JOT JOURNAL OF OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Extending the Object-Capability Model with Fine-Grained Type-Based Capabilities

Roland Wismüller*, **Damian Ludwig****, **and Felix Breitweiser***
*University of Siegen, Germany
**Federal Office for Information Security, Bonn, Germany

ABSTRACT Although the Principle Of Least Authority (POLA) is a well-recognized guideline for building secure software systems, there is actually a lack of concepts that really encourage programmers to use POLA consequently. The best support for POLA is currently offered by secure languages based on the Object-Capability (OCap) paradigm, where object references serve as capabilities for accessing objects. However, the OCap model just controls the overall accessibility of objects, it does not directly support fine-grained control over the use of specific operations on these objects. While access control at the level of individual methods can be implemented by using the membrane pattern, this approach creates a heavy burden for the programmer, may lead to performance problems, and suffers from the fact that it is difficult to determine the minimum permissions that must be granted.

In this paper, we present a type-based split capability model, where the access permissions granted by a reference are restricted by the type of the variables used to send and receive that reference. In this way, required and granted permissions are directly represented in the type definition of a software component's interface. Furthermore, compliance with access restrictions can often be checked statically when a software component is deployed, thus avoiding the run-time overhead of using a membrane. In the case where membranes are needed to enforce access control at run-time, these membranes are automatically built by the run-time system.

As a foundation for this model, we specify type checking rules that prevent software components from from amplifying their authority by downcasting a reference to a more permissive type. Finally, we identify the necessary requirements for the run-time system as well as the run-time overhead induced by our security model.

KEYWORDS access control, object-capability model, capabilities, type system.

1. Introduction

There is a great deal of agreement in the scientific community that software systems employing untrusted components should be built on the Principle Of Least Authority (POLA), which demands that a subject should receive just the authority it needs to successfully achieve its intended purpose (Miller & Shapiro 2003). Surprisingly, there are nevertheless only few concepts, which really encourage and support the programmer to use POLA. One of the most prominent paradigms in this area is the object-capability (OCap) model (Miller 2006; Murray 2008). OCap supports POLA in the sense that a subject (which in OCap actually is an object in an active role) can access an object only when it received a reference to that object from someone else. The reference acts as a capability, which cannot be forged by the subject itself. Thus, OCap allows to easily ensure that a subject can only access the objects that it actually needs for performing its task. However, a severe limitation of OCap is that it essentially only limits the ability to access objects, but not the ability to perform specific operations on these objects.

As a motivating (and drastic) example, consider a rocket

JOT reference format:
Roland Wismüller, Damian Ludwig, and Felix Breitweiser. *Extending the Object-Capability Model with Fine-Grained Type-Based Capabilities.* Journal of Object Technology. Vol. 23, No. 1, 2024. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2024.23.1.a1

base equipped with a number of rockets and two kinds of subjects: the president and service technicians. There are two possible operations for each rocket: `getStatus()` determines the healthiness of the rocket, while `launch()` fires the rocket. In an object oriented programming language, we may model this as follows:[1]

```
class Rocket {
  ...
  public int getStatus() { ... }
  public void launch() { ... }
}
```

The service technician is also modeled as an object:

```
class ServiceTechnician {
  ...
  public void service(Rocket r) { ... }
}
```

Now, the OCap model perfectly ensures that the service technician can access a rocket only if he receives a reference to that rocket via a call to `service()` executed by an authorized subject. However, OCap does not prevent him from invoking `launch()` on that rocket!

We may be tempted to introduce an interface that only includes the relevant method `getStatus()`:

```
interface Serviceable {
  int getStatus();
}
class Rocket implements Serviceable {
  ...
}
class ServiceTechnician {
  ...
  public void service(Serviceable r) {
    ...
  }
}
```

However, of course, in traditional OO languages, nothing prevents the service technician from implementing `service()` like this:

```
public void service(Serviceable r) {
  ((Rocket)r).launch();
}
```

The way how this problem is traditionally solved in OCap is to introduce a *membrane* (also called a *facet*) (Miller 2006; Mettler et al. 2010), i.e., to use a forwarding object that only provides the permitted methods. In our example, we would have to define a new class:

```
class RocketMembrane
      implements Serviceable {
  private Rocket delegate;
  public RocketMembrane(Rocket r) {
    delegate = r;
  }
  public int getStatus() {
    return delegate.getStatus();
  }
}
```

We then would pass such a restricted membrane to the service technician:

```
st.service(new RocketMembrane(rocket));
```

This practice is actually secure, but it has a number of serious drawbacks:

1. The burden on the programmer for implementing POLA is extremely high, since in the OCap model POLA demands to provide an individually tailored membrane for each use of an object. Thus, in practice, classes for many different membranes must be written.

2. There is no safe way for the programmer to know what kind of authority is actually required by a specific subject. While in our example, the argument type `Serviceable` of `service()` gives us some hint, a programmer can usually not rely on the fact that the subject only requires the methods of this interface, as there sometimes are legal reasons for downcasting an object reference. As an example, assume that some subset of the rockets has an additional operation `test()` for performing an extended self test:

```
interface Testable {
  int test();
}
// TestableRocket also implements
// Serviceable
class TestableRocket extends Rocket
  implements Testable {
  public int test() { ... }
}
```

   In this case, the service technician should be able to use that method, if needed:

```
public void service(Serviceable r) {
  ...
  if (problem_detected &&
       (r instanceof Testable)) {
    int diag = ((Testable)r).test();
    ...
  }
}
```

   This means that the class `RocketMembrane` must be extended to also allow the use of the method `test()`, although the programmer cannot infer this fact from the signature of `service()`.

3. Since the programmer cannot be sure whether a subject is honest or not, he must always create a membrane to restrict the use of an object passed to that subject. In the case of a honest subject, this results in an unnecessary overhead.

[1] We use the widely-used OO language Java, rather than a special OCap language like E (Miller 2006) for the best possible clarity. For simplicity, we assume that the rocket status is simply encoded as an `int` value.

4. Finally, the explicit creation of membranes may lead to an unacceptable overhead in large-scale systems, since the membranes will typically cascade. As an example, assume that we have a service assistant as another kind of subject. The service technician can delegate some tasks to his assistant, but the assistant is not allowed to execute the self test. To enforce this restriction, `service()` must wrap its argument into a membrane that just provides the `getStatus()` method, before passing it to the assistant. However, the argument is already a membrane, so we now have two levels of indirection. Note that we cannot avoid cascading, since for security reasons it obviously must not be possible for the service technician to remove the old membrane.

In this paper, we will show how all of these drawbacks can be avoided by regarding types as specifications of access rights at the level of individual methods, such that a reference together with the static type of the variable containing it acts as a fine-grained capability. In particular, we will present the following contributions:

- We develop a model of split capabilities, where a certain part of the capability (the object reference) can be freely passed around between subjects while another part (the static type of the variable containing the reference) is permanently located at a specific subject. This will provide a clear view on how types can be regarded as capabilities.
- From the requirements imposed by this model, we derive a type system that enables an easy and yet secure cooperation of untrusted components. Besides enabling the split capability model, the type system supports both structural and nominal typing and also provides a generic way to support unsafe covariance. We will present a formal model of the type system, together with a proof of its fundamental security property.
- Finally, we will identify the requirements for an execution environment (i.e., a virtual machine) based on that type system, together with an analysis of the sources of overhead imposed by it.

2. A Type-Based Split Capability Model

In order to get a clearer understanding of the OCap model and how it may be improved, we will now analyze the model in more detail. As the main goal of OCap is to support POLA, we first need to know what *authority* actually means.

Intuitively, we may describe the authority of a subject s as the set of effects that s can cause on any object in the system of consideration. I.e., in a pure object oriented model the authority would be the set of pairs (m, o), such that s can achieve an invocation of m on o. However, this is only half of the truth, as authority may also be passed between subjects. Thus, the authority of s also includes its ability to pass authority to other subjects, as well as its ability to receive authority from other subjects. In order to unify these different abilities, we will use the generic term *operation* to denote any effect that can be achieved on an object, i.e. calling a method as well as sending and receiving of authority to / from it.[2]

However, the authority of s does not only include actions performed by s itself, but also actions to which s may persuade other subjects (confused deputy, c.f. (Miller et al. 2003)). As the latter is virtually impossible to grasp, since it depends on the behavior of all other subjects in the system, we need to find a more restricted, but usable concept. We thus define the *direct authority* of a subject s as the set of operations that s can perform *itself*, i.e., without the help of another subject.

We can now analyze how the direct authority of a subject s is extended, if another subject s' passes s a capability for some object o. In the traditional OCap model, this capability just consists of a reference r to o. Since OCap requires a run-time system that implements memory safety, it prevents s from manipulating r in such a way that it references some other object o'. As OCap does not induce any further limitations on the use of r, the direct authority of s is extended by all operations on o that are supported by that object. This includes the ability to call any method provided by o,[3] but we still have to analyze the ability of s to send / receive capabilities to / from o. So how can s pass a capability to o? The only way is to call a method on o that accepts a reference as an argument. I.e., as soon as o provides a method with a reference argument, s is able to pass any capability to o. Likewise, if o provides a method returning a reference, s is able to receive any capability from o.

In OCap, it is actually possible to restrict the direct authority granted by a reference r by wrapping r's target object o into a membrane. This membrane may provide only a subset of o's methods, but it may also wrap method arguments and/or results into other membranes, thereby restricting the authority that s can send to / receive from o. The drawback here is that the code for these membranes must be explicitly provided by the programmer, and that the restriction of authority happens at run-time, thus impeding static security analysis.

Now, how does the situation change, when our system is programmed using a statically typed programming language? In order to analyze the situation more thoroughly, let us consider the case where subject s' passes a reference r to subject s as an argument of a method called on s.[4] In order to do so, s' must have a reference r_s to s, which has been declared with some type T. Now, T is not necessarily identical to the actual type S of s, but is usually some supertype of S, as shown in the code snippet below.[5]

```
1 interface AT {        // subtype of AS
2   void ma(AAT arg);
3   RAT mr();
4 }
```

[2] Note that in the OCap model, an object can also act as a subject.

[3] Depending on how strict the object model is implemented by the run-time system, this may be limited to public methods or may also include protected and private methods. E.g., in Java private methods can actually be called from outside the class by using reflection.

[4] The situation where r is passed as a result of a method called by s on s' is similar.

[5] Despite the Java-like syntax, the example cannot be expressed in Java, as Java does not support contravariant argument types when overriding methods. Furthermore, to emphasize the influence of the types' structure on subtyping, the example is based on structural subtyping.

```
5  interface T {          // supertype of S
6    void pass(AT arg);
7  }
8
9  interface AS {         // supertype of AT
10   void ma(AAS arg);
11   RAS mr();
12 }
13 class S {              // subtype of T
14   public void pass(AS arg) { ... }
15 }
16
17 ...
18   T  rs = ...;  // Reference to an
19                 // instance s of class S
20   AT r  = ...;  // Reference to an
21                 // instance of a class
22                 // implementing AT
23   rs.pass(r);   // pass r to s
```

This in turn means that the type AT, which s' is using to send r, is usually a subtype of the type AS used by s to receive the reference. As the type AT actually restricts the operations that can be performed using reference r, we can view AT as a specification of the authority s' is willing to pass to s. On the other hand, AS is a specification of the requirements that s imposes on the received reference (Hagimont et al. 1996). Furthermore, via the argument type AAT of method `ma()` in AT, s' also specifies restrictions on references passed to the target object of r, and via the return type RAT of `mr()` also on references received from that object. Likewise AAS and RAS specify the corresponding requirements of s.

However, in traditional programming languages, these specifications are not binding, as it is allowed to downcast a reference to a different type, provided that this type is a supertype of the actual type of the referenced object. This is why classical OCap is right in its analysis of authority and its use of membranes to manifest any restrictions in the objects themselves.

So what we need is to restrict cast operations on references in such a way that subjects are prevented from amplifying their authority. Under this condition, the (dynamic) object reference r together with the static type AS used by s to receive the reference in the above example forms a capability (r, AS), where AS grants the authority to use the specified operations. A notable aspect here is that the only information transferred at run-time is the reference r, which is just the address of the object and does not include any run-time type or other access control information. The reference r just grants the permission to *use* the referenced object, while the receiving type AS defines *how* exactly the object may be used. On the one hand, this drastically reduces the run-time overhead for implementing fine-grained capabilities in comparison to a naïve capability-based implementation where the information on the permitted operations is explicitly passed alongside the reference. On the other hand, as already mentioned above, AS and AT provide a precise documentation of the authority actually required by s and granted by s', respectively.

So if the sender s' has a capability (r, AT) and passes it on

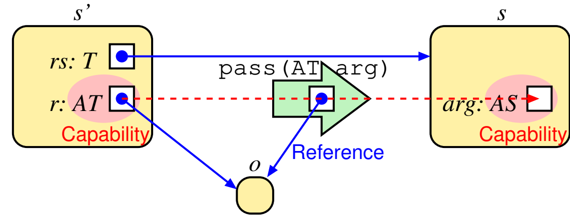

Figure 1 Subject s' has a capability (r, AT) consisting of a reference r and the static type AT of the variable containing r. When this capability is passed on to s, only r is transferred.

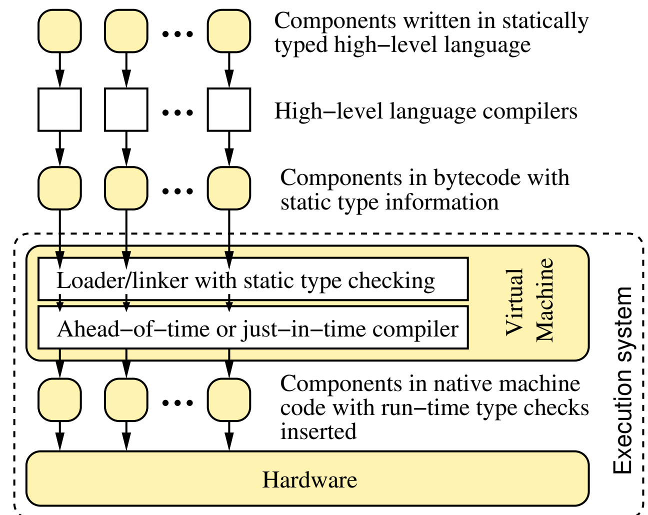

Figure 2 System model: Software components are represented by typed bytecode. When a component is loaded, a static type check is performed and it is compiled into machine code for execution.

to the receiver s, the capability is split, and only r is actually transferred (see Fig. 1). When s successfully receives r, it has a capability (r, AS). Traditional static type checking will ensure that AT actually provides all the methods required by AS. In addition, we must prevent s from casting r to a type that grants more authority than AS. This can be achieved by designing proper rules for permitted type casts in the programming language. However, if security is just based on the properties of the high-level programming language, this means that the components integrated into a secure system must all be available as source code, which is often infeasible due to the protection of intellectual property rights. Thus, components should be delivered in a statically typed intermediate language, similar to the Java bytecode. In that way, the virtual machine executing the bytecode can perform static type checking when a component is loaded. In addition, run-time type checks can be inserted where required. Fig. 2 shows the system model assumed.

So far, the model restricts the interaction of *any* two objects in a software system, which limits the expressiveness of any (high-level or intermediate) language based on that model. However, this is not necessary in practice, since within a single code unit where all the code is written by the same group of programmers, we will not have security concerns. Only when we deploy

software components from different, possibly untrusted sources, we want to enforce security constraints for the interaction of these components. To this end, we give up the requirement that objects have a one-to-one correspondence with subjects, which means that whenever an object is created, this object is associated with a new subject. Rather, we also allow that a newly created object is associated with the same subject than the object creating it. In this way, we have the flexibility to associate all objects belonging to a single code unit with the same subject, thus, removing the need for security checks when passing references between these objects. This concept is further elaborated in Sect. 4.3.3.

To summarize: in the type-based split capability model, all resources are realized as objects. An object can be accessed only via a capability (r, S) which consists of a reference r to the object and the static type S of the variable where that reference is stored in. While r just controls whether or not the object can be accessed at all, S determines which operations can be performed on the object. When a capability is passed around, only the reference r is transferred. After r has been received, the receiver has a capability (r, T) where T is the static type of the variable where r is stored into. A proper type system ensures that this assignment is only successful, if the type cast from S to T is legal, which implies that it does not amplify authority. The type system further ensures that the receiver cannot cast the reference to a type that is more permissive than T, so that the direct authority granted by (r, T) is limited to the operations that are permitted by *both* S and T.

3. Type System: Basic Ideas

We will now derive the basic ideas and structure of a type system that allows to implement the type-based split capability model. Since in this context, only references are relevant, we do not yet include simple data types into our discussion. We further limit this introductory discussion to interface types, i.e., we are just considering (public) methods.

