [23] and for the treatment of object transfer in relation to ownership see Müller and Rudich [37]).

References

1. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales, Australia (2001)
2. Clarke, D.G., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Freeman-Benson, B.N., Chambers, C. (eds.) Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1998), Vancouver, British Columbia, Canada, October 18-22, pp. 48–64. ACM (1998)
3. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic Java. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Portland, Oregon, USA, October 22-26. ACM (2006)
4. 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)
5. 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 (2007)
6. 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)
7. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
8. Oracle Corporation: Java SE 6 RMI documentation, `http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html`
9. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Portland, Oregon, USA, October 22-26, pp. 57–74. ACM (2006)

10. Ernst, M.D.: Type Annotations Specification (JSR 308) and The Checker Framework: Custom pluggable types for Java, http://types.cs.washington.edu/jsr308/
11. Welsch, Y., Schäfer, J.: Location Types for Safe Distributed Object-Oriented Programming. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 194–210. Springer, Heidelberg (2011)
12. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI, pp. 235–245 (1973)
13. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. Inf. Comput. 196(2), 127–155 (2005)
14. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)
15. Östlund, J., Wrigstad, T.: Welterweight Java. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 97–116. Springer, Heidelberg (2010)
16. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (1994)
17. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for java using annotated constraints. In: Northrop, L.M., Vlissides, J.M. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, pp. 43–55. ACM (2001)
18. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Aiken, A., Morrisett, G. (eds.) Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, pp. 213–223. ACM (2003)
19. Lu, Y., Potter, J.: On Ownership and Accessibility. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 99–123. Springer, Heidelberg (2006)
20. Dietl, W., Ernst, M.D., Müller, P.: Tunable Static Inference for Generic Universe Types. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 333–357. Springer, Heidelberg (2011)
21. Pierce, B.C. (ed.): Advanced Topics in Types and Programming Languages. MIT Press (2005)
22. Le Berre, D., Parrain, A.: The SAT4J library, release 2.2, system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
23. Mycroft, A.: Location—the other confinement form. In: Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, IWACO 2011 (2011)
24. Gay, D., Aiken, A.: Memory management with explicit regions. In: Davidson, J.W., Cooper, K.D., Berman, A.M. (eds.) Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, pp. 313–323. ACM (1998)
25. Geilmann, K., Poetzsch-Heffter, A.: Modular checking of confinement for object-oriented components using abstract interpretation. In: Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, IWACO 2011 (2011)
26. Poetzsch-Heffter, A., Schäfer, J.: Modular Specification of Encapsulated Object-Oriented Components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 313–341. Springer, Heidelberg (2006)
27. Aldrich, J., Chambers, C.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
28. Dietl, W.: Universe Types: Topology, Encapsulation, Genericity, and Tools. PhD thesis, ETH Zurich, Switzerland (2009)

29. 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)
30. 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)
31. Grothoff, C.: Expressive Type Systems for Object-Oriented Languages. PhD thesis, University of California, Los Angeles (2006)
32. 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)
33. Tofte, M., Talpin, J.P.: Region-based memory management. Inf. Comput. 132(2), 109–176 (1997)
34. Flanagan, C., Freund, S.N.: Type inference against races. Sci. Comput. Program. 64, 140–165 (2007)
35. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Crocker, R., Steele Jr., G.L. (eds.) Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, Anaheim, CA, USA, October 26-30, pp. 302–312. ACM (2003)
36. 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)
37. Müller, P., Rudich, A.: Ownership transfer in universe types. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Steele Jr., G.L. (eds.) Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, Montreal, Quebec, Canada, October 21-25, pp. 461–478. ACM (2007)

A Proof of the Preservation Theorems

In the following, we present the proofs of Theorem 2 and Theorem 1. As a fist step, we state a few lemmata that relate the definitions given in Sections 3.1 and 3.2.

Lemma 1. *For all* $l_1, l_2, l_3 : \mathsf{dtype}(l_1, l_3) <: \mathsf{dtype}(l_1, l_2) \triangleright_K \mathsf{dtype}(l_2, l_3)$.

Proof. By case analyis ($l_1 = l_2 = l_3$, $l_1 = l_2 \neq l_3$, $l_1 \neq l_2 = l_3$, $l_1 = l_3 \neq l_2$, $l_1 \neq l_2 \neq l_3 \neq l_1$).

Lemma 2. *For all* $l_1, l_2 : \mathsf{dtype}(l_1, l_2) = \mathsf{dtype}(l_2, l_1)$.

