ServiceJ: Service-Oriented Programming in Java

Sven De Labey *Marko van Dooren*
Eric Steegmans

Report CW451, June 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

ServiceJ: Service-Oriented Programming in Java

Sven De Labey *Marko van Dooren*
Eric Steegmans

Report CW 451, June 2006

Department of Computer Science, K.U.Leuven

Abstract

While object-oriented programming languages such as Java and C# deliver the main mechanism for implementing enterprise systems, these languages have not kept pace with the rapidly evolving technology of Service-Oriented Computing. The main reason is their insufficient support for dealing with *service volatility* and *service distribution*.

In this paper, we present ServiceJ, an extension of Java with built-in support for Service-Oriented Computing. ServiceJ bridges the gap between Service-Oriented Computing and Object-Oriented Programming Languages in two ways. First, it captures the volatile nature of services by supporting dynamic service selection and binding. Second, it adequately deals with distribution problems by offering a transparent failover mechanism that can be configured using declarative language constructs. We present a formal service language, Featherweight ServiceJ, in which the soundness of ServiceJ and its future extensions can easily be proven.

Keywords : Transparent Failover, Language Concepts, Service Oriented Computing, Dynamic Binding, Formal Model, Featherweight ServiceJ.

ServiceJ: Service Oriented Programming in Java

Sven De Labey, Marko van Dooren, and Eric Steegmans

K.U.Leuven, Dept. Computer Science, Celestijnenlaan 200A,
3001 Leuven, Belgium
{Sven.DeLabey, Marko.vanDooren, Eric.Steegmans}@cs.kuleuven.be

1 Introduction

The service-oriented paradigm is gaining massive industry support, mainly because it advocates interoperability and loose coupling between heterogeneous enterprise systems. Service-Oriented Architectures (SOAs) [1] introduce a common platform for cross-enterprise business operations. Consequently, they are able to reflect dynamic business requirements in a much better way than any other paradigm has done before.

Still, SOAs impose a number of challenges on the developers of service applications. These challenges originate from the *inherent volatility* and the *distributed nature* of remote services. Being volatile, service applications must cope with newly joining services, service migration, incompatible service updates and the removal of old services. Being distributed, applications must deal with availability problems resulting from network errors or server outages.

Based on object-oriented programming languages, platforms such as Java Enterprise Edition 5.0 [2] and .Net 3.0 [3], deliver the main platforms for implementing enterprise systems. But their current programming models are not suited for the implementation of SOA applications as they fail to provide language support for handling the distribution and volatility issues mentioned above.

In this paper, we present ServiceJ, an extension of Java with built-in support for Service-Oriented Computing. ServiceJ improves other object-oriented solutions in two ways. First, it deals with the volatile nature of services by supporting dynamic service selection and dynamic service binding. Second, it copes with distribution issues by providing transparent failover. Other properties, such as service composition and mediation, are outside the scope of this paper.

This paper is structured as follows. Section 2 discusses some important requirements for dependable service applications, and Section 3 shows how Java fails to deal with these requirements. Section 4 shows how the Service Path enables dynamic binding whereas Section 5 presents ServiceJ and its new language constructs. Both sections explain the new concepts from a conceptual, informal point of view. Section 6, on the other hand, presents Featherweight ServiceJ, the formal model of ServiceJ. We prove the type soundness of ServiceJ using this model and show that our new concepts increase the availability of a dependable SOA application. Section 7 shows how ServiceJ can be used in various classes of enterprise systems. Section 8 evaluates the new concepts of our language extension and compares it with Java. Finally, Section 9 discusses related work and Section 10 concludes.

2 Requirements for Dependable Service Applications

In this section, we discuss the main requirements of a programming model for SOA *clients*. By "client", we mean both *end clients* (applications that depend on services) and *intermediary clients* (service providers that depend on other providers to deliver their services).

An example of a SOA client is the Travel Agent (Fig. 1) that depends on an external flight booking service. Two competing providers, A and B, deliver a suitable service for booking flights. To increase availability, both providers have installed redundant services ($A_{1,2,3}$ and $B_{1,2}$) on various servers in different countries. Also, A and B periodically deploy new servers, creating additional endpoints. These are shown as dashed boxes (e.g. B_{new}).

This example is used to show how the design of a programming model for SOA applications is affected by *volatility* and *distribution*.

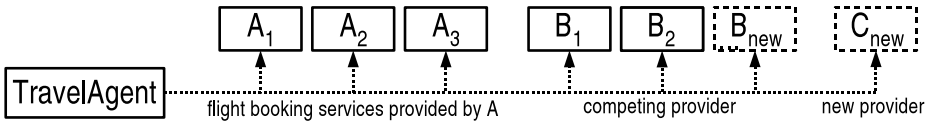


Fig. 1: The Travel Agent may choose between multiple service endpoints

Volatility. In a typical SOA, services are volatile for various reasons. Services may be migrated to other servers to improve load balancing (e.g. A_3 may move to a server in a different country), new services are published (B_{new}), and deprecated services disappear without warning their dependents – both end clients and intermediary clients. Also, another competing enterprise C may join the market delivering alternative flight booking services (C_{new}). In such a rapidly evolving environment, it is not manageable to hardwire service addresses in the source code of depending applications. Such a design would not reflect the inherent volatility of remote services. Therefore, a programming model for dependable SOA applications must advocate *dynamic service binding and discovery*.

Distribution. The ability of the Travel Agent to succesfully execute a business operation depends on the proper working of the network and on the reachability of its service providers. Consequently, a dependable SOA application must deal with (1) network problems and (2) provider failures. *Network problems* include router failures and network partitioning. *Provider failures* include server outages (e.g. A_1 crashing) and overloaded servers with unacceptable response times. Similar problems arise when one of the providers itself depends on an unreachable server. In that case, the failure of the service provider is *propagated* to all clients that invoke operations on one of the services of that provider.

An example of failure propagation is shown in Figure 2. Suppose the server on which service `(b)` resides, becomes unavailable or overloaded. Services `(c)` and `(d)` depend on `(b)`, so they fail to execute their business logic. This failure

propagates to `(e)` because `(e)` depends on `(c)`, which is unable to complete its task. But the same task could have been completed by using an alternative execution strategy. For example, `(c)` detecting that `(b)` is failing could route its requests to `(b-replicated)`. Similarly, `(e)` detecting that `(c)` is failing, could route its calls to `(c-replicated)`.

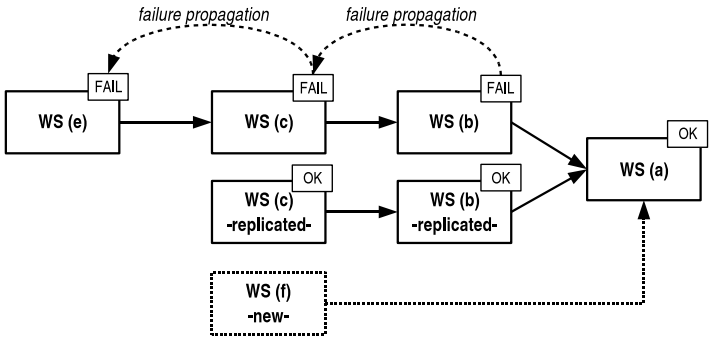


Fig. 2: Availability problems are propagated to dependent services

A programming model for SOA applications must cope with availability problems by providing *failover*. For example, the Travel Agent detecting that A_1 has crashed, should contact A_2, A_3, or even the competing service B_1. Moreover, the failover mechanism should be *transparent*. This means that an application programmer should not be forced to merge the implementation of the failover mechanism with the implementation of his business logic. Such algorithms should ideally be provided by the runtime system.

3 The Java Programming Model for Service Applications

The Java Enterprise Edition 5.0 programming model [2] provides *container injection* to initialize references to remote service endpoints. Programmers use annotations [4] to inform the container about the location of the WSDL [5] file. In the Travel Agent application, the following code triggers the container to inject a reference to the service located at A_1:

```
@WebServiceRef{wsdlLocation="http://www.A1.com/...wsdl",type=...}
FlightService myFlightService;         //container initializes reference
```

This approach improves the older versions of Java Enterprise Edition (J2EE 1.3 and J2EE 1.4). In those editions, application developers had to initialize these service references themselves. This was done using JNDI (Java Naming and Directory Interfaces) lookups and RMI narrowing operations. The problem with this approach was that application developers had to write a lot of boilerplate code in order to initialize a reference to a remote service. Therefore, the Service

Locator design pattern was introduced. Using this pattern, developers only had to pass the information specific to a particular service. The Service Locator then returned a reference to the proper endpoint. The major advantage of the `@webServiceRef` annotation is that the container *transparently* provides the functionality of a Service Locator.

While the container injection mechanism improves the older approach of JNDI lookups, there are still some major problems in the context of SOA applications. The main reason is that container injection fails to provide support for the requirements listed in Sect. 2:

- *Volatility issues.* Service addresses are hardwired in the source code of the application (`wsdlLocation`). Thus, manual reconfiguration is needed to handle service migration and removal. But this is not manageable because the annotations that require updates are scattered throughout the code. An alternative approach is to externalize service addresses in a deployment descriptor [2], but this requires (1) intensive manual reconfiguration of a verbose XML file and (2) redeployment of the service client. This is exactly why Java EE introduced a lightweight approach based on annotations.
- *Distribution issues.* Another disadvantage of hardwired addresses is that the client cannot support failover to a replicated endpoint. If a service provider crashes, the client must either wait until that provider recovers, or wait until someone manually binds the failing reference to another provider and then redeploys the client component. Both solutions are not manageable.
- *Reusability issues.* Reusing the client component in an application that cooperates with different providers requires intensive reconfiguration because every annotation in the source code (or each reference in the deployment descriptor) must be updated to point to the services of those new providers.

The main problem is that Java Enterprise Edition assumes that services reside at *static servers* with *zero downtime*. This is a very unrealistic assumption in a business environment where services are dynamically and automatically migrated to other services, for instance, to improve load balancing. Thus, annotations are a bad solution for SOA applcations because they force developers to define the service endpoints at compile-time, without support for changing them at runtime.

3.1 A Framework Cannot Solve the Problem.

To increase support for service volatility and service distribution, developers could build a framework with operations for contacting a *list of remote servers*, thus providing a means for dynamic binding and failover.

Static Service Definitions. The operations of this framework would require the same information about remote services as the `@WebServiceRef` annotation (for contacting the servers in the list (i.e the URL and the port of the service to be invoked). But these properties are *service-specific*, so they must be provided by the application developers. Hardcoding such properties in the client,

however, leads to static service definitions and reusability problems. Breaking a partnership with a service provider, for example, requires a modification of the list of service properties, leaving out those services that belong to the former business partner. A solution would be to externalize deployment information into a specific list of services. This solution is presented in Sect. 4.

Service Volatility. Another problem is that the framework is not dynamic enough to handle the *volatility* of services. If an enterprise deploys five different applications that depend on the same external service and that service is moved elsewhere, then all five applications break. The problem can be solved by updating the properties that are passed to the framework, but this implies that all five applications must be updated. The failure is not isolated.

Code Bloat. If a service cannot be reached, a `RemoteException` is thrown back to the client. Failover can be implemented by using the catch clause of this `RemoteException` to locate a different service and then reinvoke the operation on that new service. An example is given in the code below, assuming that "`failover`" points to a framework class containing operations to find alternative services from a statically defined list of providers.

```
try{
   provider.doSomething();            // business logic
}
catch(RemoteException ex){            // failover logic
   if(failover.hasNextServer()){
      this.setProvider(failover.getNextServer());
      provider.doSomething();
   }
}
```

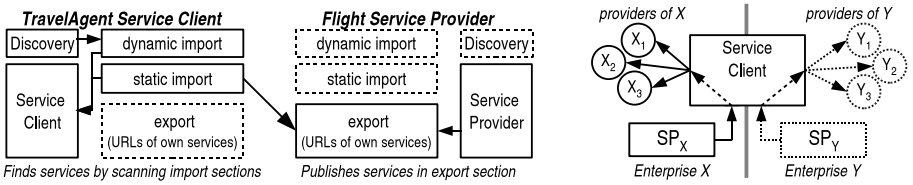
There are two problems with this approach. First, the framework does not achieve its primary objective, i.e. supporting *transparent* failover because developers have to write retry logic every time they want to invoke a remote service. Second, the implementation of the failover mechanism is mixed with the implementation of the business logic (`doSomething()`), seriously reducing the readability of the code.

Towards Agile Service Applications. We have developed an extension of Java that solves the problems mentioned above, as such evolving to the programming model defined in Sect. 2. This extension, called ServiceJ, is presented in the next two sections. Sect. 4 introduces the Service Path and shows how it is used to ease the configuration of dependable SOA applications. This solution is aimed at *deployers*. Sect. 5 shows how ServiceJ relies on that Service Path to provide transparent failover. This solution is targeted at *application developers*.

4 Service Path – Support for Dynamic Binding

Instead of relying on hardcoded information from `@WebServiceRef` annotations, ServiceJ scans through the Service Path (`SP`) environment to look for compatible services. This approach is comparable to the Java `CLASSPATH` environment that tells the class loader where to find third-party classes [6]. It also shares properties with the Domain Name System [7]. The `SP` has three sections (see Fig.3a):

- The *export section* is used by service providers to publish the addresses of the services they deliver. Clients may retrieve this list via a URL.
- Clients add URLs pointing to the export sections of their service providers to their *static import section*. ServiceJ visits these URLs in order to find compatible services. This section may also contain URLs pointing directly to WSDL files, although this approach is less robust in the face of volatility.
- The *dynamic import section* is used by external discovery engines to add URLs of newly discovered services and exported `SP`s.



(a) Using the Service Path to find service locations (b) Reusable service clients

Fig. 3: The Service Path environment

The `SP` approach improves the annotation mechanism in two ways:

- *Volatility.* A provider that needs to replicate, migrate, or remove one of its services only needs to update its export section. The import sections of the clients that depend on those services require no reconfiguration.
- *Reusability.* The externalization of service URLs increases the reusability of a client. Enterprise Y can reuse a client developed by X by providing it with its custom `SP` (Fig.3b). ServiceJ automatically finds the new providers by scanning the customized `SP` without a need for source code modification. Unlike a Deployment Descriptor, which requires a URL for each service reference in the client, the `SP` only requires one URL per business partner.

5 ServiceJ – Support for Transparent Failover

The principle of *Separation of Concerns* defines separate concerns for application *developers* and *deployers* [2]. The `SP` is typically used by the latter to configure service addresses. But application developers should concentrate on the business logic and must not be bothered with issues originating from service volatility and distribution. Therefore, ServiceJ provides a failover mechanism that transparently deals with these issues. Developers may customize this mechanism using high level language constructs such as *type qualifiers* and *declarative operations*.

5.1 Extending Types with Type Qualifiers

In Java EE 5.0, transparent failover is not supported because the container can only inject a single reference per annotated field. ServiceJ, on the other hand, offers transparent failover by scanning the `SP` to replace unreachable services with alternative services. In addition, developers are able to configure this failover mechanism in a declarative way by using *type qualifiers*, similar to the qualifiers of Javari [8]. ServiceJ provides two type qualifiers:

- The **pool** qualifier is used to indicate that all compatible services found in the `SP` are considered to be *equivalent*. For example, the Travel Agent would decorate a service reference with the **pool** qualifier to indicate that it does not differentiate between the services provided by A and those provided by B. A pool of flight booking services is declared as: **pool** `FlightService fsP`;
- The **sequence** qualifier is used to indicate that some services in the `SP` are *preferred above others*. Developers of the Travel Agent use a **sequence** qualifier to indicate that ServiceJ must first select, for instance, the cheapest flight provider. In case of failure, it may contact the second cheapest provider, etc. A sequence is declared as: **sequence** `FlightService fsSeq`;. Developers must provide the sorting attribute in an **orderby** clause (see Sect. 5.2).

The hierarchy of type qualifiers is shown in Fig.4. The **sequence** qualifier is a subtype of the **pool** qualifier because it induces an order on the members of a **pool**. Consequently, all operations that can be invoked on a **pool** can also be invoked on a **sequence** (see Sect. 5.2). Regular Java references are the most specific qualifiers in order to ensure compatibility with other Java applications.

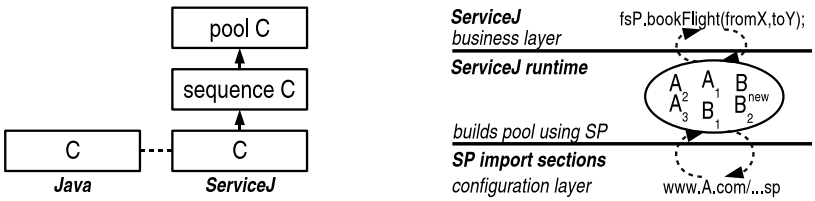


Fig. 4: ServiceJ extends the Java type system (left) and cooperates with the `SP` (right)

Initialization. ServiceJ initializes **pool** and **sequence** references by scanning the import sections of the `SP` (Fig. 4). These sections contain URLs pointing to WSDL files and exported `SP`s (Sect.4). If a URL points to the *WSDL file* of a compatible service, then that service is added to the pool. If it points to an *SP exported by a provider*, then that `SP` is scanned for compatible service URLs.

Transparent Failover. ServiceJ improves the injection mechanism with support for transparent failover. If an unreachable service is invoked, ServiceJ searches for another service and then reinvokes the operation on that new endpoint. The following code books a flight by calling a service found in the `SP` and transparently contacts another provider in case of a failure:

```
pool FlightService fsP;          // implicit initialization
fsP.bookFlight(fromX,ToY);       // transparent failover
```

5.2 Declarative Operations for Manipulating Pools and Sequences

The ServiceJ runtime environment is responsible for managing the contents of service pools based on information found in the `SP`. Often, however, developers need to impose additional restrictions on the contents of a pool. Therefore, we introduce two declarative operations: one for constraining pool membership (**where**), and one for sorting the members of a pool (**orderby**).

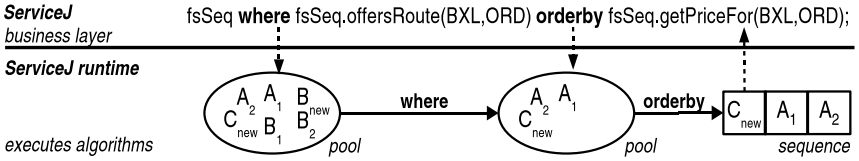


Fig. 5: Developers indicate where constraint handling (**where**) and sorting (**orderby**) is required. The runtime system is responsible for manipulating the pool contents

Constraints. Developers may need to impose membership conditions or QoS constraints on the contents of a service pool. The Travel Agent, for example, may wish to express that its reference to the booking service must invoke a service that offers a flight from airport `BXL` to airport `ORD`. Such constraints can be enforced using a **where** clause and a boolean condition (see also Fig. 5):

```
pool FlightService fsP where fsP.offersRoute(BXL,ORD);
```

Sorting. The **sequence** qualifier is used to denote a sorted list of services. ServiceJ transparently sorts the members of a pool, but the sorting attribute must be provided by application developers using an **orderby** clause. For example, TravelAgent developers may wish to sort the pool based on the price asked by the provider to reserve a seat (see also Fig. 5):

```
sequence FlightService fsSeq where fsSeq.offersRoute(BXL,ORD)
                             orderby fsSeq.getPriceFor(BXL,ORD);
```

6 Formalization of ServiceJ as a Java Extension

This Section presents Featherweight ServiceJ (FSJ): a formal language based on Featherweight Java (FJ) [9]. We discuss its syntax and subtyping, its expression typing, and its reduction rules. At the end of this Section, we use FSJ to prove the type soundness of the extensions that were developed in Section 5.

6.1 FSJ Syntax and Subtyping

Syntax. The FSJ syntax (see Fig. 6) is an extension of the FJ syntax. FSJ uses a class table CT that maps classes to their respective class definitions. For example, $CT(C)$ maps the class name C to its definition, which is typically written as **class** $C \triangleleft D\{\overline{T}\ \overline{f}, K\ \overline{M}\}$. Every class extends at most one other class

(inheritance is written as $C \triangleleft D$). A class contains a set of fields $\overline{f}$ (a shorthand for $f_1, f_2, \ldots, f_n$) with types $\overline{T}$, a constructor K and a suite of methods $\overline{M}$.

The required level of fault-tolerance is indicated by developers with a qualifier Q. This qualifier can be a **pool**, a **sequence** (see Sect.5) or a **singleton**. The **singleton** qualifier is used to denote regular Java references. While its use is implicit in ServiceJ, we chose to explicitize it in FSJ in order to clearly differentiate between fault-tolerant references (**pool** and **sequence**) and regular Java references (**singleton**).

A type T can be a **bool**[1] or a combination of a qualifier Q and a class name C. An expression e can be a variable (x), a field access ($e.f$), a method call ($e.m(\overline{e})$), a cast ($(T)e$), an object initialization (**new** $C(\overline{e})$), a field assignment ($set\ e.f = e\ then\ e$) or a local variable ($let\ x = e\ in\ e$). FSJ adds special expressions for developing dependable SOA applications: a function for constraining pool membership (e **where** e) and a sorting function (e **orderby** e).

Subtyping. The subtyping rules for FSJ are straightforward (see Fig. 6). The subtyping relation (denoted <:) is reflexive (S-Ref) and transitive (S-Trans). We add subtyping rules for qualifiers (S-Qual), leading to the hierarchy of qualifiers shown in Fig.4. A type S is a subtype of T if its qualifier Q_S and its class name C_S are subtypes of Q_T and C_T, respectively (S-Typ).

FSJ — Syntax

$CT(C)$	::=	**class** $C \triangleleft D\ \{\overline{T}\ \overline{f}; K\ \overline{M}\}$	C, D, E		class names
K	::=	$C(\overline{T}\ \overline{f})$ { **super**$(\overline{f})$; **this**.$\overline{f} = \overline{f}$; }	$Type$	::=	**bool** $\mid T$
M	::=	$T\ m(\overline{T}\ \overline{x})$ { **return** e; }	T	::=	$Q\ C$
e	::=	$x \mid e.f \mid e.m(\overline{e}) \mid (T)e \mid$ **new** $C(\overline{e})$	Q	::=	**pool**
	$\mid$	e **where** $e \mid e$ **orderby** e		$\mid$	**sequence**
	$\mid$	$set\ e.f = e\ then\ e \mid let\ x = e\ in\ e$		$\mid$	**singleton**

FSJ — Subtyping

$$\text{S-Def}\ \frac{CT(C) = \textbf{class}\ C \triangleleft D\ \{...;...\}}{C <: D} \qquad \text{S-Typ}\ \frac{Q <: Q' \qquad C <: C'}{Q\ C <: Q'\ C'}$$

$$\text{S-Qual}\ \frac{}{\begin{array}{c}\textbf{singleton} <: \textbf{sequence} \\ \textbf{sequence} <: \textbf{pool}\end{array}} \qquad \text{S-Ref}\ \frac{}{T <: T} \qquad \text{S-Trans}\ \frac{S <: T \qquad T <: U}{S <: U}$$

Fig. 6: Featherweight ServiceJ – Syntax and Subtyping

6.2 FSJ Field and Method Lookup

Field Lookup. There are no significant differences between the field lookup of FSJ and that of FJ. The first rule, Fields-Object, states that `Object` contains

[1] The primitive types of FSJ are limited to booleans to keep the formal model concise. The addition of other primitive types such as **int** and **double** is straightforward.

no fields. The second rule, FIELDS-C, states that the fields of a class C consist of those fields that C introduces ($\overline{T}\ \overline{f}$), along with all fields inherited from its direct and indirect superclasses ($\overline{U}\ \overline{g}$).

$$\textsc{Fd-Obj}\ \frac{}{fields(\textbf{Object}) = \bullet} \qquad \textsc{Fd-C}\ \frac{fields(D) = \overline{U}\ \overline{g} \qquad CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{T}\ \overline{f}; K\ \overline{M}\}}{fields(C) = \overline{U}\ \overline{g}, \overline{T}\ \overline{f}}$$

Fig. 7: FSJ – Field Lookup

Method Lookup. FSJ method lookup is closely related to the method lookup of FJ. The type of a method in class C is denoted as $mtype(m, C)$. The result is written as $\overline{T} \rightarrow U$, where $\overline{T}$ are the types of the arguments and U is the return type. The type of a method m offered by a class C with methods $\overline{M}$ can be found in C itself (MTYPE-C) if $m \in \overline{M}$, or in one of its superclasses (MTYPE-CSUB) if $m \notin \overline{M}$. Similarly, the body of a method offered by class C can be found in the class itself (MBODY-C) if $m \in \overline{M}$, or in one of its superclasses (MBODY-SUB) if $m \notin \overline{M}$. Thus, our extended type system does not interfere with the typical lookup strategy of Java. Method *invocation*, however, will contain severe adaptations to the invocation semantics of Featherweight Java because we provide runtime support for failover. These semantics are part of the reduction rules (see Sect. 6.5).

$$\textsc{MType-C}\ \frac{\textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{T}\ \overline{f}; K\ \overline{M}\} \qquad U\ m(\overline{T}\ \overline{x})\{\ \textbf{return}\ e;\ \}\ \in\ \overline{M}}{mtype(m, C) = \overline{T} \rightarrow U}$$

$$\textsc{MType-CSub}\ \frac{\textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{T}\ \overline{f}; K\ \overline{M}\} \qquad m \notin \overline{M}}{mtype(m, C) = mtype(m, D)}$$

$$\textsc{MBody-C}\ \frac{\textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{T}\ \overline{f}; K\ \overline{M}\} \qquad U\ m(\overline{T}\ \overline{x})\{\ \textbf{return}\ e;\ \}\ \in\ \overline{M}}{mbody(m, C) = \overline{x}.e}$$

$$\textsc{MBody-CSub}\ \frac{\textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{T}\ \overline{f}; K\ \overline{M}\} \qquad m \notin \overline{M}}{mbody(m, C) = mbody(m, D)}$$

Fig. 8: FSJ – Method Lookup

6.3 FSJ Expression Typing

Expression typing rules are used to determine the type of an expression. These rules are depicted in Fig. 9. FSJ uses a typing environment Γ and writes "$\Gamma \vdash e : T$" to state that e has type T in Γ. For example, we may write $\Gamma \vdash e.f_i : T_i$, which means that accessing field f_i of expression e gives an expression of type T_i. This notation is borrowed from Featherweight Java, and, in some sense, from ClassicJava [10].