3.1. Assertions and Permissions

A traditional reference type actually specifies *assertions* on the referenced object, i.e., in our context, which methods the object must provide (including the methods' argument and result types). It is possible to "override" these assertions by downcasting a reference. In this case, a run-time check ensures that the assigned object actually provides the methods required by the target type.

As our type system must not allow an amplification of authority, a type must not only specify assertions, but also *permissions*, such that a downcast for permissions is not possible. How can we achieve this? For the sake of a clear discussion, we use the following observation: A type T actually maps method names to a corresponding state, i.e., in the traditional case, a method m is either available in T with specified argument and return types or it is not listed in the type. We will denote this state as $f_T(m)$ and explicitly model a type as a function mapping strings (i.e. possible method names) to a state. We now give the two traditional states a refined meaning:

1. The type asserts that the method is provided by the referenced object *and permits the use of that method*. We will denote this state as *avail*, as an abbreviation for *available*.
2. The type does not assert that the method is provided by the referenced object *and also denies the use of that method*. This state is named *denied*.

Now, since we must not allow to downcast permissions, this interpretation would mean that we cannot allow downcasting at all. Thus, we introduce a third state:

3. The type does not assert that the method is provided by the referenced object *but permits the use of that method*. We call this state *optional*.[6]

In order to check whether this extension can lead to a consistent type system, we should first consider the subtype relation. Roughly speaking, S is subtype of T ($S <: T$), if S is "as least as good" as T, i.e., every object (reference) of type S can be regarded as also having type T. This means that S must provide stronger assertions and/or more permissions than T, which in our model translates to the requirement that for each m, $f_S(m)$ must provide stronger assertions and/or more permissions than $f_T(m)$, denoted as $f_T(m) \leq f_S(m)$. From the definition of our states, it is obvious that we have *denied* $<$ *optional* $<$ *avail*. Now let us check the cases where this definition of the ordering prevents S from being a subtype of T. Note that $S \not<: T$ implies that an assignment of an object reference of type S to a variable of type T (probably including an explicit type cast to T) is either forbidden at all, or is only possible with some additional actions at run-time.

- If we have some method m with $f_S(m) =$ *denied* and $f_T(m) =$ *avail*, then we cannot allow such an assignment at all, as it would result in an amplification of authority (S denies the use of m while T permits it).
- If for some m we have $f_S(m) =$ *optional* and $f_T(m) =$ *avail*, then we need to check at run-time whether the referenced object actually provides a method m.[7]
- Finally, we may have a method m with $f_S(m) =$ *denied* and $f_T(m) =$ *optional*. As in the first case, we must prevent an amplification of authority. However, now T does not require the referenced object to provide method m. Thus, we actually can permit the type cast, provided that at run-time we ensure that the object finally assigned does not have this method. If the object to be assigned originally had a method m, we can wrap it into a membrane that does not provide m. Note that all the information needed to construct this membrane is contained in the involved types (S, T, and the type of the object to be assigned), thus, it can be created transparently by the run-time system.

If we have a closer look at the last case, we may ask what type the membrane should have, especially, what the status of m

[6] Optional methods in a similar sense are provided by, e.g., Objective C and Swift. Java's `default` methods have a different semantics in that they just allow that a class implementing an interface does not have to provide these methods.

[7] Actually, we also have to check whether the argument and return types are consistent. This will be discussed in later sections of the paper.

Table 1 Semantics of method states

	Type permits the use of m	Type asserts that m is available in the object	Type asserts that m is un-available in the object
denied	**F**	**F**	**F**
optional	**T**	**F**	**F**
avail	**T**	**T**	**F**
unavail	**F**	**F**	**T**

should be. As the membrane is introduced just to give a guarantee that m is not available, we need a new state asserting that the referenced object does *not* provide m (and thus, does not permit the use of that method). We call this state *unavail*, as an abbreviation for *unavailable*. As the type M of the membrane must be a subtype of T (since the reference of type T will ultimately refer to an object of type M), we must define *optional* $<$ *unavail*, thus resulting in a partial order for our method states. Table 1 summarizes the meaning of the four resulting states.

3.2. Back to the Example

Now that we have outlined the essential principles of our type system, let us examine whether it can solve the example problem from Sect. 1. To fulfill his duty, the service technician needs access to the rocket's `getStatus()` method, and its `test()` method, if available. This can be easily and clearly specified in the argument type of `service()`:[8]

```
interface Serviceable {
  int getStatus();      // state: avail
  optional int test(); // state: optional
}
class ServiceTechnician {
  ...
  public void service(Serviceable r) {
    ...
  }
}
```

Since the state of method `launch()` is *denied* (the method is not mentioned in the interface), the service technician will not be able to downcast the received reference in such a way that he can call that method. Thus, security is enforced just by the static argument type of `service()`, not by wrapping the rocket object into a membrane. Of course, the service technician would be able to access `launch()`, if it were included in the `Serviceable` interface, however, you would not hire a technician explicitly asking for the permission to launch the serviced rocket, would you?

[8] We introduce a new keyword `optional` to specify that the state of a method is *optional*.

By including `test()` as an optional method in the interface, the service technician will be able to call that method, if it is available. In our implementation, we do not allow a direct call to an optional method, as this may result in the call failing at run-time. Rather, we restrict run-time errors to type cast operations by forcing the programmer to first downcast the reference to a type that specifies the method as *avail*, e.g.:

```
interface Testable {
  int test(); // state: avail
}
...
public void service(Serviceable r) {
  ...
  if (problem_detected &&
      (r instanceof Testable)) {
    int diag = ((Testable)r).test();
    ...
  }
}
```

Note that in this scenario, the caller of `service()` actually can decide at run-time to deny the use of `test()` by simply casting the passed reference to a type that does not include that method:

```
interface NonTestable {
  int getStatus(); // state: avail
}
...
st.service((NonTestable)rocket);
```

In this case, the run-time system will automatically wrap the rocket object into a membrane that misses the `test()` method.

So the idea of our type system nicely solves all the problems discussed in the introduction. In the following subsections, we will now provide a formal description of the type system based on that idea, which also includes support for local object types allowing unconstrained downcasts for objects owned by the acting subject. After that, we will extend the type system with support for unspecified reference types, unsafe subtyping, including unsafe covariant types, and confined types. We will also show how to compute adapters in order to implement coercions for type casts.

4. Type System: Details and Formal Description

For a better understanding of the following presentation, we will first clarify some assumptions and requirements on the execution system that we have in mind. As already outlined in Sect. 2, we assume a model very similar to that of the Java language. Software components are written in a statically typed programming language and are then compiled into bytecode that preserves the relevant part of the static type information (c.f. Fig. 2). The virtual machine can load new components at run-time. When a component is loaded, a static type checker first checks all assignments present in the loaded code unit. If the type check succeeds, the intermediate code will be compiled

into native machine code incorporating additional run-time type checks where necessary. In the following discussion, we assume that the bytecode does not require explicit type cast operations, but rather includes an implicit type cast operation whenever a value is assigned to a variable (including passing arguments and results to and from methods).

The ultimate goal of our approach is to allow the inclusion of third party software components from possibly untrusted sources in a secure way and thus, encourage software reuse. As the interoperability of software components greatly benefits from supporting both nominal and structural subtyping (Gil & Maman 2008; Malayeri & Aldrich 2009), our type system supports both of these typing disciplines.

4.1. Modeling of Types

In order to formalize the rules of the type system, especially the subtype relation and the conditions for a legal downcast, we model a type as a tuple comprising four components: the machine's *representation* of the type's elements, a *category* classifying the elements, the *permissions* granted by the type and the *protocol* defining the available methods.

1. The **representation** specifies how the execution environment, e.g., the virtual machine or the processor, stores a value of that type. For the following discussion, we distinguish simple data types, e.g., 8-bit integer or 64-bit floating point numbers, and references to objects (or arrays). We thus define $\mathcal{R} = \{int8, ..., float64, reference\}$ as the set of possible representations. The representation is important, since the run-time system may have to change it when a valid type cast is performed.

2. The purpose of the **category** is to capture all properties of the type's elements that are not explicitly modeled otherwise. The type system does not impose any restrictions on what exactly a category is, except that the set $\mathcal{C}$ of categories must have an order $\leq$ with a smallest element $\perp \in \mathcal{C}$, such that $(\mathcal{C}, \leq)$ is a meet-semilattice. The intended meaning of $c_1 \leq c_2$ is that c_2 provides more and/or stronger assertions than c_1, which means that c_2 is a subcategory of c_1.

   The main use of the category is to represent the explicit relationships specified by nominal subtyping. E.g., when a class c_2 is declared as a subclass of c_1, we use c_1 and c_2 with $c_1 \leq c_2$ as the types' categories.

   The category is also used to identify arrays and to capture the possible range of integer and floating point numbers. E.g., we define a category C_{int8} for integers in the range $[-128, 127]$, and similar for the other simple data types. Note that this specification is different from the representation, as we may have a type with elements stored as 16-bit values, which are nevertheless constrained to the range $[-128, 127]$.

3. The **permissions** are modeled as a mapping from identifiers to the states *avail*, *optional*, *denied* and *unavail* introduced above. For reference types, identifiers will include method names, but in general they can also represent other features of the type's elements (see Sections 4.3.3 and 6.3).

4. Finally, for reference types, i.e., if the representation is *reference*, the **protocol** specifies the argument and return types[9] for each (public) method of the referenced object. Again, we model the protocol as a mapping from method names to method signatures[10], where the special value $\perp$ is used, if the type does not specify a signature, e.g., because the named method does not exist.

Our formal representation of types is summarized in Fig. 3.

As an example, the interface `Serviceable` discussed in Sect. 3.2 will be represented like this:

$$T_{\texttt{Serviceable}} = (reference, C_{\texttt{Serv}}, f_{\texttt{Serv}}, p_{\texttt{Serv}}), \text{ with}$$

$$f_{\texttt{Serv}}(x) = \begin{cases} avail & \text{if } x \in \mathcal{N} \\ & \quad \vee\, x = \text{"}\texttt{getStatus}\text{"} \\ optional & \text{if } x = \text{"}\texttt{test}\text{"} \\ denied & \text{otherwise} \end{cases}$$

$$p_{\texttt{Serv}}(x) = \begin{cases} ((), (T_{\texttt{int}})) & \text{if } x = \text{"}\texttt{getStatus}\text{"} \\ & \quad \vee\, x = \text{"}\texttt{test}\text{"} \\ \perp & \text{otherwise} \end{cases}$$

$$T_{\texttt{int}} = (int32, C_{int32}, f_{\texttt{int}}, \lambda x.\perp), \text{ with}$$

$$f_{\texttt{int}}(x) = \begin{cases} avail & \text{if } x \in \mathcal{N} \\ denied & \text{otherwise} \end{cases}$$

In this example, the types do not restrict any features that may be defined by $\mathcal{N}$ (see Sections 4.3.3 and 6.3 for examples). The category $C_{\texttt{Serv}}$ allows to control the subtyping behavior: A subtype of `Serviceable` must have a category C with $C_{\texttt{Serv}} \leq C$. So, if $C_{\texttt{Serv}} = \perp$, any type can be a subtype of `Serviceable`, as long as it is structurally compatible (structural subtyping). If $C_{\texttt{Serv}} \neq \perp$, any subtype must have a category C that was explicitly declared as a subcategory of $C_{\texttt{Serv}}$ (nominal subtyping).

4.2. Subtyping Rules

The formal rules defining the subtype relation in our type system are shown in Fig. 4. Note that since our focus is on the permissions granted and the assertions given by a type, we prefer the notation $T \leq S$ rather than $S <: T$ to denote that S is a subtype of T.

We have already explained how the refined method states influence the subtype relation in Section 3.1. In addition, we require the usual contravariance in argument types and covariance in return types for all methods that are defined (i.e., permitted) by both S and T.

[9] We do not restrict our type system to a single return type.

[10] The type system does not include special support for method overloading. Overloading can be implemented by using a name mangling scheme as in C++.

Type: $\mathcal{T} = \mathcal{R} \times \mathcal{C} \times \mathcal{F} \times \mathcal{P}$

A type is a tuple (representation, category, permissions, protocol).

Representation: $\mathcal{R} = \{int8, int16, int32, int64, float32, float64, reference\}$

Specifies, how elements of this type are stored in the execution environment.

Category: any meet-semilattice $(\mathcal{C}, \leq)$ with

- $\{\bot, C_{int8}, C_{int16}, C_{int32}, C_{int64}, C_{float32}, C_{float64}, C_{class}, C_{adapter}, C_{array}\} \subseteq \mathcal{C}$
- $\forall x \in \mathcal{C} : \bot \leq x$
- $C_{int64} < C_{int32} < C_{int16} < C_{int8}$; $C_{float64} < C_{float32} < C_{int16}$; $C_{float64} < C_{int32}$
- $C_{class} < C_{adapter}$; $C_{class} < C_{array}$

The predefined categories C_{class}, $C_{adapter}$, and C_{array} will be explained in Sect. 4.4 and following.

The other predefined categories represent the value ranges for integer and floating point numbers. Since the order is based on assertion strength we have, e.g., $C_{int64} < C_{int32}$, which actually means that each value allowed by C_{int32} is also allowed by C_{int64}.

Types can freely introduce new categories, as needed. However, subcategories of $C_{adapter}$ and C_{array} may only be created by the type system itself (see Sections 4.4 and 4.6).

Permissions: $\mathcal{F} = \mathcal{A}^{\mathcal{N} \cup \mathbb{A}^*}$

The permissions granted by a type are defined by a total function[a] mapping feature identifiers and strings (i.e., possible method names) to assertions.

Here $\mathcal{N}$ is an arbitrary finite set of feature identifiers, $\mathbb{A}^*$ denotes the set of all strings (i.e., possible method names) and $\mathcal{A} = \{denied, optional, avail, unavail\}$ is the set of possible states.

Protocol: $\mathcal{P} = (\mathcal{M} \cup \{\bot\})^{\mathbb{A}^*}$

A protocol is a total function[a] mapping a method name to a method signature or $\bot$.

Here $\mathcal{M} = \mathcal{T}^* \times \mathcal{T}^*$, representing the method's argument types and its return types.

As an invariant, we require that for any given method name, a type's protocol specifies a method signature, if and only if the type permits that method, i.e., its permissions map the method to one of the states *avail* or *optional*.

[a] Although $\mathbb{A}^*$ is an infinite set, using a wild-card entry still allows the implementation of this function as a lookup-table.

Figure 3 Formal modeling of types.

Given the order of states $s, t \in \mathcal{A}$ specified by the following table:

PoA

$\downarrow t \ \leq \ s \rightarrow$	*denied*	*optional*	*avail*	*unavail*
denied	**T**	**T**	**T**	**T**
optional	**F**	**T**	**T**	**T**
avail	**F**	**F**	**T**	**F**
unavail	**F**	**F**	**F**	**T**

Then, the relation $\leq \subseteq \mathcal{T} \times \mathcal{T}$ is the greatest fixed point of the following rules:

$$\text{PoT}\ \frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ r_T = r_S \\ c_T \leq c_S \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : f_T(x) \leq f_S(x) \\ \forall x \in \mathbb{A}^* : (p_T(x) \neq \bot \wedge p_S(x) \neq \bot) \ \Rightarrow \ p_T(x) \leq p_S(x) \end{array}}{T \leq S}$$

$$\text{PoM}\ \frac{\begin{array}{c} t = (A_t, R_t) \in \mathcal{M} \\ s = (A_s, R_s) \in \mathcal{M} \\ |A_t| = |A_s| \\ |R_t| = |R_s| \\ \forall i \in [1, |A_s|] : (A_s)_i \leq (A_t)_i \\ \forall i \in [1, |R_t|] : (R_t)_i \leq (R_s)_i \end{array}}{t \leq s}$$

Figure 4 Subtype relation: S is subtype of T, if and only if $T \leq S$.

There are two other extensions compared to traditional structural subtyping rules. First the condition $r_T = r_S$ in rule POT indicates that we can only regard an element of type S as also being an element of T, if it already is represented in the way required by T. In all other cases, e.g., when we assign from an 8-bit integer variable to a 16-bit integer variable, the assignment may still be legal as shown in the next section, however, it will definitely require a conversion at run-time.

Second, the condition $c_T \leq c_S$ in POT enables us to handle structural and nominal subtyping in a single framework. As just explained above, a type T supporting structural subtyping has $c_T = \perp$. Thus, the condition will always be satisfied and subtyping is just based on structural compatibility.