Proof. Directly from definition of dtype(_,_).

Lemma 3. *If* $L_1, L_2, L_3, L_4 \in \{\mathsf{Near}, \mathsf{Far}, \mathsf{Somewhere}\}$ *and* $L_1 <: L_2$ *and* $L_3 <: L_4$, *then* $L_1 \triangleright_K L_3 <: L_2 \triangleright_K L_4$.

Proof. By case analysis.

Lemma 4. *If* $L_1 <: L_2$, *then* $\mathsf{abs}(L_1) <: \mathsf{abs}(L_2)$.

Proof. By case analysis.

Lemma 5. *If* $L_1 \triangleright_{\mathsf{From}} L_2 <: L_3$*, then* $\mathsf{abs}(L_1) \triangleright_{\mathsf{From}} \mathsf{abs}(L_2) <: \mathsf{abs}(L_3)$.

Proof. By case analysis.

We introduce the predicate $\mathsf{diffFar}(L_1, L_2)$ that determines whether L_1 and L_2 are both named far types, but with different names. Put formally, $\mathsf{diffFar}(L_1, L_2) = \mathsf{True}$ if $L_1 = \mathsf{Far}(n_1)$ and $L_2 = \mathsf{Far}(n_2)$ and $n_1 \neq n_2$, and False otherwise.

Lemma 6. *If* $L_1 \triangleright_{\mathsf{To}} L_2 <: L_3$ *and* $\neg\mathsf{diffFar}(L_1, L_2)$*, then* $\mathsf{abs}(L_1) \triangleright_{\mathsf{To}} \mathsf{abs}(L_2) <: \mathsf{abs}(L_3)$.

Proof. By case analysis.

Proof of Theorem 2:

Assume a well-formed configuration ζ for a program P such that $\vdash P$ and $\zeta \rightsquigarrow \zeta'$. This means that the heap entries (Definition 4) are well-formed, the stack frames (Definition 5) for ζ are well-formed and ζ is well-formed with respect to named far locations. Then prove that ζ' is well-formed as well. In the presentation of the proof we focus on the location type aspects. We assume that conditions of a standard class-based type system hold, of which we name a few:

Condition 1: We do not allow this on the left hand side of an assignment, and thus the object (and its location) referenced by this remains the same (for a certain stack frame) after reduction steps.

Condition 2: If a variable points to an object with dynamic class type c, the variable is also statically typed as c.

The proof then proceeds by a standard case analysis on the reduction rules used. We show the first rule in detail; in the presentation of the other rules, we omit uninteresting cases (e.g. transfer of null values). Throughout the proof, l_{this} is used as abbreviation of $\mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{this})))$.

Case R-ASSIGN:

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

We distinguish two cases:

Case $x = y$**:** The configuration then remains unchanged and the claim follows trivially.

Case $x \neq y$**:**

Case $\mathcal{D}(y) = \mathsf{null}$**:** Thus $\mathcal{D}'(x) = \mathsf{null}$ and the claim follows trivially.

Case $\mathcal{D}(y) \neq \mathsf{null}$**:** Let $l_y := \mathsf{loc}(\mathcal{H}(\mathcal{D}(y)))$. We have

1. $l_{\mathsf{this}} = \mathsf{loc}(\mathcal{H}(\mathcal{D}'(\mathsf{this})))$ by condition 1,
2. $l_y = \mathsf{loc}(\mathcal{H}(\mathcal{D}'(x)))$ as $\mathcal{D}'(x) = \mathcal{D}(y)$,
3. $\mathsf{anno}(c, m, y) <: \mathsf{anno}(c, m, x)$ by typing rules WF-ASSIGN and WF-VAR,
4. $\mathsf{abs}(\mathsf{anno}(c, m, y)) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$ by Lemma 4 from step 3,

5. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c, m, y))$ by well-formedness assumption of stack frame $(x \leftarrow y \cdot \overline{S}, \mathcal{D})^{c,m}$, and
6. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$ by the previous two steps using transitivity of the subtype relation.

We have two proof goals as the heap remains unchanged. We first want to prove that $(\overline{S}, \mathcal{D}')^{c,m}$ is well-formed. By Definition 5 this amounts to proving that for all z with $\mathcal{D}'(z) = \iota$, we have $\mathsf{dtype}(l_{\mathsf{this}}, \mathsf{loc}(\mathcal{H}(\iota))) <: \mathsf{anno}(c, m, z)$. As only x is modified, it is enough to prove this for x (i.e. $\mathsf{dtype}(l_{\mathsf{this}}, \mathsf{loc}(\mathcal{H}(\mathcal{D}'(x)))) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$), which follows directly from steps 2 and 6.