- T-Var. The type of a variable can be found in the typing environment Γ. This rule is similar to the typing rule for variables in FJ.
- T-Get and T-Set. A service never directly exposes fields to its clients. Therefore, field inspection and assignment are allowed only on regular Java references ($Q =$ **singleton**). It is important to note, however, that the result of a field inspection can have *any* qualifier. Thus, it is allowed to inspect fields of type **pool** and **sequence** as long as the reference used to inspect those fields (e_0) has type **singleton**. It is interesting to note that field assignment is defined as an *expression* instead of a statement. After assigning e_2 to $e_1.f$, the expression moves on to e_3 (shown as *then* e_3). This is done to keep the formal model concise – an entire program is written as a single expression.
- T-Invk. Method invocation does not impose restrictions on the qualifier (Q) of the invocation target (e_0). Thus, contrary to field access and assignment, method invocation is allowed on **pool** and **sequence** references. ServiceJ uses different strategies for invoking operations on these reference, but this is shown in the reduction rules (see Sect. 6.5). T-Invk requires that all actual arguments passed to a method be conform to the formal arguments of the method signature ($\overline{S} <: \overline{U}$).
- T-New. Constructor invocation using the **new** operator does not consult the `SP`. Thus, the result is a reference to a single object (a **singleton**) without support for fault-tolerance. T-New is not used to initialize **pool** and **sequence** references. These are transparently initialized by ServiceJ. Further details about the initialization of **pool** and **sequence** references can be found in Sect. 6.5.
- T-Whr. The **where** operation constrains pool membership (e_1). It filters out those services that fail to satisfy the **bool** constraint (e_2). The selection algorithm is transparently executed by ServiceJ and is explained in Sect. 6.5.
- T-Ord. The **orderby** operation accepts an argument of type `Comparable` and changes the initial type (S) of an expression to *toSequence*(S) which converts a **pool** to a **sequence**. This is necessary because the result of **orderby** is always a *sorted* list. The sorting algorithm itself is provided by ServiceJ. Its execution strategy is discussed in Sect. 6.5.
- T-UCast and T-DCast. Qualifiers and class names can be upcast and downcast independently of eachother. T-SCast deals with casts to a type that is *unrelated* to the type of the expression (i.e. the cast is neither a downcast nor an upcast). In this case, FSJ signals a "*stupid warning*" [9].

FSJ – Expression Typing

$$\text{T-Var}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{T-Get}\ \frac{fields(C_0) = \overline{T}\ \overline{f} \quad \Gamma \vdash e_0 : \textbf{singleton}\ C_0}{\Gamma \vdash e_0.f_i\ :\ T_i} \qquad \text{T-Whr}\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1\ \textbf{where}\ e_2 : T}$$

$$\text{T-Set}\ \frac{fields(C) = \overline{T}\ \overline{f} \quad S <: T_i \quad \Gamma \vdash e_1 : \textbf{singleton}\ C \quad \Gamma \vdash e_2 : S \quad \Gamma \vdash e_3 : U}{\Gamma \vdash set\ e_1.f_i = e_2\ then\ e_3 : U} \qquad \text{T-New}\ \frac{fields(C) = \overline{U}\ \overline{f} \quad \Gamma \vdash \overline{e} : \overline{T} \quad \overline{T} <: \overline{U}}{\Gamma \vdash \textbf{new}\ C(\overline{e}) : \textbf{singleton}\ C}$$

$$\text{T-Invk}\ \frac{\Gamma \vdash \overline{e}\ :\ \overline{S} \quad \Gamma \vdash e_0\ :\ T_0 \quad \overline{S} <: \overline{U} \quad mtype(m, T_0) = \overline{U} \to T}{\Gamma \vdash e_0.m(\overline{e}) : T} \qquad \text{T-Let}\ \frac{\Gamma \vdash e_1 : T \quad \Gamma, x : T \vdash e_2 : U}{\Gamma \vdash let\ x = e_1\ in\ e_2 : U}$$

$$\text{T-UCast}\ \frac{Q <: Q' \quad \Gamma \vdash e_0 : Q\ C \quad C <: C'}{\Gamma \vdash (Q'\ C')e_0 : Q'\ C'} \qquad \text{T-DCast}\ \frac{Q' <: Q \quad \Gamma \vdash e_0 : Q\ C \quad C' <: C \quad Q \neq Q' \vee C \neq C'}{\Gamma \vdash (Q'\ C')e_0 : Q'\ C'}$$

$$\text{T-SCast}\ \frac{\Gamma \vdash e_0 : Q\ D \quad stupid\ warning \quad C \not<: D\ \wedge\ D \not<: C}{\Gamma \vdash (C)e_0 : Q\ C} \qquad \text{T-Ord}\ \frac{\Gamma \vdash e_1 : S \quad \Gamma \vdash e_2 : T \quad U = toSequence(S) \quad T <: \textbf{singleton}\ \texttt{Comparable}}{\Gamma \vdash e_1\ \textbf{orderby}\ e_2 : U}$$

Fig. 9: Featherweight ServiceJ – Expression Typing

6.4 Method and Class Typing

The rules for method and class typing are shown in Fig. 10. The rules of Featherweight ServiceJ closely follow those of Featherweight Java:

- T-Method deals with method overriding. A method override is valid in a subclass C (where $C <: D$) if the argument types ($\overline{T}$) and the return type (T_0) of that method are *exactly the same* as the respective types ($\overline{U} \to U_0$) of the definition of that method in the superclass D, that is: $\overline{T} = \overline{U}$ and $T_0 = U_0$. This rule is actually too strong, as the latest edition of Java (version 5.0) allows types of overriden methods to be strengthened. Thus, instead of requiring an *exact* match, we could also require the overriding method to have *conform types* (in that case, we require $\overline{T} <: \overline{U}$ and $T_0 <: U_0$ instead of $\overline{T} = \overline{U}$ and $T_0 = U_0$). However, this extension is orthogonal to the extensions introduced by ServiceJ.
- T-Class requires that every method in C ($m \in M$) is valid according to T-Method. It also requires that fields are initialized by the class that defined those fields. Thus, a call to the constructor in the parent class ($\textbf{super}(\overline{g})$) is needed to initialize inherited fields ($\overline{T}\ \overline{g}$), whereas newly introduced fields ($\overline{U}\ \overline{f}$) are initialized by the constructor of the class itself ($\textbf{this}.\overline{f} = \overline{f}$). Note that **pool** and **sequence** references are initialized implicitly by the runtime system (just like `@WebServiceRef` fields are initialized by the container), so $this.f = f$ should only be written by a programmer when f refers to a **singleton** reference.

$$\text{T-Method}\ \frac{\begin{array}{c}\overline{x} : \overline{T}, \textbf{this} : C \vdash e_0 : S_0 \qquad S_0 <: T_0 \\ \textbf{class}\ C\ \textbf{extends}\ D\ \{...\} \\ \text{if } mtype(m, D) = \overline{U} \to U_0, \text{ then } \overline{T} = \overline{U} \text{ and } T_0 = U_0\end{array}}{T_0\ m(\overline{T}\ \overline{x})\{\ \textbf{return}\ e_0;\ \}\ \text{ OK IN } C}$$

$$\text{T-Class}\ \frac{\begin{array}{c}fields(D) = \overline{U}\ \overline{g} \\ \overline{M}\ \text{OK IN}\ C \\ K = C(\overline{T}\ \overline{f},\ \overline{U}\ \overline{g})\{\textbf{super}(\overline{g});\ \textbf{this}.\overline{f} = \overline{f}\}\end{array}}{\textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{U}\ \overline{f};\ K\ \overline{M}\}\ \text{OK}}$$

Fig. 10: Featherweight ServiceJ – Method Typing and Class Typing

6.5 FSJ Basic Reduction Rules

The reduction rules of FSJ are more complex than those of FJ because we save the value of each field in a *store* $\mathcal{S}$. This is required in order to support field assignment, which does not exist in FJ. ClassicJava [10], however, introduces a store to save the type and value of fields. Our store is more complex, as we have extended the type system of Java.

Featherweight ServiceJ distinguishes between *basic reduction rules* and *advanced reduction rules*. Basic reduction rules consider everything except **where** and **orderby** expressions. These operations each have their own set of advanced reduction rules that implement the constraining algorithm and the sorting algorithm, respectively.

Featherweight ServiceJ Store. In Featherweight ServiceJ, reduction rules heavily depend on the store environment $\mathcal{S}$. It is important that the reader is somewhat familiar with the contents of this store in order to comprehend our discussion of the basic reduction rules.

- **Store ($\mathcal{S}$).** Similar to [10], we use a store $\mathcal{S}$ in order to support field assignment. A store entry $\sigma \in \mathcal{S}$ is a triple $\langle v, Q\ C, \mathcal{M}\rangle$ where v is a value with $Q\ C$ as its type and $\mathcal{M}$ as its *map*. Thus, we can use the store to retrieve the type and/or the map of a variable v. To do this, we introduce some extra functions:
  - $\mathcal{S}_T(v_i)$ is used to retrieve the type of v_i. This function is a projection from $\mathcal{S}(v) = \langle Q\ C, \mathcal{M}\rangle$ to $Q\ C$.
  - $\mathcal{S}_\mathcal{M}(v_i)$ is used to retrieve the mapping belonging to v_i. This function is a projection from $\mathcal{S}(v) = \langle Q\ C, \mathcal{M}\rangle$ to $\mathcal{M}$.

  Necessarily, we have $\mathcal{S}(v_i) \equiv \langle \mathcal{S}_T(v_i), \mathcal{S}_\mathcal{M}(v_i)\rangle$ for each $v_i \in \mathcal{S}$.
- **Mappings ($\mathcal{M}$).** The contents of the *map* $\mathcal{M}$ depend on the qualifier Q of a store entry $\sigma = \langle v, Q\ C, \mathcal{M}\rangle$:

- $Q = \textbf{pool} \vee Q = \textbf{sequence} \Rightarrow \mathcal{M} \equiv \mathcal{R}$
  For **pool** and **sequence** references, we use $\mathcal{M}$ to save all references to compatible service endpoints. Thus, every pool member is stored here and $\mathcal{M} \equiv \mathcal{R}$. ServiceJ implements transparent failover by searching $\mathcal{M}$ for an alternative service endpoint (see the definition of $failover(\mathcal{R})$). There is no need to define a mapping from fields to values, as **pool** and **sequence** references prohibit direct field access and assignment. When we are sure that a variable v_i has type **pool** or **sequence**, we will retrieve its mapping by writing $\mathcal{S}_{\mathcal{R}}(v_i)$ in stead of $\mathcal{S}_{\mathcal{M}}(v_i)$.
- $Q = \textbf{singleton} \Rightarrow \mathcal{M} \equiv \mathcal{F}$
  For **singleton** references, we use $\mathcal{M}$ to map fields to values. We use $\mathcal{F}$ to denote such field mappings ($\mathcal{M} \equiv \mathcal{F}$). For example, if $\sigma = \langle \textbf{singleton}\ C, \mathcal{F} \rangle$ and $fields(C) = \overline{T}\ \overline{f}$, then $\mathcal{F}$ contains a value v_i for each field f_i. This is written as $\mathcal{F}[f_i \rightarrow v_i]$. When we are sure that a variable v_i has type **singleton**, we will retrieve its mapping by writing $\mathcal{S}_{\mathcal{F}}(v_i)$ in stead of $\mathcal{S}_{\mathcal{M}}(v_i)$.

Featherweight ServiceJ Basic Reduction Rules. In this Section, we discuss the basic reduction rules. Section 6.6 and Section 6.7 deal with the advanced rules for executing the **where** and **orderby** operations. The basic reduction rules are shown in Fig. 11.

- **Field access and assignment.**
  - R-Fd. The expression typing rule for field access (T-Get) requires that the target expression is a **singleton** because field access is allowed only on **singleton** references. For the same reason, we assume that v is a **singleton** and thus $\mathcal{M} \equiv \mathcal{F}$. This field mapping $\mathcal{F}$ is then used to get the proper value v_i for the requested field f_i. This is shown as $\mathcal{F}(f_i) = v_i$. The store $\mathcal{S}$ is not modified during the evaluation of f_i.
  - R-Set. Similar to R-Fd, we require the type of the target reference (v_1) to be a **singleton** in order to succesfully assign a field. Again, we can use the field mapping $\mathcal{F}$ to overwrite the value of f_i with the assigned value v_i. This is shown as $\mathcal{F}[f_i \rightarrow v_i]$. The expression reduces to e_3, which again shows that field assignment is an *expression* rather than a statement. The store $\mathcal{S}$ is modified during reduction because f_i has received a new value v_i. When the transition is complete, the old value cannot be recalled.
- **Method Invocation.** The mapping $\mathcal{M}$ defines the main difference between method invocation on **singleton** references and method invocation on **pool** and **sequence** references:
  - R-InvkS. If a method is invoked on a **singleton** reference, then every occurence of the **this** reference in the method body ($\overline{x}.e_0$) is replaced by the invocation target v_0. This rewriting rule does *not* provide transparent failover because there exist no alternative invocations targets besides v_0. Indeed, the mapping function $\mathcal{M}$ of a **singleton** is a *field mapping* $\mathcal{F}$

rather than a collection of alternative service endpoints $\mathcal{R}$. Thus, R-InvkS models the classic evaluation of a Java method invocation.
- R-InvkP. If a method is invoked on a **pool** or a **sequence** reference, then the **this** reference is *not* overwritten with the invocation target v_0. Instead, a function $failover(\mathcal{S}(v_0))$ is called in order to retrieve an available service endpoint from the *reference mapping* $\mathcal{R}$ (note that $\mathcal{R}$ is retrieved from $\mathcal{S}(v_0) = \langle Q\ C, \mathcal{R}\rangle$). Thus, the **this** reference is replaced with an available service endpoint, as such supporting transparent failover.

– **The fail predicate.** A failure is denoted as $\langle fail, \mathcal{S}\rangle$. There are two cases in which such a fail predicate may occur. First, the state $\langle fail, \mathcal{S}\rangle$ occurs when $failover(\mathcal{R})$ fails to find a compatible service in the Service Path environment. This is shown in the formal definition of $failover(\mathcal{R})$. In that case, there is no invocation target for R-InvkP, so the program must fail. Second, a $fail$ predicate may occur if a **where** operation is executed such that all services are removed from $\mathcal{R}$ (i.e. the constraint formulated in the **where** clause is too strict). There is again no valid invocation target for R-InvkP, so the system must fail. The second case is explained in more detail in Sect. 6.6. The occurence of a $fail$ predicate will necessarily[2] lead to a program that gets stuck during evaluation. This is inherently supported by the model because no reduction rule matches with an expression that is equivalent with the $fail$ predicate.
– **Type casting.** The type of an expression ($\mathcal{S}_T(v)$) must be at least as specific as the type to which it is cast. Casting a variable v to a type that is unrelated to the type of v can never occur as there is no rule that matches with such request. Consequently, failed dynamic casts lead to stuck programs. Note that the store $\mathcal{S}$ is not changed after the cast has been applied. This is not different from the semantics of Java.
– **Local variables.** Local variables (i.e. expressions such as those in T-Let) are not added to the store. Instead, we immediately rewrite the expression in which they occur, substituting the local variable x with its value y. This is shown as $[y/x]e$ in $let\ x = y\ in\ e$.
– **Object Initialization.** During the initialization of an instance of class C with $fields(C) = \overline{T}\ \overline{f}$, each field f_i is set to a value v_i. R-New saves this field mapping in $\mathcal{F}$. This is shown as $\mathcal{F} = [\overline{f} \rightarrow \overline{v}]$. Also, the newly initialized object must be added to the store. This is done by adding a new element v_0 to $\mathcal{S}$. Following the typing rule for object initialization (T-New), v_0 has type **singleton** C. Thus, $\mathcal{S}(v_0) = \langle$**singleton** $C, \mathcal{F}\rangle$.

[2] There are cases in which the $fail$ predicate does not lead to an overall program failure. This situation may occur when the $fail$ predicate is sent to a method as an actual argument that is not used within the path of the method body taken by the program. For example, the $fail$ parameter may be assigned in the else-branch of a condition whereas the program follows the then-branch. Although substituted (see R-InvkS or R-InvkP), $fail$ will not occur in the resulting expression, so the program will not get stuck. A detailed discussion would require extending FSJ to support conditional branching, but this is outside the scope of this paper.

Definition 1 (Store). *A store $\mathcal{S}$ is a triple $\langle v, Q\ C, \mathcal{M}\rangle$ where:*
- *v is a variable.*
- *$Q\ C$ is the type of v where Q is a* type qualifier *and C is a* class name*.*
- *$\mathcal{M}$ is the mapping of v and:*
  - *If Q is a **singleton**, then $\mathcal{M}$ represents the mapping from $fields(C) = \overline{T}\ \overline{f}$ to values $\overline{v}$. This mapping is written as $\mathcal{F}[\overline{f} \rightarrow \overline{v}]$.*
  - *If Q is a **pool** or a **sequence**, then $\mathcal{M}$ contains the alternative pool members for v. This mapping is written as $\mathcal{R}$.*

⊡

Definition 2 (Failover). *Let $\mathcal{S}(v) = \langle Q\ C, \mathcal{R}\rangle$ and let $\mathcal{R}$ be the reference mapping with pool members v_i $(i : 1 \ldots n)$.*
The failover operation

$$failover : \langle Q\ C, \mathcal{R}\rangle \rightarrow v$$

is defined as follows:

$$failover(\langle Q\ C, \mathcal{R}\rangle) = \begin{cases} v_i & \exists v_i \in \mathcal{R} : avail(v_i) \wedge \forall j < i : \neg avail(v_j) \\ fail & otherwise \end{cases}$$

⊡

Lemma 1. (Sequence Creation Preserves Typing)
Given a type T where $T = Q\ C$,
Then $toSequence(T) = S$ for some $S = Q'\ C'$ where $Q' <: Q$ and $C = C'$.

Proof. Immediate from the definition of $toSequence(T)$ and from the syntax of Q which states that $Q = \textbf{pool} \vee Q = \textbf{sequence} \vee Q = \textbf{singleton}$.

Having $T = Q\ C$, there are three cases for some class C:

- $Q\ C \equiv \textbf{singleton}\ C$. Then $toSequence(\textbf{singleton}\ C) = \textbf{singleton}\ C$ and S-Refl stating that $\textbf{singleton}\ C <: \textbf{singleton}\ C$ finishes the case.
- $Q\ C \equiv \textbf{pool}\ C$. Then $toSequence(\textbf{pool}\ C) = \textbf{sequence}\ C$. We have $\textbf{singleton}\ C <: \textbf{pool}$ and $C = C$, thus finishing the case.
- $Q\ C \equiv \textbf{sequence}\ C$. Then $toSequence(\textbf{sequence}\ C) = \textbf{sequence}\ C$ and S-Refl stating that $\textbf{sequence}\ C <: \textbf{sequence}\ C$ finishes the case.

□

Definition 3 (Sequence creation). *Let T be a type with qualifier Q and class name C. The function*

$$toSequence : T \to T$$

is defined as follows:

$$toSequence(Q\ C) = \begin{cases} \textbf{\textit{sequence}}\ C & if\ Q = \textbf{\textit{pool}} \vee Q = \textbf{\textit{sequence}} \\ \textbf{\textit{singleton}}\ C & if\ Q = \textbf{\textit{singleton}} \end{cases}$$

⊡

FSJ — Reduction Rules

$$\text{R-Fd}\ \frac{\mathcal{S}(v_0) = \langle \textbf{singleton}\ C, \mathcal{F}\rangle \quad fields(C) = \overline{T}\ \overline{f} \quad \mathcal{F}(f_i) = v_i}{\langle v_0.f_i, \mathcal{S}\rangle \to \langle v_i, \mathcal{S}\rangle}$$

$$\text{R-Set}\ \frac{\mathcal{S}(v_0) = \langle \textbf{singleton}\ C, \mathcal{F}\rangle}{\langle set\ v_0.f_i = v_i\ then\ e_b, \mathcal{S}\rangle \to \langle e_b, \mathcal{S}[v_0 \to \langle \textbf{singleton}\ C, \mathcal{F}[f_i \to v_i]\rangle]\rangle}$$

$$\text{R-New}\ \frac{v_0 \notin dom(\mathcal{S}) \qquad fields(C) = \overline{T}\ \overline{f}}{\langle \textbf{new}\ C(\overline{v}), \mathcal{S}\rangle \to \langle v_0, \mathcal{S}[v_0 \to \langle \textbf{singleton}\ C, \mathcal{F}[\overline{f} \to \overline{v}]\rangle]\rangle}$$

$$\text{R-Cast}\ \frac{\mathcal{S}_T(v) <: U}{\langle (U)v, \mathcal{S}\rangle \to \langle v, \mathcal{S}\rangle}$$

$$\text{R-InvkP}\ \frac{\mathcal{S}(v_0) = \langle Q\ C, \mathcal{R}\rangle \qquad mbody(m, C) = \overline{x}.e_0}{\langle v_0.m(\overline{v}), \mathcal{S}\rangle \to \langle\ [\overline{v}/\overline{x}, failover(\mathcal{S}(v_0))/\textbf{this}]e_0, \mathcal{S}\rangle}$$

$$\text{R-InvkS}\ \frac{\mathcal{S}(v_0) = \langle Q\ C, \mathcal{F}\rangle \quad mbody(m, C) = \overline{x}.e_0}{\langle v_0.m(\overline{v}), \mathcal{S}\rangle \to \langle [\overline{v}/\overline{x}, v_0/\textbf{this}]e_0, \mathcal{S}\rangle}$$

$$\text{R-Let}\ \frac{}{\langle let\ x = y\ in\ e, \mathcal{S}\rangle \to \langle [y/x]e, \mathcal{S}\rangle}$$

$$\text{R-Op}\ \frac{v_0 \notin dom(\mathcal{S})}{\begin{array}{l}\langle v_1\ \textbf{orderby}\ e_c, \mathcal{S}\rangle \to \langle v_0, \mathcal{S}[v_0 \to \langle toSequence(\mathcal{S}(v_1)), sortBy(\mathcal{S}_{\mathcal{M}}(v_1), e_c)\rangle]\rangle \\ \langle v_1\ \textbf{where}\ e_b, \mathcal{S}\rangle \to \langle v_1, \mathcal{S}[v_1 \to constrainBy(\mathcal{S}(v_1), e_b)]\rangle\end{array}}$$

Fig. 11: Featherweight ServiceJ – reduction rules

6.6 Advanced Reduction Rules for Constraint Handling

Before we discuss the reduction rules of the **where** operation, we first provide a high level algorithm in pseudo code. Contrary to the reduction rules, this algorithm does not take into account the side effects that may occur when reducing the bool constraint (e_b). This is not a problem, however, as the algorithm is used to *illustrate* rather than implement the constraint handling process.