For nominal subtyping, we extend $\mathcal{C}$ with a new element c for each class `C` defined in a program. If a class `C'` has been declared as a subclass of `C`, we also extend the order $\leq$ on $\mathcal{C}$ with the element (c, c'), thus ensuring $c \leq c'$ for the representative $c' \in \mathcal{C}$ of class `C'`. The same is true for nominal interfaces. This means that if T has been declared as a nominal type, another type T' can only be subtype of T, if it is structurally compatible *and* was explicitly declared as extending T.

4.3. Legal Casts

As we have seen in Sect. 2, the security of the type-based split capability model depends on the fact that all assignments executed at run-time are type-safe. Thus, we have to answer the question under which conditions an assignment is legal. In the following, we assume that (explicit or implicit) type casts only occur as part of an assignment, including assignments of arguments and results during a method call.

In general, there are two situations when a type check may be performed:

- When a code unit, i.e., a component is loaded, the loader must perform a static type check for all assignments in the loaded code unit.[11] In that way, most type errors will already be detected at load-time rather than later at run-time. Among others, the static type check will verify that the component's code conforms to the specification of the required permissions given by the component's interface. Likewise, if a new component B is connected to another, already loaded component A, i.e., the main class of B is instantiated[12] and a reference to that instance is passed to a method in A, the static type check ensures that B does not require more permissions than A is willing to grant (and vice versa). This means that subsequent run-time errors due to missing permissions are precluded.
  The static type check can have three results: When the checked assignment is illegal, loading the code unit will be aborted with an error. Otherwise, if the type checker determines that there is a chance for the assignment to fail with a type error at run-time, e.g., in case of a downcast, a proper run-time check is inserted into the code. Finally, if the type checker can prove that there cannot be any type error at run-time, the overhead for a run-time type check is avoided.
- At run-time, an inserted type check for an assignment from a variable s of type S (denoted as $s : S$) to another variable $t : T$ ensures that the value in / the object referenced by s can actually be assigned to t. In contrast to the static type check, the run-time check also has access to the type of the actual value or object that is to be assigned.

4.3.1. Static type checks

When performing static type checks while loading a code unit, we consider the situation that a variable $s : S$ is assigned to another variable $t : T$. If the static type check permits the assignment, we must ensure that whenever the assignment is executed at run-time, the value or object $o' : O'$ that is ultimately assigned to t has type T, which means that $O' <: T$ must hold.

In the easiest case, we have $S <: T$, which means that for all objects $o : O$ to which s may refer at run-time (or, for all values which s may contain, respectively),[13] we can simply assign o to t without the need for any further action. This is because we have $O <: S$ by induction hypothesis, which in this case implies $O <: T$.

If $S \not<: T$, we may be able to amend the assignment with a type coercion performed at run-time. I.e., we can transform o into an $o' : O'$ with $O' <: T$ and then assign o'. Traditional type systems usually just introduce coercions for value types, in order to convert the value's representation.

As we have seen in Sect. 3, we sometimes also need to introduce coercions for objects, in the form of a membrane. A membrane $o' : O'$ for an object $o : O$ provides a subset of the methods available in o and delegates them to o. If necessary, the membrane can also create other membranes for arguments and/or return values of the methods. Such a membrane is necessary when the status of some method is *denied* in S, but *optional* in T.

Actually, coercions for objects do not have to be limited to membranes, but can be arbitrary adapters that meet the following properties:

- The adapter must have a subtype of T.
- The adapter can modify the type's representation, its permissions (method states) and the argument and result types of the methods in its protocol, but it cannot modify the category or the methods' semantics.

This means that the adapter delegates some or all of the available methods to o, while applying legal type conversions to the arguments and results. In that way, the type system gains an increased flexibility for supporting code reuse. E.g., it is possible to assign an object with a method `m(int)` to a variable whose type declares `m` as `m(short)`. The adapter will take care of converting the argument from `short` to `int`, which will always succeed.

We intentionally restrict the possibility to create adapters to cases where the type conversion of arguments and results

[11] Even though the high-level language compiler also performs a static type check of the code, a malicious programmer may create the bytecode in some other way, so it must be (re-)checked at load-time.

[12] Remember that we consider a purely object oriented programming model.

[13] To simplify the language, we will refrain in future from explicitly mentioning values contained in variables and rather just speak of objects referenced by variables.

cannot fail at run-time. Otherwise, a programmer may be highly astonished, if a method call fails with a type error, although the method was called with the declared argument and result types. E.g., with the types defined in the example of Sect. 3.2, we are in principle able to create an adapter for a method `testRocket(Testable d)` that accepts an argument of type `Serviceable` and casts it to `Testable` before forwarding the method call. However, if a programmer now calls the adapter's method, passing a `Rocket` object as an argument, the call would fail with a type error (since the `Rocket` object cannot be assigned to a variable of type `Testable`), although `Rocket` is a subtype of `Serviceable`.

4.3.2. Run-time type checks

In general, even if a static cast from S to T is allowed, it may be possible that a variable $s : S$ can refer to some object $o : O$ that cannot be properly converted for a legal assignment to $t : T$. If the static type checker cannot rule out this possibility, it augments the assignment with an additional run-time type check, which determines whether there is some way to convert o into an object of type T. This basically requires that o provides all methods required by T and that the arguments and results of these methods can be properly converted.

4.3.3. Handling Downcasts

In order to prevent an amplification of authority, our type system must be more restrictive than traditional ones. This actually limits the expressiveness of programming languages based on that type system, as for instance, it does not permit a downcast to a subtype. However, if we use the justified assumption that the result of creating a new object o, which should be associated with the subject s creating it, results in a reference r allowing unrestricted access to o, then we can actually allow s to downcast any reference to that object o without restrictions, as this will not result in more authority than s already has by means of r. Exploiting this fact will basically remove all security-related restrictions of the type system, if we stay in the context of a single subject (c.f. the next-to-last paragraph of Sect. 2).

In order to implement this idea, the run-time system must be able to determine the subject s associated with each object o, which can be achieved by storing a subject-ID in each object. At run-time, we then can determine whether o is associated with the same subject that executes the assignment currently being checked.

A meaningful static type checking is enabled by introducing a feature $local \in \mathcal{N}$ (c.f. Fig. 3). For a reference type T involved in a checked assignment, the assertion $f_T(local) = avail$ means that the referenced object is guaranteed to be associated with the subject that performs that assignment, while $f_T(local) = optional$ means that this guarantee cannot be given. At run-time, we now use two different types for an object, depending on whether or not it is associated with the subject executing the assignment being checked. In the first case, the object's type O asserts $f_O(local) = avail$, while in the second case, it asserts $f_O(local) = unavail$.

With that, we can allow an unrestricted downcast from type S to type T, whenever $f_T(local) = avail$ and $f_S(local) \neq unavail$, i.e., T asserts that the referenced object is owned by the subject performing the cast and the cast actually will succeed or, in case of a static type check, at least has a chance to succeed at run-time.

Note that we must now take special care when performing a static type check of an assignment that transfers a reference to some other object o', e.g., when checking the assignment to an argument in a method call. In this case, the assertion $f_S(local) = avail$ of a source type S just guarantees that the referenced object is local for the subject s performing the call. For the subject s' owning o', it is local only if we have $s = s'$. This is guaranteed, if and only if the type R of the reference used for calling the method asserts $f_R(local) = avail$. Thus, if $f_R(local) \neq avail$, we must use a modified type S' as the source type of the type check, where S' differs from S just by having $f_{S'}(local) = optional$. The same consideration applies to the assignment of the results of a method call.

4.3.4. Type checking rules

Fig. 5 shows the definition of the relation $\text{isLegal}(T, S, a, u)$ that determines whether a type conversion from a type S to a type T is permitted. The parameter a is always set to $\mathbf{F}$ in top-level calls of $\text{isLegal}()$. It is set to $\mathbf{T}$ in rule LEGALT in the cases where an unrestricted downcast is possible. Parameter u determines whether the cast may be unsafe, i.e., is allowed to fail at run-time. Thus, for static type checks, we use $u = \mathbf{T}$, for checking a type conversion at run-time $u = \mathbf{F}$.

The type cast is permitted, if and only if the following four conditions hold:

- S and T have the same representation, or both types are value types. In the latter case, we can convert the value into the required representation. Note that a possible range check for values is based on the value type's category, not its representation.
- If the cast is not allowed to fail at run-time, the category c_S of S must be a subcategory of (i.e., a category with stronger or equal assertions than) the category c_T of T. Otherwise, we can ignore the relationship of the category in S and T, since even if c_S is not a subcategory of c_T, there is still a chance that the object assigned at run-time actually has a subcategory of c_T.
- All features and methods that are guaranteed to be available by T must also be guaranteed to be available by S, unless we allow run-time errors. In that case, we have a chance for the cast to succeed at run-time, even if T guarantees that some feature or method x is available, but S does not, provided that S at least allows the use of x or we are allowed to perform an unrestricted downcast.
- For all methods m specified (i.e., permitted) by both S and T, we require that all arguments and results can be legally casted.

A static type check may actually have three different results:

- If we have $S <: T$, the cast is unconditionally legal, and we do not need a check at run-time. Note that $S <: T$ implies $\text{isLegal}(T, S, \mathbf{F}, \mathbf{T})$.
- If we have $\text{isLegal}(T, S, \mathbf{F}, \mathbf{T})$, but $S \not<: T$, some run-time action is required to convert the type. This conversion

The relation isLegal $\subseteq \mathcal{T} \times \mathcal{T} \times \mathbb{B}$ is the greatest fixed point of the following rules:

$$\text{LegalT}\;\frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ a, u \in \mathbb{B} \\ a' = a \vee (f_T(\mathit{local}) = \mathit{avail} \wedge f_S(\mathit{local}) \neq \mathit{unavail}) \\ r_T = r_S \vee r_T \neq \mathit{reference} \wedge r_S \neq \mathit{reference} \\ c_T \leq c_S \vee u \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : \text{isLegal}(f_T(x), f_S(x), a', u) \\ \forall x \in \mathbb{A}^* : (p_T(x) \neq \bot \wedge p_S(x) \neq \bot) \;\Rightarrow\; \text{isLegal}(p_T(x), p_S(x), a', u) \end{array}}{\text{isLegal}(T, S, a, u)}$$

$$\text{LegalA}\;\frac{\begin{array}{c} t, s \in \mathcal{A} \\ a, u \in \mathbb{B} \\ t = \mathit{avail} \;\Rightarrow\; (s = \mathit{avail} \;\vee\; u \wedge (s \notin \{\mathit{denied}, \mathit{unavail}\} \vee a)) \end{array}}{\text{isLegal}(t, s, a, u)}$$

$$\text{LegalM}\;\frac{\begin{array}{c} t = (A_t, R_t) \in \mathcal{M} \\ s = (A_s, R_s) \in \mathcal{M} \\ a, u \in \mathbb{B} \\ |A_t| = |A_s| \\ |R_t| = |R_s| \\ \forall i \in [1, |A_s|] : \text{isLegal}((A_s)_i, (A_t)_i, a, u) \\ \forall i \in [1, |R_t|] : \text{isLegal}((R_t)_i, (R_s)_i, a, u) \end{array}}{\text{isLegal}(t, s, a, u)}$$

Figure 5 Type checking rules: static type checking permits an assignment, if isLegal$(\mathrm{T}, \mathrm{S}, \mathbf{F}, \mathbf{T})$, while a run-time type checks requires isLegal$(\mathrm{T}, \mathrm{S}, \mathbf{F}, \mathbf{F})$.

may succeed or may fail, depending on the concrete value or object that is assigned. The details of computing this run-time action are presented in Sect. 4.4.
– Finally, if $\neg$isLegal$(T, S, \mathbf{F}, \mathbf{T})$, there is no chance that the cast can succeed at run-time, thus, the system may flag an error and refuse to load the code unit.

For static type checks, the equations in Fig. 5 do not impose any restrictions on the category. This is acceptable, since in the case where $c_T \not\leq c_S$, we have $S \not<: T$ and thus, a run-time check will be performed. However, it is desirable that the static type check is as strict as possible. In principle, the run-time check can only succeed, if the type O of the object that is actually assigned has a category c_O with $c_T \leq c_O$. By induction hypothesis, we must always have $O <: S$, which implies $c_S \leq c_O$. This means that if the static type check can prove that there is no $x \in \mathcal{C}$ such that $c_T \leq x$ and $c_S \leq x$ (i.e., there is no least upper bound for c_T and c_S), it can flag a type error. The problem here is that $(\mathcal{C}, \leq)$ may be extended by code units loaded at a later time. However, in the case where c_T and c_S are visible only in their local code unit, the condition for checking the category can be tightened.

4.4. Computing Coercions

Let us again consider the situation where at run-time an object v referenced by a variable $s : S$ should be assigned to another variable $t : T$. So far, we have only discussed whether or not such an assignment is allowed. If the assignment is actually allowed, we may still have to determine a suitable coercion that converts v into an object w that can be directly assigned to t, i.e., the type of w must be a subtype of T.

Note that due to the extension defined in Sect 4.3.3, the type $V_r = (r_{V_r}, c_{V_r}, f_{V_r}, p_{V_r})$ of v actually depends on the subject r receiving v, i.e., the subject that is performing the assignment. I.e., if r is the owner of v, we have $f_{V_r}(local) = avail$, otherwise $f_{V_r}(local) = unavail$. Thus, in the following, V_r and W_r are the types of v and w as seen by that subject, denoted as $V_r = \text{type}_r(v), W_r = \text{type}_r(w)$.

From the three possible results of the assignment's static type check defined in Sect. 4.3.4, we only need to consider the second case, where isLegal$(T, S, \mathbf{F}, \mathbf{T})$ but $S \not<: T$. If $S <: T$, we can directly assign v to t, as (by induction hypothesis) $V <: S$ and thus $V <: T$ must hold. On the other hand, if $\neg$isLegal$(T, S, \mathbf{F}, \mathbf{T})$, the assignment fails.

Now, if T is a value type, it is easy to see that also S and V_r must be value types (rules LEGALT and POT). Thus, we just need to convert the value of v into the representation required by T, e.g., from *int32* to *float64*. We also may (and should) perform a range check during this conversion, based on the actual value of v and the category of T.

Otherwise, T, S and V_r must be reference types. In this case, we may have to build an adapter that properly converts the interface of v. The adapter is an object w that delegates all method calls to the object v, possibly converting the arguments and results as needed. Thus, the code of a method in w typically looks something like this:

```
RW method(AW arg) {
  RV result = delegate.method((AV)arg);
  return (RW)result;
}
```

Here, RW and AW are the result and argument types defined by w, while RV and AV are the result and argument types used in the implementation of v. Note that w does not necessarily need to provide all the methods implemented by v.

The main question now is: how can we determine the exact type W_r (and thus, also the class, i.e., the code) of the adapter object w? Obviously, W_r must be a subtype of T. Since an assignment is not allowed to amplify the authority of the subject r executing the assignment, W_r must not provide more authority than S, except in the case where the object v is owned by r, i.e., $f_{V_r}(local) = avail$. On the other hand, if an operation is provided by v and is actually permitted by S, w should also provide it.

We use a two-step approach to compute W_r. First, if necessary, we determine a restricted subtype R of T that does not provide more authority than S. As this step does not depend on v, it can be executed during static type checking when a code unit is loaded. Later, at run-time, we determine the type W_r from R and V_r.

For a type T and a type S with isLegal$(T, S, \mathbf{F}, \mathbf{T})$, the restricted subtype $R = T \cap_r S <: T$ is computed using the rules shown in Fig. 6. If T is a value type, there is no authority to restrict, thus, $R = T$ (rule RSUBT1). For reference types, we have to restrict T's permissions, such that R does not grant more authority than S (rule RSUBT2). This is done in the following way (c.f., rule RSUBA): If an operation x is not allowed by T or is allowed by S, there is no need for a restriction, thus, $f_R(x) = f_T(x)$. However, if x is allowed by T, but is not allowed by S (i.e., $f_S(x) \in \{denied, unavail\}$), the restricted subtype must also not allow it. As $T \cap_r S$ is only defined if isLegal$(T, S, \mathbf{F}, \mathbf{T})$, we just have to consider the case $f_T(x) = optional$. A look at rule POA shows that we have to set $f_R(x) = unavail$ (since $optional \leq unavail$). The last equation in rule RSUBT2 ensures that the invariant $p_R(x) \neq \perp \Leftrightarrow f_R(x) \in \{avail, optional\}$ specified in Fig. 3 is met.