The second proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. As only x is modified, we need to only consider the case $(\mathsf{anno}(c, m, x), l_x) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c, m, x) = \mathsf{Far}(n)$. By step 3, we then know that $\mathsf{anno}(c, m, y) = \mathsf{Far}(n)$ and consequently $(\mathsf{anno}(c, m, y), l_y) \in \mathsf{reach}(\zeta)$. As $\mathsf{anno}(c, m, x) = \mathsf{anno}(c, m, y) = \mathsf{Far}(n)$ and $l_x = l_y$ (from step 1) and by well-formedness assumption of ζ, the claim follows.

Case R-NewSame:

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

We have

1. $\mathsf{anno}(c, m, y) <: \mathsf{anno}(c, m, x)$ by typing rules WF-Assign and WF-NewSame,
2. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c, m, y))$ by well-formedness assumption of stack frame $(x \leftarrow \mathsf{new}\ c'\ \mathsf{in}\ y \cdot \overline{S}, \mathcal{D})^{c,m}$,
3. $\mathsf{abs}(\mathsf{anno}(c, m, y)) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$ by Lemma 4 applied to step 1,
4. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$ from steps 2 and 3 by transitivity of subtype relation, and
5. $l_y = \mathsf{loc}(\mathcal{H}'(\mathcal{D}'(x)))$ directly from the rule R-NewSame.

The first proof goal is to show that $(\overline{S}, \mathcal{D}')^{c,m}$ is well-formed. As only x is modified, this amounts to proving that $\mathsf{dtype}(l_{\mathsf{this}}, \mathsf{loc}(\mathcal{H}'(\mathcal{D}'(x)))) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$ which follows from steps 4 and 5. The second proof goal is to show that $\mathcal{H}'$ is well-formed, which is trivially the case as all the fields of the new object contain null. The third proof goal is to show that ζ' is well-formed wrt. named Far locations, which is similar to the rule R-Assign, as the newly created object is only reachable from x.

Case R-NewFresh:

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

We have

1. $\mathsf{Far}(n) <: \mathsf{anno}(c,m,x)$ for some i by typing rules WF-ASSIGN and WF-NEWFRESHP,
2. $\mathsf{Far} <: \mathsf{abs}(\mathsf{anno}(c,m,x))$ by Lemma 4 from previous step,
3. $\mathsf{dtype}(l_{\mathsf{this}}, l) = \mathsf{Far}$ as l is fresh,
4. $\mathsf{dtype}(l_{\mathsf{this}}, l) <: \mathsf{abs}(\mathsf{anno}(c,m,x))$ by steps 2 and 3, and
5. $l = \mathsf{loc}(\mathcal{H}'(\mathcal{D}'(x)))$ directly from the rule R-NEWFRESH.

The proof goals are the same as for the previous rule R-NEWSAME. The claim $\mathsf{dtype}(l_{\mathsf{this}}, \mathsf{loc}(\mathcal{H}'(\mathcal{D}'(x)))) <: \mathsf{abs}(\mathsf{anno}(c,m,x))$ follows from steps 4 and 5. The new heap entry is trivially well-formed as all the fields contain null. The new configuration is well-formed wrt. named Far locations as l is fresh and thus l is different to all other locations.

Case R-FIELDGET:

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

We only consider the case $\mathcal{D}''(f) \neq \mathsf{null}$. Let $l_f := \mathsf{loc}(\mathcal{H}(\mathcal{D}''(f)))$. We have