Algortihm 1. Constraining pools and sequences with the **where** clause

WHERE (v, $\mathcal{R}$, e_b)
*# v: the variable on which the **where** operation is invoked*
 # $\mathcal{R}$: the reference mapping that must be constrained
 # e_b: the constraint

```
1   R_acc ← ∅;
2   e_backup ← e_b;
3   while R ≠ ∅ do
4       fix w ∈ R;                          # take any element from R
5       if [w/v]e_b →* true then            # expression reduced to true
6           R_acc ← R_acc ∪ {w};            # so add w to the accumulator
7           R ← R \ {w};                    # and remove w from R
8           e_b ← e_backup;                 # and proceed with next reference from R
9       else                                # OR expression reduced to false
10          R ← R \ {w};                    # so remove w from R
11          e_b ← e_backup;                 # and proceed with next reference from R
12      end if
13  end while                               # repeat until every pool entry is checked
14  return R_acc;                           # return references that succeeded the test
```

- R-Wh-Start triggers the execution of the algorithm. This expression is expanded with an empty stack ($\emptyset$) and a backup expression e_b.
  - The stack will be used as an *accumulator* ($\mathcal{R}_{acc}$) containing those references of $\mathcal{R}$ that comply with the boolean condition e_b. The accumulator will be filled during execution, and it will serve as the new reference mapping for v when its current reference mapping has been filtered (the old reference mapping will be discarded).
  - The backup expression (e_b) is needed because e_b must be reduced for *each element* of $\mathcal{R}$. Thus, after the first expression has reduced to **true** ($[w/v]e_b \rightarrow$ **true**) or **false** ($[w/v]e_b \rightarrow$ **false**), we loose e_b. But we still need this expression because all other elements of $\mathcal{R}$ need to be reduced with e_b. Therefore, we keep a backup version that can be used whenever another element of $\mathcal{R}$ must be reduced. This is shown in R-Where-True and R-Where-False. In the pseude code above, this is shown as $e_b \leftarrow e_{backup}$ (lines 8 and 11).

- R-Wh-Stop. The algorithm stops if there are no remaining references in $\mathcal{R}$. This means that every pool member has been tested for pool membership with e_b, and that $\mathcal{R}_{acc}$ contains only those references that comply with that boolean constraint. We can now reduce the entire **where** expression by replacing $\mathcal{R}$ with $\mathcal{R}_{acc}$. This means that the reference mapping ($\mathcal{R}$) of v now only contains those reference that comply with e_b. In the pseude code above, this is shown as **return** $\mathcal{R}_{acc}$ (line 14).
- R-Wh-True. If an expression e_b reduces to **true** for a particular pool member w, then that pool member is added to the accumulator (denoted as $\mathcal{R}_{acc} \cup \{w\}$). The reduced expression (**true**) is replaced by the original expression e_{tmp} in order to start a new iteration with a new pool member of $\mathcal{R}$. The implementation of this step is shown in lines 5 – 8 in the pseudo code. Note however, that side effects are dealt with only by the reduction rules.
- R-Wh-False. If an expression e_b reduces to **false** for a particular pool member w, then that pool member is *not* added to the accumulator because it failed to comply with the constraints imposed on the pool. The reduced expression (**false**) is replaced by the original expression e_{tmp} in order to start a new iteration with a new pool member of $\mathcal{R}$. This step is implemented in lines 9 – 11 of the pseudo code.
- R-Wh-Iter represents the iteration step itself. It selects a pool member w from the collection of pool members $\mathcal{R}$ and replaces each occurence of v in the boolean condition e_b with w. This is shown as $[w/v]e_b$. This is similar to method invocation, where w is the actual argment for a formal parameter v. Remember that v represents the original reference on which the **where** operation is invoked. In the pseudo code, this is shown in lines 3 – 4.

$$\textsc{R-Wh-Start}\ \frac{}{\langle v_1\ \textbf{where}\ e_b, \mathcal{S}\rangle \rightarrow \langle v_1\ \textbf{where}\ e_b, \mathcal{S}, \emptyset, e_b\rangle}$$

$$\textsc{R-Wh-Iter}\ \frac{\mathcal{R} \neq \emptyset}{\begin{array}{l}\langle v\ \textbf{where}\ e_{tmp}, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R}\rangle], \mathcal{R}_{acc}, e_{tmp}\rangle \rightarrow \\ \langle v\ \textbf{where}\ [w/v]e_{tmp}, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R} \setminus \{w\}\rangle], \mathcal{R}_{acc}, e_{tmp}, w\rangle\end{array}}$$

$$\textsc{R-Wh-Stop}\ \frac{\mathcal{R} = \emptyset}{\begin{array}{l}\langle v\ \textbf{where}\ e_{tmp}, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R}\rangle], \mathcal{R}_{acc}, e_{temp}\rangle \rightarrow \\ \langle v, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R}_{acc}\rangle]\rangle\end{array}}$$

$$\textsc{R-Wh-True}\ \frac{}{\begin{array}{l}\langle v\ \textbf{where}\ true, \mathcal{S}, \mathcal{R}_{acc}, e_{tmp}, w\rangle \rightarrow \\ \langle v\ \textbf{where}\ e_{tmp}, \mathcal{S}, \mathcal{R}_{acc} \cup \{w\}, e_{tmp}\rangle\end{array}}$$

$$\textsc{R-Wh-False}\ \frac{}{\begin{array}{l}\langle v\ \textbf{where}\ false, \mathcal{S}, \mathcal{R}_{acc}, e_{tmp}, w\rangle \rightarrow \\ \langle v\ \textbf{where}\ e_{tmp}, \mathcal{S}, \mathcal{R}_{acc}, e_{tmp}\rangle\end{array}}$$

Fig. 12: Featherweight ServiceJ – Advanced Reduction Rules for **where**

6.7 Advanced Reduction Rules for Sorting

The advanced reduction rules for sorting **pool** and **sequence** references are shown in Fig. 13. We have developed the reduction rules in case the sorting expression (e_c) reduces to an **int** or a **double**. This algorithm can be extended to work with `Comparable` expressions, but this requires additional reduction rules to deal with possible side effects in the `compareTo(Comparable,Comparable)` method. Such side effects cannot occur with integer and double expressions because the invocation of `compareTo` can be replaced with the built-in $<$ operator, which is free of any side effects. The semantics of this operator are equal to the semantics of the $<$ operator for real numbers $(\mathbb{R}, <)$.

Algorithm. For illustrative purposes, we have included a highlevel algorithm which implements the sorting function. This algorithm assumes that the reduction of $[w/v]e_c$ has no side effects (line 7). However, this assumption is dropped in the reduction rules. Sorting is done in two passes:

1. For each pool member $w \in \mathcal{R}$: reduce $[w/v]e_c$ to an **int** or a **double**. This result is denoted as α. Then, add the tuple $\langle \alpha, w \rangle$ to an unsorted list $\mathcal{L}_{unsorted}$. At the end of the first pass, we have added a tuple $\langle \alpha, w \rangle$ to $\mathcal{L}_{unsorted}$ for each $w \in \mathcal{R}$ where α is an encoding of the *score* for each pool member.
2. Sort the tuples of $\mathcal{L}_{unsorted}$ according to their α-values. Thus, the pool member with the best score is assigned to the head of the list. Assign this sorted list to $\mathcal{L}_{sorted}$ and return it to the caller.

Algortihm 2. Sorting pools and sequences with the **orderby** clause

ORDERBY $(v,\ \mathcal{R},\ e_c)$
*# v: the variable on which the **orderby** operation is invoked*
 # $\mathcal{R}$: the reference mapping that must be sorted
 # e_c: the sorting expression

1	$\alpha \leftarrow 0$;	
2	$\mathcal{L}_{unsorted} \leftarrow [\,]$;	
3	$\mathcal{L}_{sorted} \leftarrow [\,]$	
4	$e_{backup} \leftarrow e_c$;	
5	**while** $\mathcal{R} \neq \emptyset$ **do**	
6	fix $w \in \mathcal{R}$;	*# take any element from $\mathcal{R}$*
7	$\alpha \xleftarrow{*} [w/v]e_c$	*# reduce expression to α*
8	$\mathcal{L}_{unsorted} \leftarrow \mathcal{L}_{unsorted} \cup \langle \alpha, w \rangle$;	*# add $\langle \alpha, w \rangle$ to the accumulator*
9	$\mathcal{R} \leftarrow \mathcal{R} \setminus \{w\}$;	*# and remove w from $\mathcal{R}$*
10	$e_c \leftarrow e_{backup}$;	*# and restore the sorting expression*
11	**end while**	
12	$\mathcal{L}_{sorted} \leftarrow quicksort(\mathcal{L}_{unsorted})$	*# sort according to α-values*
13	**return** $\mathcal{L}_{sorted}$;	*# return sorted list*

Reduction rules for orderby. The advanced reduction rules for **orderby** clauses are shown in Fig. 13. We discuss each rule and refer to the pseudo code algorithm where possible. It is important to note that these reduction rules do not implement a sorting function themselves. The sorting function is implemented by $sort(\mathcal{L})$. We have externalized this algortihm from the reduction rules because it relies on the $<$ operator and this operator introduces no side effects. An extension for sorting `Comparable` expressions, however, would rely on a `compareTo` operation to implement the sorting function. This would require additional reductions with possible side effects. Thus, such an extension would require a hardcoded sorting algorithm inside the reduction rules. This extension is not shown here.

- R-Ord-Start is the entry point of the algorithm. It introduces an empty list that will be used to add tuples $\langle \alpha, w \rangle$ containing a pool member w and the reduced sorting expression for that member (the α-value). In the pseudo code, this is shown on lines 1 through 4.
- R-Ord-Stop contains the stop condition. The algorithm ends if the sorting expression has been reduced for each member of $\mathcal{R}$. In that case, $\mathcal{R} = \emptyset$. The sorting function $sort(\mathcal{L})$ is applied on the collection of $\langle \alpha, w \rangle$-tuples, resulting in a sorted list of pool members ($\mathcal{L}_{sorted}$ in the pseudo code).
- R-Ord-Iter represents a normal iteration step ($\mathcal{R} \neq \emptyset$). It extracts an element w from $\mathcal{R}$ and rewrites the sorting expression e_c with this pool member ($[w/v]e_c$). After this expression has reduced, however, we loose the original information about e_c and w. Therefore, we replicate both expressions and reduce only the original version (shown at the end of the reduction rule). The replicated versions are used to restore the expression in R-Ord-Int. The iteration step is shown on lines 5 through 7 in the pseudo code.
- R-Ord-Int. If the sorting expression has reduced for some pool member w, then the result is an **int** or a **double**. This value is denoted as α. R-Ord-Int saves a pool member together with its α-value in an unsorted list of tuples (denoted aq $\mathcal{L}$). This step is shown on line 8 in the pseudo code.

$$\text{R-Ord-Start}\ \frac{}{\langle v\ \textbf{orderby}\ e_c, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R}\rangle]\rangle \rightarrow \langle\ v\ \textbf{orderby}\ e_c, \mathcal{S}, \mathcal{R}_{tmp}, [\,]\rangle}$$

$$\text{R-Ord-Stop}\ \frac{\mathcal{R}_{tmp} = \emptyset \qquad v_0 \notin dom(\mathcal{S}) \qquad \mathcal{S}(v) = \langle Q\ C, \mathcal{R}\rangle}{\langle v\ \textbf{orderby}\ e_c, \mathcal{S}, \mathcal{R}_{tmp}, \mathcal{L}\rangle \rightarrow \langle v_0, \mathcal{S}[v_0 \rightarrow \langle toSequence(Q\ C), sort(\mathcal{L})\rangle]\rangle}$$

$$\text{R-Ord-Iter}\ \frac{\mathcal{R}_{tmp} \neq \emptyset}{\begin{array}{l}\langle v\ \textbf{orderby}\ e_c, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R}\rangle], \mathcal{L}\rangle \rightarrow \\ \langle v\ \textbf{orderby}\ [w/v]e_c, \mathcal{S}[v \rightarrow \langle Q\ C, \mathcal{R}\rangle], \mathcal{R}_{tmp} \setminus \{w\}, \mathcal{L}, w, e_c\rangle\end{array}}$$

$$\text{R-Ord-Int}\ \frac{}{\langle v\ \textbf{orderby}\ \alpha, \mathcal{S}, \mathcal{L}, x, e_c\rangle \rightarrow \langle v\ \textbf{orderby}\ e_c, \mathcal{S}, \mathcal{L} \cup \langle \alpha, x\rangle\rangle}$$

Fig. 13: Featherweight ServiceJ – Advanced Reduction Rules for **orderby**

6.8 Congruence Rules

The congruence rules are rather straightforward (see Fig. 14). Some expressions need multiple congruence rules. For example, method invocation consists of an expression representing the invocation target (e_0) and expressions representing the arguments of that operation ($\overline{e}$). Therefore, we need two congruence rules. RC-Invk-Recv deals with the invocation target (e_0 reducing to e_0'). RC-Invk-Arg deals with one of the actual parameters (e_i reducing to e_i'). A similar approach is needed to represent the congruence rules of the **where** operation (RC-Where-Arg and RC-Where-Recv) and the **orderby** operation (RC-Order-Arg and RC-Order-Recv).

$$\textsc{RC-New}\ \frac{\langle e_i, \mathcal{S}\rangle \rightarrow \langle e_i', \mathcal{S}'\rangle}{\langle \textbf{new}\ C(\ldots, e_i, \ldots), \mathcal{S}\rangle \rightarrow \langle \textbf{new}\ C(\ldots, e_i', \ldots), \mathcal{S}'\rangle} \qquad \textsc{RC-Let}\ \frac{\langle e_i, \mathcal{S}\rangle \rightarrow \langle e_i', \mathcal{S}'\rangle}{\langle let\ x = e_i\ in\ e, \mathcal{S}\rangle \rightarrow \langle let\ x = e_i'\ in\ e, \mathcal{S}'\rangle}$$

$$\textsc{RC-Fd}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle e_0.f, \mathcal{S}\rangle \rightarrow \langle e_0'.f, \mathcal{S}'\rangle} \qquad \textsc{RC-Invk-Arg}\ \frac{\langle e_i, \mathcal{S}\rangle \rightarrow \langle e_i', \mathcal{S}'\rangle}{\langle e_0.m(\ldots, e_i, \ldots), \mathcal{S}\rangle \rightarrow \langle e_0.m(\ldots, e_i', \ldots), \mathcal{S}'\rangle}$$

$$\textsc{R-Invk-Recv}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle e_0.m(\overline{e}), \mathcal{S}\rangle \rightarrow \langle e_0'.m(\overline{e}), \mathcal{S}'\rangle} \qquad \textsc{RC-Cast}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle (Q\ C)e_0, \mathcal{S}\rangle \rightarrow \langle (Q\ C)e_0', \mathcal{S}'\rangle}$$

$$\textsc{RC-Where-Arg}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle e\ \textbf{where}\ e_0, \mathcal{S}\rangle \rightarrow \langle e\ \textbf{where}\ e_0', \mathcal{S}'\rangle} \qquad \textsc{RC-Where-Recv}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle e_0\ \textbf{where}\ e, \mathcal{S}\rangle \rightarrow \langle e_0'\ \textbf{where}\ e, \mathcal{S}'\rangle}$$

$$\textsc{RC-Order-Arg}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle e\ \textbf{orderby}\ e_0, \mathcal{S}\rangle \rightarrow \langle e\ \textbf{orderby}\ e_0', \mathcal{S}'\rangle} \qquad \textsc{RC-Order-Recv}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle e_0\ \textbf{orderby}\ e, \mathcal{S}\rangle \rightarrow \langle e_0'\ \textbf{orderby}\ e, \mathcal{S}'\rangle}$$

$$\textsc{RC-Set-L}\ \frac{\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle}{\langle set\ e_0.f\ =\ e_v\ then\ e_b, \mathcal{S}\rangle \rightarrow \langle set\ e_0'.f\ =\ e_v\ then\ e_b, \mathcal{S}'\rangle} \qquad \textsc{RC-Set-R}\ \frac{\langle e_v, \mathcal{S}\rangle \rightarrow \langle e_v', \mathcal{S}'\rangle}{\langle set\ v.f\ =\ e_v\ then\ e_b, \mathcal{S}\rangle \rightarrow \langle set\ v.f\ =\ e_v'\ then\ e_b, \mathcal{S}'\rangle}$$

Fig. 14: FSJ – Congruence Rules

6.9 Store Typing

We already pointed out that we need a store $\mathcal{S}$ to save types T, field mappings $\mathcal{F}$ and reference mappings $\mathcal{R}$. Thus, we need to define extra rules to ensure that the store $\mathcal{S}$ is consistent with the typing environment Γ. These rules are shown in Fig. 15

- T-StoreTyp states that the store type ($\mathcal{S}_T(x)$) of a variable x must be conform to the type of that variable in Γ. Formally, $\Gamma \vdash \mathcal{S}_T(x) <: \Gamma(x)$. If this statement is true for every variable x in the typing environment ($\forall x \in \Gamma$), then the typing of the store $\mathcal{S}$ is conform to the typing environment Γ.
- T-FMap is used for the field mappings of **singleton** references. It is easy to show that this rule is never used for **pool** and **sequence** references, as direct field assignment and field inspection are prohibited for these qualifiers (this is enforced by the expression typing rules in Sect. 6.3). The rule requires that the type of the value ($\mathcal{S}_T(\mathcal{F}(f_i))$) of every field f_i of a class C must be conform to the declared type T_i of that field. More concise, if a field f has type T, then only values of type $S <: T$ are legal values for f. This rule is equivalent to the assignment rules of Java.
- T-RMap imposes similar conditions on **pool** and **sequence** references as T-FMap does on **singleton** references. It states that a reference mapping ($\mathcal{R}$) is valid only if every pool member $v \in \mathcal{R}$ has type **singleton** C. The class name C is needed for reasons of type conformance, whereas the **singleton** qualifier is needed to ensure that no nested pools are created. For example, if we dropped this requirement, it would be possible to add a **pool** reference to a **pool**. This would lead to a hierarchy of pools for one reference, which needlessly complicates the semantics of the **pool** and **sequence** qualifiers.
- T-Store imposes three requirements on the contents of a store. First, the store typing $\vdash_T \mathcal{S}$ must be consistent with the typing environment Γ. Second, the field mapping of all **singleton** references must be legal ($\mathcal{S}_\mathcal{F}(x)\ OK$). This is enforced by T-FMap. Third, all **pool** and **sequence** references must have legal reference mappings ($\mathcal{S}_\mathcal{R}(x)\ OK$). This is enforced by T-RMap.
- T-State requires that the store type of every expression e is consistent with the type of that expression in Γ. Moreover, it requires that all previous rules hold for that state (i.e: $\vdash \mathcal{S} : \Gamma$). This rule is very important for the proof of the type soundness theorem as we will subsequently prove the consistency of $\mathcal{S}$ after each evaluation step.

Theorem 1 (Subject Reduction). *Given* $\vdash \langle e_0, \mathcal{S}\rangle : T$.
If $\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle$, *then* $\vdash \langle e_0', \mathcal{S}'\rangle : U$ *for some* $U <: T$.

Proof. See Appendix. □

Theorem 2 (Progress). *If* $\vdash \langle e_0, \mathcal{S}\rangle : T$, *then:*
- e_0 *is a variable* x, *or*
- $\langle e_0, \mathcal{S}\rangle \longrightarrow \langle e_0', \mathcal{S}'\rangle$ *for some* $\mathcal{S}'$ *and* e', *or*
- $\langle e_0, \mathcal{S}\rangle$ *is stuck due to the occurence of the* $fail$ *predicate, or*
- $\langle e_0, \mathcal{S}\rangle$ *is stuck in a failed dynamic downcast.*

Proof. See Appendix. □

$$\text{T-StoreTyp}\ \frac{\Gamma \vdash \mathcal{S}_T(x) : T \quad T <: \Gamma(x) \quad \forall x \in dom(\Gamma)}{\vdash_T \mathcal{S} : \Gamma} \qquad \text{T-RMap}\ \frac{\mathcal{S}(v) = \langle Q\ C, \mathcal{R} \rangle \quad \forall v \in \mathcal{R} : \mathcal{S}_T(v) <: \textbf{singleton}\ C}{\mathcal{S}_{\mathcal{M}}(v)\ \text{OK}}$$

$$\text{T-FMap}\ \frac{\mathcal{S}(v) = \langle Q\ C, \mathcal{F} \rangle \qquad fields(C) = \overline{T}\ \overline{f} \quad \forall f_i \in fields(C) : \mathcal{S}_T(\mathcal{F}(f_i)) <: T_i}{\mathcal{S}_{\mathcal{M}}(v)\ \text{OK}} \qquad \text{T-State}\ \frac{\vdash \mathcal{S} : \Gamma \qquad \Gamma \vdash e : T}{\vdash \langle e, \mathcal{S} \rangle : T}$$

$$\text{T-Store}\ \frac{\vdash_T \mathcal{S} : \Gamma \qquad \forall x \in dom(\Gamma) : \mathcal{S}_{\mathcal{M}}(x)\ \text{OK}}{\vdash S : \Gamma}$$

Fig. 15: FSJ – Store Typing

Theorem 3 (Type Soundness). *Given* $\vdash e : T$*, then either*

- $\langle e, \emptyset \rangle \xrightarrow{*} \langle x, \mathcal{S} \rangle$ *for some* $\mathcal{S}$ *and* x
- $\langle e, \emptyset \rangle \xrightarrow{*} \langle e', \mathcal{S} \rangle$ *where* $\langle e', \mathcal{S} \rangle$ *is stuck due to a failed dynamic downcast or due to the occurence of a* $fail$ *predicate.*
- $\langle e, \emptyset \rangle$ *executes forever.*

Proof. See Appendix. □

Theorem 4 (Availability). *If a method call on a* ***singleton*** $S(v_s) = \langle Q\ C, \mathcal{F} \rangle$ *succeeds, then a method call on a* ***pool/sequence*** $S(v_p) = \langle Q\ C, \mathcal{R} \rangle$ *with* $v_s \in \mathcal{R}$ *will also succeed.*

Proof. See Appendix. □

7 Applications

This Section shows some applications of ServiceJ. The language is useful for ensuring dynamic binding and fault tolerance in the *internal* SOA of an enterprise (Sect. 7.1), for dealing with cooperative partners (Sect. 7.2), and for outsourcing a part of an internal enterprise SOA (Sect. 7.3). We also consider an application that uses pools and sequences in local Java applications, although this part of ServiceJ is subject to future research (Sect. 7.4).

7.1 Internal Enterprise SOA

A part of an internal enterprise SOA is shown in Fig.16, which depicts a replicated database server containing customer data. The primary server, A_1, is maintained by the enterprise itself. This server delivers the fastest processing time and its data is guaranteed to be consistent. A second server, A_2, has a much higher response time and a lower throughput. Moreover, its data is not guaranteed to be consistent with the primary server, as the deployers have chosen for a lazy synchronization strategy. The third server, B_{ext}, is maintained by an external backup center. Its processing time is rather slow because the server does not reside on the same LAN as the enterprise SOA. Moreover, the enterprise is charged a cost each time the data is accessed or manipulated.

The enterprise wants to implement a global policy for all applications accessing customer data. This policy has two important statements:

- The primary server should be used whenever it is available. Only if it fails, the backup server may be contacted. If that server fails too, then the external backup center may be used.
- The bindings to the external backup center should not be hardcoded, as the enterprise continuously seeks for providers with a better quality of service.

While this policy is extremely hard to implement in the current version of Java, it requires only one line of code in ServiceJ. This is achieved by using the Service Path environment and a **sequence** qualifier:

– The Service Path environment is used to define the locations of the three servers. As such, the bindings are not hardcoded, and the management of the external backup server is easy. There is no need to update the applications that depend on a customer database if a server is moved.
– The preference policy can be implemented using a **sequence** qualifier. For example, the replicated components might contain an operation `getOrder()` that returns an integer representing the order in which these systems are to be contacted. The primary database server may then implement this method by returning "1", the backup system may return "2", etc. The runtime system of ServiceJ will automatically sort the providers, and contact them in the right order. The application developers may then implement the enterprise policy by writing the following code:

```
sequence CustomerDB cust orderby cust.getOrder();
```

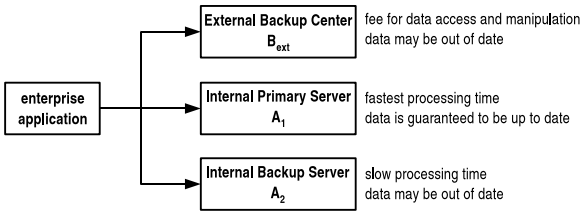


Fig. 16: Replication of a database server in an internal SOA

7.2 Cooperation Between Business Partners

A service client uses the import section of the Service Path environment to insert URLs pointing to export sections of the providers on which it depends. These export sections contain the locations of the services that a provider offers. However, a provider can extend this section such that it also contains URLs pointing to the export sections its *business partners*.

An example is shown in Fig. 17. This figure shows the import section of a client which depends on two service providers. The left provider has added four service URLs to its export section. But this section also contains four URLs pointing to other export sections. During service pool initialization, ServiceJ will scan the export section of the provider and add all service URLs to the pool. But it will also recursively scan the other `SP` URLs found in the export section, leading to an even bigger collection of alternatives.