Note that the restriction of permissions must be done recursively, i.e., also for all argument and results types of the methods specified in T (rules RSUBM and RSUBTS). Since argument types are contravariant, we also need to define a restricted supertype $R' = T \cap^r S$ of S that does not provide more authority than T. The rules for this restricted supertype are shown in Fig. 7. Similar as before, we can leave the permissions of S unchanged, if an operation is not allowed by S or is allowed by T (rule RSUPA). The restricted supertype must not allow an operation x, if is allowed by S, but not by T. In these cases, we set $f_{R'}(x) = denied$ (since $denied \leq optional$ and $denied \leq avail$).

When the actual assignment is performed at run-time, in addition to S, T and $R = T \cap_r S$, we also know the object v and its type V_r as seen by the receiving subject r. We then follow the procedure outlined in Algorithms 1 and 2. Note that the algorithms also include the case where v is transferred to another subject, via an assignment to an argument or result value of a

Given the relation $t \cap_r s$ for states $s, t \in \mathcal{A}$ specified by the following table:[a]

rSubA

$\downarrow t \cap_r s \rightarrow$	*denied*	*optional*	*avail*	*unavail*
denied	*denied*	*denied*	*denied*	*denied*
optional	*unavail*	*optional*	*optional*	*unavail*
avail	—	*avail*	*avail*	—
unavail	*unavail*	*unavail*	*unavail*	*unavail*

Then, for two types S and $T = (r_T, c_T, f_T, p_T)$, such that $f_T(\mathit{local}) \neq \mathit{avail}$ and isLegal$(T, S, \mathbf{F}, \mathbf{T})$, the restricted subtype $T \cap_r S$ is defined by the largest fixed point of the following rules:

$$\text{rSubT1}\ \frac{T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \quad S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \quad r_T \neq \mathit{reference}}{T \cap_r S = T}$$

$$\text{rSubT2}\ \frac{\begin{array}{c} T = (\mathit{reference}, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ R = (\mathit{reference}, c_T, f_R, p_R) \in \mathcal{T} \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : f_R(x) = f_T(x) \cap_r f_S(x) \\ \forall x \in \mathbb{A}^* : p_R(x) = \begin{cases} p_T(x) \cap_r p_S(x) & \text{if} \quad p_T(x) \neq \bot \wedge p_S(x) \neq \bot \\ \bot & \text{otherwise} \end{cases} \end{array}}{T \cap_r S = R}$$

$$\text{rSubM}\ \frac{t = (A_t, R_t) \in \mathcal{M} \quad s = (A_s, R_s) \in \mathcal{M}}{t \cap_r s = (A_s \cap^r A_t, R_t \cap_r R_s)}$$

$$\text{rSubTS}\ \frac{T, S, R \in \mathcal{T}^* \quad |T| = |S| = |R| \quad \forall i \in [1, |R|] : R_i = T_i \cap_r S_i}{T \cap_r S = R}$$

[a] Note that $t \cap_r s$ is undefined for $t = \mathit{avail}$ and $s \in \{\mathit{denied}, \mathit{unavail}\}$. However, this combination does not occur when computing a restricted subtype due to the restricted domain of this operation.

Figure 6 Restricted subtype.

Algorithm 1 Run-time assignment from variable s to variable t.

Require: The static type check for the assignment $t \leftarrow s$ succeeded
1: $r \leftarrow$ the subject owning the variable t
2: $T \leftarrow$ the type of t
3: $S \leftarrow$ the type of s
4: $v \leftarrow$ the value of / object referenced by s
5: $v' \leftarrow \text{cast}_{r,T,S}(v)$
6: **if** $v' = \bot$ **then**
7:     **raise** run-time error
8: **else**
9:     $t \leftarrow v'$

Given the relation $t \cap^r s$ for states $s, t \in \mathcal{A}$ specified by the following table:

rSupA

$\downarrow t \cap^r s \rightarrow$	*denied*	*optional*	*avail*	*unavail*
denied	*denied*	*denied*	*denied*	*unavail*
optional	*denied*	*optional*	*avail*	*unavail*
avail	*denied*	*optional*	*avail*	*unavail*
unavail	*denied*	*denied*	*denied*	*unavail*

Then, for two types S and T, such that isLegal$(T, S, \mathbf{F}, \mathbf{T})$, the restricted supertype $T \cap^r S$ is defined by the largest fixed point of the following rules:

$$\text{rSupT1}\;\frac{T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \quad S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \quad r_S \neq \textit{reference}}{T \cap^r S = S}$$

$$\text{rSupT2}\;\frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (\textit{reference}, c_S, f_S, p_S) \in \mathcal{T} \\ R = (\textit{reference}, c_T, f_R, p_R) \in \mathcal{T} \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : f_R(x) = f_T(x) \cap^r f_S(x) \\ \forall x \in \mathbb{A}^* : p_R(x) = \begin{cases} p_T(x) \cap^r p_S(x) & \text{if} \quad p_T(x) \neq \bot \wedge p_S(x) \neq \bot \\ \bot & \text{otherwise} \end{cases} \end{array}}{T \cap^r S = R}$$

$$\text{rSupM}\;\frac{t = (A_t, R_t) \in \mathcal{M} \quad s = (A_s, R_s) \in \mathcal{M}}{t \cap^r s = (A_s \cap_r A_t, R_t \cap^r R_s)}$$

$$\text{rSupTS}\;\frac{T, S, R \in \mathcal{T}^* \quad |T| = |S| = |R| \quad \forall i \in [1, |R|] : R_i = T_i \cap^r S_i}{T \cap^r S = R}$$

Figure 7 Restricted supertype.

Algorithm 2 Run-time cast.

1: **function** $\text{cast}_{r,T,S}(v)$
2: $\quad (r_T, c_T, f_T, p_T) \leftarrow T$
3: $\quad (r_S, c_S, f_S, p_S) \leftarrow S$
4: $\quad V_r \leftarrow \text{type}_r(v)$ ▷ The type of v as seen by the receiving subject
5: $\quad$ **if** $r_T \neq \textit{reference}$ **then**
6: $\quad\quad$ **return** $\text{coerce}_{r,T}(v)$ ▷ Convert the value
7: $\quad$ **else if** $f_T(\textit{local}) = \textit{avail}$ **then** ▷ No access restrictions
8: $\quad\quad$ **return** $\text{coerce}_{r,T}(\text{unwrap}(v))$
9: $\quad$ **else**
10: $\quad\quad R \leftarrow T \cap_r S$
11: $\quad\quad$ **if** $v = \text{unwrap}(v)$ **then** ▷ Enforce access restrictions imposed by S
12: $\quad\quad\quad$ **return** $\text{coerce}_{r,R}(v)$
13: $\quad\quad$ **else if** isLegal$(R, V_r, \mathbf{F}, \mathbf{T})$ **then** ▷ Enforce access restrictions imposed by both S and the previous adapter
14: $\quad\quad\quad R' \leftarrow R \cap_r V_r$
15: $\quad\quad\quad$ **return** $\text{coerce}_{r,R'}(\text{unwrap}(v))$
16: $\quad\quad$ **else**
17: $\quad\quad\quad$ **return** $\bot$

Algorithm 3 Auxiliaries: Unwrap an adapter, compute the coercion of a value or object, create an adapter.

1: **function** unwrap(v)
2: $(r_V, c_V, f_V, p_V) \leftarrow \text{type}(v)$ ▷ The type of v as seen by its owner
3: **if** $v \neq \text{null} \ \wedge \ C_{adapter} \leq c_V$ **then**
4: **return** the object wrapped by v
5: **else**
6: **return** v
7: **function** coerce$_{r,R}(v)$
8: $(r_R, c_R, f_R, p_R) \leftarrow R$
9: **if** $r_R = \textit{reference} \wedge v = \text{null}$ **then**
10: **return** null
11: **else**
12: $V_r \leftarrow \text{type}_r(v)$ ▷ The type of v as seen by the receiving subject
13: **if** $\neg\text{isLegal}(R, V_r, \mathbf{F}, \mathbf{F})$ **then**
14: **return** $\bot$ ▷ Assignment will raise a run-time error
15: **else if** $R \leq V_r$ **then**
16: **return** v ▷ v already has type R
17: **else if** $r_R \neq \textit{reference}$ **then**
18: **if** v fits into the representation r_R **then**
19: **return** the value of v converted into the representation r_R
20: **else**
21: **return** $\bot$
22: **else**
23: **return** wrap$_R(v)$
24: **function** wrap$_R(v)$
25: $(r_R, c_R, f_R, p_R) \leftarrow R$
26: $(r_V, c_V, f_V, p_V) \leftarrow \text{type}(v)$ ▷ The type of v as seen by its owner
27: $c_W = $ new category with $c_V \leq c_W \wedge C_{adapter} \leq c_W$
28: $f_W = \lambda x. \begin{cases} \textit{avail} & \text{if } x = \textit{local} \ \vee \ (f_V(x) = \textit{avail} \wedge f_R(x) \notin \{\textit{denied}, \textit{unavail}\}) \\ \textit{unavail} & \text{otherwise} \end{cases}$
29: $p_W = \lambda x. \begin{cases} p_R(x) & \text{if } f_V(x) = \textit{avail} \\ \bot & \text{otherwise} \end{cases}$
30: $w \leftarrow$ new object with the following properties:
31: $-$ $\text{type}(w) = (r_R, c_W, f_W, p_W)$
32: $-$ $\text{owner}(w) = \text{owner}(v)$
33: $-$ w forwards all its method calls to v
34: **return** w

method call. As explained at the end of Sect. 4.3.3, an assertion $f_S(local) = avail$ given by the source type S may no longer be valid for the receiving subject. Thus, in principle we might have to remove that assertion by setting $f_S(local) = optional$. However, since S is only used to compute the restricted subtype R, this difference is irrelevant here, since rule RSUBA gives the same result for *avail* and *optional*.

In addition to the assignment of a simple object, we also need to handle the case where v already is an adapter. Now, if the assignment's target type T specifies $f_T(local) = avail$, we have the guarantee that either v is owned by the receiving subject r or the assignment will fail in the process. Thus, we should not impose any access restrictions on v, which means that we can ignore the restrictions specified by S, but also those enforced by a potential adapter. I.e., if v actually is an adapter, we can unwrap it and continue with the original object instead. In any case, we still need to convert the object to the target type T, which may require a new adapter that properly converts argument and return values of some methods. Using the two auxiliary functions defined in Algorithm 3, this means that we just assign $\text{coerce}_{r,T}(\text{unwrap}(v))$ (line 8, Alg. 2). The function coerce may return the special value $\perp$, which denotes that a run-time error must be raised (line 6-7, Alg. 1). Thus, if v actually is not owned by the receiving subject, the assignment fails.

If the target type T does not assert $f_T(local) = avail$, we must take care that the access restrictions imposed by the source type S are still enforced after the assignment. For this effect, we use the restricted subtype $R = T \cap_r S$ as the new target type. So, if v is not an adapter, we can just assign $\text{coerce}_{r,R}(v)$ (line 11-12, Alg. 2). In the other case, we do not wrap the adapter into another one, but rather compute a new adapter for the original object that performs the function of both of them. To that end, we must include the access restrictions of the old adapter in the target type, i.e., we have to use $R' = R \cap_r V_r$ as the new target type (note that V_r manifests the old adapter's access restrictions). As $R \cap_r V_r$ only exists if $\text{isLegal}(R, V_r, \mathbf{F}, \mathbf{T})$, we need to check this condition and raise a run-time error, if it is not fulfilled.[14] Otherwise, we assign $\text{coerce}_{r,R'}(\text{unwrap}(v))$ (line 13-17).

Algorithm 3 shows how a coercion of a value or object v of type V_r to a target type R is computed. Lines 9-16 handle the trivial cases, where no coercion is required or the assignment is illegal. Otherwise, if v is a value, it is converted into the required representation, if possible (lines 17-21). If v is an object, we return an adapter with type R (line 23). To do so, we create a new category c_W that is a subcategory of both c_V and $C_{adapter}$ (line 27), in order to mark w as being an adapter. The adapter is owned by the same subject as the original object (line 32) and provides all features and methods that are both available in v and permitted by R. (lines 28, 29, and 31). All methods are just forwarded to v, however, this may include type casts for arguments and results.

Note that setting $\text{owner}(w) = \text{owner}(v)$ implies that if $\text{owner}(v) = r$, $\text{type}_r(w)$ will assert *local*, which means that we cannot create a coercion for the case where $f_R(local) = unavail$. While this case could be handled by setting $\text{owner}(w)$ to some reserved subject, we simply disallow that any type T of a variable specifies $f_T(local) \in \{denied, unavail\}$, as restricting a reference's target to non-local objects does not make much sense, anyway.

4.5. Classes

Up to now, we have restricted our discussion to reference types, where we only considered the public methods offered by the referenced objects. I.e., we just looked at interface types. However, we also must be able to adequately express the type of an object as an instance of a class, including things like non-public methods and attributes. As a security constraint, we require that these elements of an object can only be accessed by the subject owning that object. In other words, they are only accessible via a reference with type $T = (r_T, c_T, f_T, p_T)$ and $f_T(local) = avail$. Since these non-public elements are not otherwise relevant in the context of access control, our type system is independent of their exact nature and does not need to model them explicitly. Rather, we just represent classes as categories. Any instance of a class C created via `new` C then receives the type $\text{type}(C) := (reference, C, f_C, p_C)$ with

- $f_C(x) = avail$ for $x = local$ and all other features or methods that are actually provided by class C, otherwise $f_C(x) = unavail$,
- $p_C(m) = (A_m, R_m)$, where A_m and R_m are the argument and return types of the method m as implemented by class C.

The owner of the new instance usually is the subject executing the `new` operation. However, we can also allow to extend the `new` operator in such a way that a new subject is created together with the new object.

For consistency reasons, we require a number of constraints for classes and class types:

- First, for each class $C \in \mathcal{C}$, the relation $C_{class} \leq C$ must hold in order to mark C as being a class.
- If D is defined as being a subclass of C, i.e., $C_{class} \leq C \leq D$, $\text{type}(D)$ must be a subtype of $\text{type}(C)$, i.e., $\text{type}(C) \leq \text{type}(D)$.
- For any type $T = (r_T, c_T, f_T, p_T)$ with $C_{class} \leq c_T$, the type T' that only differs from T by specifying $f_{T'}(local) = avail$ must be a supertype of $\text{type}(c_T)$, i.e., $T' \leq \text{type}(c_T)$.

The second constraint ensures the consistency between subclassing and subtyping, while the last one safeguards consistency between classes and the reference types using them. If a type T specifies that the referenced object must be an instance of a given class c_T, it actually doesn't make sense to require at the same time some conditions on features or methods that c_T does not meet. The only exception here is the feature *local* which does not depend on the object itself, but its owner.

A subtle problem with classes is the construction of adapters based on delegation as shown in Algorithm 3. The adapter is an object of a class c_W with $c_V \leq c_W$ *by definition*. However,

[14] In contrast to the statement given in Sect. 4.3.4, we can and must allow an unsafe cast here, since even if v cannot be converted to R without the possibility of a run-time error, we still may have a chance after unwrapping v.

c_W can not really be a subclass of c_V, because the non-public elements of c_V cannot be accessed via the adapter. We show in the appendix (Theorem 7) that this situation can only occur for assignments with a target type T where $f_T(local) \neq avail$, which means that T does not allow to access the non-public elements anyway.

4.6. Arrays

In order to smoothly integrate arrays into the type system, we model arrays as objects of special classes implementing the array's access methods. E.g., the type for a one-dimensional array of `float` values will provide the methods

```
float read(int index);
void write(int index, float value);
int length();
```

and will specify a category c with $C_{array} \leq c$. The big advantage of this representation is that we do not need any special handling within the type system. This also implies that arrays can be assigned to any reference with a type permitted by the rules in Fig 5, thus, e.g. also allowing read-only array types. It is also easily possible to support multidimensional arrays or to add additional methods, like

```
void swap(int index1, int index2);
```

The latter enables the definition of a restricted type, which allows to modify the order of the array elements while still denying arbitrary write access to the array.

A consequence of this modeling is, of course, that the bytecode based on the type system must represent array accesses as explicit method calls. Without some special handling, this would be prohibitively expensive at run-time. However, when compiling the bytecode to machine code, we can avoid the overhead of a method call by replacing it with a direct memory access, provided that static type checking proves that the target object of the call must always be a native array. As the predefined category C_{array} is not exposed to the bytecode, array types can only be created via constructors in the type system. Thus, if the reference used for the call has a type $T = (r_T, c_T, f_T, p_T)$ with $C_{array} \leq c_T$, we can be sure that the target object is a native array.