1. $\mathsf{anno}(c', f) \triangleright_{\mathsf{From}} \mathsf{anno}(c,m,y) <: \mathsf{anno}(c,m,x)$ by typing rules WF-ASSIGN and WF-FIELDGET,
2. $\mathsf{dtype}(l_y, l_f) <: \mathsf{abs}(\mathsf{anno}(c', f))$ by well-formedness assumption of heap entry $(l_y, c', \mathcal{D}'')$,
3. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c,m,y))$ by well-formedness assumption of stack frame $(x \leftarrow y.f \cdot \overline{S}, \mathcal{D})^{c,m}$,
4. $\mathsf{dtype}(l_y, l_f) \triangleright_{\mathsf{From}} \mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c', f)) \triangleright_{\mathsf{From}} \mathsf{abs}(\mathsf{anno}(c,m,y))$ from steps 2 and 3 by Lemma 3,
5. $\mathsf{dtype}(l_{\mathsf{this}}, l_f) <: \mathsf{dtype}(l_y, l_f) \triangleright_{\mathsf{From}} \mathsf{dtype}(l_{\mathsf{this}}, l_y)$ by Lemma 1 and Lemma 2,
6. $\mathsf{abs}(\mathsf{anno}(c', f)) \triangleright_{\mathsf{From}} \mathsf{abs}(\mathsf{anno}(c,m,y)) <: \mathsf{abs}(\mathsf{anno}(c,m,x))$ by Lemma 5 from step 1,
7. $\mathsf{dtype}(l_{\mathsf{this}}, l_f) <: \mathsf{abs}(\mathsf{anno}(c,m,x))$ by steps 5, 4 and 6 using transitivity of the subtype relation, and
8. $l_f = \mathsf{loc}(\mathcal{H}(\mathcal{D}'(x)))$ as $\mathcal{D}''(f) = \mathcal{D}'(x)$.

We have two proof goals as the heap remains unchanged. The first proof goal is to show that $(\overline{S}, \mathcal{D}')^{c,m}$ is well-formed. As only x is modified, it is sufficient to show that $\mathsf{dtype}(l_{\mathsf{this}}, \mathsf{loc}(\mathcal{H}(\mathcal{D}'(x)))) <: \mathsf{abs}(\mathsf{anno}(c,m,x))$ which follows directly from steps 7 and 8.

The second proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. As only x is modified, we need to only consider the case $(\mathsf{anno}(c,m,x), l_f) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c,m,x) = \mathsf{Far}(n)$.

By step 1, we then know that either

- $\mathsf{anno}(c,m,y) = \mathsf{Near}$ and $\mathsf{anno}(c', f) = \mathsf{Far}(n)$ or
- $\mathsf{anno}(c,m,y) = \mathsf{Far}(n)$ and $\mathsf{anno}(c', f) = \mathsf{Near}$.

In the first case, as $(\mathsf{anno}(c', f), l_f) \in \mathsf{reach}(\zeta)$ and ζ well-formed, the claim follows directly from step 8. In the second case, $l_y = l_f$ and the claim then follows similarly to the first case.

Case R-FieldSet:

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

We only consider the case $\mathcal{D}(y) \neq \mathsf{null}$. Let $l_x := \mathsf{loc}(\mathcal{H}(\mathcal{D}(x)))$ and $l_y := \mathsf{loc}(\mathcal{H}(\mathcal{D}(y)))$. We have

1. $\mathsf{anno}(c, m, y) \triangleright_{\mathsf{To}} \mathsf{anno}(c, m, x) <: \mathsf{anno}(c', f)$ by typing rules WF-Assign and WF-FieldSet,
2. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c, m, y))$ by well-formedness assumption of stack frame $(x.f \leftarrow y \cdot \overline{S}, \mathcal{D})^{c,m}$,
3. $\mathsf{dtype}(l_{\mathsf{this}}, l_x) <: \mathsf{abs}(\mathsf{anno}(c, m, x))$ by well-formedness assumption of stack frame $(x.f \leftarrow y \cdot \overline{S}, \mathcal{D})^{c,m}$, and
4. $l_y = \mathsf{loc}(\mathcal{H}'(\mathcal{D}''(f)))$ as $\mathcal{D}''(f) = \mathcal{D}(y)$.

We distinguish two cases:

Case $\neg\mathsf{diffFar}(\mathsf{anno}(c, m, x), \mathsf{anno}(c, m, y))$**:** Similarly to the case R-FieldGet we have $\mathsf{dtype}(l_x, l_y) <: \mathsf{abs}(\mathsf{anno}(c', f))$ from steps 1, 2, and 3 by Lemma 1, Lemma 2, Lemma 3, Lemma 6 and transitivity of subtyping.
One proof goal is this time to show that the heap entry $(l_x, c', \mathcal{D}'')$ is well-formed. As only f is modified, we have to prove that $\mathsf{dtype}(l_x, \mathsf{loc}(\mathcal{H}'(\mathcal{D}''(f)))) <: \mathsf{abs}(\mathsf{anno}(c', f))$. This results directly from step 4.
The other proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. As only f is modified, we need to only consider the case $(\mathsf{anno}(c', f), l_y) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c', f) = \mathsf{Far}(n)$. By step 1 and case assumption, we then know that $\mathsf{anno}(c, m, x) = \mathsf{Near}$ and $\mathsf{anno}(c, m, y) = \mathsf{Far}(n)$. The claim then follows directly.