7.3 Outsourcing

An important trend in today's enterprises is to outsource those segments of an enterprise that are not part of the *core competencies*. For example, an enterprise

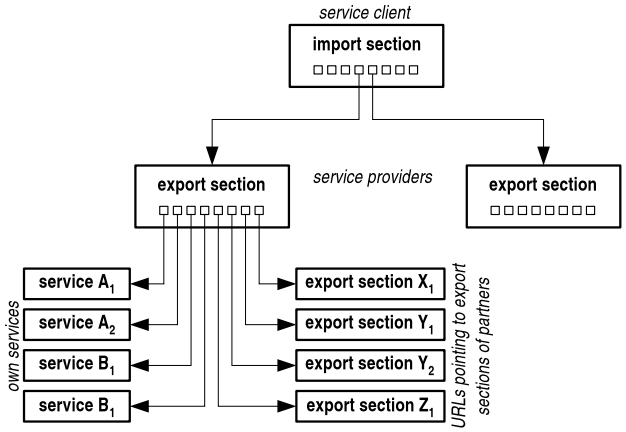


Fig. 17: The export section may contain URLs pointing to other export sections

that delivers business-to-business IT solutions may outsource its entire marketing or accounting division. In this section, we show how service pools and service paths ease the software integration problems that inherently occur when some activities of an enterprise are outsourced.

In small organizations, business activities (marketing, finance, accounting,...) are typically handled inhouse. Consequently the software supporting these activities is optimized for communicating with local applications. In other words, no thought is given to communication outside the internal SOA.

When a business activity is outsourced, however, every reference to the software system supporting that activity should be redirected to an external service provider. Reintegrating the enterprise system requires two major modifications to the enterprise SOA. First, every client that depends on the system being outsourced, must be updated to refer to the new system, which is now an externally managed application. This involves modifying and recompiling every system that depends on the software system being outsourced. There is no "modification isolation". Second, the system being outsourced must be updated in the same way, because it might contain references to the internal enterprise SOA, which the outsourcer sees as an external system. These integration problems occur each time the enterprise decides to move to another outsourcer.

In our approach, moving a business activity to an external provider is supported by the service path. Developers only need to update the location of the registry service to point to the outsourcer instead of an internal registry. Our service pools don't differentiate between local or remote services, so contrary to J2EE 5.0, no changes are needed to the JNDI lookup mechanism. Moreover, no changes are needed to the enterprise system whenever the outsourcer decides to add more or less replicated service endpoints. The same actions are taken to integrate the outsourced system with the enterprise SOA. Thus, instead of

changing, recompiling and reconfiguring a major part of the service architecture, our approach only requires a minor modification of the service path.

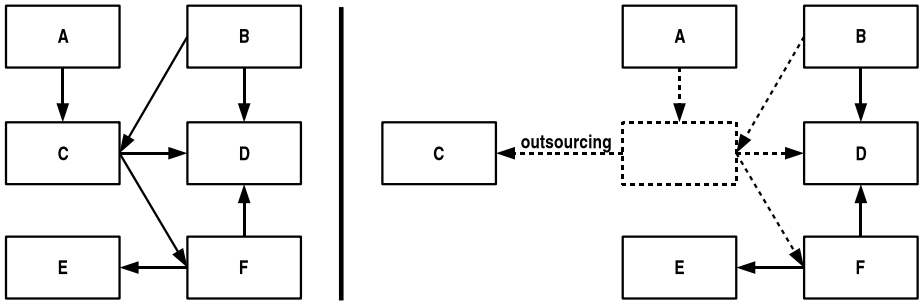


Fig. 18: An enterprise system with six components (left) where one component is outsourced (right)

7.4 Local Service Pools

A part of our future work is concerned with the development of *local pools*. These pools are used in scenarios where both the client and the "server" are classes in the same JVM. For example, a local banking application may model the holder and his/her account as two classes `Person` and `BankAccount`. One of the business methods of a person may then state that "the person may succesfully execute a payment by withdrawing the required amount of money from any of his accounts". This business method can be implemented using a local service pool. For further details about local service pools, we refer to future work.

8 Evaluation

In this Section, we compare ServiceJ to Java. We compare how both approaches deal with service volatility and service availability. We also consider how both languages provide support for the discovery of services. Finally, we compare the level of abstraction that both language offer to the programmer.

Service Volatility. Using Java Enterprise Edition 5.0, developers may define service endpoint locations in two ways. First, they may use the `wsdlLocation` attribute of the `@WebServiceRef` annotation. The disadvantage of this approach is that service endpoints are hardcoded in the source code of a service client. An alternative approach is to externalize service endpoint URLs in a Deployment Descriptor (DD). This approach, however, does not scale well, as *each* service reference must be bound to a service endpoint. Thus, if multiple references depend on the same provider URL, developers need to duplicate the endpoint URL

in their DD. Consequently, more than one endpoint reference must be updated if some provider moves to another location.

ServiceJ, on the other hand, relies on the Service Path environment to bind service references to the proper endpoint URLs. This `SP` does not point directly to service URLs. Instead, it contains references to the exported SP sections of providers. As such, service clients no longer depend on the deployment strategy of their providers. Providers willing to migrate their services, may update the new service locations in their export sections. The next time a service is to be located, ServiceJ scans through these export sections and automatically finds the new location. This approach assumes that the URL of the export section is stable. This is more robust than assuming that no service provider will ever migrate one of its services.

Availability. Java fails to support fault tolerant service invocations in two ways. First, the `@WebServiceRef` does not support the definition of multiple `wsdlLocation` attributes. Thus, developers have no means for defining the locations of backup services. Second, the container is not able to inject multiple service references into a single field. Thus, whenever a reference points to a service that has become unavailable, the container is not able to rebind this reference to another service endpoint.

ServiceJ extends the Java type system with **pool** and **sequence** references. Contrary to regular Java references, these references may contain refer to multiple service endpoints, although only one is active at a time. If this active reference contacts a failing service endpoint, then ServiceJ will rebind this reference to another service endpoint and reinvoke the operation. In addition, this failover mechanism is completely transparent for application developers.

Service Discovery. Java provides no means for discovering services. Even if such discovery engine existed, it would not be of much use, as the static binding strategy of Java completely disable any support for binding references to dynamically discovered service endpoints.

Similar to Java, ServiceJ does not provide a discovery engine itself. But it provides a way for integrating with external discovery engines (see also Sect. 9). The *dynamic import section* of the Service Path environment allows such external discovery systems to add the URLs of newly found `SP` export sections. Also, URLs pointing directly to WSDL definitions of services may be added, although this approach is less flexible as service migration cannot be handled.

Transparency. In Java, developers could choose to implement a framework to support failover. However, this approach would not be transparent because the business logic of an application is interleaved with code to provide failover. Typically, the catch block of a RemoteException would be used to bind the failing reference to a different endpoint. This strategy severely pollutes the business logic of a service client.

In contrast, ServiceJ provides failover in a transparent way. Services that have become unreachable are transparently swapped with other services from the same pool. In addition, developers are able to customize this failover mechanism in a declarative way. At this moment, ServiceJ offers operations for constraining and sorting the pool (**where** and **orderby**).

9 Related Work

Cardelli et al. propose *service combinators* [11] to deal with connectivity problems in web computing. The sequential execution operator (denoted "`?`") allows to increase the fault-tolerance level of web clients. For example, the statement `(S1?S2).bookFlight();` contacts `S2` if `S1` is unreachable. Service combinators are used in WebOz [12], WebL [13], XL [14] and Aries [15]. While service combinators provide a basic means for fault-tolerance, these languages do not capture the volatility of services because addresses (`S1` and `S2`) are hardwired in the source code. ServiceJ, on the other hand, provides dynamic binding to addresses found in the Service Path, thus removing the need to modify source code each time a service moves to a new location. Moreover, ServiceJ proposes a more powerful approach of fault-tolerance by offering declarative operations for sorting and constraint handling.

BPEL [16] and BPELJ [17] provide a language for specifying business processes and interaction protocols. The main problems with these languages are their verbose programming model and their lack of support for dynamic binding. The authors of [18] have extended BPEL with the `<evaluate>` tag to enable dynamic binding. Using this tag, the target of a service reference can be bound to the result of a method invocation. A similar approach is proposed in GPSL [19]. While both proposals provide basic support for dynamic binding, their evaluation mechanisms are too weak to support *transparent* fault-tolerance. Application developers must pollute their business logic with code for handling availability problems, thus reducing the readability of the source code. ServiceJ solves these problems by providing its failover algorithms in a transparent way. As such, application developers can concentrate on the implementation of the business logic. Moreover, ServiceJ advocates a *non-verbose programming model* by offering lightweight type qualifiers and declarative operations.

Similar to BPEL, ActiveXML suffers from hardwired service addresses scattered throughout the source code. The authors of [20] extend ActiveXML with dynamic binding by implementing *implicit service calls*. Instead of using hardcoded addresses, their approach relies on an ontology to locate the WSDL descriptions of services. The main drawback of this approach is that the *base url* (e.g. `www.A1.com/wsdl/`) must still be hardcoded in the client, making it impossible to contact other business partners in case the primary provider fails. ServiceJ, on the other hand, does not require base urls to be hardcoded and is able to bind a single reference to multiple services. Moreover, hot swapping of crashing services is executed transparently, while ActiveXML and its proposed extension require manual intervention of application developers.

Jini [21, 22] is a Java framework that offers an environment for dynamically discovering and binding services. Discovery is limited, however, because it relies on multicast messages. Such messages are mostly blocked by WAN routers as they consume too much bandwidth. Thus, application developers are forced to send unicast messages to hardcoded service locations. This programming model is not adequate for SOA applications because it requires extensive reconfiguration in an environment with volatile services. Moreover, application developers must write a lot of boilerplate code in order to implement fault-tolerance, whereas ServiceJ provides a lightweight solution with declarative operations.

A lot of solutions based on Aspect-Oriented Programming (AOP) have been proposed to deal with crosscutting concerns such as security, logging and fault-tolerance. The proposal that most ressembles our approach is the Web Services Management Layer (WSML) [23]. This layer acts as a mediator between the service client and the web services on which that client depends. WSML enables hot swapping of unreachable services. ServiceJ provides transparent failover in a similar way, but adds support for fine-tuning the pool of alternative services by means of declarative operations (**where** and **orderby**).

Currently, ServiceJ does not provide a discovery engine. The dynamic import section of the Service Path, however, allows external discovery systems to add references to newly discovered services. Examples of discovery systems that may be used to complement our approach include SelfServ [24], Osiris [25], Meteor-S [26], SPiDeR [27], and Obduro [28].

10 Conclusion and Future Work

We have presented ServiceJ, an extension of Java with support for dependable SOA applications. Support is provided at the programming level using language concepts, and at the runtime level using transparent algorithms.

ServiceJ improves SOA support of existing OOPL in two ways. First, the Service Path environment allows deployers to manage service configurations in a central way. In addition, its dynamic section provides an entry point for discovery mechanisms to include newly found services. Second, ServiceJ provides failover in a transparent way. Using type qualifiers (**pool** and **sequence**), developers may customize this failover mechanism. In addition, developers are equiped with declarative operations (**where** and **orderby**) to customize fault-tolerant invocations even further.

Future research includes the definition of new type qualifiers to provide parallel service invocations and aggregation of query results from heterogeneous services. We are also investigating how the Service Path can be extended with additional functionality, such as an inheritance mechanism. We are currently working on a prototype compiler.

References

1. Papazoglou, M.: SOC: Concepts, Characteristics and Directions. In: Proc. of the 4th Int. Conf. on Web Information Systems Engineering. (2003)

2. Java EE 5.0 Specification: http://www.jcp.org/en/jsr/detail?id=244. (2006)
3. C# Language Specification: http://www.ecma-international.org/. (2005)
4. JAX-WS 2.0 Specification: http://www.jcp.org/en/jsr/detail?id=224. (2005)
5. WSDL Specification v1.1: www.w3.org/TR/wsdl. (2001)
6. The JVM Specification: http://java.sun.com/docs/books/vmspec/. (1999)
7. Mockapetris, P.: Domain names. concepts and facilities. Internet RFC 1034. (1987)
8. Tschantz, M.S., et al.: Javari: adding reference immutability to Java. In: OOPSLA '05: Proc. of the 20th Conf. on OOP, Systems, Languages, and Applications. (2005)
9. Igarashi, A., et al.: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: OOPSLA99. Volume 34(10). (1999) 132–146
10. Flatt, M., et al.: Classes and Mixins. In: POPL98: Proc of the 25th Symposium on Principles of Programming Languages. (1998) 171–183
11. Cardelli, L., Davies, R.: Service Combinators for Web Computing. In: IEEE Transactions on Software Engineering **25**(3) (1999) 309–316
12. Hadim, M., et al.: Service Combinators for Web Computing in Distributed Oz. In: Proc. Intl. Conf. on Parallel and Distr. Processing Techniques and Appl. (2000)
13. Kistler, T., Marais, H.: WebL - A Programming Language for the Web. In: Proc. 7th Intl. Conf. on WWW, The Netherlands, Elsevier (1998) 259–270
14. Florescu, D., Gruenhagen, A., Kossmann, D.: XL: A Platform for Web Services. In: Proc. First Biennial Conf. on Innovative Data Systems Research. (2003)
15. Pereira, F., Valente, M., et al.: Tactics for Remote Method Invocation. In: Journal of Universal Computer Science **10**(7) (2004) 824–842
16. BPEL4WS Specification v1.1: http://www.ibm.com/developerworks. (2003)
17. BPELJ: BPEL for Java technology: http://www.ibm.com/developerworks. (2004)
18. Karastoyanova, D., et al.: An Approach to Parametrizing Web Service Flows. In: ICSOC05– Proc. of the 3rd Int. Conf. on Service Oriented Computing. (2005)
19. Cooney, D., et al.: Programming and Compiling Web Services with GPSL. In: ICSOC05– Proc. of the 3rd Int. Conf. on Service Oriented Computing. (2005)
20. Benbernou, S., et al.: Implicit Service Calls in ActiveXML Through OWL-S. In: ICSOC05– Proc. of the 3rd Int. Conf. on Service Oriented Computing. (2005)
21. Jini Architecture Specification: (http://www.jini.org)
22. Hasselmeyer, P.: On Service Discovery Process Types. In: ICSOC05– Proc. of the 3rd Int. Conf. on Service Oriented Computing. (2005)
23. Verheecke, B., et al.: AOP for Dynamic Configuration and Management of Web Services. In: International Journal on Web Services Research **1**(3) (2004) 25–41
24. Benatallah, B., Sheng, Q., et al.: The Self-Serv Environment for Web Services Composition. In: IEEE Internet Computing, 7(1):40-48. (2003)
25. Schuler, C., Weber, R., et al.: Peer-to-Peer Process Execution with OSIRIS. In: ICSOC03– Proc. of the 1st Int. Conf. on Service Oriented Computing. (2003)
26. Gomadam, K., et al.: Demonstrating Dynamic Configuration and Execution of Web Processes. In: ICSOC05– Proc. of the 3rd Int. Conf. on SOC. (2005)
27. Sahin, O., et al.: Spider: P2p-based web service discovery. In: ICSOC05– Proc. of the 3rd Int. Conf. on Service Oriented Computing. (2005)
28. Vogels, W.: Tracking Service Availability in Long Running Business Activities. In: ICSOC03– Proc. of the 1st Int. Conf. on Service Oriented Computing. (2003)

A Proofs

Lemma.
If $mtype(m, D) = \overline{T} \rightarrow T_0$, *then* $mtype(m, C) = \overline{T} \rightarrow T_0$ *for all* $C \leq D$

Proof. By induction on the derivation of $C \leq D$.

Trivial case. $C = D$. Then $mtype(m, C) = mtype(m, D) = \overline{T} \rightarrow T_0$
Induction Hypothesis. Assume $mtype(m, C_n) = mtype(m, C_1)$ $\forall C_n \leq \ldots \leq C_1$
Induction Step.
We have to prove that $mtype(m, C_{n+1}) = mtype(m, C_1)$ $\forall C_{n+1} \leq \ldots \leq C_1$
There are two cases:

- m *is not defined in* C_{n+1}. Then, by rule MType-CSub, we have: $mtype(m, C_{n+1}) = mtype(m, C_n) = mtype(m, C_1) = \overline{T} \rightarrow T_0$ by the induction hypothesis.
- m *is defined in* C_{n+1}. Then, $U\ m(\overline{U}\ x)$ is part of the definition of C_{n+1}. We also have $mtype(m, C_n) = \overline{T} \rightarrow T_0$ by the induction hypothesis. Using rule MType-C, we get: $\overline{U} = \overline{T}$ and $U = T$. Thus, $mtype(m, C_{n+1}) = mtype(m, C_n) = mtype(m, C_1) = \overline{T} \rightarrow T_0$.

B Preservation

If $\Gamma, \overline{x} : \overline{W} \vdash e : T$ *and* $\Gamma \vdash \overline{d} : \overline{V}$ *where* $\overline{V} <: \overline{W}$
Then $\Gamma \vdash [\overline{d}/\overline{x}]e : S$ *for some* $S <: T$

PROOF. By induction on the derivation of $\Gamma, \overline{x} : \overline{W} \vdash e : T$.

Case T-VAR $e \equiv x$ $T = \Gamma(x)$

- **Case** $x \notin \overline{x}$
  Then, the substitution has no effect and $[\overline{d}/\overline{x}]x = x$. Thus, $\Gamma([\overline{d}/\overline{x}]x) = \Gamma(x) = T$ and S-REFL finishes the case.

- **Case** $x \in \overline{x}$. Take $x = x_i$
  We have $\Gamma, \overline{x} : \overline{W} \vdash x : T$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$
  Because $x = x_i$, we have $\Gamma(x) = T = W_i$.
  Then, $[\overline{d}/\overline{x}]x = [d_i/x_i]x = d_i : V_i$ with $V_i <: W_i$ and $W_i = T$, so $V_i <: T$.
  Taking $S = W_i$ gives $\Gamma \vdash [\overline{d}/\overline{x}]x : S$ with $S <: T$ and this finishes the case.

Case T-FIELD $e \equiv e_0.f_i$ $\Gamma, \overline{x} : \overline{W} \vdash e_0 : \mathbf{singleton}\ D_0$
$fields(D_0) = \overline{S}\ \overline{f}$
$T = S_i$

We have $\Gamma, \overline{x} : \overline{W} \vdash e_0 : \mathbf{singleton}\ D_0$ and $\Gamma \vdash \overline{d} : \overline{V}$ where $\overline{V} <: \overline{W}$. By the induction hypothesis, we have $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S$ for some $S <: \mathbf{singleton}\ D_0$. Take $S = Q\ D$. Then, $Q\ D <: \mathbf{singleton}\ D_0$ and by S-TYP we have that $Q <: \mathbf{singleton}$ and $D <: D_0$. Thus, $Q = \mathbf{singleton}$ because no qualifier can be more specific than **singleton**. By FIELDS-C, we have $fields(D) = fields(D_0),\ \overline{U}\ \overline{g}$ for some $\overline{U}\ \overline{g}$. We can now apply T-FIELD with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : \mathbf{singleton}\ D$
- $fields(D) = fields(D_0),\ \overline{U}\ \overline{g}$

The result is $\Gamma \vdash [\overline{d}/\overline{x}]e_0.f_i : S_i$. We also know that $S_i = T$, so the case is finished by S-REFL.

Case T-INVK $e \equiv e_0.m(\overline{e})$ $mtype(m, T_0) = \overline{U} \to T$
$\Gamma, \overline{x} : \overline{W} \vdash \overline{e_0} : \overline{T_0}$
$\Gamma, \overline{x} : \overline{W} \vdash \overline{e} : \overline{T}$ and $\overline{T} <: \overline{U}$

We have $\Gamma, \overline{x} : \overline{W} \vdash\ e_0.m(\overline{e}) : T$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. We can apply the induction hypothesis to both e_0 and $\overline{e}$.

1. For $\Gamma, \overline{x} : \overline{W} \vdash e_0 : T_0$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$, there exists some $S_0 <: T_0$ such that $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$.
2. For $\Gamma, \overline{x} : \overline{W} \vdash \overline{e} : \overline{T}$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$, there exist $\overline{S} <: \overline{T}$ such that $\Gamma \vdash [\overline{d}/\overline{x}]\overline{e} : \overline{S}$

By Lemma 1, we know that $mtype(m, S_0) = mtype(m, T_0) = \overline{U} \rightarrow T$. By S-TRANS we know that $\overline{S} <: \overline{U}$ because $\overline{S} <: \overline{T}$ and $\overline{T} <: \overline{U}$. Thus, we can apply T-INVK with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$
- $\Gamma \vdash [\overline{d}/\overline{x}]\overline{e} : \overline{S}$ and $\overline{S} <: \overline{U}$
- $mtype(m, S_0) = \overline{U} \rightarrow T$

The result is $\Gamma \vdash e_0.m(\overline{e}) : T$ and S-REFL finishes the case.

Case T-NEW $\quad e \equiv new\ D(\overline{e}) \quad \Gamma, \overline{x} : \overline{W} \vdash \overline{e} : \overline{S}$ and $\overline{S} <: \overline{T}$, $fields(D) = \overline{T}\ \overline{f}$

We have $\Gamma, \overline{x} : \overline{W} \vdash$ **new** $D(\overline{e}) : T$, where $T = Q\ D$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. We can apply the induction hypothesis to $\overline{e}$. There exist $\overline{U} <: \overline{S}$ such that $\Gamma \vdash [\overline{d}/\overline{x}]\overline{e} : \overline{U}$ By S-TRANS, we have $\overline{U} <: \overline{T}$, because $\overline{U} <: \overline{S}$ and $\overline{S} <: \overline{T}$. We can now apply T-NEW with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]\overline{e} : \overline{U}$ with $\overline{U} <: \overline{T}$
- $fields(D) = \overline{T}\ \overline{f}$

The result is $\Gamma \vdash$ **new** $D([\overline{d}/\overline{x}]\overline{e})$: **singleton** D. We have **singleton** $<: Q$ by S-POOL and S-SEQ. We also have $D <: D$ by S-REFL. Application of S-TYP gives **singleton** $D <: Q\ D$ and this finishes the case.

Case T-UCAST $\quad e \equiv (T)e_0 \quad \Gamma, \overline{x} : \overline{W} \vdash e_0 : S$ and $S <: T$

We have $\Gamma, \overline{x} : \overline{W} \vdash e_0 : S$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. By the induction hypothesis, there exists some $U <: S$ such that $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : U$. We can apply S-TRANS on $U <: S$ and $S <: T$, delivering $U <: T$. We can now apply T-UCAST with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : U$
- $U <: T$

The result is $\Gamma \vdash (T)[\overline{d}/\overline{x}]e_0 : T$ and S-REFL finishes the case.

Case T-DCAST $\quad e \equiv (T)e_0 \quad \Gamma, \overline{x} : \overline{W} \vdash e_0 : S$, $T <: S$ and $T \neq S$

We have $\Gamma, \overline{x} : \overline{W} \vdash e_0 : S$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. By the induction hypothesis, there exists some $U <: S$ such that $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : U$. Thus, $T <: S$ and $U <: S$. There are three possibilities:

1. **SubCase:** $U <: T$.
   We can now apply T-UCAST with the following premises:
   - $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : U$
   - $U <: T$

   The result is $\Gamma \vdash (T)[\overline{d}/\overline{x}]e_0 : T$ and S-REFL finishes the case.
2. **SubCase:** $T <: U$.
   We can now apply T-DCAST with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : U$
- $T <: U$

The result is $\Gamma \vdash (T)[\overline{d}/\overline{x}]e_0 : T$ and S-Refl finishes the case.

3. **SubCase:** $T \not<: U \wedge U \not<: T$.
   We can now apply T-SCast with the following premises:
   - $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : U$
   - $T \not<: U \wedge U \not<: T$

   The result is $\Gamma \vdash (T)[\overline{d}/\overline{x}]e_0 : T$ with a *stupid warning* and S-Refl finishes the case.

Case T-SCast $\quad e \equiv (C)e_0 \quad\quad \Gamma, \overline{x} : \overline{W} \vdash e_0 : Q_0\ D_0$, $C \not<: D_0 \wedge D_0 \not<: C$

We have $\Gamma, \overline{x} : \overline{W} \vdash e_0 : Q_0\ D_0$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. By the induction hypothesis, there exists some $Q_0'\ D_0' <: Q_0\ D_0$ such that $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : Q_0'\ D_0'$. Then, by S-Typ we have $Q_0' <: Q_0$ and $D_0' <: D_0$. We prove that D_0' and C are unrelated, i.e. $D_0' \not<: C \wedge C \not<: D_0'$

1. $D_0' \not<: C$.
   This is because $D_0' <: D_0$ and $D_0 \not<: C$.Thus, S-Trans cannot be applied and $D_0' \not<: C$.
2. $C \not<: D_0'$.
   Proof by contradiction. Assume $C <: D_0'$. We also have $D_0' <: D_0$, so we can apply S-Trans. The result is $C <: D_0$, but this contradicts with the fact that $C \not<: D_0$.