5. An Extended Example

In order to illustrate the working of the proposed type system, we extend the example given in the introduction as shown in Fig. 8. The service technician now receives a reference to a rocket base that allows to get access to all the rockets in that base. The service technician specifies an interface to the rocket base that only allows to call the `getStatus()` method, and the `test()` method, if available. This means that the president can pass a reference to the rocket base without any further action at run-time, and can still be sure that the service technician cannot launch any of the rockets.

Furthermore, if the president wants to deny access to the `test()` method, he can simply "cast away" that method when passing the rocket base, as shown in the middle right section of Fig. 8. Since the status of `test()` in `Serviceable` is *optional*, the assignment of the reference is still legal (c.f. rule LEGALA in Fig. 5). However, as `NonTestable` is no longer a subtype of `Serviceable`, `NonTestableBase` is not a subtype of `ServiceableBase`. Thus, the assignment of the argument in the call to `service()` must be handled at run-time according to Algorithms 1 and 2. Since the method does not require its argument to be *local*[15] and the actual argument is not already wrapped into an adapter, we coerce it into the type R = `ServiceableBase` $\cap_r$ `NonTestableBase` (c.f. lines 10-12 of Algorithm 2). The computation of this type first does two covariant recursions before it finally computes `Serviceable` $\cap_r$ `NonTestable`. Since the status of `test()` is *optional* in `Serviceable` and *denied* in `NonTestable`, the resulting status is *unavail* (rule RSUBA). So the result for R is the type shown in the lower left of Fig. 8.

The adapter class computed according to the rules in Algorithm 3 is just an implementation of this interface, where all methods are simply delegated to the adapter's delegate object. The method `service()` now receives an instance of `BaseAdapter` as its argument. When it calls `getRockets()` on this adapter, the type cast of the return value (from `Rocket[]` to `rServiceableArray`) will result in the creation of an `ArrayAdapter` instance via the same argumentation as above. Finally, reading an element via the `read()` method will return a `RocketAdapter` that does not provide the ability to invoke the `test()` method.

6. Extensions

In the following subsections, we present some extensions that we have implemented in the type system presented in Sect. 4.

6.1. Unspecified Reference Type

In practice, there is a need for a type system to also support an unspecified (generic) reference type. This is required, for instance, if a subject requests the run-time system to dynamically load a new component specified by its name. In that case, the run-time system must return a reference to some object contained in that component, otherwise, OCap will not allow to interact with it. As there is no way to know the type of this reference in advance, we need a generic reference type. Generic references also are required for implementing generic containers, etc.

In languages with traditional type systems, a reference type not defining any (application-specific) methods can be used for that purpose, e.g., `void*` in C++ or `Object` in Java. In our type system, however, such a type would forbid to call any (application-specific) method on the referenced object. Downcasting the reference is only possible for the owner of the object, so this is not a solution. E.g., for a component loaded at run-time, the run-time system should create a new subject owning that component, in order to isolate it from its environment.

Thus, we need to introduce a new reference type $T_{\texttt{ref}} = (reference, \perp, f, p)$ that does not specify any required method

[15] The status of the feature *local* is actually not shown in Fig. 8, however, it should be clear that the service technician does not require the rocket base to be owned by himself.

```
// Interface as provided by the
// rocket base
class Rocket {
  public int getStatus() { ... }
  public int test() { ... }
  public void launch() { ... }
}
class RocketBase {
  public Rocket[] getRockets() { ... }
}
```

```
// Interface as required by the service
// technician
interface Serviceable {
  int getStatus();
  optional int test();
}
interface ServiceableArray {
  int length();
  Serviceable read(int i);
}
interface ServiceableBase {
   ServiceableArray getRockets();
}
class ServiceTechnician {
  public void service(ServiceableBase b) {
    ...
  }
}
```

```
// Interface used by the president in
// order to further restrict authority
interface NonTestable {
  int getStatus();
}
interface NonTestableArray {
  int length();
  NonTestable read(int i);
}
interface NonTestableBase {
  NonTestableArray getRockets();
}
```

```
// Code executed by the president
// to mandate the service technician
ServiceTechnician st;
RocketBase base;
... // Initialize st and base
...
// The cast results in creating a membrane
st.service((NonTestableBase) base);
```

```
// Restricted subtypes
interface rServiceable {
  int getStatus();
  unavail test(); // status: unavail
}


interface rServiceableArray {
  int length();
  rServiceable read(int i);
}




interface rServiceableBase {
  rServiceableArray getRockets();
}
```

```
// Adapter classes
class RocketAdapter {  // implements
  Rocket delegate;     // rServiceable
  int getStatus() {
    return delegate.getStatus();
  }
}
class ArrayAdapter {   // implements
  Rocket[] delegate;   // rServiceableArray
  int length() { ... }
  rServiceable read(int i) {
    // cast will create a RocketAdapter
    return (rServiceable)delegate[i];
  }
}
class BaseAdapter {    // implements
  RocketBase delegate; // rServiceableBase
  rServiceableArray getRockets() {
    // cast will create an ArrayAdapter
    return (rServiceableArray)
      delegate.getRockets();
  }
}
```

Figure 8 An example for restricting authority at run-time.

signature, but still allows to call all methods of the referenced object. An obvious way to achieve this is to let f map all methods to a new method state *permitted*, while p maps all methods to $\bot$. The state *permitted* does not provide any guarantee on the method's signature, but still allows to call it, which means that we have *denied* $<$ *permitted* $<$ *optional*. With this definition, handling this state properly in the rules for subtyping, legal assignment and restricted types requires no further modification for the covariant part of these rules:

- An assignment where $T_{\texttt{ref}}$ is the target and S the source type is always legal, but we may need to introduce a membrane that enforces the access restrictions imposed by S.
- In the reversed case, we always need a run-time type check to ensure that the assigned object actually has type S.

The contravariant case, which occurs when $T_{\texttt{ref}}$ is used as an argument type of a method, however, is more complicated. Recall the example from Sect. 2 and assume that we assign $r_s : T$ to some variable $u : U$ with U defined as

```
interface U {
  void pass(ref arg);
}
```

Since *AT* may provide less authority than *AS*, we need to introduce an adapter (i.e., membrane) that prevents an amplification of authority when passing the argument to the method `pass()`. As this adapter will be assigned to u, its type W must be a subtype of U. This in turn means that the argument type *AW* of `pass()` in W must be a supertype of $T_{\texttt{ref}}$. More concrete, the construction of the adapter outlined in Sect. 4.4, where the adapters for arguments (and results) of methods are (implicitly and recursively) built by type-casting these arguments, requires that $AW = AT \cap^r T_{\texttt{ref}}$.

How can we construct such a type? Obviously, the methods in *AW* must specify a signature, otherwise we are not able to represent the access restrictions imposed by *AT*. So we need a new method state, named *restricted*, that requires a signature and fulfills the condition *restricted* $<$ *permitted*. This state permits to call the method, but only via the specified signature, and only if the actually called method has a compatible signature. This means that when we cast from a type S to a type T where the state of some method m is *restricted* in S, we have to check the signature specified in S versus the signature in T. As a result, we may have to build a membrane in order to enforce the access restrictions specified by S. On the other hand, if the state of m is *restricted* in T, the signature specified in T must not influence the result of the type check. To see that, consider the situation where T has been constructed as a supertype of $T_{\texttt{ref}}$, i.e., $T \leq T_{\texttt{ref}}$. Now, S can be casted to $T_{\texttt{ref}}$, independent of the signature of m in S. This, however, also means that S can be casted to the supertype T, independent of the signature of m in both S and T.

Although the method signature is not specified with the state *permitted* and is ignored in the rules where the target state is *restricted*, we cannot simply omit the recursive inspection of the method's argument and return types during a type check, because this would result in ignoring the access restrictions imposed by these types. In fact, the unspecified signature denotes that the argument and return types are not restricted. Thus, for the recursive inspection, we can replace the unspecified signature with the most general signature that still is compatible with the other signature, thus avoiding the introduction of a large number of special cases in the typing rules in Figures 4 to 7. Fig. 9 shows the two functions that replace the method signature at the source and the target position of the typing rules, if needed. E.g., the last condition of the rule POT is then modified to

$$\forall x \in \mathbb{A}^* : (s_T \neq \bot \wedge s_S \neq \bot) \Rightarrow s_T \leq s_S$$
$$\text{with } s_T = \text{sDst}(f_T(x), p_T(x), f_S(x), p_S(x))$$
$$\text{and } s_S = \text{sSrc}(f_T(x), p_T(x), f_S(x), p_S(x))$$

A careful look at Fig. 4 and 5 as well as Algorithms 1 to 3 shows that every reference type can be assigned to $T_{\texttt{ref}}$, although not every reference type is a subtype of $T_{\texttt{ref}}$, because in case that the source type maps a possible method name to state *denied*, we may need to introduce a membrane in order to avoid an amplification of authority. On the other hand, when $T_{\texttt{ref}}$ is assigned to some other reference type T, the static type check always succeeds (i.e., isLegal$(T, T_{\texttt{ref}}, \mathbf{F}, \mathbf{T})$ holds), but since normally we have $T \not\leq T_{\texttt{ref}}$, there will be a run-time action computing a coercion. This will perform the necessary run-time type check between the actually assigned object and T (c.f. line 13 in Algorithm 3).

6.2. Unsafe Subtyping

Especially when using arrays or other containers, it is often desirable that a type system supports unsafe covariance. I.e., a variable of type S can be assigned to a variable of type T, although the co- and contravariance rules for the results and arguments for some method declared by S and T are not met (Meijer & Drayton 2004). For instance, in Java, an assignment like

```
Object[] a = new String[10];
```

is permitted. While this is not a problem when reading from the array, writing with a wrong type, as in

```
a[0] = new Integer(1);
```

will result in a run-time error.

In the presented type system, we can support a generic form of unsafe subtyping by the following extension (see Appendix B):

- Method signatures are extended by two sets specifying (the indices of) the arguments and return values that are required to be safe. Here, "safe" means that no run-time type error will be raised when a subtype of the argument type is passed to the method or a supertype of the result type is used to receive the result.
  In Fig. 3, we then have $\mathcal{M} = \mathcal{T}^* \times \wp(\mathbb{N}) \times \mathcal{T}^* \times \wp(\mathbb{N})$.
- In the subtyping rule POM, where we now have $t = (A_t, a_t, R_t, r_t)$ and $s = (A_s, a_s, R_s, r_s)$, we add the conditions $a_t \subseteq a_s$ and $r_t \subseteq r_s$. This ensures that for $T \leq S$ to hold, the methods of S must be at least as safe as the

unspec : $\mathcal{T} \to \mathcal{T}$

$$\text{unspec}((r, c, f, p)) := \begin{cases} T_{\texttt{ref}} & \text{if} \quad r = \textit{reference} \\ (r, \bot, f, p) & \text{otherwise} \end{cases}$$

This function returns the most general type with the same representation as the given one.

unspec : $\mathcal{M} \to \mathcal{M}$

$$\text{unspec}((A, R)) := ((\text{unspec}(A_i))_{i=1..|A|}, (\text{unspec}(R_i))_{i=1..|R|})$$

This function returns a method with the same number of arguments and results as the input, but with the most general types.

sDst : $\mathcal{A} \times (\mathcal{M} \cup \{\bot\}) \times \mathcal{A} \times (\mathcal{M} \cup \{\bot\}) \to (\mathcal{M} \cup \{\bot\})$

$$\text{sDst}(a_T, m_T, a_S, m_S) := \begin{cases} \text{unspec}(m_S) & \text{if} \quad a_T = \textit{permitted} \\ & \quad\quad \wedge\ a_S \in \{\textit{restricted}, \textit{optional}, \textit{avail}\} \\ & \quad \vee\ a_T = \textit{restricted} \\ & \quad\quad \wedge\ a_S \in \{\textit{optional}, \textit{avail}\} \\ m_T & \text{otherwise} \end{cases}$$

For given method states and method signatures in a target type T and a source type S, sDst(a_T, m_T, a_S, m_S) returns the signature that should be checked against the signature m_S in the source type. If the method is *permitted* in T and the source type specifies a signature, the returned signature is the most general one compatible with m_S. The same is true if the method is *restricted* in T and *optional* or *avail* in S in order to ensure transitivity of the subtype relation.

sSrc : $\mathcal{A} \times (\mathcal{M} \cup \{\bot\}) \times \mathcal{A} \times (\mathcal{M} \cup \{\bot\}) \to (\mathcal{M} \cup \{\bot\})$

$$\text{sSrc}(a_T, m_T, a_S, m_S) := \begin{cases} \textit{unspec}(m_T) & \text{if} \quad a_S = \textit{permitted} \\ & \quad\quad \wedge\ a_T \in \{\textit{restricted}, \textit{optional}, \textit{avail}\} \\ m_S & \text{otherwise} \end{cases}$$

For given method states and method signatures in a target type T and a source type S, sSrc(a_T, m_T, a_S, m_S) returns the signature that should checked against the signature m_T in the target type. If the method is *permitted* in S and the target type specifies a signature, the returned signature is the most general one compatible with m_T.

Figure 9 Creating a matching unspecified signature.

corresponding methods in T, ensuring that objects of type S can be used wherever type T is required.

- As the rules for legal type casts in Fig 5 already cover the case where subsequent run-time errors are permitted, we just need to set $u = \mathbf{T}$ in the recursive calls when checking an unsafe argument or result. I.e., we replace the corresponding conditions in LEGALM with

$$\forall i \in [1, |A_s|] : \text{isLegal}((A_s)_i, (A_t)_i, a, u \vee i \notin a_t)$$
$$\forall i \in [1, |R_t|] : \text{isLegal}((R_t)_i, (R_s)_i, a, u \vee i \notin r_t)$$

- Finally, rules RSUBM and RSUPM in Fig. 6 and 7 must be modified such that each method in the restricted type receives the safety attributes of the corresponding method in the target type:

$$t \cap_r s = (A_s \cap^r A_t, a_t, R_t \cap_r R_s, r_t)$$
$$t \cap^r s = (A_s \cap_r A_t, a_s, R_t \cap^r R_s, r_s)$$

6.3. Confined Types

In order to demonstrate the benefits of having introduced additional features to a type, we show how to implement confined types (Bokowski & Vitek 1999; Clarke et al. 1998; P. Müller & Poetzsch-Heffter 1999) in our type system. Expressed in the terms introduced in Sect. 2, a confined type gives the guarantee that a reference of that type can never be passed to another subject. This means that under all circumstances an object with a confined type cannot be accessed by any other subject than its owner.

All we need to do for that is to extend the set $\mathcal{N}$ of feature identifiers in Fig 3 with a new element *transferable*. In addition, we perform the following *static* check at load-time: whenever a method is called using a reference variable with a type $T = (r_T, c_T, f_T, p_T)$ where $f_T(\mathit{local}) \neq \mathit{avail}$ (i.e., the call may pass arguments to or receive results from a different subject), the types of all the passed actual arguments as well as all the return types of the called method must map the *transferable* feature to *avail*. The rest, including the potential creation of an adapter when the *transferable* feature is "cast away", is handled by the type system without any modifications.

In a similar fashion, it is also possible to integrate other reference annotations, like *read-only* or *borrowed* (Boyland et al. 2001) into the type system.

7. Execution Environment

A type system alone is of course not sufficient to ensure security properties at run-time. There are some additional requirements that the run-time system has to meet.

First of all, it must provide **type safety**. This implies that the run-time system must enforce the use of a typed intermediate representation (bytecode) as the only means to introduce new code into the system. The bytecode must be type-checked according to the rules outlined in Sect. 4.3.4. Ideally, the code should be statically type checked at load time and rejected in case of a type error, in order to avoid excessive type checking at run-time. A specific issue in this context is the stack used for passing arguments and results to and from methods. As the stack is a generic data structure, its elements cannot be statically typed. While static checking is still possible using verification techniques (Leroy 2003), our COSMA virtual machine (Wismüller & Ludwig 2019) avoids exposing the stack to the intermediate language by providing a call instruction that explicitly specifies the argument and result variables, together with their types.

Second, the run-time system must provide **object encapsulation** and **memory safety** (Miller 2006). This means that (except for value types), data in memory can only be accessed via references that cannot be forged. I.e., the only way to create a reference is by creating a new object. It must not be possible to create or modify a reference at will, using e.g. an integer-to-pointer conversion or pointer arithmetic. Also, accessing data by exceeding the bounds of an array must be prevented by proper bound-checks. Another problem is static mutable state that can be accessed without having a reference, as this allows to pass information, especially capabilities, between subjects without having a capability permitting that. Instead of forbidding static mutable state, i.e. static class members, at all, COSMA deals with this problem by providing each subject defining or using a class with its own private copy of the static members.

Finally, the run-time system must be able to **create the code for the adapter classes** as needed. This may be done on demand only or can already be done when loading a code unit, using data flow analysis to determine which value or object types may reach a given assignment in the code.