Case $\mathsf{diffFar}(\mathsf{anno}(c, m, x), \mathsf{anno}(c, m, y))$**:** Assume wlog that $\mathsf{anno}(c, m, x) = \mathsf{Far}(n_1)$ and $\mathsf{anno}(c, m, y) = \mathsf{Far}(n_2)$ with $n_1 \neq n_2$. We then get $\mathsf{Far}(n_2) <: \mathsf{anno}(c', f)$ from step 1. We also have $l_x \neq l_y$ by well-formedness assumption of the old configuration. Thus $\mathsf{dtype}(l_x, l_y) = \mathsf{Far}$.
One proof goal is this time to show that the heap entry $(l_x, c', \mathcal{D}'')$ is well-formed. As only f is modified, we have to prove that $\mathsf{dtype}(l_x, \mathsf{loc}(\mathcal{H}'(\mathcal{D}''(f)))) <: \mathsf{abs}(\mathsf{anno}(c', f))$. This follows directly from step 4 and the results in the previous paragraph.
The other proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. As only f is modified, we need to only consider the case $(\mathsf{anno}(c', f), l_y) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c', f) = \mathsf{Far}(n_2)$. By case assumption, we know that $\mathsf{anno}(c, m, y) = \mathsf{Far}(n_2)$ from which the claim follows directly.

Case R-Call:

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

We consider only the case $\mathcal{D}(z_i) \neq \mathsf{null}$. Let $l_y := \mathsf{loc}(\mathcal{H}(\mathcal{D}(y)))$ and $l_{z_i} := \mathsf{loc}(\mathcal{H}(\mathcal{D}(z_i)))$. Let $\overline{x} := \mathsf{params}(c', m')$ and $\mathcal{F}' := (\overline{S}', \mathcal{D}')^{c',m'}$. We have

1. $\mathsf{anno}(c, m, z_i) \triangleright_{\mathsf{To}} \mathsf{anno}(c, m, y) <: \mathsf{anno}(c', m', x_i)$ by typing rules WF-ASSIGN and WF-CALL,
2. $\mathsf{dtype}(l_{\mathsf{this}}, l_{z_i}) <: \mathsf{abs}(\mathsf{anno}(c, m, z_i))$ by well-formedness assumption of stack frame $\mathcal{F}$,
3. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{abs}(\mathsf{anno}(c, m, y))$ by well-formedness assumption of stack frame $\mathcal{F}$,
4. $\mathcal{F}'$ is of the form $(\overline{S}', \mathcal{D}')^{c',m'}$ where $\mathcal{D}'(\mathsf{this}) = \mathcal{D}(y)$ and $\mathcal{D}'(x_i) = \mathcal{D}(z_i)$ by definition of $\mathsf{initF}(,,,)$, and
5. $l_{x_i} = l_{z_i}$ as $\mathcal{D}(z_i) = \mathcal{D}'(x_i)$.

We distinguish two cases:

Case $\neg\mathsf{diffFar}(\mathsf{anno}(c, m, y), \mathsf{anno}(c, m, z_i))$**:** Similarly to the case R-FIELDGET we have $\mathsf{dtype}(l_y, l_{z_i}) <: \mathsf{abs}(\mathsf{anno}(c', m', x_i))$ from steps 1, 2 and 3 by Lemma 1, Lemma 2, Lemma 3, Lemma 6 and transitivity of subtyping.

The first proof goal is to show that $\mathcal{F}'$ is well-formed. We only need to consider the parameters $\overline{x}$, as the local variables are initialized with null. We thus need to show that $\mathsf{dtype}(l_y, l_{x_i}) <: \mathsf{abs}(\mathsf{anno}(c', m', x_i))$ which follows directly from step 5.

The other proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. We need to only consider the case $(\mathsf{anno}(c', m', x_i), l_y) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c', m', x_i) = \mathsf{Far}(n)$. By step 1 and case assumption, we then know that $\mathsf{anno}(c, m, y) = \mathsf{Near}$ and $\mathsf{anno}(c, m, z_i) = \mathsf{Far}(n)$. The claim then follows directly.