We can now apply T-SCast with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : Q_0'\ D_0'$
- $D_0' \not<: C \wedge C \not<: D_0'$

The result is $\Gamma \vdash (C)[\overline{d}/\overline{x}]e_0 : Q\ C$ with a *stupid warning*

Case T-Whr $\quad e \equiv e_0$ **where** $e_1 \quad\quad \Gamma, \overline{x} : \overline{W} \vdash e_0 : T$, $\Gamma, \overline{x} : \overline{W} \vdash e_1 :$ **bool**

We have $\Gamma, \overline{x} : \overline{W} \vdash e_0 : T$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. By the induction hypothesis, we have $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S$ for some $S <: T$. In our model, subtyping is not supported for primitive types, so $\Gamma \vdash [\overline{d}/\overline{x}]e_1 :$ **bool**. We can now apply T-Whr with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S$
- $\Gamma \vdash [\overline{d}/\overline{x}]e_1 :$ **bool**

The result is $\Gamma \vdash [\overline{d}/\overline{x}]e_0$ **where** $[\overline{d}/\overline{x}]e_1 : S$. This finishes the case because $S <: T$

Case T-ORD $\quad e \equiv e_0$ **orderby** e_1 $\quad\quad \Gamma, \overline{x} : \overline{W} \vdash e_0 : S$
$U = toSequence(S)$
$\Gamma, \overline{x} : \overline{W} \vdash e_1 : T$
$T <:$ **singleton Comparable**

We have $\Gamma, \overline{x} : \overline{W} \vdash e_0 : S$, and $\Gamma \vdash \overline{x} : \overline{V}$ with $\overline{V} <: \overline{W}$. By the induction hypothesis, there exist some $S' <: S$ such that $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S'$. We also have $\Gamma, \overline{x} : \overline{W} \vdash e_1 : T$, and $\Gamma \vdash \overline{x} : \overline{V}$ with $\overline{V} <: \overline{W}$. Again, by the induction hypothesis, there exist some $T' <: T$ such that $\Gamma \vdash [\overline{d}/\overline{x}]e_1 : T'$. We can now apply S-TRANS on $T' <: T$ and $T <:$ **singleton Comparable**. This gives $T' <:$ **singleton Comparable**. We now apply T-ORD with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S'$
- $\Gamma \vdash [\overline{d}/\overline{x}]e_1 : T'$
- $T' <:$ **singleton Comparable**
- $U' = toSequence(S')$

The result is $\Gamma \vdash [\overline{d}/\overline{x}]e_0$ **orderby** $[\overline{d}/\overline{x}]e_1 : U'$
Next, we prove that $U' <: U$. Let $U = Q_U\ C_U$ and $U' = Q'_U\ C'_U$. We also know that $U = toSequence(S)$ and $U' = toSequence(S')$. Let $S = Q_S\ C_S$ and $S' = Q'_S\ C'_S$. Moreover, the *toSequence* function only modifies type qualifiers Q. Thus $C_U = C_S$ and $C'_U = C'_S$. We also have that $S' <: S$, so $C'_S <: C_S$ and thus $C'_U <: C_U$.

Now, we have to prove that $Q'_U <: Q_U$. This is done by a case analysis. We know that $S' <: S$, so $Q'_S <: Q_S$, giving six cases:

- $Q'_S =$ **pool** $\wedge\ Q_S =$ **pool**. Then,
  $toSequence(Q'_S\ C'_S) =$ **sequence** C'_S
  $toSequence(Q_S\ C_S) =$ **sequence** C_S
  and the case is finished by S-TYP.
- $Q'_S =$ **sequence** $\wedge\ Q_S =$ **pool**. Then,
  $toSequence(Q'_S\ C'_S) =$ **sequence** C'_S
  $toSequence(Q_S\ C_S) =$ **sequence** C_S
  and the case is finished by S-TYP.
- $Q'_S =$ **sequence** $\wedge\ Q_S =$ **sequence**. Then,
  $toSequence(Q'_S\ C'_S) =$ **sequence** C'_S
  $toSequence(Q_S\ C_S) =$ **sequence** C_S
  and the case is finished by S-TYP.
- $Q'_S =$ **singleton** $\wedge\ Q_S =$ **pool**. Then,
  $toSequence(Q'_S\ C'_S) =$ **singleton** C'_S
  $toSequence(Q_S\ C_S) =$ **sequence** C_S
  and the case is finished by S-TYP.
- $Q'_S =$ **singleton** $\wedge\ Q_S =$ **sequence**. Then,
  $toSequence(Q'_S\ C'_S) =$ **singleton** C'_S

$toSequence(Q_S\ C_S) = \textbf{sequence}\ C_S$
and the case is finished by S-Typ.
- $Q'_S = \textbf{singleton} \wedge Q_S = \textbf{singleton}$. Then,
  $toSequence(Q'_S\ C'_S) = \textbf{singleton}\ C'_S$
  $toSequence(Q_S\ C_S) = \textbf{singleton}\ C_S$
  and the case is finished by S-Typ.

Case T-Set $e \equiv set\ e_0.f_i = e_1\ then\ e_2$

$\Gamma, \overline{x} : \overline{W} \vdash e_0 : \textbf{singleton}\ C$
$\Gamma, \overline{x} : \overline{W} \vdash e_1 : S$
$\Gamma, \overline{x} : \overline{W} \vdash e_2 : T$
$fields(C) = \overline{U}\ \overline{f}$
$S <: U_i$

We have $\Gamma, \overline{x} : \overline{W} \vdash set\ e_0.f_i = e_1\ then\ e_2 : T$ and $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{W}$. By the induction hypothesis, we have:

1. $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : Q'\ C'$ for some $Q'\ C' <: \textbf{singleton}\ C$.
   By S-Typ, we have $Q' <: \textbf{singleton}$ and $C' <: C$. Thus, $Q' = \textbf{singleton}$.
2. $\Gamma \vdash [\overline{d}/\overline{x}]e_1 : S'$ for some $S' <: S$
3. $\Gamma \vdash [\overline{d}/\overline{x}]e_2 : T'$ for some $T' <: T$.
   By Fields-C, we have $fields(C') = fields(C), \overline{T}\ \overline{g}$ for some $\overline{T}\ \overline{g}$.
   By S-Trans, we have $S' <: U_i$ because $S' <: S$ and $S <: U_i$.

We can now apply T-Set with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_0 : \textbf{singleton}\ C'$
- $\Gamma \vdash [\overline{d}/\overline{x}]e_1 : S'$
- $\Gamma \vdash [\overline{d}/\overline{x}]e_2 : T'$
- $fields(C') = fields(C), \overline{T}\ \overline{g}$
- $S' <: U_i$

The result is $\Gamma \vdash set\ [\overline{d}/\overline{x}]e_0.f_i = [\overline{d}/\overline{x}]e_1\ then\ [\overline{d}/\overline{x}]e_2 : T'$. This finishes the case because $T' <: T$.

Case T-Let $e \equiv let\ y = e_1\ in\ e_2$

$\Gamma \vdash e_1 : T_y$
$\Gamma, y : T_y \vdash e_2 : U$

We have $\Gamma, \overline{x} : \overline{W} \vdash let\ y = e_1\ in\ e_2 : T$ and $\Gamma \vdash \overline{d} : \overline{V}$ for some $\overline{V} <: \overline{W}$. By the induction hypothesis, we have $\Gamma \vdash [\overline{d}/\overline{x}]e_1 : T'_y$ for some $T'_y <: T_y$. We can also apply the induction hypothesis to e_2, giving $\Gamma, y : T_y \vdash [\overline{d}/\overline{x}]e_2 : U'$ for some $U' <: U$. Finally, we can apply T-Let with the following premises:

- $\Gamma \vdash [\overline{d}/\overline{x}]e_1 : T'_y$
- $\Gamma, y : T_y \vdash [\overline{d}/\overline{x}]e_2 : U'$

The result is $\Gamma \vdash let\ y = [\overline{d}/\overline{x}]e_1\ in\ [\overline{d}/\overline{x}]e_2 : U'$ for some $U' <: U$. This finishes the case.

□

C Progress

If $\vdash \langle e_0, \mathcal{S} \rangle : T$, then:
- e_0 is a variable x, or
- $\langle e_0, \mathcal{S} \rangle \longrightarrow \langle e_0', \mathcal{S}' \rangle$ for some $\mathcal{S}'$ and e', or
- $\langle e_0, \mathcal{S} \rangle$ is stuck due to the occurence of the $fail$ predicate, or
- $\langle e_0, \mathcal{S} \rangle$ is stuck in a failed dynamic downcast.

Proof. By induction on $\vdash \langle e_0, \mathcal{S} \rangle : T$.

Case $\langle e_0, \mathcal{S} \rangle \equiv \langle x, \mathcal{S} \rangle$

In that case, the expression e_0 is a variable x, so the theorem is true by definition (see the first case of the theorem).

Case $\langle e_0, \mathcal{S} \rangle \equiv \langle e.f_i, \mathcal{S} \rangle$

Substituting e_0 in the theorem premise gives $\vdash \langle e.f_i, \mathcal{S} \rangle : T_i$.
From T-State, we have for $\vdash \langle e.f_i, \mathcal{S} \rangle : T_i$ that:
(a) $\vdash \mathcal{S} : \Gamma$
(b) $\Gamma \vdash e.f_i : T_i$

From T-Get and (b), we have:
(c) $\Gamma \vdash e :$ **singleton** C
(d) $fields(C) = \overline{T}\ \overline{f}$

For this case, we need rules concerning the typing of the store $\mathcal{S}$. From (a) and T-StoreTyp, the following holds $\forall x \in dom(\Gamma)$:
(e) $\Gamma \vdash \mathcal{S}_T(x) : T$
(f) $T <: \Gamma(x)$

We also need rules concerning the field mapping $\mathcal{F}$ of the expression e. From (a) and T-FMap, we have $\forall x \in dom(\Gamma)$ where $\mathcal{S}(x) = \langle Q\ C, \mathcal{F} \rangle$:
(g) $\forall f_i \in fields(C) : \mathcal{S}_T(\mathcal{F}(f_i)) <: T_i$

SubCase $\langle x.f_i, \mathcal{S} \rangle$

Substituting e by x in (c), we get $\Gamma \vdash x :$ **singleton** C. From (e) and (f) we get that the typing of the store $\mathcal{S}$ is consistent with the typing environment Γ. Thus, the store $\mathcal{S}$ must map the type of x to **singleton** C. We have $\mathcal{S}(x) = \langle$**singleton** $C, \mathcal{F} \rangle$ where $\mathcal{F}$ is the field mapping belonging to x. This mapping is defined as $\mathcal{F} : f_i \rightarrow v_i$. Moreover, by (g), we know that is mapping is well defined. Formally, for all v_i we have $\mathcal{S}(v_i) <: T_i$.

We can now fill in R-Field with the following premises:
- $\mathcal{S}(x) = \langle$**singleton** $C, \mathcal{F} \rangle$ by (e) and (f)
- $\mathcal{F} : f_i \rightarrow v_i$ by (g)
- $fields(C) = \overline{T}\ \overline{f}$ by (d)

Using this reduction rule, $\langle x.f_i, \mathcal{S} \rangle$ reduces to $\langle v_i, \mathcal{S}' \rangle$. Thus, the second reduction path of the theorem is taken and the case is finished.

SubCase $\langle e.f_i, \mathcal{S}\rangle$

We can apply the induction hypothesis on e, giving rise to these cases:

- e is a variable x. Then, $\langle e.f_i, \mathcal{S}\rangle \equiv \langle x.f_i, \mathcal{S}\rangle$ and this case is already proven.
- $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$ for some $\mathcal{S}'$ and e'. Then, $\langle e.f_i, \mathcal{S}\rangle$ reduces by RC-Field to $\langle e'.f_i, \mathcal{S}'\rangle$ for some $\mathcal{S}'$ and e'. This finishes the case.
- e is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. In that case, $e.f_i$ will never be evaluated because this requires e to be reduced. Thus, $e.f_i$ is also stuck, and this finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e.m(\overline{e}), \mathcal{S}\rangle$

Then $e \equiv e_0.m(\overline{e})$. Substituting e for $e_0.m(\overline{e})$ in the theorem premise gives $\vdash \langle e_0.m(\overline{e}), \mathcal{S}\rangle : T$.

From T-State, we have for $\vdash \langle e_0.m(\overline{e}), \mathcal{S}\rangle : T$ that:
(a) $\vdash \mathcal{S} : \Gamma$
(b) $\Gamma \vdash e_0.m(\overline{e}) : T$

From (a) and T-StoreTyp, the following holds $\forall x \in dom(\Gamma)$:
(c) $\Gamma \vdash \mathcal{S}_T(x) : T$
(d) $T <: \Gamma(x)$

From T-Invk and (b), we get that:
(e) $\Gamma \vdash \overline{e} : \overline{S}$ and $\overline{S} <: \overline{U}$
(f) $\Gamma \vdash e_0 : T_0$, and let $T_0 = Q_0\ C_0$ without loss of generality
(g) $mtype(m, T_0) = \overline{U} \rightarrow T$

The proof of this case is slightly more complicated because the runtime semantics of **singleton** method invocation differ significantly from the runtime semantics of method invocation on **pool** and **sequence** expressions. We prove both cases separately.

SubCase $T_0 \equiv Q_0\ C_0$ and $Q_0 \equiv$ **singleton** This subcase describes the proof for method invoation where the invocation target is an expression e of type **singleton** C_0. We consider three subsubcases for such **singleton** expressions.

SubSubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x.m(\overline{y}), \mathcal{S}\rangle$

By (c) and (d), we know that the typing of the store $\mathcal{S}$ is consistent with the typing environment Γ. Thus, $\mathcal{S}(e_0) \equiv \mathcal{S}(x) = \langle Q_0\ C_0, \mathcal{M}\rangle$. For this case, we took Q_0 to be a **singleton**, so $\mathcal{M} \equiv \mathcal{F}$ (by definition of $\mathcal{S}$) and we get $\mathcal{S}(x) = \langle$**singleton** $C_0, \mathcal{F}\rangle$. We also know that the field mapping $\mathcal{F}$ is well-defined by T-FMap:

(h) $\forall f_i \in fields(C) : \mathcal{S}_T(\mathcal{F}(f_i)) <: T_i$ where $fields(C) = \overline{T}\ \overline{f}$.

Also, by (g) we have that $mtype(m, T_0) = \overline{U} \rightarrow T$. By definition, $mbody(m, T_0) = \overline{z}.e_0$, whether the method is found in C_0 itself (MBody) or in one of its direct or indirect superclasses (Mbody-CSub).

We can now apply R-InvkS with the following premises:

- $\mathcal{S}(x) = \langle \textbf{singleton}\ C_0, \mathcal{F} \rangle$
- $mbody(m, T_0) = \overline{z}.e_0$

The expression $\langle x.m(\overline{y}), \mathcal{S} \rangle$ reduces to $\langle [\overline{y}/\overline{z}, x/\textbf{this}]e_0, \mathcal{S}' \rangle$. Thus, the second reduction path of the theorem is followed, and this finishes the case.

SubSubCase $\langle e_0, \mathcal{S} \rangle \equiv \langle e.m(\overline{e}), \mathcal{S} \rangle$

We can apply the induction hypothesis on e. According to the theorem, either of four possible cases holds for e. We prove that $\langle e.m(\overline{e}), \mathcal{S} \rangle$ reduces for each of these cases. We take the last two cases together as they each deal with expressions that got stuck during the reduction:

- e is a variable x. Then, there are two possibilities. If all the arguments of $m(\overline{e})$ are variables $\overline{y}$, then $\langle x.m(\overline{y}), \mathcal{S} \rangle$ reduces according to R-InvkS. This case has already been proven. The second case is when at least one e_i in $\overline{e}$ is not a variable. This case is proven below (in the next subsubcase).
- $\langle e, \mathcal{S} \rangle$ reduces to $\langle e', \mathcal{S}' \rangle$. In that case we can use the congruence rules to further reduce the expression. More precisely, $\langle e.m(\overline{e}), \mathcal{S} \rangle$ reduces by RC-Invk-Recv to $\langle e'.m(\overline{e}), \mathcal{S}' \rangle$. This finishes the case.
- The expression e is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. In that case, the method will never be invoked, and $\langle e.m(\overline{e}), \mathcal{S} \rangle$ will also be stuck due to the same problem. This finishes the case.

SubSubCase $\langle e_0, \mathcal{S} \rangle \equiv \langle x.m(\overline{e}), \mathcal{S} \rangle$

In that case, we can apply the induction hypothesis on each of the actual arguments of $m(\overline{e})$. Fix such an argument e_i without loss of generality. This expression e_i reduces according to either of the four reduction paths enumerated in the theorem. We prove that $\langle x.m(\overline{e}), \mathcal{S} \rangle$ reduces in each of these cases (the last two cases are taken together):

- e_i is a variable y. If this holds for all e_i in $\overline{e}$, then the expression reduces by R-InvkS. This case is already proven above. However, if the statement does not hold, then this means that there exists at least one e_j that is not a variable. This expression reduces by one of the other cases.
- $\langle e_i, \mathcal{S} \rangle \rightarrow \langle e_i', \mathcal{S}' \rangle$. Then, we can use the congruence rules to reduce $\langle x.m(\overline{e}), \mathcal{S} \rangle$. Taking RC-Invk-Arg, we have that $\langle x.m(e_1, \ldots, e_j, \ldots, e_n), \mathcal{S} \rangle$ reduces to $\langle x.m(e_1, \ldots, e_i', \ldots, e_n), \mathcal{S}' \rangle$. Thus, the expression reduces according to the second reduction path of the theorem, as such finishing the case.
- e_i is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. Either way, method invocation will not be executed. The expression $x.m(\overline{e})$ will be stuck because of the same problem. This finishes the case.

This concludes the proof for method invocation on **singleton** expressions. We now prove the case in which a method is invoked on an expression of type **pool** C_0.

SubCase $T_0 \equiv Q_0\ C_0$ and $Q_0 \equiv$ **pool**

This subcase describes the reduction of a method invocation where the invocation target e_0 has type T_0 where $T_0 \equiv$ **pool** C_0. This leads to the same subsubcases as the proof for method invocation on **singleton** references, but we need different reduction rules and congruence rules. We only give the proof for method invocation on **pool** references. The proof for $T_0 \equiv$ **sequence** C_0 follows the same reasoning with the same reduction and congruence rules for each step. Note that we refer back to facts from the beginning of this case (denoted with (a) to (g)).

SubSubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x.m(\overline{y}), \mathcal{S}\rangle$

By (c) and (d), we know that the typing of the store $\mathcal{S}$ is consistent with the typing environment Γ. Thus, $\mathcal{S}(e_0) \equiv \mathcal{S}(x) = \langle Q_0\ C_0, \mathcal{M}\rangle$. For this case, we took Q_0 to be a **pool**, so $\mathcal{M} \equiv \mathcal{R}$ (by definition of $\mathcal{S}$) and we get $\mathcal{S}(x) = \langle$**pool** $C_0, \mathcal{R}\rangle$. We also know that the reference mapping $\mathcal{R}$ is well-defined by T-RMap. This means that the type of every pool member must be conform to the type of x:

(h) $\forall v \in \mathcal{R} : \mathcal{S}_T(v) <:$ **singleton** C_0

Also, by (g) we have that $mtype(m, T_0) = \overline{U} \rightarrow T$. By definition, $mbody(m, T_0) = \overline{z}.e_0$, whether the method is found in C_0 itself (MBody) or in one of its direct or indirect superclasses (MBody-CSub).

We can now apply R-InvkP with the following premises:
- $\mathcal{S}(x) = \langle$**pool** $C_0, \mathcal{R}\rangle$
- $mbody(m, T_0) = \overline{z}.e_0$

The expression $\langle x.m(\overline{y}), \mathcal{S}\rangle$ reduces to $\langle[\overline{y}/\overline{z}, failover(\mathcal{R})/$**this**$]e_0, \mathcal{S}'\rangle$. Thus, the second reduction path of the theorem is followed, and this finishes the case.

SubSubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle e.m(\overline{e}), \mathcal{S}\rangle$

We can apply the induction hypothesis on e. According to the theorem, either of four possible cases holds for e. We prove that $\langle e.m(\overline{e}), \mathcal{S}\rangle$ reduces for each of these cases. We take the last two cases together as they each deal with expressions that got stuck during the reduction:

- e is a variable x. Then, there are two possibilities. If all the arguments of $m(\overline{e})$ are variables $\overline{y}$, then $\langle x.m(\overline{y}), \mathcal{S}\rangle$ reduces according to R-InvkP. This case has already been proven. The second case is when at least one e_i in $\overline{e}$ is not a variable. This case is proven below (in the next subsubcase).
- $\langle e, \mathcal{S}\rangle$ reduces to $\langle e', \mathcal{S}'\rangle$. In that case we can use the congruence rules to further reduce the expression. More precisely, $\langle e.m(\overline{e}), \mathcal{S}\rangle$ reduces by RC-Invk-Recv to $\langle e'.m(\overline{e}), \mathcal{S}'\rangle$. This finishes the case.
- The expression e is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. In that case, the method will never be invoked, and $\langle e.m(\overline{e}), \mathcal{S}\rangle$ will also be stuck due to the same problem. This finishes the case.

SubSubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x.m(\overline{e}), \mathcal{S}\rangle$

In that case, we can apply the induction hypothesis on each of the actual arguments of $m(\overline{e})$. Fix such an argument e_i without loss of generality. This expression e_i reduces according to either of the four reduction paths enumerated in the theorem. We prove that $\langle x.m(\overline{e}), \mathcal{S}\rangle$ reduces in each of these cases (the last two cases are taken together):

- e_i is a variable y. If this holds for all e_i in $\overline{e}$, then the expression reduces by R-InvkP. This case is already proven above. However, if the statement does not hold, then this means that there exists at least one e_j that is not a variable. This expression reduces by one of the other cases.
- $\langle e_i, \mathcal{S}\rangle \rightarrow \langle e_i', \mathcal{S}'\rangle$. Then, we can use the congruence rules to reduce $\langle x.m(\overline{e}), \mathcal{S}\rangle$. Taking RC-Invk-Arg, we have that $\langle x.m(e_1, \ldots, e_j, \ldots, e_n), \mathcal{S}\rangle$ reduces to $\langle x.m(e_1, \ldots, e_i', \ldots, e_n), \mathcal{S}'\rangle$. Thus, the expression reduces according to the second reduction path of the theorem, as such finishing the case.
- e_i is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. Either way, method invocation will not be executed. The expression $x.m(\overline{e})$ will be stuck because of the same problem. This finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle$**new** $C(\overline{e}), \mathcal{S}\rangle$

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle$**new** $C(\overline{x}), \mathcal{S}\rangle$.

This expression reduces by R-New for some $v_0 \notin dom(\mathcal{S})$, finishing the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle$**new** $C(\overline{e}), \mathcal{S}\rangle$.