A possible realization of the run-time system is to use an ahead-of-time compiler, as implemented in COSMA. When a code unit is loaded, we create all possibly required adapter classes in advance after the static type check of the code. Then, the intermediate code is compiled into native machine code. As long as we do not require a run-time action for an assignment (see Sect. 4.3.4), the efficiency of this machine code is not restricted by the type system.

If there is a need for a run-time action, the compiler amends the assignment with the code computing the coercion. For reference types, this code is outlined in Algorithm 4. Using a unique ID identifying the assignment and the type of the assigned object as the key, a description of the necessary actions is fetched from a precomputed hash table (line 4). If there is no entry, the source object cannot be assigned, i.e., one of the type checks in line 13 of Algorithm 2 or line 13 of Algorithm 3 failed when the required adapter classes were computed for this assignment. Otherwise, any old adapter is removed and the object is wrapped into a new adapter, if necessary.

In order to execute the run-time cast action, the virtual machine must be able to determine the owner and the type for each object. Since the required adapter classes are precomputed at load time, the type is just used as a hash table key, so no detailed type information is required at run-time. Thus, types can simply be represented by integer numbers. For value types, we additionally need the representation for computing conversions, while for reference types actually adopted by objects, we also have to store the corresponding non-*local* type.

Algorithm 4 Run-time cast action for assigning a reference to an object o to a variable t.

```
1: if o ≠ null then
2:     id = unique integer identifying this assignment
3:     r = owner(o)
4:     a = castAction(id, type_r(o))                    ▷ Hash-table lookup
5:     if a = null then   ▷ Precomputing of adapter classes has determined that the assignment must fail
6:         raise run-time error
7:     else
8:         if a.removeAdapter then                      ▷ Boolean: do we need to unwrap the object?
9:             o = unwrap(o)
10:        if a.instantiateAdapter ≠ null then          ▷ Function creating an adapter for object o owned by subject r
11:            o = a.instantiateAdapter(o, r)
12: t = o                                               ▷ Perform the actual assignment
```

8. Overhead

It is a well known fact that security never comes for free. So we will carefully analyze all sources of overhead induced by our proposal, both in terms of time and memory.

First, we need to maintain a **type identifier for each object**, resulting in a memory overhead of typically 32 bits per object. However, most modern object oriented languages already include run-time type information (RTTI). E.g., RTTI was introduced into C++ already in 1993 (Stroustrup 1994), so it should no longer be considered as additional overhead.

In addition, we require a **mapping from objects to their owners**. Depending on the maximum number of subjects supported, this adds a memory overhead of 16 to 32 bits per object. An exception would be objects that are confined to a single subject (c.f. Sect. 6.3).

Since our type system is not solely based on nominal typing, but also supports structural typing, **dynamic dispatch** of methods may become more complex, since it is no longer possible to use simple virtual function tables as it is done in C++. However, efficient dynamic dispatch is an issue in most modern object oriented languages, so there has been a lot of research in appropriate methods (M. Müller 1995; Gil & Zibin 2007; Milton & Schmidt 1994).

Obviously, also the **introduction of adapters** during the execution of certain type casts creates an overhead both in terms of memory and time. Adapters not only need memory for the adapter objects themselves, but also for the code associated with the adapter's class. In addition, accessing an object via an adapter requires an additional method call due to the indirection. However, as we will show below, the overhead introduced by the proposed type system is actually much smaller than the overhead required by using traditional type systems and security enforcing patterns.

Adapters are created for two reasons:

1. In order to introduce a type conversion, if source and target type have a mismatch in the value type of some argument or result, c.f. Sect. 4.3.2. Unless the programmer is able to modify one of the interfaces involved, even with traditional type systems, a (manually created) adapter would be necessary for handling that mismatch.
2. To enforce access restrictions when passing an object reference to another subject. Again, this situation requires to introduce a membrane even in the traditional OCap model, provided that the programmer cares about implementing security in the sense of POLA.

As traditional type systems allow an unrestricted type cast of references, implementing POLA actually requires to introduce a membrane whenever *any* access restriction should be applied for an object passed to another subject. In contrast, the proposed type-based split capability model does not create an adapter, if the receiving type already enforces that access restriction. An adapter is only needed if the target type permits some feature that is available in the assigned object, but denied by the source type. In addition, adapters will not be cascaded, while a manual application of the membrane pattern inherently will lead to cascaded membranes (c.f. Sect. 1).

Finally, the **static type check** when loading a code unit is a significant overhead in time. However, as it occurs only once, this is not really an issue for the run-time performance.

In summary, we can conclude that the type-based split capability model based on the proposed type system does not induce more run-time overhead than other modern object oriented languages, while providing a significantly higher level of security with much less burden on the programmer than the traditional OCap model.

9. Related Work

9.1. Object-Capability Model

A good introduction into the object-capability model, as well as a motivation for the principle of least authority is given by Miller et al. (Miller et al. 2004). The authors also argue that access control concerns should be considered directly when designing a system. A detailed discussion of the properties of capability systems, as well as a couple of misconceptions about them is given in (Miller et al. 2003). Two of these misconceptions, concerning the revocation of capabilities and confinement guarantees, are further discussed in (Miller & Shapiro 2003) and (Miller 2006). A formal analysis of the security properties of the OCap model, especially of some of the security-enforcing

patterns like the membrane pattern, is presented in (Murray 2008).

An extension of the classical OCap model towards fine-grained access control at the level of individual methods has been implemented in the Oviedo3 system (Díaz-Fondón et al. 1999), which consists of an object oriented abstract machine and an accompanying operating system. However, Oviedo3 implements capabilities as explicit, protected data structures and thus induces overhead both in memory usage and in execution time required for checking the permissions prior to each method invocation.

9.2. Secure Languages

The goal of secure languages is to provide support for the secure composition of possibly untrusted software components. A pioneer in this area is the work of Miller on the E language (Miller 2006), which provides access control on the basis of the OCap model. Miller also presents the use of security-enforcing patterns, like membranes, in order to enable access control at a finer granularity. In addition, the thesis points out the requirements that must be met by a secure language: memory safety, object encapsulation, no ambient authority, no static mutable state, and an API without security leaks.

Joe-E is an adaptation of the Java language with the goal to meet the stated requirements (Mettler et al. 2010). Especially, Joe-E verifies that static state only contains immutable data, but no capabilities. Joe-E also introduces the concept of immutable interfaces to guarantee that methods cannot keep or leak any information between invocations. Security in Joe-E is based on compile-time checking and the use of hardened libraries that do not provide insecure features like reflection and ambient authority, which are present in the standard Java class library.

In a very similar way, a secure version of OCaml is provided by the Emily language (Stiegler 2007). A difference to the previous approaches is the fact that Emily does not use dynamically loaded components, but uses the powerbox pattern to restrict the authority of applications launched at run-time.

Finally, Maffeis et al. define a secure subset of JavaScript supporting the mutual isolation of web applications (Maffeis et al. 2010). The paper also provides a formal proof that a capability safe language also is authority safe and thus can provide isolation guarantees.

A general limitations of these approaches is that they can provide security guarantees only in the case where all software integrated into a system has provably been written in a secure language, because otherwise, the system may be attacked from below the high-level language level (Stiegler 2000). As this is not practical for reasons of licensing and protection of intellectual property, using a secure intermediate language or bytecode is a more feasible solution.

9.3. Type-Based Security

The idea of specifying required capabilities as part of a component's interface was first elaborated by Hagimont et al. (Hagimont et al. 1996). In this work, capabilities were not yet considered to be part of a type and were still implemented traditionally in the operating system kernel.

The starting point for considering types as capabilities is the work done in the area of alias protection (Clarke et al. 1998; P. Müller & Poetzsch-Heffter 1999; Dietl & Müller 2008). The goal here is to ensure that compound objects cannot leak references to internal objects to the outside world. E.g., in an implementation of a sorted tree, code outside of the tree implementation should not be allowed to modify the tree nodes, as this can destroy the tree's invariant of being sorted.

Clarke et al. introduce a *rep* keyword as a part of an *ownership type* to denote that a given reference is not allowed to leave the context of the containing object (Clarke et al. 1998). Müller and Poetzsch-Heffter introduce an explicit hierarchical structure of contexts called *universes*, as well as the notion of read-only references (P. Müller & Poetzsch-Heffter 1999). A type can be parameterized with the universe, to which its objects belong. The typing rules then ensure that an object cannot be referenced from outside its universe, except when using a read-only reference. In (Dietl & Müller 2008), the authors show that ownership types and universes can also be implemented by dependent classes that are parameterized with a reference to a superordinate class.

Bokowski and Vitek (Bokowski & Vitek 1999) explore a similar idea, but with a context in security: references to objects with *confined types* are not allowed to leave their protection domain, e.g., a Java package. In (Fong 2005), Fong shows that confined types can be enforced at link time by the code consumer, thus being useful for secure cooperation of untrusted components.

Boyland et al. made the step towards understanding types as capabilities by generalizing existing reference annotations for sharing, like *read only*, *unique*, or *borrowed* as combinations of different access rights (Boyland et al. 2001). Likewise, also confined types can be understood using the concept of capabilities that are represented by types (Fong 2005). This idea is further elaborated by Fong in (Fong & Zhang 2004), (Fong 2006), (Fong 2008), and (Fong & Orr 2006). In the first paper, an additional type system is introduced that allows to specify the permitted operations on an object as it is passed from one method to the next. For that purpose, the programmer can (and must) specify a labeled transition system that determines how the permissions evolve when a reference is passed from one method to another. In the later publications, a hierarchy of confinement domains is used to determine how capabilities can be passed on. The programmer must explicitly define this hierarchy in the form of trust relationships between domains and must assign a confinement domain to each reference type and each method implementation. Fong then defines a number of constraints that must be met by the type system in order to allow types being used as capabilities granting calling rights for individual methods. Among others, a domain not trusted by a target type is not allowed to perform a downcast operation. The constraints have been implemented via a static type checker and also by restricting the visibility of symbols at load time.

The approach presented in this paper differs from the aforementioned ones in providing a *single* comprehensive type system that combines traditional type information with capabilities. In addition, it permits a limited downcast even in untrusted do-

mains by introducing optional methods and automatic creation of membranes. Thus, the type system encourages a discipline of programming, where component interfaces specify both the strictly required permissions, as well as the optionally desired ones. Another major difference is the fact that in the mentioned approaches, subjects are uniquely associated with types or classes and thus are static. Thus, multi-client software cannot be modeled adequately, since no distinction is made between instances of the same class, even when they are acting on behalf of different clients. In our system, there is the choice to dynamically create a new subject whenever an object is instantiated.

Type systems have also been used to implement information-flow security, i.e., to prevent classified data to be leaked to a domain with a lower security level. To do so, reference types are basically augmented with the security level of the referenced object. A survey on this topic is provided in (Sabelfeld & Myers 2003), recent research is presented, e.g., in (Toro et al. 2018), (Gollamudi & Chong 2016), and (Runge et al. 2023).

9.4. Nominal and Structural Subtyping

Currently, most programming languages are based on either nominal or structural subtyping. With structural subtyping, S is a subtype of T, if S provides at least all the methods (and attributes) of T with a compatible signature (i.e., covariant result types and contravariant argument types). Nominal subtyping in addition requires that S has been explicitly *declared* to be a subtype of T. Both subtyping schemes have their advantages and disadvantages, which are discussed, e.g., in (Gil & Maman 2008) and (Malayeri & Aldrich 2008). An empirical evaluation of the usefulness of structural subtyping has been conducted by Malayeri and Adrich (Malayeri & Aldrich 2009).

A substantial advantage of structural subtyping is that code reuse is much easier. However, structural subtyping completely ignores the semantics of the type's objects. Liskov and Wing have shown that meaningful subtyping must also take into account the behavior (i.e. semantics) of objects and methods (Liskov & Wing 1994). With nominal subtyping, the explicit declaration of subtype relationship implicitly provides the required guarantees. Thus, for integrating both subtyping disciplines, the semantics should not be hidden in the type name but represented explicitly.

Previous approaches for integrating nominal or structural subtyping did not fully conform to this observation. The Java extension Whiteoak (Gil & Maman 2008) just provides structural typing in *addition* to the nominal subtyping used by standard Java. In brand types (Malayeri & Aldrich 2008; Jones et al. 2015), a type may specify a *brand*, which is using nominal subtyping similar to a class, as well as additional methods, which are structurally subtyped. Osterman defines a flexible type system that allows (nominal) subtyping of classes without inheriting the implementation (including the attributes), structural subtyping of interfaces, and the declaration of supertypes (Ostermann 2008). However, the type system has the serious flaw that the subtype relation is no longer transitive.

10. Conclusion and Future Work

The principle of least authority is an established guideline for developing secure software systems from mutually untrusted components. A well researched way to implement POLA is the use of the object-capability model. However, OCap alone is not sufficient, as by itself it only enforces coarse-grained access control at the level of objects. In this paper, we have shown that fine-grained access control at the level of individual methods can efficiently be implemented by extending OCap with the concept of split capabilities. In this model, only one part of the capability, the object reference, is explicitly passed between subjects. While the object reference just permits to access the referenced object at all, the second part of the capability, the static type of the variable storing the reference, determines which methods can be invoked. This part of the capability is not transferred between subjects, rather, strict type checking of the receiver's type with respect to the sender's type ensures that no amplification of authority is possible.

By including a type assertion that restricts a reference variable to only refer to objects owned by the local subject, it is possible to define a type system that does not limit the code's expressiveness within a single component, but only enforces access restrictions at the interface between mutually untrusted components.

Implementing fine-grained access control via the type system is also inherently more efficient than using traditional OCap together with the membrane pattern. First, if the receiver or a reference specifies that it does not need permission for a certain operation, the type system guarantees that the reference does not grant the authority for that operation. So in that case, no membrane is needed. Even if a membrane is required because the receiver requests an optional permission and the sender doesn't want to grant it, the type system guarantees that there will be no cascading of membranes.

The automatic creation of adapters for objects also enhances the flexibility when re-using components, as it allows type casts of references even if the return types or argument types of some method does not fulfill the covariance or contravariance conditions. This also includes unsafe subtyping. Furthermore, the type system gains additional flexibility by adding permissions for an extensible set of features, thus allowing to easily extend the type system with support for additional type capabilities, e.g., confined types.

As a proof of concept, the type system has been fully implemented and tested as part of the COSMA virtual machine (Wismüller & Ludwig 2019). The virtual machine dynamically loads software components represented by a statically typed bytecode and either directly interprets the code or uses an ahead-of-time compiler to translate it into native machine code. The current release of the virtual machine is publicly available in the WWW.[16] The virtual machine is complemented by a compiler for a high-level language that incorporates the extended facilities of the presented type system by supporting additional type modifiers.

Our future work focuses on extensions of the type system as

[16] https://www.bs.informatik.uni-siegen.de/forschung/cosma

well as widening the applications of the virtual machine.

We have argued above that in some cases we do not need to create a membrane, if the authority is already suitably restricted by the target type of an assignment. This also means that there are situations where during an assignment an existing membrane could be removed without any impact on security or functionality, thus reducing the overhead for accessing the object. The problem with this optimization is that it requires a run-time check in assignments that otherwise would not need one. I.e., we need to trade the overhead of the additional run-time check against the overhead of accessing the object via a membrane. Using static analysis to estimate how often the assignment and the subsequent accesses are executed will enable a reasonable decision.

The COSMA virtual machine is currently based on implicit memory reclamation via a garbage collector to avoid the problem of dangling references. Some modern programming languages, especially Rust, avoid the necessity for a garbage collector by using the concept of reference borrowing (Jung et al. 2021). In future work, we will try to integrate reference borrowing with the concepts presented in this paper, since a safe way to ultimately delete an object without affecting other software components also provides a security benefit.

Another future extension will concern the underlying trust model. In the presented approach, all subjects are mutually suspicious. In some situations, it will be more appropriate to consider a hierarchical trust relationship as in the work of Fong (Fong 2008). As in contrast to Fong's approach, subjects are created dynamically in our model, we need to establish new trust relationships whenever a new subject is created. This happens either when a new software component is loaded, or when a new subject is created while instantiating an object using the special `new` operator mentioned in Sect. 4.5. In the first case, the trust relationship may be based on the identity of the component's author, which requires a secure authentication based on a cryptographic signature. In the second case, the code of the instantiated class that is executed by the newly created subject s' is provided by the creating subject s, which means that s' inherently trusts s. On the other hand, s does not (need to) trust s', which enables the programmer to realize sub-components with reduced authority.