Case $\mathsf{diffFar}(\mathsf{anno}(c, m, y), \mathsf{anno}(c, m, z_i))$**:** Assume wlog that $\mathsf{anno}(c, m, y) = \mathsf{Far}(n_1)$ and $\mathsf{anno}(c, m, z_i) = \mathsf{Far}(n_2)$ with $n_1 \neq n_2$. By step 1, we get $\mathsf{Far}(n_2) <: \mathsf{anno}(c', m', x_i)$. We also have $l_y \neq l_{z_i}$ by well-formedness assumption of the old configuration. Thus $\mathsf{dtype}(l_y, l_{z_i}) = \mathsf{Far}$.

The first proof goal is to show that $\mathcal{F}'$ is well-formed. We only need to consider the parameters $\overline{x}$, as the local variables are initialized with null. We thus need to show that $\mathsf{dtype}(l_y, l_{x_i}) <: \mathsf{abs}(\mathsf{anno}(c', m', x_i))$ which follows directly from step 5 and the results in the previous paragraph.

The other proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. We need to consider the case $(\mathsf{anno}(c', m', x_i), l_{x_i}) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c', m', x_i) = \mathsf{Far}(n_2)$. By case assumption, we know that $\mathsf{anno}(c, m, z_i) = \mathsf{Far}(n_2)$ from which the claim follows directly.

Case R-RETURN:

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

We only consider the case where $\mathcal{D}(\mathsf{result}) \neq \mathsf{null}$. Let $l_y := \mathsf{loc}(\mathcal{H}(\mathcal{D}'(y)))$ and $l_{\mathsf{result}} := \mathsf{loc}(\mathcal{H}(\mathcal{D}(\mathsf{result})))$. We have

1. $\mathsf{anno}(c', m', \mathsf{result}) \triangleright_{\mathsf{From}} \mathsf{anno}(c, m, y) <: \mathsf{anno}(c, m, x)$ by typing rules WF-ASSIGN and WF-CALL,
2. $\mathsf{dtype}(l_{\mathsf{this}}, l_y) <: \mathsf{anno}(c, m, y)$ by well-formedness assumption of stack frame $\mathcal{F}$,
3. $\mathsf{dtype}(l_y, l_{\mathsf{result}}) <: \mathsf{anno}(c', m', \mathsf{result})$ by well-formedness assumption of stack frame $(\bullet, \mathcal{D})^{c',m'}$ and condition 1,
4. $\mathsf{dtype}(l_{\mathsf{this}}, l_{\mathsf{result}}) <: \mathsf{anno}(c, m, x)$ from steps 1, 2 and 3 similarly to the case R-FIELDGET by Lemma 1, Lemma 2, Lemma 3 and transitivity of subtyping, and
5. $l_{\mathsf{result}} = \mathsf{loc}(\mathcal{H}(\mathcal{D}''(x)))$ as $\mathcal{D}(\mathsf{result}) = \mathcal{D}''(x)$.

The first proof goal is to show that the stack frame $(\overline{S}, \mathcal{D}'')^{c,m}$ is well-formed. As only x is modified, we need to show that $\mathsf{dtype}(l_{\mathsf{this}}, \mathsf{loc}(\mathcal{H}(\mathcal{D}''(x)))) <: \mathsf{anno}(c, m, x)$ which follows directly from steps 4 and 5.

The second proof goal is to show that the new configuration ζ' is well-formed wrt. named Far locations. As only x is modified, we need to only consider the case $(\mathsf{anno}(c, m, x), l_{\mathsf{result}}) \in \mathsf{reach}(\zeta')$ where $\mathsf{anno}(c, m, x) = \mathsf{Far}(n)$.

By step 1, we then know that either

- $\mathsf{anno}(c, m, y) = \mathsf{Near}$ and $\mathsf{anno}(c', m', \mathsf{result}) = \mathsf{Far}(n)$ or
- $\mathsf{anno}(c, m, y) = \mathsf{Far}(n)$ and $\mathsf{anno}(c', m', \mathsf{result}) = \mathsf{Near}$.

In the first case, as $(\mathsf{anno}(c', m', \mathsf{result}), l_{\mathsf{result}}) \in \mathsf{reach}(\zeta)$ and ζ well-formed, the claim follows directly from step 5. In the second case, $l_y = l_{\mathsf{result}}$ and the claim then follows similarly to the first case.