We can apply the induction hypothesis on the arguments passed to the constructor. Without loss of generality, fix such an expression e_i. By the induction hypothesis, $\langle e_i, \mathcal{S}\rangle$ reduces by either of four reduction paths. We prove that the theorem holds for **new** $C(\overline{e})$ for each of these paths:

- e_i is a variable x_i. If this holds for all e_i in $\overline{e}$, then $\langle$**new** $C(\overline{e}), \mathcal{S}\rangle \equiv \langle$**new** $C(\overline{x}, \mathcal{S}\rangle$; which reduces by R-New. This case is already proven.
- $\langle e_i, \mathcal{S}\rangle \rightarrow \langle e_i', \mathcal{S}'\rangle$. Then, we can use the congruence rules to reduce the expression. The expression $\langle$**new** $C(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle$ reduces to $\langle$**new** $C(e_1, \ldots, e_i', \ldots, e_n), \mathcal{S}'\rangle$ by RC-New-Arg. This finishes the case.
- $\langle e_i, \mathcal{S}\rangle$ is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. Either way, $\langle$**new** $C(\overline{e}), \mathcal{S}\rangle$ is also stuck, as there exists an e_i in $\overline{e}$ that cannot reduce. This finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle let\ x = e_y\ in\ e, \mathcal{S}\rangle$

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle let\ x = y\ in\ e, \mathcal{S}\rangle$ Then, the expression $\langle let\ x = y\ in\ e, \mathcal{S}\rangle$ reduces to $\langle [y/x]e, \mathcal{S}'\rangle$ by R-Let and this finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle let\ x = e_y\ in\ e, \mathcal{S}\rangle$

In this case, we can apply the induction hypothesis on e_y. Thus, there are four possibilities for e_y (we take the last two possibilities together):

- e_y is a variable z. This case is already proven.
- $\langle e_y, \mathcal{S}\rangle \rightarrow \langle e'_y, \mathcal{S}'\rangle$. Then, we can use the congruence rules to further reduce the expression. In this case, $\langle let\ x = e_y\ in\ e, \mathcal{S}\rangle$ reduces to $\langle let\ x = e'_y\ in\ e, \mathcal{S}'\rangle$ by RC-Let.
- e_y is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. In this case, "$let\ x = e_y\ in\ e$" will also be stuck because e_y must first reduce before the entire expression can reduce.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle (U)e, \mathcal{S}\rangle$

SubCase $\langle (U)e, \mathcal{S}\rangle \equiv \langle (U)x, \mathcal{S}\rangle$

Substituting e_0 with $(U)x$ in the theorem, we get $\vdash \langle (U)x, \mathcal{S}\rangle : T$. From T-State, we get $\vdash \mathcal{S} : \Gamma$ and $\Gamma \vdash (U)x : T$. From $\vdash \mathcal{S} : \Gamma$, we get by T-Store that $\vdash_T$: $\mathcal{S} : \Gamma$, which means that the typing of the store $\mathcal{S}$ is consistent with the typing environment Γ. This is enforced by T-StoreTyp. Consequently, we have $\Gamma \vdash \mathcal{S}_T(y) : T$ where $T <: \Gamma(y)$ for all $y \in dom(\Gamma)$.

From T-UCast, T-DCast, and T-SCast, we have $\Gamma \vdash x : T_0$. Because the typing of the store is consistent with Γ, this means that $\mathcal{S}(x) = \langle T_0, \mathcal{M}\rangle$ for some $\mathcal{M}$. The remainder of this case is not concerned with the mapping $\mathcal{M}$ because type casting does not manipulate any fields. We work with the typing function of the store, i.e. $\mathcal{S}_T(x) = T_0$.

There are three possible reduction paths, based on the relationship between T_0 and U:

- $T_0 <: U$. Then, $\mathcal{S}_T(x) <: U$ and the expression reduces by R-Cast. This finishes the case.
- $T_0 \not<: U$. Then, no evaluation step can take place, as the premise for R-Cast is not satisfied. Thus, the program gets stuck in a bad cast. This finishes the case.
- T_0 and U are unrelated. Then, no evaluation step can take place, as the premise for R-Cast is not satisfied. Thus, the program gets stuck in a bad cast. This finishes the case.

SubCase $\langle (U)e_0, \mathcal{S}\rangle \equiv \langle (U)e, \mathcal{S}\rangle$ Then, we can apply the induction hypothesis on e. There are four cases. We prove that the theorem is true for each of these cases. The last two possibilities are taken together as they both deal with an expression that got stuck:

- e is a variable x. Then $\langle (U)e, \mathcal{S}\rangle \equiv \langle (U)x, \mathcal{S}\rangle$ and this case is already proven.
- $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}\rangle$. Then, we can use the congruence rules to further reduce the expression and $\langle (U)e, \mathcal{S}\rangle$ reduces to $\langle (U)e', \mathcal{S}\rangle$ by RC-Cast.
- The expression e is stuck due to the occurence of a $fail$ predicate, or due to a failed dynamic downcast. Then, $(U)e$ is also stuck because e must reduce before the cast can be applied.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ e_s.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$

We prove this case as follows. First, we prove the trivial case in which both the left-hand side (LHS) and the right-hand side (RHS) have reduced to values x and y. Then, we look at the more general case in which the LHS has reduced. Finally, a last subcase proves the most general case in which both the LHS and the RHS are unreduced subexpressions.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ x.f = y\ then\ e_{next}, \mathcal{S}\rangle$

We can substitute e_0 with $set\ x.f = y\ then\ e_{next}$ in the theorem premise. Then, we have $\vdash \langle set\ x.f = y\ then\ e_{next}, \mathcal{S}\rangle : T$. By T-State, we have $\vdash \mathcal{S} : \Gamma$ and $\Gamma \vdash set\ x.f = y\ then\ e_{next} : T$. Because $\vdash \mathcal{S} : \Gamma$ holds, we have by T-Store that $\vdash_T \mathcal{S} : \Gamma$. This means that the typing of the store $\mathcal{S}$ is consistent with the typing environment Γ.

By T-Set, we have $\Gamma \vdash x :$ **singleton** C. Because the typing of the store $\mathcal{S}$ is consistent with Γ, we have $\mathcal{S}(x) = \langle$**singleton** $C, \mathcal{M}\rangle$. Moreover, the mapping $\mathcal{M}$ is a *field* mapping because x is a **singleton**. Thus, $\mathcal{M} = \mathcal{F}$ and by T-FMap, we have that this field mapping is consistent:

- $fields(C) = \overline{T}\ \overline{f}$
- $\forall f_i \in fields(C) : \mathcal{S}_T(\mathcal{F}(f_i)) <: T_i$ where $\mathcal{F} : f_i \rightarrow v_i$

Thus, we have $\mathcal{S}(x) = \langle$**singleton** $C, \mathcal{F}\rangle$ and the case is finished because this expression ($\langle set\ x = y\ then\ e_{next}, \mathcal{S}\rangle$) reduces by R-Set.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$

We can apply the induction hypothesis on e_v. Thus, $\vdash \langle e_v, \mathcal{S}\rangle : T_0$ and there are four cases. We prove that the theorem holds for $\vdash \langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle : T$ by considering each case. The two last cases are taken together as they both deal with expressions that got stuck:

- e_v is a variable y. This case is already proven.
- $\langle e_v, \mathcal{S}\rangle \rightarrow \langle e_v, \mathcal{S}'\rangle$. Then, we can apply one of the congruence rules to further reduce the expression. Indeed, by RC-Set-R, the expression $\langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$ reduces to $\langle set\ x.f = e'_v\ then\ e_{next}, \mathcal{S}'\rangle$. This finishes the case.
- e_v is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. Then, $set\ x.f = e_v\ then\ e_{next}$ is also stuck by the same failure, and this finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ e_s.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$

We can apply the induction hypothesis on e_s. Thus, $\vdash \langle e_s, \mathcal{S}\rangle : T_0$ and there are four cases. We prove that the theorem holds for $\vdash \langle set\ e_s.f = e_v\ then\ e_{next}, \mathcal{S}\rangle : T$ by considering each case. The two last cases are taken together as they both deal with expressions that got stuck:

- e_s is a variable x. Then, we have $\langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$ and this case is already proven.
- $\langle e_s, \mathcal{S}\rangle \rightarrow \langle e_s, \mathcal{S}'\rangle$. Then, we can apply one of the congruence rules to further reduce the expression. Indeed, by RC-Set-L, the expression $\langle set\ e_s.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$ reduces to $\langle set\ e'_s.f = e_v\ then\ e_{next}, \mathcal{S}'\rangle$. This finishes the case.
- e_s is stuck due to the occurence of the $fail$ predicate, or due to a failed dynamic downcast. Then, $set\ e_s.f = e_v\ then\ e_{next}$ is also stuck by the same failure, and this finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e\ \textbf{where}\ e_b, \mathcal{S}\rangle$

First, we prove the cases in which the argument of the **where** operation has reduces to a value (**true** or **false**). Then, we prove the case in which the argument is an unreduced expression e_b. Finally, we prove the case in which both the receiver and the argument of the **where** operation are unreduced expressions e and e_b.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x\ \textbf{where}\ true, \mathcal{S}\rangle$

In that case, the expression reduces by R-Wh-True and the result is $\langle\langle v\ \textbf{where}\ e_{tmp}, \mathcal{S}\rangle\rangle$ where e_{tmp} was pushed on a temprary stack during the initialization of the **where**-algorithm.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle y\ \textbf{where}\ false, \mathcal{S}\rangle$

In that case, the expression reduces by R-Wh-False and the result is $\langle\langle v\ \textbf{where}\ e_{tmp}, \mathcal{S}\rangle\rangle$ where e_{tmp} was pushed on a temprary stack during the initialization of the **where**-algorithm.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x\ \textbf{where}\ e_b, \mathcal{S}\rangle$

There are two reduction paths, depending on the contents of the reference mapping $\mathcal{R}$.

- $\mathcal{R} = \emptyset$. Then, the expression reduces by R-Wh-Stop. This means that every pool member $v \in \mathcal{R}$ has been checked for the condition e_b and the algortihm ends. The result after the reduction is a variable v.
- $\mathcal{R} \neq \emptyset$. Then, the expression reduces by R-Wh-Iter. This means that the **where** algorithm takes another iteration step for another $v \in \mathcal{R}$. The result after the reduction step is $\langle v\ \textbf{where}\ [w/v]e_{tmp}, \mathcal{S}\rangle$ for some $w \in \mathcal{R}$.

Either way, the expression takes a reduction step, thus finishing the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle e \textbf{ where } e_b, \mathcal{S}\rangle$

We can apply the induction hypothesis on e. This means that there are four cases (we take the last two cases together as both cases deal with an expression that got stuck):

- e is a variable v. This case was already proven in the cases above.
- $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$. Then, we can apply one of the congruence rules. Indeed, by R-WHERE-RECV, we have for $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$ that $\langle e \textbf{ where } e_b, \mathcal{S}\rangle \rightarrow \langle e' \textbf{ where } e_b, \mathcal{S}'\rangle$. This finishes the case.
- e got stuck. Then, e **where** e_b is also stuck because e must first reduce to a variable v before we can further reduce this expression.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e \textbf{ orderby } e_c, \mathcal{S}\rangle$

We first consider the case in which both the receiver and the argument of the **orderby** operation have reduced to variables x and y. Then, we consider the case in which the argument e_c is an unreduced expression. Finally, we consider the case in which both the receiver and the argument are unreduces expressions.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x \textbf{ orderby } y, \mathcal{S}\rangle$

This expression reduces by R-ORD-INT and the result is n expression $\langle v \textbf{ orderby } e_c, \mathcal{S}\rangle$ where e_c was pushed on a temporary stack at the beginning of the execution of the **orderby** algorithm.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x \textbf{ orderby } e_c, \mathcal{S}\rangle$

There are two possible reduction paths, depending on the contents of $\mathcal{R}_{tmp}$, which is a temporary copy of $\mathcal{R}$ from which a pool member is extracted each iteration step:

- $\mathcal{R} = \emptyset$. Then, the expression reduces by R-STOP. This means that every pool member from $\mathcal{R}$ has been processed by the algorithm.
- $\mathcal{R} \neq \emptyset$. Then, the expression reduces by R-ITER. This means that there are still pool members in $\mathcal{R}$ that have not yet been processed by the algorithm. Thus, we need an extra iteration step.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle e \textbf{ orderby } e_c, \mathcal{S}\rangle$

Again, we can apply the induction hypothesis on e. There are four cases (we take the last two cases together as both cases deal with an expression that got stuck):

- e is a variable v. This case was already proven in the cases above.
- $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$. Then, we can apply one of the congruence rules. Indeed, by R-ORDERBY-RECV, we have for $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$ that $\langle e \textbf{ orderby } e_c, \mathcal{S}\rangle \rightarrow \langle e' \textbf{ orderby } e_c, \mathcal{S}'\rangle$. This finishes the case.
- e got stuck. Then, e **orderby** e_c is also stuck because e must first reduce to a variable v before we can further reduce this expression.

□

D Subject Reduction

Given $\vdash \langle e_0, \mathcal{S}\rangle : T$ and $\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle$,
then $\vdash \langle e_0', \mathcal{S}'\rangle : U$ for some $U <: T$.

Proof By induction on $\langle e_0, \mathcal{S}\rangle \rightarrow \langle e_0', \mathcal{S}'\rangle$.

Proof strategy. This proof considers each expression in turn. For each case, we prove that the reduced expression $\langle e_0', \mathcal{S}'\rangle$ is in a consistent state, i.e. that T-State holds. In order to prove this, it is sufficient to prove two propositions for each reduced expression $\langle e_0', \mathcal{S}'\rangle$:

- $\vdash \mathcal{S}' : \Gamma$. The new store is consistent with the typing environment.
- $\Gamma \vdash : e_0' : U$. The reduced expression has type $U <: T$ in Γ.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle x, \mathcal{S}\rangle$

The expression is a variable, so no reduction step can be taken.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e.f_i, \mathcal{S}\rangle$

First, we prove the case in which e has reduced to a variable x. In that case, we can use R-Get to reduce the expression. In a second subcase, we consider the general case in which e is not reduced to a variable. Then, we can use one of the congruence rules to further reduce the expression.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x.f_i, \mathcal{S}\rangle$

We have $\langle x.f_i, \mathcal{S}\rangle : T_i$. The expression $\langle x.f_i, \mathcal{S}\rangle$ reduces by R-Fd. The result is $\langle v_i, \mathcal{S}'\rangle$ where $v_i = \mathcal{F}(f_i)$. For this expression, we have to prove that T-State holds, i.e. that $\vdash \langle v_i, \mathcal{S}'\rangle : U_i$ for some $U_i < T_i$. This means we have to prove that (1) $\vdash \mathcal{S}' : \Gamma$ and that (2) $\Gamma \vdash x.f_i : U_i$ for some $U_i <: T_i$.

1. By R-Fd, the expression $\langle x.f_i, \mathcal{S}\rangle$ reduces to $\langle v_i, \mathcal{S}\rangle$. Thus, the store does not change during reduction, and $\mathcal{S} = \mathcal{S}'$. By applying T-State on the original expression $\langle x.f_i, \mathcal{S}\rangle$, we have $\vdash \mathcal{S} : \Gamma$. Thus, $\vdash \mathcal{S}' : \Gamma$ because $\mathcal{S} = \mathcal{S}'$.
2. We have to prove that $\Gamma \vdash v_i : U_i$ for some $U_i <: T_i$. We do this by tracing back how v_i was added to the field mapping $\mathcal{F}$. There are two possibilities: (1) the field was explicitly assigned using R-Set and (2) the field was initialized during object construction by R-New. We consider each case separately.
   (a) *The field was assigned during object construction, without ever being reassigned by* R-Set. Then, the evaluation step of R-New added a new value v_0 to the store with a field mapping $\mathcal{F}$. This field mapping is a function $\mathcal{F} : f_i \rightarrow v_i$, for each $f_i \in fields(C)$. Moreover, f_i is also the i^{th} actual argument passed to the constructor. By T-New, we know that the typing of these arguments is $\Gamma \vdash \overline{v} : \overline{U}$ for $fields(C) = \overline{T}\ \overline{f}$ with $\overline{U} <: \overline{T}$. Thus, for f_i we have assigned an expression v_i with $\Gamma \vdash v_i : U_i <: T_i$. Having $\Gamma \vdash v_i : U_i$ for some $U_i <: T_i$ finishes the case.

(b) *The field is reassigned by* R-SET. In this case, the field was assigned during object construction by R-NEW, but this assignment was overwritten during an explicit field assignment, which caused a modification of the field mapping $\mathcal{F}$. By R-SET, we know that the field mapping $\mathcal{F}$ is modified by binding y to f_i (this is denoted as $\mathcal{F}[f_i \rightarrow y]$). Thus $\mathcal{F}(f_i) = v_i = y$. Filling in the facts in T-SET gives $\Gamma \vdash x : \textbf{singleton}\ C$ with $fields(C) = \overline{T}\ \overline{f}$. Thus, f_i has type T_i. We also know by T-SET that $\Gamma \vdash y : S_i$ and $S_i <: T_i$. Having $\Gamma \vdash y : S_i$ for some $S_i <: T_i$ finishes the case by choosing $U = S_i$.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle e.f_i, \mathcal{S}\rangle$

We can apply the induction hypothesis on e. This means that $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$ where $\vdash \langle e, \mathcal{S}\rangle : T$ and $\vdash \langle e', \mathcal{S}'\rangle : U$ for some $U <: T$. Let $T = Q_T\ C_T$ and $U = Q_U\ C_U$ without loss of generality. Then, because $U <: T$, we have by S-TYP that $Q_U <: Q_T \wedge C_U <: C_T$.

We also know by FD-C and FD-OBJ that $fields(C_U)$ is equal to those fields that C_U introduces itself, together with those fields that C_U inherits from its direct and indirect superclasses. Because $C_U <: C_T$, this can be written as $fields(C_U) = fields(C_T),\ \overline{V}\ \overline{g}$ for some $\overline{V}\ \overline{g}$. Thus, $fields(C_T) \subseteq fields(C_U)$. We know that $f_i \in fields(C_T)$ by T-FIELD, so by FD-C, $f_i \in fields(C_U)$ and f_i has the same type, say T_i, in C_T and C_U because it is the same field.

The theorem premise states that $\vdash \langle e.f_i, \mathcal{S}\rangle : T_i$, which means by T-STATE that $\Gamma \vdash e.f_i : T_i$ where T_i is the type of f_i (T-GET). Thus, $\vdash \langle e'.f_i, \mathcal{S}'\rangle : T_i$. Let $U_i = T_i$ and the case is finished by S-REFL which states that the subtyping relation is reflexive, that is $T_i <: T_i$.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ e_f.f_i = e_v\ then\ e_{next}, \mathcal{S}\rangle$

First, we consider the trivial case in which both the receiver x and the target y have reduced to values. Then, we can use R-SET to reduce the expression. In a second subcase, we consider the case in which the receiver x has reduced to a variable, whereas the target is still an expression to be evaluated (e_v). In that case, we use one of the congruence rules to further reduce this subexpression. A third subcase considers the case in which both the receiver and the target are subexpressions e_f and e_v. Again, we use one of the congruence rules to reduce this expression.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ x.f_i = y\ then\ e_{next}, \mathcal{S}\rangle$

The theorem premise states that $\vdash \langle set\ x.f_i = y\ then\ e_{next}, \mathcal{S}\rangle : T$. This expression reduces by R-FD. Thus, we have to prove that the result of applying R-FD gives $\vdash \langle e_{next}, \mathcal{S}[x \rightarrow \langle \textbf{singleton}\ C, \mathcal{F}[f_i \rightarrow y]\rangle]\rangle : U$ for some $U <: T$. By T-STATE, it is sufficient that (1) $\Gamma \vdash e_{next} : U$ for some $U <: T$ and (2) that $\vdash \mathcal{S}' : \Gamma$ where $\mathcal{S}' = \mathcal{S}[x \rightarrow \langle \textbf{singleton}\ C, \mathcal{F}[f_i \rightarrow y]\rangle]$

1. The first statement is easy to prove. We know from the theorem premise that $\vdash \langle set\ x.f_i = y\ then\ e_{next}, \mathcal{S}\rangle : T$. Using T-State, we get that $\Gamma \vdash set\ x.f_i = y\ then\ e_{next} : T$. We can now apply T-Set on this expression, giving $\Gamma \vdash e_{next} : T$. Thus, letting $U = T$ finishes the case because S-Ref states that the subtyping relation is reflexive, i.e. $T <: T$.
2. To prove the second statement, we must show that the new store $\mathcal{S}'$ is still consistent with the typing environment Γ. In other words, we must show that R-Fd does not allow field assignments which violate the expression typing rules. By T-Store, it is sufficient to prove two statements for $\mathcal{S}'$. First, we have to prove that the typing of $\mathcal{S}'$ is valid, i.e. $\vdash_T \mathcal{S}' : \Gamma$. Second, we have to prove that the mapping $\mathcal{M}$ of each store entry is valid, i.e. $\forall x \in dom\Gamma : \mathcal{S}_{\mathcal{M}}(x)\ OK$.
   - *Store Typing* ($\vdash_T \mathcal{S}' : \Gamma$). By T-STyp, it is sufficient to prove that $\mathcal{S}'_T(v) <: \Gamma(x)$ for all $v \in dom(\Gamma)$. This is straightforward for all $v \neq x$ because R-Fd did not change the typing of those v. Thus, $\mathcal{S}_T(v) = \mathcal{S}'_T(v)$. For x, we have $\mathcal{S}(x) = \langle \textbf{singleton}\ C, \mathcal{F}\rangle$ and after applying R-Fd, we have $\mathcal{S}'(x) = \langle \textbf{singleton}\ C, \mathcal{F}'\rangle$. Thus, for x, we also have $\mathcal{S}_T(x) = \mathcal{S}'_T(x)$. Consequently, we have $\forall v \in dom(\Gamma) : \mathcal{S}_T(v) = \mathcal{S}'_T(v)$. Knowing from the theorem premise that $\vdash_T \mathcal{S} : \Gamma$ and $\mathcal{S}_T = \mathcal{S}'_T$, we have $\vdash_T \mathcal{S}' : \Gamma$.
   - *Store Mapping* ($\forall x \in dom(\Gamma) : \mathcal{S}_{\mathcal{M}}(x)\ OK$). Let $\mathcal{F}'$ be the new field mapping after applying R-Fd. That is, $\mathcal{F}' = \mathcal{F}[f_i \rightarrow y]$. Because R-Fd only changes the field mapping $\mathcal{F}$ of x, it is sufficient to prove for x that $\mathcal{S}'_{\mathcal{F}}(x)\ OK$. Thus, by T-FMap, it is sufficient to prove for $\mathcal{S}'(x) = \langle \textbf{singleton}\ C, \mathcal{F}\rangle$ with $fields(C) = \overline{T}\ \overline{f}$ that for all $f_j \in fields(C)$ we have $\mathcal{S}'_T(\mathcal{F}(f_j)) <: T_i$. There are two cases (remember that f_i is the field to which y is assigned):
     - $f_j \neq f_i$. In that case, f_j is not reassigned by R-Fd so those parts of the field mapping are the same in $\mathcal{F}$ and $\mathcal{F}'$. Consequently, we have $\mathcal{S}'_T(\mathcal{F}(f_j)) = \mathcal{S}_T(\mathcal{F}(f_j)) <: T_j$.
     - $f_j = f_i$. In that case, f_i is reassigned by R-Fd and we have $\mathcal{F}' : f_i \rightarrow y$. Thus, we must prove that $\mathcal{S}'_T(y) <: T_i$. By T-Set, we have for $\Gamma \vdash set\ x.f = y\ then\ e_{next} : T_{next}$ and $fields(C) = \overline{T}\ \overline{f}$ that $\Gamma \vdash y : S$ for $S <: T_i$. But we already know that the typing of the new store $\mathcal{S}'$ is consistent with Γ, i.e. $\vdash \mathcal{S}' : \Gamma$. By T-STyp, this means that $\mathcal{S}'_T(y) <: S$. Then, the case is finished by letting $U = \mathcal{S}'_T(y)$. Indeed, by S-Trans we have $U <: T_i$ from $U <: S$ and $S <: T_i$.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle$

We can apply the induction hypothesis on e_v. Then, $\vdash \langle e_v, \mathcal{S}\rangle : T$ and $\langle e_v, \mathcal{S}\rangle \rightarrow \langle e'_v, \mathcal{S}'\rangle$ with $\vdash \langle e'_v, \mathcal{S}'\rangle : U$ for some $U <: T$. We can now apply one of the congruence rules (RC-Set-R). This gives $\langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle \rightarrow \langle set\ x.f = e'_v\ then\ e_{next}, \mathcal{S}'\rangle$.

The theorem premise states that $\vdash \langle set\ x.f = e_v\ then\ e_{next}, \mathcal{S}\rangle : W$. We have to show that $\vdash \langle set\ x.f = e'_v\ then\ e_{next}, \mathcal{S}'\rangle : V$ for some $V <: W$. We already

know from $\vdash \langle e'_v, \mathcal{S}' \rangle : U$ that $\mathcal{S}' : \Gamma$, so by T-State, we only have to show that $\Gamma \vdash \mathit{set}\ x.f = e'_v\ \mathit{then}\ e_{next} : V$ for some $V <: W$. By applying T-Set on $\Gamma \vdash \mathit{set}\ x.f = e_v\ \mathit{then}\ e_{next} : W$ we know that $\Gamma \vdash e_{next} : W$. By applying T-Set on $\Gamma \vdash \mathit{set}\ x.f = e'_v\ \mathit{then}\ e_{next} : V$, we have that $\Gamma \vdash e_{next} : V$. So, $V = W$ and the case is finished by S-Refl.

SubCase $\langle e_0, \mathcal{S} \rangle \equiv \langle \mathit{set}\ e_f.f = e_v\ \mathit{then}\ e_{next}, \mathcal{S} \rangle$

We can apply the induction hypothesis on e_f. Then, $\vdash \langle e_f, \mathcal{S} \rangle : T$ and $\langle e_f, \mathcal{S} \rangle \rightarrow \langle e'_f, \mathcal{S}' \rangle$ with $\vdash \langle e'_f, \mathcal{S}' \rangle : U$ for some $U <: T$. We can now apply one of the congruence rules (RC-Set-L). This gives $\langle \mathit{set}\ e_f.f = e_v\ \mathit{then}\ e_{next}, \mathcal{S} \rangle \rightarrow \langle \mathit{set}\ e'_f.f = e_v\ \mathit{then}\ e_{next}, \mathcal{S}' \rangle$.