The implementation of the COSMA virtual machine is currently extended in several ways. First of all, we are modifying the Java compiler provided by OpenJDK, such that it creates bytecode for COSMA. The extended features of our type system are implemented using annotations, thus avoiding a need to modify the Java syntax. The current prototype already allows to compile parts of the Java Collection framework, which will enable us to use a reasonably large code base for benchmarking. Based on this compiler, we will prepare a quantitative evaluation of the overheads presented in Sect. 8 in the near future.

We are also designing and implementing an object-oriented operating system kernel based on COSMA. In contrast to traditional operating systems, we do not need any hardware support for protection. Memory protection is ensured by the type-based capability model, while the higher privileges of the kernel are realized by the fact that it is the only software component loaded as native machine code rather than COSMA bytecode. Access to the kernel is provided (and controlled) by passing a reference as an argument to the constructor of a component's top-level class.

Since the protection mechanisms presented in this paper do not rely on any hardware support, the model is especially useful for embedded systems based on low-power microcontrollers. For this application scenario, we are currently implementing a host/target environment, where the host computer is responsible for checking and linking software components. It will then compile the linked bytecode into native machine code and flash the resulting image into the micro-controller. In that way, we can realize strict access control for software components while avoiding to burden the microcontroller with the code and run-time overhead for extensive checking of types or permissions.

References

Bokowski, B., & Vitek, J. (1999, November). Confined Types. In *Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)* (p. 82-96). Denver, CO, USA: ACM.

Boyland, J., Noble, J., & Retert, W. (2001, June). Capabilities for Sharing - A Generalisation of Uniqueness and Read-Only. In *15th European Conf. on Object-Oriented Programming (ECOOP '01)* (p. 2-27). Budapest, Hungary: Springer Verlag.

Clarke, D. G., Potter, J. M., & Noble, J. (1998, October). Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)* (p. 48-64). Vancouver, CA: ACM.

Dietl, W., & Müller, P. (2008, January). Ownership Type Systems and Dependent Classes. In *Intl. Workshop on Foundations of Object-Oriented Languages (FOOL'08).* San Francisco, CA, USA: ACM.

Díaz-Fondón, M. A., Álvarez Gutiérrez, D., García-Mendoza-Sánchez, A., Álvarez García, F., Tajes-Martínez, L., & Cueva-Lovelle, J. M. (1999, September). Integrating Capabilities into the Object Model to Protect Distributed Object Systems. In *Proc. Intl. Symp. on Distributed Objects and Applications* (p. 374-383). Edinburgh, GB: IEEE. Retrieved from http://dx.doi.org/10.1109/DOA.1999.794067

Fong, P. W. L. (2005, October). Link-Time Enforcement of Confined Types for JVM Bytecode. In *Proc. 3rd Annual Conf. on Privacy, Security and Trust (PST'05)* (p. 191-202). St. Andrews, Canada: IEEE.

Fong, P. W. L. (2006, September). Discretionary capability confinement. In *Proc. 11th European Symposium On Research In Computer Security (ESORICS'06)* (Vol. 4189, p. 127-144). Hamburg, Germany: Springer.

Fong, P. W. L. (2008, April). Discretionary Capability Confinement. *International Journal of Information Security*, *7*(2), 137-154.

Fong, P. W. L., & Orr, S. A. (2006, December). A Module System for Isolating Untrusted Software Extensions. In *Proc. 22nd Annual Computer Security Applications Conf.*

(ACSAC'06) (p. 203-212). Miami Beach, Florida, USA: IEEE.

Fong, P. W. L., & Zhang, C. (2004, April). *Capabilities as alias control: Secure cooperation in dynamically extensible systems* (Technical Report No. CS-2004-3). Regina, Canada: Dept. of Computer Science, Univ. of Regina.

Gil, J., & Maman, I. (2008, October). Whiteoak: Introducing Structural Typing into Java. In *Proc. 23rd Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'08)* (p. 73–90). Nashville, TN: ACM. Retrieved from https://dl.acm.org/doi/10.1145/1449764.1449771

Gil, J., & Zibin, Y. (2007, November). Efficient Dynamic Dispatching with Type Slicing. *ACM Transactions on Programming Languages and Systems*, *30*(1), Article 5.

Gollamudi, A., & Chong, S. (2016, November). Automatic Enforcement of Expressive Security Policies using Enclaves. In *Proc. 2016 ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)* (p. 494-513). Amsterdam, Netherlands: ACM.

Hagimont, D., Mossière, J., de Pina, X. R., & Saunier, F. (1996, May). Hidden Software Capabilities. In *Proc. 16th Intl. Conf. on Distributed Computing Systems (ICDCS '96)* (p. 282-289). Hong Kong: IEEE.

Jones, T., Homer, M., & Noble, J. (2015, July). Brand Objects for Nominal Typing. In J. T. Boyland (Ed.), *29th European Conference on Object-Oriented Programming (ECOOP'15)* (p. 999–1023). Prague, CZ: Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Jung, R., Jourdan, J.-H., Krebbers, R., & Dreyer, D. (2021, March). Safe Systems Programming in Rust. *Communications of the ACM*, *64*(4), 144–152.

Leroy, X. (2003). Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, *30*, 235–269.

Liskov, B. H., & Wing, J. M. (1994, November). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, *16*(6), 1811-1841.

Maffeis, S., Mitchell, J. C., & Taly, A. (2010, May). Object Capabilities and Isolation of Untrusted Web Applications. In *Proc. of IEEE Symp. Security and Privacy* (p. 125-140). Oakland, CA, USA: IEEE.

Malayeri, D., & Aldrich, J. (2008). Integrating Nominal and Structural Subtyping. In J. Vitek (Ed.), *ECOOP 2008 - Object-Oriented Programming* (p. 260-284). Paphos, Cyprus: Springer.

Malayeri, D., & Aldrich, J. (2009, March). Is Structural Subtyping Useful? An Empirical Study. In G. Castagna (Ed.), *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009* (p. 95–111). York, UK: Springer-Verlag.

Meijer, E., & Drayton, P. (2004, October). Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages.* Vancouver, CA: ACM.

Mettler, A., Wagner, D., & Close, T. (2010, February). Joe-E: A Security-Oriented Subset of Java. In *17th Network and Distributed Systems Symposium* (p. 357-374). San Diego, CA: The Internet Society. Retrieved from https://www.cs.berkeley.edu/~daw/papers/joe-e-ndss10.pdf

Miller, M. S. (2006). *Robust composition: Towards a unified approach to access control and concurrency control* (Ph.D. Thesis, Johns Hopkins University, Baltimore, Maryland). Retrieved from http://erights.org/talks/thesis/markm-thesis.pdf

Miller, M. S., & Shapiro, J. S. (2003, December). Paradigm Regained: Abstraction Mechanisms for Access Control. In *Advances in Computing Science - ASIAN 2003. Progamming Languages and Distributed Computation* (Vol. 2896, p. 224-242). Mumbai, India: Springer.

Miller, M. S., Tulloh, B., & Shapiro, J. S. (2004). The Structure of Authority: Why Security Is not a Separable Concern. In *Proc. 2nd Intl. Conf. on Multiparadigm Programming in Mozart/Oz* (p. 2-20). Charleroi, Belgium: Springer. Retrieved from http://erights.org/talks/no-sep/secnotsep.pdf

Miller, M. S., Yee, K.-P., & Shapiro, J. S. (2003). *Capability Myths Demolished* (Technical Report No. SRL2003-02). Baltimore, MD: Systems Research Laboratory, Johns Hopkins University. Retrieved from http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf

Milton, S., & Schmidt, H. W. (1994, January). *Dynamic Dispatch in Object-Oriented Languages* (Tech. Rep.). Department of Computer Science, The Australian National University, Canberra. Retrieved from https://www.researchgate.net/publication/2459095_Dynamic_Dispatch_in_Object-Oriented_Languages

Müller, M. (1995). *Message Dispatch in Dynamically-Typed Object-Oriented Languages* (Unpublished master's thesis). Univ. of New Mexico, Albuquerque, NM 87131.

Müller, P., & Poetzsch-Heffter, A. (1999). Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter & J. Meyer (Eds.), *Programming Languages and Fundamentals of Programming* (p. 131-140). Germany: Fernuniversität Hagen. (Technical Report 263)

Murray, T. (2008, June). Analysing object-capability security. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security* (p. 177-194). Pittsburgh, PA, USA. Retrieved from https://www.cs.ox.ac.uk/files/2690/AOCS.pdf

Ostermann, K. (2008, January). Nominal and Structural Subtyping in Component-Based Programming. *Journal of Object Technology*, *7*(1), 121-145. Retrieved from http://www.jot.fm/issues/issue_2008_01/article4

Runge, T., Servetto, M., Potanin, A., & Schaefer, I. (2023, March). Immutability and Encapsulation for Sound OO Information Flow Control. *ACM Transactions on Programming Languages and Systems*, *45*(1), Article 3.

Sabelfeld, A., & Myers, A. C. (2003, January). Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, *21*(1), 1-15.

Stiegler, M. (2000). *The E Language in a Walnut.* Retrieved from http://www.skyhunter.com/marcs/ewalnut.html

Stiegler, M. (2007, January). Emily: A High Performance

Language for Enabling Secure Cooperation. In *Fifth Intl. Conf. on Creating, Connecting and Collaborating through Computing C5'07* (p. 163-169). Kyoto, Japan: IEEE.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Toro, M., Garcia, R., & Tanter, E. (2018, December). Type-Driven Gradual Security with References. *ACM Transactions on Programming Languages and Systems*, *40*(4), Article 16.

Wismüller, R., & Ludwig, D. (2019, June). Secure Cooperation of Untrusted Components Using a Strongly Typed Virtual Machine. *International Journal on Advances in Security*, *12*(1&2), 53-68. Retrieved from http://www.iariajournals.org/security/sec_v12_n12_2019_paged.pdf

About the authors

Roland Wismüller is a full Professor for Operating Systems and Distributed Systems at the Department of Electrical Engineering and Computer Science, University of Siegen, Germany. His research interests include techniques for building secure component-based systems, including virtual machines, programming languages and type systems, as well as parallel and distributed systems and the application of machine learning techniques for the analysis of network data. You can contact the author at roland.wismueller@uni-siegen.de or visit https://www.bs.informatik.uni-siegen.de.

Damian Ludwig is a security analyst at the Federal Office for Information Security, Germany. Prior to that, he had been a research assistant to Prof. Wismüller at the University of Siegen, Germany, where he contributed to COSMA. He is interested in advanced separation techniques and next-generation operating systems. You can contact the author at jonas-damian.ludwig@bsi.bund.de.

Felix Breitweiser is a research assistant at the Department of Electrical Engineering and Computer Science, University of Siegen, Germany. His main research interests are type systems and programming language design, as well as the didactics of computer science. You can contact the author at felix.breitweiser@uni-siegen.de or visit https://www.bs.informatik.uni-siegen.de/mitarbeiter/breitweiser.

A. Appendix: Operational Semantics

In the following, we present a small-step semantics for the bytecode instructions of our virtual machine. As we are only concerned about object references and invocations of methods, we omit all other details of the machine, e.g., numerical instructions, jump instructions, and object field access. Furthermore, we restrict the presentation to methods with exactly one argument and one return value. We use the following abstract types:

- Γ for the type environment
- $\mathcal{V}$ for the set of variables
- $\mathcal{T}$ for the set of types
- *Class* for classes
- *Stmt* for the set of statements:

`load0 dst`	load *null* into variable `dst`
`new c dst`	instantiate class `c` and store the object reference in variable `dst`
`mov src dst`	load the value from variable `src` into variable `dst`
`call ref m arg res`	call method `m` on the object referenced by `ref` with the argument value stored in variable `arg` and store the return value in variable `res`
`ret src`	Return the value stored in variable `src`

- $Impl : (arg : \mathcal{V}) \times (ret : \mathcal{T}) \times (var : \mathcal{V}^*) \times (code : Stmt^*)$ for method implementations, where *arg* is the method's argument, *ret* its return type, *var* the method's local variables, and *code* the actual code of the method
- $Obj : (methods : Impl^{\mathbb{A}^*})$ for objects, where *methods* is the object's method table
- $Val = Obj \cup \{null\}$

A *Frame* is defined as a tuple $(obj : Obj) \times (method : Impl) \times (pc : \mathbb{N}) \times (var : Val^{\mathcal{V}}) \times (res : \mathcal{V}) \times (resT : \mathcal{T})$, where *obj* represents the current object, *method* the currently executing method with the program counter *pc*, *var* the variable storage with the current values of the method's argument and local variables, *res* is the caller's variable that receives the method's result, and *resT* the result type expected by the caller.

The semantic rules are formulated as state transitions of one stack configuration to another. Stacks are written as $s :: t$ where s is the topmost frame and t is the rest of the stack. If a frame s is changed, we use the notation $s[field \leftarrow value]$ to indicate an update. $var[l := o]$ denotes an update of location l in the variable storage, such that its new value is o.

The three auxiliary functions used in the semantic rules have the following meaning:

- $instantiate : Obj \times Class \rightarrow Obj$ returns a new instance of a given class owned by a given subject.
- $vars : Impl \times Val \rightarrow Val^{\mathcal{V}}$ is the initial variable mapping for a given method when called with a given argument value:
$vars(m, a) = \lambda v. \begin{cases} a & \text{if } v = m.arg \\ null & \text{if } v \in m.vars \end{cases}$
- $relocate : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ takes care of adapting the assertion for the feature *local* when method arguments or results may be passed between objects with different owners. With $\tau = (r_\tau, c_\tau, f_\tau, p_\tau)$ and $t = (r_t, c_t, f_t, p_t)$, we have
$relocate(\tau, t) = \begin{cases} t & \text{if } f_\tau(local) = avail \\ (r_t, c_t, f'_t, p_t) & \text{otherwise} \end{cases}$, where $f'_t = \lambda x. \begin{cases} optional & \text{if } x = local \\ f_t(x) & \text{otherwise} \end{cases}$

In the following rules, a short line on the right hand side separates the conditions that are solely based on static type information from conditions that can only be evaluated at run-time. The conditions above this marker can already be checked when the code unit containing the instruction is loaded into the virtual machine. For the last rule, this is possible, as the method containing the `ret` instruction is known from its context.

$$\textsc{Load0}\ \frac{\Gamma \vdash dst : (reference, c_\tau, f_\tau, p_\tau) = \tau \quad __}{s :: t \ \xrightarrow{\texttt{load0 dst}}\ s[pc \leftarrow pc + 1, var \leftarrow var[dst := null]] :: t}$$

$$\textsc{New}\ \frac{\begin{array}{c} c : Class \\ \text{type}(c) = \tau_1 \\ \Gamma \vdash dst : \tau_2 \\ \text{isLegal}(\tau_2, \tau_1, \mathbf{F}, \mathbf{T}) \quad __ \\ \text{owner}(s.obj) = r \\ instantiate(r, c) = o \\ \text{cast}_{r,\tau_2,\tau_1}(o) = o' \neq \bot \end{array}}{s :: t \ \xrightarrow{\texttt{new c dst}}\ s[pc \leftarrow pc + 1, var \leftarrow var[dst := o']] :: t}$$

$$\textsc{Mov}\ \frac{\begin{array}{c}\Gamma \vdash src : \tau_1 \\ \Gamma \vdash dst : \tau_2 \\ \text{isLegal}(\tau_2, \tau_1, \mathbf{F}, \mathbf{T}) \\ s.var(src) = o \\ \text{owner}(s.obj) = r \\ \text{cast}_{r,\tau_2,\tau_1}(o) = o' \neq \bot\end{array}}{s :: t \xrightarrow{\texttt{mov src dst}} s[pc \leftarrow pc + 1, var \leftarrow var[dst := o']] :: t}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\Gamma \vdash ref : (r_\tau, c_\tau, f_\tau, p_\tau) = \tau \\ \Gamma \vdash arg : \tau_1 \\ \Gamma \vdash res : \tau_2 \\ f_\tau(m) = avail \\ p_\tau(m) = (A_m, R_m) \\ \text{isLegal}(A_m, relocate(\tau, \tau_1), \mathbf{F}, \mathbf{T}) \\ \text{isLegal}(\tau_2, relocate(\tau, R_m), \mathbf{F}, \mathbf{T}) \\ s.var(ref) = o \\ s.var(arg) = a \\ \text{owner}(o) = r \\ o.methods(m) = m' \\ \text{cast}_{r,A_m,relocate(\tau,\tau_1)}(a) = a' \neq \bot\end{array}}{s :: t \xrightarrow{\texttt{call ref m arg res}} (o, m', 0, vars(m', a'), res, relocate(\tau, R_m)) :: s[pc \leftarrow pc + 1] :: t}$$

$$\textsc{Ret}\ \frac{\begin{array}{c}\Gamma \vdash src : \tau_1 \\ \text{isLegal}(s.method.ret, \tau_1, \mathbf{F}, \mathbf{T}) \\ \Gamma \vdash s.res : \tau_2 \\ s.var(src) = o \\ \text{owner}(s.obj) = r \\ \text{owner}(s'.obj) = r' \\ \text{cast}_{r',\tau_2,s.retT}(\text{cast}_{r,s.method.ret,\tau_1}(o)) = o' \neq \bot\end{array}}{s :: s' :: t \xrightarrow{\texttt{ret src}} s'[var \leftarrow var[s.res := o']] :: t}$$