Proof of Theorem 1:

Let P be a program such that $\vdash P$ and all location annotations in P are from $\{\mathsf{Near}, \mathsf{Far}, \mathsf{Somewhere}\}$. Thus, for all c, m, x, f,

$$\mathsf{abs}(\mathsf{anno}(c, m, x)) = \mathsf{anno}(c, m, x)$$
$$\mathsf{abs}(\mathsf{anno}(c, f)) = \mathsf{anno}(c, f)$$

Let ζ be a configuration for P that is well-formed according to Definition 1 and let ζ' be a successor configuration. According to Definition 6, there is no n, l with $(\mathsf{Far}(n), l) \in \mathsf{reach}(\zeta)$. Thus, ζ is also well-formed according to Definition 8. Theorem 2 guarantees that ζ' is well-formed according to Definition 8. Because of the two equations above, ζ' is also well-formed according to Definition 1. Thus, Theorem 1 holds.

B Soundness and Completeness of Type Inference

In the following, we present the proof of Theorem 3, i.e., we show that the constraint generation rules in Fig. 15 precisely collect the typing constraints defined by the typing rules in Fig. 10. P denotes programs that are fully annotated with pairwise distinct location type variables, i.e., all type occurrences in P are of the form $\alpha\, c$.

Soundness. We have to show: If $\sigma \vDash \overline{\mathcal{Q}}_P$, then $\vdash \sigma P$.

Let $\mathcal{T}_P^c$ be the derivation tree for P according to the constraint generation rules in Fig. 15. Each node of $\mathcal{T}_P^c$ corresponds to a rule application. Without loss of generality, we assume that each node is annotated with the conditions mentioned in the corresponding rule. Since $\sigma \vDash \overline{\mathcal{Q}}_P$, σ satisfies all constraints appearing in $\mathcal{T}_P^c$ and $\mathsf{locv}(P) \subseteq \mathsf{dom}(\sigma)$. To show $\vdash \sigma P$, we have to construct a derivation tree $\mathcal{T}_P$ for σP according to the typing rules in Fig. 10. Note that P uniquely determines the number of nodes and structure of $\mathcal{T}_P^c$ and of $\mathcal{T}_P$, if it exists. Consequently, there is a bijection between the nodes of the two derivation trees.

We inductively construct $\mathcal{T}_P$ starting with the leaf nodes using rules WF-NEWFRESH, WF-NEWFRESHP, WF-NEWSAME, WF-VAR, WF-FIELDGET, WF-FIELDSET, WF-CALL. The rule instances in $\mathcal{T}_P$ are constructed from the corresponding nodes in $\mathcal{T}_P^c$. We describe the construction here for the most interesting rules WF-NEWFRESH, WF-NEWFRESHP, and WF-CALL; for the other rules, the construction works analogously.

Case WF-NEWFRESH, WF-NEWFRESHP: Let N be a leaf node of $\mathcal{T}_P^c$ annotated by $\overline{V} \vdash \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} : \delta, \delta <: \gamma, \gamma = \mathsf{Far}$. As σ is a solution, $\sigma(\delta) <: \mathsf{Far}$.
For the construction of $\mathcal{T}_P$, we use $\sigma(\overline{V}) \vdash \mathsf{new}\ c\ \mathsf{in}\ \mathsf{fresh} : \sigma(\delta)$ as a leaf node corresponding to N where $\sigma(\overline{V})$ denotes the substitution of the location type variables in $\overline{V}$ by σ. If $\sigma(\delta) = \mathsf{Far}$, this is an instance of WF-NEWFRESH; otherwise, it is an instance of WF-NEWFRESHP.

Case WF-CALL Let N be a leaf node of $\mathcal{T}_P^c$ annotated by

$$\overline{V} \vdash y.m(\overline{z}) : \gamma, \beta \triangleright_{\mathsf{From}} \alpha = \gamma \cdot \overline{\mathcal{Q}_1} \cdot \ldots \cdot \overline{\mathcal{Q}_n}$$

For the construction of $\mathcal{T}_P$, we use $\sigma(\overline{V}) \vdash y.m(\overline{z}) : \mathsf{anno}(c, m, \mathsf{result}) \triangleright_{\mathsf{From}} \sigma(\alpha)$ as a leaf node corresponding to N. This is a correct instance of WF-CALL, because the assumptions of the constraint generation rule yield:

- $\alpha\ c\ y \in \overline{V}$ and $\alpha_i \_ z_i \in \overline{V}$; thus $\sigma(\alpha)\ c\ y \in \sigma(\overline{V})$ and $\sigma(\alpha_i) \_ z_i \in \sigma(\overline{V})$.
- $\overline{x} = \mathsf{params}(c, m)$
- $\alpha_i \triangleright. \alpha <: \mathsf{anno}(c, m, x_i)$ and, as σ is a solution, $\sigma(\alpha_i) \triangleright. \sigma(\alpha) <: \mathsf{anno}(c, m, x_i)$

Note that for the result type L of the expression in $\mathcal{T}_P$ in a leave node we have $L = \sigma(\gamma)$ where γ is the corresponding location type variable in $\mathcal{T}_P^c$.

Induction Step. Similar to the leaf nodes, we construct the inner nodes of $\mathcal{T}_P$ by instantiating the appropriate typing rule. The only interesting case is the assignment:

Case WF-ASSIGN: Let N be an inner node of $\mathcal{T}_P^c$ annotated by

$$\overline{V} \vdash x \leftarrow E : \beta <: \alpha \cdot \overline{\mathcal{Q}}$$

having a subtree annotated by $\overline{V} \vdash E : \beta, \overline{\mathcal{Q}}$. For the construction of $\mathcal{T}_P$, we use $\sigma(\overline{V}) \vdash x \leftarrow E$ as an inner node corresponding to N. This is a correct instance of WF-ASSIGN, because:

- there is a subtree in $\mathcal{T}_P$ annotated by $\sigma(\overline{V}) \vdash E : \sigma(\beta)$
- $\alpha \_ x \in \overline{V}$; thus $\sigma(\alpha) \_ x \in \sigma(\overline{V})$.
- $\sigma(\beta) <: \sigma(\alpha)$

Completeness. We have to show: If $\vdash \sigma P$ with $\mathsf{dom}(\sigma) = \mathsf{locv}(P)$, then there is a solution σ' of $\overline{\mathcal{Q}}_P$ such that $\sigma' \mid_{\mathsf{dom}(\sigma)} = \sigma$.

Let $\mathcal{T}_P$ be the derivation tree for σP according to the typing rules in Fig. 10 and $\mathcal{T}_P^c$ the derivation tree for P according to the constraint generation rules in Fig. 15. As P uniquely determines the number of nodes and structure of $\mathcal{T}_P^c$ and of $\mathcal{T}_P^c$ there is a bijection between the nodes of the two derivation trees.

σ defines location types for all location type variables occurring in P at variable, parameter, field, and result type declarations. Solutions of $\overline{\mathcal{Q}}_P$ have to be also defined for the fresh variables introduced by the application of the constraint generation rules CG-M, CG-NEWFRESH, CG-FIELDSET, CG-FIELDGET, CG-CALL. Fresh variables are different in different rule applications, i.e., at different tree nodes of $\mathcal{T}_P^c$. We define a solution σ' with $\sigma' \mid_{\mathsf{dom}(\sigma)} = \sigma$ as follows on the fresh variables introduced by the rule applications in $\mathcal{T}_P^c$:

CG-M: $\sigma'(\delta) = \mathsf{Near}$
CG-NEWFRESH: $\sigma'(\gamma) = \mathsf{Far}$ and $\sigma'(\delta) = L$ where L is the type used in the corresponding rule application of WF-NEWFRESH or WF-NEWFRESHP in $\mathcal{T}_P$.
CG-FIELDSET: $\sigma'(\delta) = \sigma(\gamma) \triangleright_{\mathsf{To}} \sigma(\alpha)$
CG-FIELDGET: $\sigma'(\gamma) = \sigma(\mathsf{anno}(c, f)) \triangleright_{\mathsf{From}} \sigma(\alpha)$
CG-CALL: $\sigma'(\gamma) = \sigma(\mathsf{anno}(c, m, \mathsf{result})) \triangleright_{\mathsf{From}} \sigma(\alpha)$ and $\sigma'(\gamma_i) = \sigma(\alpha_i) \triangleright_{\mathsf{To}} \sigma(\alpha)$

Just as demonstrated in the soundness proof above, one can check rule by rule that the assumptions satisfied in the application of the typing rules imply that

- the assumptions in the application of the constraint generation rules are satisfied and
- σ' satisfies the constraints.

That is, we have found a solution for $\overline{\mathcal{Q}}_P$.