The theorem premise states that $\vdash \langle \mathit{set}\ e_f.f = e_v\ \mathit{then}\ e_{next}, \mathcal{S} \rangle : W$. We have to show that $\vdash \langle \mathit{set}\ e'_f.f = e_v\ \mathit{then}\ e_{next}, \mathcal{S}' \rangle : V$ for some $V <: W$. We already know from $\vdash \langle e'_f, \mathcal{S}' \rangle : U$ that $\mathcal{S}' : \Gamma$, so by T-State, we only have to show that $\Gamma \vdash \mathit{set}\ e'_f.f = e_v\ \mathit{then}\ e_{next} : V$ for some $V <: W$. By applying T-Set on $\Gamma \vdash \mathit{set}\ e_f.f = e_v\ \mathit{then}\ e_{next} : W$ we know that $\Gamma \vdash e_{next} : W$. By applying T-Set on $\Gamma \vdash \mathit{set}\ e'_f.f = e_v\ \mathit{then}\ e_{next} : V$, we have that $\Gamma \vdash e_{next} : V$. So, $V = W$ and the case is finished by S-Refl.

Case $\langle e_0, \mathcal{S} \rangle \equiv \langle \mathit{let}\ x = e_y\ \mathit{in}\ e_{let}, \mathcal{S} \rangle$

First, we prove the case in which e_y has reduced to a value y. This case depends on the preservation lemma which was proven earlier in this paper. Then, we consider the general case where e_y reduces to an expression e'_y. This case depends on the induction hypothesis for *let* expressions.

SubCase $\langle e_0, \mathcal{S} \rangle \equiv \langle \mathit{let}\ x = y\ \mathit{in}\ e_{let}, \mathcal{S} \rangle$

We have $\vdash \langle \mathit{let}\ x = y\ \mathit{in}\ e_{let}, \mathcal{S} \rangle : T$. The expression $\langle \mathit{let}\ x = y\ \mathit{in}\ e_{let}, \mathcal{S} \rangle$ reduces by R-Let to $\langle [y/x]e_{let}, \mathcal{S} \rangle$. Thus, we have to prove that $\vdash \langle [y/x]e_{let}, \mathcal{S} \rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove that (1) $\vdash \mathcal{S} : \Gamma$ and (2) $\Gamma \vdash [y/x]e_{let} : U$ for some $U <: T$.

1. $\vdash \mathcal{S} : \Gamma$. The theorem premise stated that $\vdash \langle \mathit{let}\ x = y\ \mathit{in}\ e_{let}, \mathcal{S} \rangle$ and by T-State, this means that $\vdash \mathcal{S} : \Gamma$. This statement still holds after reduction by R-Let because the reduction step does not manipulate the store $\mathcal{S}$ (thus, $\mathcal{S} = \mathcal{S}'$ in the theorem).
2. $\Gamma \vdash [y/x]e_{let} : U$ for some $U <: T$. By T-Let, we have (1) $\Gamma, x : V \vdash e : T$ and (2) $\Gamma \vdash y : V$. Then, we can apply the term substitution lemma on $\Gamma, x : V \vdash e : T$. The result is $\Gamma \vdash [y/x]e_0 : U$ for some $U <: T$. This finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle let\ x = e_y\ in\ e_{let}, \mathcal{S}\rangle$

We can apply the induction hypothesis on e_y. Then, $\vdash \langle e_y, \mathcal{S}\rangle : W$ and $\langle e_y, \mathcal{S}\rangle \rightarrow \langle e'_y, \mathcal{S}'\rangle$ with $\vdash \langle e'_y, \mathcal{S}'\rangle : V$ for some $V <: W$.

The theorem premise states that $\vdash \langle let\ x = e_y\ in\ e_{let}, \mathcal{S}\rangle$. Knowing that $\langle e_y, \mathcal{S}\rangle \rightarrow \langle e'_y, \mathcal{S}'\rangle$, we can apply RC-Let to reduce the expression. This gives $\langle let\ x = e_y\ in\ e_{let}, \mathcal{S}\rangle \rightarrow \langle let\ x = e'_y\ in\ e_{let}, \mathcal{S}'\rangle$. Thus, in this case, we have to prove that $\vdash \langle let\ x = e'_y\ in\ e_{let}, \mathcal{S}'\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove two statements: (1) $\vdash \mathcal{S}' : \Gamma$ and (2) $\Gamma \vdash let\ x = e_y\ in\ e_{let} : U$ for some $U <: T$.

1. $\vdash \mathcal{S}' : \Gamma$. By applying the induction hypothesis on $\langle e_y, \mathcal{S}\rangle$, we got that $\vdash \langle e'_y, \mathcal{S}'\rangle : V$. By T-State, this gives $\vdash \mathcal{S}' : \Gamma$.
2. $\Gamma \vdash let\ x = e'_y\ in\ e_{let} : U$ for some $U <: T$. We can apply T-Let on the original expression ($\Gamma \vdash let\ x = e_y\ in\ e_{let} : T$). This gives $\Gamma, x : W \vdash e_{let} : T$ for $\Gamma \vdash e_y : W$. We also have by the induction hypothesis that $\langle e'_y, \mathcal{S}'\rangle$. By T-State, this means that $\Gamma \vdash e'_y : V$. We can now apply T-Let on $\Gamma \vdash e'_y : V$ and $\Gamma, x : V \vdash e_{let} : T$. The result is $\Gamma \vdash let\ x = e'_y\ in\ e_{let} : T$. Letting $U = T$ finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle (V)e, \mathcal{S}\rangle$

We first consider the case in which e has reduced to a value x and then consider the more general case using one of the congruence rules.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle (V)x, \mathcal{S}\rangle$

In this case, we have $\vdash \langle (V)x, \mathcal{S}\rangle : T$. There are two possibilities. First, the program may be stuck because the cast is invalid. Then, R-Cast will not initiate an evaluation step because its precondition ($\mathcal{S}_T(x) <: V$) is not satisfied. Consequently, there is nothing to prove because the second precondition ($\langle e_0, \mathcal{S}\rangle \rightarrow \langle e'_0, \mathcal{S}'\rangle$) of the theorem is not satisfied. Second, the program is not stuck because the cast is valid. In that case, the precondition of R-Cast is satisfied, and an evaluation step is taken. This means that:

- $\mathcal{S}_T(x) <: T$
- $\langle (T)x, \mathcal{S}\rangle \rightarrow \langle x, \mathcal{S}\rangle$
- $\vdash \langle (V)x, \mathcal{S}\rangle : T$. Then, by T-State, we have $\Gamma \vdash (V)x : T$. By T-UCast, T-DCast, and T-SCast this means that $V = T$.

We have to prove that $\vdash \langle x, \mathcal{S}\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove that (1) $\vdash \mathcal{S} : \Gamma$ and (2) $\Gamma \vdash x : U$ for some $U <: T$.

1. $\vdash \mathcal{S} : \Gamma$. We have $\vdash \langle (T)x, \mathcal{S}\rangle : T$ and by T-State, this gives $\vdash \mathcal{S} : \Gamma$. The store is not changed during reduction, so $\mathcal{S}' = \mathcal{S}$ in the theorem premise.
2. $\Gamma \vdash x : U$ for some $U <: T$. We know that $\mathcal{S}_T(x) <: T$ otherwise the program would be stuck. Say, $T_x = \mathcal{S}_T(x)$. The premise of the theorem stated that $\vdash \langle (V)x, \mathcal{S}\rangle : T$ and we know that $V = T$. By T-State, this means that $\Gamma \vdash (T)x : T$. Knowing that $T_x <: T$, we can apply T-DCast. This gives $\Gamma \vdash x : T_x$. Thus, letting $U = T_x$ finishes the case because $T_x <: T$.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle (V)e, \mathcal{S}\rangle$

In this case, we can apply the induction hypothesis on e. Then, $\vdash \langle e, \mathcal{S}\rangle : B$ and $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$ with $\vdash \langle e', \mathcal{S}'\rangle : A$ for some $A <: B$. For $\langle (V)e, \mathcal{S}\rangle$, the theorem premises state that $\vdash \langle (V)e, \mathcal{S}\rangle : T$ and $\langle (V)e, \mathcal{S}\rangle \rightarrow \langle (V)e', \mathcal{S}'\rangle$. We have to prove that $\vdash \langle (V)e', \mathcal{S}'\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove that (1) $\mathcal{S} : \Gamma$ and that (2) $\Gamma \vdash (V)e' : U$ for some $U <: T$.

1. $\vdash \mathcal{S}' : \Gamma$. From $\vdash \langle e', \mathcal{S}'\rangle$, we have by T-State that $\vdash \mathcal{S}' : \Gamma$.
2. $\Gamma \vdash (V)e' : U$ for some $U <: T$. We already know that $\Gamma \vdash (V)e : T$. By T-UCast, T-DCast, and T-SCast, we get $V = T$. Thus, it is sufficient to prove that $\Gamma \vdash (T)e' : U$ for some $U <: T$. We get $\Gamma \vdash e' : A$ by applying T-State on $\vdash \langle e', \mathcal{S}'\rangle : A$. There are three possibilities, depending on the relationship between A and T:
   - $T <: A$. Then, we have a downcast and we can fill in T-DCast for $\Gamma \vdash e' : A$ and we get $\Gamma \vdash (T)e' : T$. Letting $U = T$ and then applying S-Ref finishes the case.
   - $A <: T$. Then, we have an upcast and we can fill in T-UCast for $\Gamma \vdash e' : A$ and we get $\Gamma \vdash (T)e' : T$. Letting $U = T$ and then applying S-Ref finishes the case.
   - $A \not<: T \wedge T \not<: A$. Then, we have a cast to a type that is unrelated to the type of e' (the so-called *stupid cast*). We can fill in T-SCast for $\Gamma \vdash e' : A$ and we get $\Gamma \vdash (T)e' : T$. Letting $U = T$ and then applying S-Ref finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e.m(\overline{e}), \mathcal{S}\rangle$

First, we consider the case in which both the invocation target and the actual arguments of the method have reduces to values x and $\overline{y}$. Then, we can use R-InvkS to reduce the expression. We also consider the case for method invocation on **pool** and **sequence** references. In that case, we reduce the expression using R-InvkP. A second case considers expressions for which the invocation target has reduces, but (one of the) actual arguments is still a subexpression. We use the congruence rules to further reduce this expression. In the last case, we consider an expression for which both the invocation target and the actual arguments are subexpressions e and $\overline{e}$. Again, we use one of the congruence rules to reduce the expression.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x.m(\overline{y}), \mathcal{S}\rangle$

We have $\vdash \langle x.m(\overline{y}), \mathcal{S}\rangle : T$ and this expression reduces either by R-InvkS or R-InvkP. The choice, however, is deterministic because it depends on the type qualifier of T. Let $T = Q\ C$. If $Q = \textbf{singleton}$, then $\mathcal{S}(x) = \langle \textbf{singleton}\ C, \mathcal{F}\rangle$ and R-InvkS is the only rule that matches. Conversely, if $Q = \textbf{pool}$ or $Q = \textbf{sequence}$, then $\mathcal{S}(x) = \langle \textbf{pool}\ C, \mathcal{R}\rangle$ and R-InvkP is the only rule that matches.

We will first prove the case for $Q = \textbf{singleton}$ and then indicate how this proof can be modified for $Q = \textbf{pool}$ and $Q = \textbf{sequence}$.

The theorem premise states that $\vdash \langle x.m(\overline{y}), \mathcal{S}\rangle : T$ and $\langle x.m(\overline{y}), \mathcal{S}\rangle$ reduces by R-InvkS to $\langle [\overline{y}/\overline{z}, x/\textbf{this}]e_0, \mathcal{S}\rangle$ where $mbody(m, C) = \overline{z}.e_0$. We have to prove that $\vdash \langle [\overline{y}/\overline{z}, x/\textbf{this}]e_0, \mathcal{S}\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove that (1) $\vdash \mathcal{S} : \Gamma$ and that (2) $\Gamma \vdash [\overline{y}/\overline{z}, x/\textbf{this}]e_0 : U$ for some $U <: T$.

- $\vdash \mathcal{S} : \Gamma$. The theorem premise states that $\vdash \langle x.m(\overline{y}), \mathcal{S}\rangle : T$, which means by T-State that $\vdash \mathcal{S} : \Gamma$. Moreover, $\mathcal{S} = \mathcal{S}'$ in the theorem premise because the store is not changed by R-InvkS.
- $\Gamma \vdash [\overline{y}/\overline{z}, x/\textbf{this}]e_0 : U$ for some $U <: T$. By applying T-State on the theorem premise, we get that $\Gamma \vdash x.m(\overline{y}) : T$ where $T = \textbf{singleton}\ C$. If we apply T-Invk on $\Gamma \vdash x.m(\overline{y}) : T$, we get $\Gamma \vdash x : T_0$ and $\Gamma \vdash \overline{y} : \overline{S}$. Moreover, T-Invk gives $mtype(m, T_0) = \overline{V} \to T$ and $\overline{S} <: \overline{V}$. The premise of R-InvkS gave $mbody(m, T_0) = \overline{z}.e_0$. Combining this with $mtype(m, T_0) = \overline{V} \to T$, we get that $\Gamma, \overline{z} : \overline{V} \vdash e_0 : T$. By T-Method, this also holds for subclasses of T_0. Indeed, for overriden methods, we require the types of the arguments and the return type to be the same as the types in the superclass. We can now apply the term substitution lemma with these premises:
  - $\Gamma, \overline{z} : \overline{V}, \textbf{this} : C_0 \vdash e_0 : T$
  - $\Gamma \vdash \overline{y} : \overline{S}$ with $\overline{S} <: \overline{V}$.

  The result is $\Gamma \vdash [\overline{y}/\overline{z}, x/\textbf{this} \vdash e_0] : U$ for some $U <: T$ and this finishes the case for $Q = \textbf{singleton}$.

Case for $Q = \textbf{pool}$ or $Q = \textbf{sequence}$

Instead of replacing **this** with x, we substitute **this** with $failover(v_0)$. There are two cases, depending on the result of $failover(v_0)$.

- *$failover(v_0)$ returns the fail predicate because no $v_i \in \mathcal{R}$ is available.*
  Then, we have $\Gamma \vdash [\overline{y}/\overline{z}, fail/this]e_0 : U$. If the **this** reference occurs in e_0, then the expression will get stuck in one of the reduction rules that follow this evaluation step. If **this** does not occur in e_0, then the substitution if equivalent to $\Gamma \vdash [\overline{y}/\overline{z}]e_0 : U$ where $U <: T$ by the substitution lemma.
- *$failover(v_0) = v_i$ for some $v_i \in \mathcal{R}$.*
  We know that $\vdash \langle x.m(\overline{y}), \mathcal{S}\rangle$, so by T-State, we have $\vdash \mathcal{S} : \Gamma$. Applying T-Store gives that the typing of $\mathcal{S}$ is consistent with Γ, i.e. that $\vdash_T \mathcal{S} : \Gamma$. This means that $\mathcal{R}$ behaves according to T-RMap. Then, having $\mathcal{S}(x) = \langle Q_0\ C_0, \mathcal{R}\rangle$, we have $\forall v \in \mathcal{R} : \mathcal{S}_T(v) <: \textbf{singleton}\ C_0$. Reusing the facts for $Q = \textbf{singleton}$ gives $\Gamma \vdash [\overline{y}/\overline{z}, v_i/\textbf{this} \vdash e_0] : U$ for some $U <: T$. This finishes the case for $Q = \textbf{pool}$ or $Q = \textbf{sequence}$.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x.m(\overline{e}), \mathcal{S}\rangle$

Then, we have $\vdash \langle x.m(\overline{e}), \mathcal{S}\rangle : T$. For a reduction $\langle x.m(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle \rightarrow \langle x.m(e_1, \ldots, e'_i, \ldots, e_n), \mathcal{S}'\rangle$, we have to prove that $\langle x.m(e_1, \ldots, e'_i, \ldots, e'_n), \mathcal{S}'\rangle : U$ for some $U <: T$. We can apply the induction hypothesis on e_i. Then, $\vdash \langle e_i, \mathcal{S}\rangle : T$ and $\langle e_i, \mathcal{S}\rangle \rightarrow \langle e'_i, \mathcal{S}'\rangle$ where $\vdash \langle e'_i, \mathcal{S}'\rangle : U$ for some $U <: T$. We can then apply RC-Invk-Arg to get $\langle x.m(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle \rightarrow \langle x.m(e_1, \ldots, e'_i, \ldots, e_n), \mathcal{S}'\rangle$. By T-State, it is sufficient to prove that (1) $\vdash \mathcal{S} : \Gamma$ and that (2) $\Gamma \vdash x.m(e_1, \ldots, e'_i, \ldots, e_n) : U$ for some $U <: T$.

1. $\vdash \mathcal{S}' : \Gamma$. From $\vdash \langle e'_i, \mathcal{S}\rangle$, we know by T-State that $\vdash \mathcal{S}' : \Gamma$.
2. $\Gamma \vdash x.m(e_1, \ldots, e'_i, \ldots, e_n) : U$ for some $U <: T$. Starting from $\vdash \langle x.m(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle : T$, we have by T-State that $\Gamma \vdash x.m(e_1, \ldots, e_i, \ldots, e_n) : T$. Applying T-Invk gives $\Gamma \vdash x : Q_0\ C_0$ and $mtype(m, C_0) = \overline{Y} \rightarrow T$. Thus, the formal arguments of m have types $\overline{Y}$. For the actual arguments passed to m, we know by T-Invk that $\Gamma \vdash \overline{e} : X$ with $\overline{X} <: \overline{Y}$. From $\vdash \langle e_i, \mathcal{S}\rangle : W$, we know by T-State that $\Gamma \vdash e_i : W$. Thus, $X_i = W$. From $\vdash \langle e_i, \mathcal{S}'\rangle : V$, we know by T-State that $\Gamma \vdash e'_i : V$ for some $V <: W$. Thus, $V <: X_i$. Also, $X_i <: Y_i$, so by S-Trans, we have $V <: Y_i$. We can now apply T-Invk with the following premises:
   - $\Gamma \vdash \overline{e} : \overline{X}$, letting out e_i.
   - $\Gamma \vdash e'_i : V$ with $V <: Y_i$ and $\Gamma \vdash x : Q_0\ C_0$
   - $mtype(m, C_0) : \overline{Y} \rightarrow T$

   The result is $\Gamma \vdash x.m(e_1, \ldots, e'_i, \ldots, e_n) : T$. Letting $U = T$ finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle e.m(\overline{e}), \mathcal{S}\rangle$

We have $\vdash \langle e.m(\overline{e}), \mathcal{S}\rangle$. For a reduction $\langle e.m(\overline{e}), \mathcal{S}\rangle \rightarrow \langle e'.m(\overline{e}), \mathcal{S}'\rangle$, we have to prove that $\vdash \langle e'.m(\overline{e}), \mathcal{S}'\rangle : U$ for some $U <: T$. To do this, we can apply the induction hypothesis on e. Then, $\vdash \langle e, \mathcal{S}\rangle : W$ and $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$ where $\vdash \langle e', \mathcal{S}'\rangle : V$ for some $V <: W$. Knowing that this expression reduces, we can apply RC-Invk-Recv on $\langle e.m(\overline{e}), \mathcal{S}\rangle$ to reduce this expression. Then, we have to prove that $\vdash \langle e'.m(\overline{e}), \mathcal{S}'\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove that (1) $\vdash \mathcal{S}' : \Gamma$ and that (2) $\Gamma \vdash e'.m(\overline{e}) : U$ for some $U <: T$.

1. $\vdash \mathcal{S} : \Gamma$. From $\vdash \langle e', \mathcal{S}'\rangle : V$, we know by T-State that $\vdash \mathcal{S} : \Gamma$.
2. $\Gamma \vdash e'.m(\overline{e}) : U$ for some $U <: T$. By T-State, we know for $\vdash \langle e.m(\overline{e}), \mathcal{S}\rangle : T$ that $\Gamma \vdash e.m(\overline{e}) : T$. Applying T-Invk gives $\Gamma \vdash e : Q_0\ C_0$ and $mtype(m, C_0) = \overline{S} \rightarrow T$. Thus, the formal arguments of m have types $\overline{S}$ in C_0. For the actual arguments passed to m, we know from T-Invk that $\Gamma \vdash \overline{e} : \overline{S}_e$ for some $\overline{S}_e <: \overline{S}$. By applying T-State on $\vdash \langle e, \mathcal{S}\rangle : W$, we get that $\Gamma \vdash e : T$. Thus, $W = Q_0\ C_0$. From $\vdash \langle e', \mathcal{S}'\rangle$, we know by T-State that $\Gamma \vdash e' : V$ for some $V <: W$. Thus, $V <: Q_0\ C_0$. We can now apply T-Invk with the following premises:
   - $\Gamma \vdash \overline{e} : \overline{S}_e$ and $\Gamma \vdash e' : V$
   - $mtype(m, V) = mtype(m, C_0) : \overline{Y} \rightarrow T$. This is enforced by T-Method.

   The result is $\Gamma \vdash e'.m(\overline{e}) : T$. Letting $U = T$ finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle \textbf{new } C(\overline{e}), \mathcal{S}\rangle$

We first consider the case in which the actual arguments passed to the constructor have all reduces to values $\overline{x}$. In that case, the expression reduces by R-New and a new variable x is added to the store $\mathcal{S}$. We prove that the store with this new value is still consistent with the tying environment Γ. Then, we consider the general case in which the actual arguments of the constructor are unreduces expressions $\overline{e}$. In that case, we use the congruence rules to further reduce the expression.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle \textbf{new } C(\overline{x}), \mathcal{S}\rangle$

We have $\vdash \langle \textbf{new } C(\overline{x}), \mathcal{S}\rangle : T$. This expression reduces by R-New and the result is $\vdash \langle x, \mathcal{S}[x \rightarrow \langle \textbf{singleton } C, \mathcal{F}[\overline{f} \rightarrow \overline{x}]\rangle]\rangle$. Thus, during reduction, the store $\mathcal{S}$ changes and $\mathcal{S}' = \mathcal{S}[x \rightarrow \langle \textbf{singleton } C, \mathcal{F}[\overline{f} \rightarrow \overline{x}]\rangle]$. We have to prove that $\vdash \langle x, \mathcal{S}[x \rightarrow \langle \textbf{singleton } C, \mathcal{F}[\overline{f} \rightarrow \overline{x}]\rangle]\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove that (1) $\vdash \mathcal{S}' : \Gamma'$, where $\Gamma' = \Gamma[x \rightarrow \textbf{singleton } C]$ and that (2) $\Gamma \vdash x : U$ for some $U <: T$.

- $\vdash \mathcal{S}' : \Gamma'$. Note that $\vdash \mathcal{S} : \Gamma$ and that the only change to $\mathcal{S}$ is the addition of x. We prove that the new store $\mathcal{S}'$ is consistent with the typing environment by showing (1) that $\vdash_T \mathcal{S}' : \Gamma'$ and (2) that $\forall x \in dom(\Gamma') : \mathcal{S}_{\mathcal{M}}\ OK$.
  (a) Note that $\mathcal{S}'_T(x) = \langle \textbf{singleton } C\rangle$ and $\Gamma' \vdash x : \textbf{singleton } C$. Thus, $\mathcal{S}'_T(x) = \Gamma'(x)$. Because T-State holds for $\vdash \langle \textbf{new } C(\overline{x}), \mathcal{S}\rangle$, we also have $\forall y \in dom(\Gamma) : \mathcal{S}_T(y) <: \Gamma(y)$. Because $x \notin dom(\Gamma)$, this also means that $\forall y \in dom(\Gamma) : \mathcal{S}'_T(y) <: \Gamma'(y)$. Knowing that $\mathcal{S}'_T(x) = \Gamma'(x)$, we can combine the previous statement such that it holds for $dom(\Gamma') = dom(\Gamma) \cup \{x\}$. Then, we get $\forall x \in dom(\Gamma') : \mathcal{S}'_T(x) <: \Gamma'(x)$. Thus, T-StoreTyp holds and $\vdash_T \mathcal{S}' : \Gamma$.
  (b) We also have to prove that $\forall x \in dom(\Gamma') : \mathcal{S}_{\mathcal{M}}\ OK$. Note that only the field mapping of x is manipulated during the evaluation step. Thus, it is sufficient to prove that $\mathcal{S}'_{\mathcal{F}}(x)\ OK$. We know that $\vdash \langle \textbf{new } C(\overline{x}), \mathcal{S}\rangle : T$. By T-State, this means that $\Gamma \vdash \textbf{new } C(\overline{x}) : T$. By T-Invk, we have $\Gamma \vdash \overline{x} : V$ and $fields(C) = \overline{W}\ \overline{f}$ such that $\overline{V} <: \overline{W}$. Thus for each actual argument x_i that will be assigned to f_i, we have $V_i <: W_i$. Consequently, T-FMap holds for $\mathcal{S}'_{\mathcal{F}}(x)$.
- $\Gamma' \vdash x : U$ for some $U <: T$. We know that $\Gamma' = \Gamma[x \rightarrow \textbf{singleton } C]$ and $\vdash \langle \textbf{new } C(\overline{e}) : T, \mathcal{S}\rangle$, where $T = \textbf{singleton } C$. Thus, letting $U = \textbf{singleton } C$ finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle \textbf{new } C(\overline{e}), \mathcal{S}\rangle$
Then, the theorem premise states that $\vdash \langle \textbf{new } C(\overline{e}), \mathcal{S}\rangle : T$. For a reduction $\langle \textbf{new } C(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle \rightarrow \langle \textbf{new } C(e_1, \ldots, e'_i, \ldots, e_n), \mathcal{S}'\rangle$, we have to prove that $\vdash \langle \textbf{new } C(e_1, \ldots, e'_i, \ldots, e_n), \mathcal{S}'\rangle : U$ for some $U <: T$. We can apply the induction hypothesis on e_i. Then, we have $\vdash \langle e_i, \mathcal{S}\rangle : W$ and $\langle e_i, \mathcal{S}\rangle \rightarrow \langle e'_i, \mathcal{S}'\rangle$ with $\vdash \langle e'_i, \mathcal{S}'\rangle : V$ for some $V <: W$.

Then, the expression $\langle \textbf{new}\ C(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle$ reduces by RC-New-Arg and we have $\langle \textbf{new}\ C(e_1, \ldots, e'_i, \ldots, e_n), \mathcal{S}'\rangle$. We have to prove that $\vdash \langle \textbf{new}\ C(e_1, \ldots, e'_i, \ldots, e_n), \mathcal{S}'\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S}' : \Gamma$ and (2) that $\Gamma \vdash \textbf{new}\ C(e_1, \ldots, e'_i, \ldots, e_n) : U$ for some $U <: T$.

(a) $\vdash \mathcal{S}' : \Gamma$. We have $\vdash \langle \textbf{new}\ C(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle : T$. By T-State, we have $\vdash \mathcal{S}' : \Gamma$.

(b) $\Gamma \vdash \textbf{new}\ C(e_1, \ldots, e'_i, \ldots, e_n) : U$ for some $U <: T$. By the theorem premise, we have $\vdash \langle \textbf{new}\ C(e_1, \ldots, e_i, \ldots, e_n), \mathcal{S}\rangle : T$. By T-State, we have $\Gamma \vdash \textbf{new}\ C(e_1, \ldots, e_i, \ldots, e_n) : T$. By T-New, we have $\Gamma \vdash \overline{e} : \overline{S}$ and $fields(C) = \overline{T}\ \overline{f}$ such that $\overline{S} <: \overline{T}$. We also know by the induction hypothesis that $\vdash \langle e_i, \mathcal{S}\rangle : W$ and $\vdash \langle e'_i, \mathcal{S}'\rangle : V$ for some $V <: W$. This means that $\Gamma \vdash e_i : W$ and $\Gamma \vdash e'_i : V$ for some $V <: W$. Combining this with the results from T-Invk, we get that $W = S_i$ and because $S_i <: T_i$, we have $W <: T_i$. Moreover, $V <: W$ and by S-Trans, we have $V <: T_i$. We can now apply T-New with the following premises:
- $fields(C) = \overline{T}\ \overline{f}$
- $\Gamma \vdash e'_i : V$ and $V <: T_i$
- $\Gamma \vdash \overline{e} : \overline{S}$ and $\overline{S} <: \overline{T}$

The result is $\Gamma \vdash \textbf{new}\ C(e_1, \ldots, e_i, \ldots, e_n) : T$. Letting $U = T$ finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e\ \textbf{where}\ e, \mathcal{S}\rangle$

First, we consider the trivial case in which both the receiver x and the actual argument y have reduces to values. Then, we can use R-Wh-True or R-Wh-False to reduce the expression. Next, we consider the case in which the actual argument e_b has not yet reduced. We finish by considering the case in which both the receiver e and the actual argument e_b are unreduced subexpressions.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x\ \textbf{where}\ y, \mathcal{S}\rangle$

Then, we have $\vdash \langle x\ \textbf{where}\ y, \mathcal{S}\rangle : T$ and this expression reduces by R-Wh-True or R-Wh-False, depending on the value of y. We prove the case for $y = true$. Then, by R-Wh-True, we have $\langle x\ \textbf{where}\ y, \mathcal{S}\rangle \rightarrow \langle x\ \textbf{where}\ e_b, \mathcal{S}\rangle$ where $\langle e_b, \mathcal{S}\rangle : \textbf{bool}$. We have to prove that $\vdash \langle x\ \textbf{where}\ e_b, \mathcal{S}\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S} : \Gamma$ and (2) that $\Gamma \vdash x\ \textbf{where}\ e_b : U$ for some $U <: T$.

- $\vdash \mathcal{S} : \Gamma$. By R-Wh-True, we have that $\mathcal{S}' = \mathcal{S}$ and we also know from $\vdash \langle x\ \textbf{where}\ y, \mathcal{S}\rangle : T$ that $\vdash \mathcal{S} : \Gamma$.
- $\Gamma \vdash x\ \textbf{where}\ e_b : U$ for some $U <: T$. From $\vdash \langle x\ \textbf{where}\ y, \mathcal{S}\rangle : T$, we have $\Gamma \vdash x\ \textbf{where}\ y : T$ and by T-Whr, this means that $\Gamma \vdash x : T$ and $\Gamma \vdash y : \textbf{bool}$. We also know that $\vdash \langle e_b, \mathcal{S}\rangle : bool$, which means that $\Gamma \vdash e_b : \textbf{bool}$. Thus, we can fill in T-Whr with the following premises:
  - $\Gamma \vdash x : T$
  - $\Gamma \vdash e_b : \textbf{bool}$

The result is $\Gamma \vdash x$ **where** $e_b : T$ and letting $U = T$ finishes the case.

Proof for $y = false$**.**
The proof for $y = false$ is similar to the proof for $y = true$. The only difference is that R-Wh-False is taken instead of R-Wh-True.

Proof for $y \neq true \wedge y \neq false$**.**
Then, the expression cannot take an evaluation step, so there is nothing to prove.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x$ **where** $e_b, \mathcal{S}\rangle$

Then, we have $\vdash \langle x$ **where** $e_b, \mathcal{S}\rangle : T$ and there are two reduction steps possible, depending on the content of $\mathcal{R}$. We consider each case.

- $\mathcal{S} = \langle Q\ C, \mathcal{R}\rangle$ **and** $\mathcal{R}\ not = \emptyset$.
  Then, the expression $\langle x$ **where** $e_b, \mathcal{S}\rangle$ reduces by R-Wh-Iter and the result is $\langle x$ **where** $[y/x]e_b, \mathcal{S}\rangle$ for some $y \in \mathcal{F}$. Thus, we have to prove that $\vdash \langle x$ **where** $[y/x]e_b, \mathcal{S}\rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S} : \Gamma$ and (2) that $\Gamma \vdash x$ **where** $[y/x]e_b : U$ for some $U <: T$.
  - $\vdash \mathcal{S} : \Gamma$. Applying T-State on $\vdash \langle x$ **where** $e_b, \mathcal{S}\rangle : T$ gives $\vdash \mathcal{S} : \Gamma$ and $\mathcal{S}$ is not changed by the reduction rule.
  - $\Gamma \vdash x$ **where** $[y/x]e_b : U$ for some $U <: T$. We know that $\vdash \mathcal{S} : \Gamma$ and by T-RMap, we know that $\forall v \in \mathcal{R} : \mathcal{S}_T(v) <:$ **singleton** C. Because $y \in \mathcal{R}$, we have that $\mathcal{S}_T(y) <:$ **singleton** C for $\mathcal{S}(x) = \langle Q\ C, \mathcal{R}\rangle$. Thus, we have $\Gamma \vdash y :$ **singleton** C by T-StoreTyp, and we can apply the term substitution lemma on $\Gamma \vdash e_b :$ **bool**. The result is $\Gamma \vdash [y/x]e_b :$ **bool** because no subtyping is defined on primitive types. Applying T-State on $\vdash \langle x$ **where** $e_b, \mathcal{S}\rangle : T$ gives $\Gamma \vdash x$ **where** $e_b : T$ and by T-Whr, we have $\Gamma \vdash x : T$ and $\Gamma \vdash e_b :$ **bool**. Consequently, we can apply T-Whr on $\Gamma \vdash [y/x]e_b :$ **bool** and $\Gamma \vdash x : T$. The result is $\Gamma \vdash x$ **where** $[y/x]e_b : T$. Letting $U = T$ finishes the case.
- $\mathcal{S} = \langle Q\ C, \mathcal{R}\rangle$ **and** $\mathcal{R}\ = \emptyset$.
  Then, $\langle x$ **where** $e_b, \mathcal{S}\rangle$ reduces by R-Wh-Stop and the result is $\langle x, \mathcal{S}'\rangle$. Where $\mathcal{S}'(y) = \mathcal{S}(y)$ for all $y \in dom(\Gamma)$ where $y \neq x$ and $\mathcal{S}'(x) = \langle T, \mathcal{R}'\rangle$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S}' : \Gamma$ and (2) that $\Gamma \vdash x : U$ for some $U <: T$.
  - The reference mapping of $\mathcal{R}$ has been changed such that $\mathcal{R}' \subseteq \mathcal{R}$. By T-State, we know for $\vdash \langle x$ **where** $e_b, \mathcal{S}\rangle : T$ that $\vdash \mathcal{S} : \Gamma$. Using T-RMap, we know that the reference mapping $\mathcal{R}$ is consistent for $x \in dom(\Gamma)$. This means for $\mathcal{S}(x) = \langle Q\ C, \mathcal{R}\rangle$ that $\forall v \in \mathcal{R} : \mathcal{S}_T(v) <:$ **singleton** C. Thus, T-RMap still holds after the reduction because $\mathcal{R}' \subseteq \mathcal{R}$.
  - $\Gamma \vdash x : U$ for some $U <: T$. From $\vdash \langle x$ **where** $e_b, \mathcal{S}\rangle : T$, we know from T-State that $\Gamma \vdash x$ **where** $e_b : T$ and by T-Whr, we know that $\Gamma \vdash x : T$. Letting $U = T$ finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle e \textbf{ where } e_b, \mathcal{S}\rangle$

We can apply the induction hypothesis on e. Then, we have that $\vdash \langle e, \mathcal{S}\rangle : W$ and $\langle e, \mathcal{S}\rangle$ reduces to $\langle e', \mathcal{S}'\rangle$ with $\vdash \langle e', \mathcal{S}'\rangle : V$ for some $V <: W$. Knowing that $\langle e, \mathcal{S}\rangle \rightarrow \langle e', \mathcal{S}'\rangle$, we get that $\langle e \textbf{ where } e_b, \mathcal{S}\rangle$ reduces by one of the congruence rules, namely RC-WHERE-RECV. The result is $\langle e' \textbf{ where } e_b, \mathcal{S}'\rangle$. We have to prove that $\vdash \langle e' \textbf{ where } e_b, \mathcal{S}'\rangle : U$ for some $U <: T$. By T-STATE, it is sufficient to prove (1) that$\vdash \mathcal{S}' : \Gamma$ and (2) that $\Gamma \vdash e' \textbf{ where } e_b : U$ for some $U <: T$.

- $\vdash \mathcal{S}' : \Gamma$. By the induction hypothesis, we have that $\vdash \langle e', \mathcal{S}'\rangle : T$. By T-STATE, this means that $\vdash \mathcal{S} : \Gamma$.
- $\Gamma \vdash e' \textbf{ where } e_b : U$ for some $U <: T$. By applying T-STATE on $\vdash \langle e \textbf{ where } e_b, \mathcal{S}\rangle : T$, we know that $\Gamma \vdash e \textbf{ where } e_b : T$ and by T-WHR, we have that $\Gamma \vdash e_b : \textbf{bool}$ and $\Gamma \vdash e : T$. Thus, $W = T$. From $\vdash \langle e', \mathcal{S}\rangle : V$, we know by T-STATE that $\Gamma \vdash e' : V$ and $V <: W$. Thus, $V <: T$. We can now fill in T-WHR with the following premises:
  - $\Gamma \vdash e_b : \textbf{bool}$
  - $\Gamma \vdash e' : V$ where $V <: T$

  The result is $\Gamma \vdash e' \textbf{ where } e_b : V$. Letting $U = V$ finishes the case.

Case $\langle e_0, \mathcal{S}\rangle \equiv \langle e \textbf{ orderby } e_c, \mathcal{S}\rangle$

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x \textbf{ orderby } y, \mathcal{S}\rangle$

Then, we have $\vdash \langle x \textbf{ orderby } y, \mathcal{S}\rangle : T$ and this expression reduces by R-ORD-INT and the result is $\langle x \textbf{ orderby } e_c, \mathcal{S}\rangle$ for $\vdash \langle e_c, \mathcal{S}\rangle : S$ with $S <: \textbf{singleton } \texttt{Comparable}$. We have to prove that $\vdash \langle x \textbf{ orderby } e_c, \mathcal{S}\rangle : U$ for some $U <: T$. By T-STATE, it is sufficient to prove (1) that $\vdash \mathcal{S} : \Gamma$ and (2) that $\Gamma \vdash x \textbf{ orderby } e_c : U$ for some $U <: T$.

- $\vdash \mathcal{S} : \Gamma$. The reduction step does not modify the contents of the store. We have $\vdash \langle x \textbf{ orderby } y, \mathcal{S}\rangle : T$ and by T-STATE, we have that $\vdash \mathcal{S} : \Gamma$.
- $\Gamma \vdash x \textbf{ orderby } e_c : U$ for some $U <: T$. From $\vdash \langle x \textbf{ orderby } y, \mathcal{S}\rangle : T$, we have $\Gamma \vdash x \textbf{ orderby } y : T$. By T-ORD, we have $\Gamma \vdash x : V$ such that $T = \mathit{toSequence}(V)$. This means that $V <: T$ (this was proven in a separate Lemma: "$\mathit{toSequence}(T)$ preserves the typing of T"). We also have that $\Gamma \vdash y : W$ for some $W <: \textbf{singleton } \texttt{Comparable}$. We also know that $\langle e_c, \mathcal{S}\rangle : S$ for some $S <: \textbf{singleton } \texttt{Comparable}$. We can now apply T-ORD with the following premises:
  - $\Gamma \vdash x : V$ and $T = \mathit{toSequence}(V)$
  - $\Gamma \vdash e_c : S$ for some $S <: \textbf{singleton } \texttt{Comparable}$

  The result is $\Gamma \vdash x \textbf{ orderby } e_c : T$. Letting $U = T$ finishes the case.

SubCase $\langle e_0, \mathcal{S}\rangle \equiv \langle x \textbf{ orderby } e_c, \mathcal{S}\rangle$

Then, we have $\vdash \langle x \textbf{ orderby } e_c, \mathcal{S}\rangle : T$. This expression reduces by R-WH-ITER or R-WH-STOP depending on the contents of $\mathcal{R}$. We consider each case.

– $\mathcal{R} \neq \emptyset$ for $\mathcal{S}(x) = \langle Q\ C, \mathcal{R} \rangle$
  Then, the expression reduces by R-Wh-Iter and we have $\langle x\ \textbf{orderby}\ [y/x]e_c, \mathcal{S} \rangle$ for some $y \in \mathcal{R}$. We have to prove that $\vdash \langle x\ \textbf{orderby}\ [y/x]e_c, \mathcal{S} \rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S} : \Gamma$ and (2) that $\Gamma \vdash x\ \textbf{orderby}\ [y/x]e_c : U$ for some $U <: T$.
  - $\vdash \mathcal{S} : \Gamma$. We know that $\vdash \langle x\ \textbf{orderby}\ e_c, \mathcal{S} \rangle : T$, so by T-State, we have that $\vdash \mathcal{S} : \Gamma$. Moreover, the reduction rule does not change the contents of $\mathcal{S}$.
  - $\Gamma \vdash x\ \textbf{orderby}\ [y/x]e_c : U$ for some $U <: T$. By applying T-Store on $\vdash \langle x\ \textbf{orderby}\ e_c, \mathcal{S} \rangle$, we have that $\Gamma \vdash x\ \textbf{orderby}\ e_c : T$. By T-Ord, we have that $\Gamma \vdash x : T_x$ such that $T = \mathit{toSequence}(T_x)$. We also have that $\Gamma \vdash e_c : S$ for some $S <: \textbf{singleton}\ \texttt{Comparable}$. By T-RMap, we also know that the type of y is more specific than the type of x. Consequently, we can apply the substitution lemma and this gives that $\Gamma \vdash [y/x]e_c : S'$ for some $S' <: S$. We can now fill in T-Ord with the following premises:
    * $\Gamma \vdash [y/x]e_c : S'$ where $S' <: \textbf{singleton}\ \texttt{Comparable}$
    * $\Gamma \vdash x : T_x$ and $T = \mathit{toSequence}(T_x)$

    The result is $\Gamma \vdash x\ \textbf{orderby}\ e_c : T$. Letting $U = T$ finishes the case.

– $\mathcal{R} = \emptyset$ for $\mathcal{S}(x) = \langle Q\ C, \mathcal{R} \rangle$
  Then, we have $\vdash \langle x\ \textbf{orderby}\ e_c, \mathcal{S} \rangle$ and this expression reduces by R-Ord-Stop to $\langle z, \mathcal{S}' \rangle$ for some $z \notin \mathit{dom}(\mathcal{S})$ where $\mathcal{S}'(z) = \langle T, \mathcal{R}' \rangle$. Also, $\mathcal{R} = \mathcal{R}'$, because services are not removed during the execution of the sorting algorithm. We have to prove that $\vdash \langle x, \mathcal{S}' \rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S}' : \Gamma'$ where $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \cup \{z\}$ and (2) that $\Gamma, z : \mathit{toSequence}(Q\ C) \vdash z : U$ for some $U <: T$.
  - $\vdash \mathcal{S}' : \Gamma'$. We know that $\vdash \mathcal{S} : \Gamma$ by applying T-State on $\vdash \langle x\ \textbf{orderby}\ [y/x]e_c, \mathcal{S} \rangle : T$. This means that $\mathcal{S}_T(x) <: \Gamma(x)$ for all $x \in \mathit{dom}(\Gamma)$. We also know that $\mathcal{S}'_T(x) = \mathit{toSequence}(Q\ C)$ and $\Gamma \vdash v_0 : T$ where $T = \mathit{toSequence}(Q\ C)$. Thus, $\mathcal{S}'_T(x) = \Gamma'(x)$. Thus, $\vdash_T \mathcal{S}' : \Gamma'$. Moreover, $\mathcal{R} = \mathcal{R}'$ for $\mathcal{S}'(v) = \langle \mathit{toSequence}(Q\ C), \mathcal{R}' \rangle$ and $\mathcal{S}(x) = \langle Q\ C, \mathcal{R} \rangle$. Knowing by T-RMap that $\mathcal{S}_{\mathcal{R}}(x)$ is consistent, we also have that $\mathcal{S}'_{\mathcal{R}'}(v)$ is consistent, and $\mathcal{S}' \vdash \Gamma'$.
  - $\Gamma, z : \mathit{toSequence}(Q\ C) \vdash z : U$ for some $U <: T$. By applying T-State on $\vdash \langle x\ \textbf{orderby}\ [y/x]e_c, \mathcal{S} \rangle : T$, we get $\Gamma \vdash x\ \textbf{orderby}\ [y/x]e_c : T$. Applying T-Ord gives $\Gamma \vdash x : T_x$ with $T = \mathit{toSequence}(T_x)$. Because the store is consistent with the typing environment, we have $T_x = Q\ C$. Moreover, $\Gamma \vdash v_0 : \mathit{toSequence}(Q\ C)$. Thus, $\Gamma \vdash v_0 : T$. Letting $U = T$ finishes the case.

SubCase $\langle e_0, \mathcal{S} \rangle \equiv \langle e\ \textbf{orderby}\ e_c, \mathcal{S} \rangle$

We can apply the induction hypothesis on e. Then, we have $\vdash \langle e, \mathcal{S} \rangle : W$ and this expression reduces to $\langle e', \mathcal{S}' \rangle$ with $\vdash \langle e', \mathcal{S}' \rangle : V$ for some $V <: W$. By the theorem premise, we have $\vdash \langle e\ \textbf{orderby}\ e_c, \mathcal{S} \rangle : T$ and this expression

reduces by RC-Order-Recv to $\langle e' \textbf{ orderby } e_c, \mathcal{S}' \rangle$. We have to prove that $\vdash \langle e' \textbf{ orderby } e_c, \mathcal{S}' \rangle : U$ for some $U <: T$. By T-State, it is sufficient to prove (1) that $\vdash \mathcal{S} : \Gamma$ and (2) that $\Gamma \vdash e' \textbf{ orderby } e : U$ for some $U <: T$.

- $\vdash \mathcal{S}' : \Gamma$. We know that $\vdash \langle e', \mathcal{S}' \rangle : T$ and by T-State, we have $\vdash \mathcal{S} : \Gamma$.
- $\Gamma \vdash e' \textbf{ orderby } e_c : U$ for some $U <: T$. We have that $\vdash \langle e \textbf{ orderby } e_c, \mathcal{S} \rangle : T$ and from T-State, we have $\Gamma \vdash e \textbf{ orderby } e_c : T$. Applying T-Ord, we get that $\Gamma \vdash e : T_e$ and $T = \mathit{toSequence}(T_e)$. Thus, $T <: T_e$ (this was proven in the Lemma which stated that $\mathit{toSequence}(T)$ preserves the type of an expression). Also, $W = T_e$. We know that $\vdash \langle e', \mathcal{S}' \rangle : V$ and by T-State, we have $\Gamma \vdash e' : V$. Thus, $V <: W = T_e$. Moreover, $\mathit{toSequence}(V) <: V$. We can now fill in T-Ord with the following premises:
  - $\Gamma \vdash e' : V$ and $V' = \mathit{toSequence}(V)$.
  - $\Gamma \vdash e_c : S$ with $S <:$ **singleton** `Comparable`

  The result is $\Gamma \vdash e' \textbf{ orderby } e_c : V'$ with $V' = \mathit{toSequence}(V)$ and $V' < T$ because $T = \mathit{toSequence}(T_e)$ and $V <: T_e$. Letting $V' = U$ finishes the case.

□

E Type Soundness

Given $\vdash e : T$, then either
- $\langle e, \emptyset \rangle \xrightarrow{*} \langle x, \mathcal{S} \rangle$ for some $\mathcal{S}$ and x
- $\langle e, \emptyset \rangle \xrightarrow{*} \langle e', \mathcal{S} \rangle$ where $\langle e', \mathcal{S} \rangle$ is stuck due to the fail predicate.
- $\langle e, \emptyset \rangle \xrightarrow{*} \langle e', \mathcal{S} \rangle$ where $\langle e', \mathcal{S} \rangle$ is stuck due to a failed dynamic downcast
- $\langle e, \emptyset \rangle$ executes forever.

PROOF Straightforward induction on $\langle e_0, \mathcal{S} \rangle \rightarrow \langle e_0', \mathcal{S}' \rangle$ based on the Progress theorem and the Subject Reduction theorem. □

F Availability Theorems

A method call on a **pool** or **sequence** $S(v_s) = \langle Q\ C, \mathcal{R} \rangle$ succeeds if at least one pool member $v_i \in \mathcal{R}$ is available.

Proof. Straightforward from the definition of $failover(\mathcal{R})$. This function only returns the $fail$ predicate if no service from the pool is available. Thus, whenever at least one service v_i is available, a service $v_j \in \mathcal{R}$ will be returned. Then, this v_j will be substituted with the invocation target v by R-INVKP, and the method invocation will succeed under normal conditions (i.e. if an expression is not stuck due to a dynamic downcast or due to the occurence of the $fail$ predicate). □

If a method call on a **singleton** $S(v_s) = \langle Q\ C, \mathcal{F} \rangle$ succeeds, then a method call on a **pool/sequence** $S(v_p) = \langle Q\ C, \mathcal{R} \rangle$ *with* $v_s \in \mathcal{R}$ will also succeed.

Proof. Straightforward from the previous availability theorem because at least one service is in $\mathcal{R}$. □