B. Appendix: Complete Specification of Types and Typing Rules

B.1. Representation of Types

Type: $\mathcal{T} = \mathcal{R} \times \mathcal{C} \times \mathcal{F} \times \mathcal{P}$

Representation: $\mathcal{R} = \{int8, int16, int32, int64, float32, float64, reference\}$

Category: a meet-semilattice $(\mathcal{C}, \leq)$ with

- $\{\bot, C_{int8}, C_{int16}, C_{int32}, C_{int64}, C_{float32}, C_{float64}, C_{class}, C_{adapter}, C_{array}\} \subseteq \mathcal{C}$
- $\forall x \in \mathcal{C} : \bot \leq x$
- $C_{int64} < C_{int32} < C_{int16} < C_{int8}$; $C_{float64} < C_{float32} < C_{int16}$; $C_{float64} < C_{int32}$
- $C_{class} < C_{adapter}$; $C_{class} < C_{array}$

Permissions: $\mathcal{F} = \mathcal{A}^{\mathcal{N} \cup \mathbb{A}^*}$ where

- $\mathcal{N}$ is a finite set of feature identifiers,
- $\mathbb{A}^*$ is the set of all strings (i.e., possible method names),

- $\mathcal{A} = \{denied, restricted, permitted, optional, avail, unavail\}$ is the set of assertions with the following meaning:

	Type permits the use of m	Specifies signature of m	Asserts that m has specified signature	Type asserts that m is available in the object	Type asserts that m is unavailable in the object
denied	**F**	**F**	**F**	**F**	**F**
restricted	**T**	**T**	**F**	**F**	**F**
permitted	**T**	**F**	**F**	**F**	**F**
optional	**T**	**T**	**T**	**F**	**F**
avail	**T**	**T**	**T**	**T**	**F**
unavail	**F**	**F**	**F**	**F**	**T**

Protocol: $\mathcal{P} = (\mathcal{M} \cup \{\bot\})^{\mathbb{A}^*}$ where

- $\mathcal{M} = \mathcal{T}^* \times \wp(\mathbb{N}) \times \mathcal{T}^* \times \wp(\mathbb{N})$ specifies the argument types, the safe arguments, the result types, and the safe results of a method,
- A method name m is mapped to a value different from $\bot$, if and only if the assertion associated with m is *optional*, *avail*, or *restricted*.

B.2. Subtyping Relation

The order of states $s, t \in \mathcal{A}$ is defined by the following table:

PoA

$\downarrow t \;\leq\; s \rightarrow$	*denied*	*restricted*	*permitted*	*optional*	*avail*	*unavail*
denied	**T**	**T**	**T**	**T**	**T**	**T**
restricted	**F**	**T**	**T**	**T**	**T**	**T**
permitted	**F**	**F**	**T**	**T**	**T**	**T**
optional	**F**	**F**	**F**	**T**	**T**	**T**
avail	**F**	**F**	**F**	**F**	**T**	**F**
unavail	**F**	**F**	**F**	**F**	**F**	**T**

With that, the relation $\leq \subseteq \mathcal{T} \times \mathcal{T}$ is the greatest fixed point of the following rules:

$$\textsc{PoT}\;\frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ r_T = r_S \\ c_T \leq c_S \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : f_T(x) \leq f_S(x) \\ \forall x \in \mathbb{A}^* : (s_T \neq \bot \wedge s_S \neq \bot) \;\Rightarrow\; s_T \leq s_S \\ \text{with } s_T = \text{sDst}(f_T(x), p_T(x), f_S(x), p_S(x)) \\ \text{and } \; s_S = \text{sSrc}(f_T(x), p_T(x), f_S(x), p_S(x)) \end{array}}{T \leq S}$$

$$\textsc{PoM}\;\frac{\begin{array}{c} t = (A_t, a_t, R_t, r_t) \in \mathcal{M} \\ s = (A_s, a_s, R_s, r_s) \in \mathcal{M} \\ |A_t| = |A_s| \\ |R_t| = |R_s| \\ \forall i \in [1, |A_s|] : (A_s)_i \leq (A_t)_i \\ \forall i \in [1, |R_t|] : (R_t)_i \leq (R_s)_i \\ a_t \subseteq a_s \\ r_t \subseteq r_s \end{array}}{t \leq s}$$

S is subtype of T, if and only if $T \leq S$.

B.3. Type Checking Rules

We define the relation isLegal $\subseteq \mathcal{T} \times \mathcal{T} \times \mathbb{B} \times \mathbb{B}$ as the greatest fixed point of the following rules:

$$\textsc{LegalT}\ \frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ a, u \in \mathbb{B} \\ a' = a \vee (f_T(\mathit{local}) = \mathit{avail} \wedge f_S(\mathit{local}) \neq \mathit{unavail}) \\ r_T = r_S \vee r_T \neq \mathit{reference} \wedge r_S \neq \mathit{reference} \\ c_T \leq c_S \vee (u \wedge (\mathrm{isPublic}(c_T) \vee \mathrm{isPublic}(c_S) \vee \exists x \in \mathcal{C} : c_T \leq x \wedge c_S \leq x)) \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : \mathrm{isLegal}(f_T(x), f_S(x), a', u) \\ \forall x \in \mathbb{A}^* : (s_T \neq \bot \wedge s_S \neq \bot) \Rightarrow \mathrm{isLegal}(s_T, s_S, a', u) \\ \text{with } s_T = \mathrm{sDst}(f_T(x), p_T(x), f_S(x), p_S(x)) \\ \text{and } s_S = \mathrm{sSrc}(f_T(x), p_T(x), f_S(x), p_S(x)) \end{array}}{\mathrm{isLegal}(T, S, a, u)}$$

$$\textsc{LegalA}\ \frac{\begin{array}{c} t, s \in \mathcal{A} \\ a, u \in \mathbb{B} \\ t = \mathit{avail} \Rightarrow (s = \mathit{avail} \vee u \wedge (s \notin \{\mathit{denied}, \mathit{unavail}\} \vee a)) \end{array}}{\mathrm{isLegal}(t, s, a, u)}$$

$$\textsc{LegalM}\ \frac{\begin{array}{c} t = (A_t, a_t, R_t, r_t) \in \mathcal{M} \\ s = (A_s, a_s, R_s, r_s) \in \mathcal{M} \\ a, u \in \mathbb{B} \\ |A_t| = |A_s| \\ |R_t| = |R_s| \\ \forall i \in [1, |A_s|] : \mathrm{isLegal}((A_s)_i, (A_t)_i, a, u \vee i \notin a_t) \\ \forall i \in [1, |R_t|] : \mathrm{isLegal}((R_t)_i, (R_s)_i, a, u \vee i \notin r_t) \end{array}}{\mathrm{isLegal}(t, s, a, u)}$$

B.4. Restricted Subtype

The relation $t \cap_r s$ for states $s, t \in \mathcal{A}$ is defined by the following table:

rSubA

$\downarrow t \cap_r s \rightarrow$	*denied*	*restricted*	*permitted*	*optional*	*avail*	*unavail*
denied	*denied*	*denied*	*denied*	*denied*	*denied*	*denied*
restricted	*unavail*	*restricted*	*restricted*	*restricted*	*restricted*	*unavail*
permitted	*unavail*	*optional*	*permitted*	*optional*	*optional*	*unavail*
optional	*unavail*	*optional*	*optional*	*optional*	*optional*	*unavail*
avail	—	*avail*	*avail*	*avail*	*avail*	—
unavail	*unavail*	*unavail*	*unavail*	*unavail*	*unavail*	*unavail*

Note that $t \cap_r s$ is undefined for $t = \mathit{avail}$ and $s \in \{\mathit{denied}, \mathit{unavail}\}$. However, this combination does not occur when computing a restricted subtype due to the restricted domain of this operation.

With that, for two types S and $T = (r_T, c_T, f_T, p_T)$, such that $f_T(\mathit{local}) \neq \mathit{avail}$ and isLegal$(T, S, \mathbf{F}, \mathbf{T})$, the restricted subtype $T \cap_r S$ is defined by the largest fixed point of the following rules:

$$\textsc{rSubT1}\ \frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ r_T \neq \mathit{reference} \end{array}}{T \cap_r S = T}$$

$$\text{rSubT2}\ \frac{\begin{array}{c} T = (\textit{reference}, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ R = (\textit{reference}, c_T, f_R, p_R) \in \mathcal{T} \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : f_R(x) = f_T(x) \cap_r f_S(x) \\ \forall x \in \mathbb{A}^* : p_R(x) = \begin{cases} s_T \cap_r s_S & \text{if} \quad s_T \neq \bot \wedge s_S \neq \bot \\ \bot & \text{otherwise} \end{cases} \\ \text{with } s_T = \text{sDst}(f_T(x), p_T(x), f_S(x), p_S(x)) \\ \text{and } s_S = \text{sSrc}(f_T(x), p_T(x), f_S(x), p_S(x)) \end{array}}{T \cap_r S = R}$$

$$\text{rSubM}\ \frac{\begin{array}{c} t = (A_t, a_t, R_t, r_t) \in \mathcal{M} \\ s = (A_s, a_s, R_s, r_s) \in \mathcal{M} \end{array}}{t \cap_r s = (A_s \cap^r A_t, a_t, R_t \cap_r R_s, r_t)}$$

$$\text{rSubTS}\ \frac{\begin{array}{c} T, S, R \in \mathcal{T}^* \\ |T| = |S| = |R| \\ \forall i \in [1, |R|] : R_i = T_i \cap_r S_i \end{array}}{T \cap_r S = R}$$

B.5. Restricted Supertype

The relation $t \cap^r s$ for states $s, t \in \mathcal{A}$ is defined by the following table:

rSupA

$\downarrow t \ \cap^r \ s \rightarrow$	*denied*	*restricted*	*permitted*	*optional*	*avail*	*unavail*
denied	*denied*	*denied*	*denied*	*denied*	*denied*	*unavail*
restricted	*denied*	*restricted*	*restricted*	*optional*	*avail*	*unavail*
permitted	*denied*	*restricted*	*permitted*	*optional*	*avail*	*unavail*
optional	*denied*	*restricted*	*restricted*	*optional*	*avail*	*unavail*
avail	*denied*	*restricted*	*restricted*	*optional*	*avail*	*unavail*
unavail	*denied*	*denied*	*denied*	*denied*	*denied*	*unavail*

With that, for two types S and T, such that isLegal$(T, S, \mathbf{F}, \mathbf{T})$, the restricted supertype $T \cap^r S$ is defined by the largest fixed point of the following rules:

$$\text{rSupT1}\ \frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (r_S, c_S, f_S, p_S) \in \mathcal{T} \\ r_S \neq \textit{reference} \end{array}}{T \cap^r S = S}$$

$$\text{rSupT2}\ \frac{\begin{array}{c} T = (r_T, c_T, f_T, p_T) \in \mathcal{T} \\ S = (\textit{reference}, c_S, f_S, p_S) \in \mathcal{T} \\ R = (\textit{reference}, c_T, f_R, p_R) \in \mathcal{T} \\ \forall x \in \mathcal{N} \cup \mathbb{A}^* : f_R(x) = f_T(x) \cap^r f_S(x) \\ \forall x \in \mathbb{A}^* : p_R(x) = \begin{cases} s_T \cap^r s_S & \text{if} \quad s_T \neq \bot \wedge s_S \neq \bot \\ \bot & \text{otherwise} \end{cases} \\ \text{with } s_T = \text{sDst}(f_T(x), p_T(x), f_S(x), p_S(x)) \\ \text{and } s_S = \text{sSrc}(f_T(x), p_T(x), f_S(x), p_S(x)) \end{array}}{T \cap^r S = R}$$

$$\text{rSupM}\ \frac{\begin{array}{c} t = (A_t, a_t, R_t, r_t) \in \mathcal{M} \\ s = (A_s, a_s, R_s, r_s) \in \mathcal{M} \end{array}}{t \cap^r s = (A_s \cap_r A_t, a_s, R_t \cap^r R_s, r_s)}$$

$$\text{RSUPTS}\ \frac{T, S, R \in \mathcal{T}^* \quad |T| = |S| = |R| \quad \forall i \in [1, |R|] : R_i = T_i \cap^r S_i}{T \cap^r S = R}$$

C. Appendix: Theorems and Proofs

Theorem 6 states the fundamental security property of our approach, i.e., a subject that receives a capability cannot amplify that capability, unless it actually owns the object the capability refers to. The proof of this theorem relies on some auxiliary theorems, which are also presented and proven. In particular, Theorem 3 essentially states that our type system is sound. As only reference types are relevant for the security properties, the theorems and proofs do not explicitly deal with value types.

In addition, Theorem 7 shows that defining an adapter's category as a subcategory of the wrapped object's category does not induce any problems (see last paragraph of Sect. 4.5).

Theorem 1 (Restricted subtype and supertype)**.** $\forall T, S \in \mathcal{T} : \; T \leq T \cap_r S, \; T \cap^r S \leq S.$

Proof. As the recursive structure of $\leq$ and $\cap_r$ (or $\cap^r$) is identical (the contravariance in method arguments is accounted for by swapping $\cap_r$ and $\cap^r$), we only have to examine the rules POA, RSUBA and RSUPA: Comparing RSUBA with POA shows that for all s, t we have $t \leq t \cap_r s$, while comparing RSUPA with POA shows that for all s, t we have $t \cap^r s \leq s$. □

Theorem 2 (Coercion results in a subtype)**.** *Let v be an object, r be a subject, and $T \in \mathcal{T}$. Then* $\text{coerce}_{\text{r,T}}(v) = \perp \vee T \leq \text{type}_r(\text{coerce}_{\text{r,T}}(v))$.

Proof. Let $T = (r_T, c_T, f_T, p_T)$, $V = (r_V, c_V, f_V, p_V) = \text{type}(v)$, $V_r = (r_{V_r}, c_{V_r}, f_{V_r}, p_{V_r}) = \text{type}_r(v)$, $w = \text{coerce}_{\text{r,T}}(v)$, $W = (r_W, c_W, f_W, p_W) = \text{type}(w)$, and $W_r = (r_{W_r}, c_{W_r}, f_{W_r}, p_{W_r}) = \text{type}_r(w)$.

Consider Alg. 3. If $\neg\text{isLegal}(T, V_r, \mathbf{F}, \mathbf{F})$, then $w = \perp$. If $T \leq V_r$, then $w = v$, thus $T \leq \text{type}_r(w)$. Otherwise, we have $w = \text{wrap}_T(v)$ with

$$r_W = r_T$$

$$c_V \leq c_W \wedge C_{adapter} \leq c_W$$

$$f_W(x) = \begin{cases} avail & \text{if } x = local \vee (f_V(x) = avail \wedge f_T(x) \notin \{denied, unavail\}) \\ unavail & \text{otherwise} \end{cases}$$

$$p_W(m) = \begin{cases} p_T(m) & \text{if } f_V(m) = avail \\ \perp & \text{otherwise} \end{cases}$$

$$\text{owner}(w) = \text{owner}(v)$$

Since V and V_r only differ in the state of feature *local*, $\text{isLegal}(T, V_r, \mathbf{F}, \mathbf{F})$ implies $c_T \leq c_V$ (LEGALT) and thus $c_T \leq c_W$.

Let $x \in \mathcal{N} \setminus \{local\}$. If $f_V(x) = avail \wedge f_T(x) \notin \{denied, unavail\}$, then $f_W(x) = avail$. Thus, $f_T(x) \leq f_W(x)$ (POA). Otherwise, we have $f_T(x) \neq avail$ (if $f_V(x) \neq avail$, this follows from $\text{isLegal}(T, V_r, \mathbf{F}, \mathbf{F})$ and LEGALA). Thus, $f_T(x) \leq unavail = f_W(x)$ (POA).

Let $m \in \mathcal{M}$ with $p_T(m) \neq \perp \wedge p_W(m) \neq \perp$. Then $p_W(m) = p_T(m)$, thus $p_T(m) \leq p_W(m)$.

Now $\text{isLegal}(T, V_r, \mathbf{F}, \mathbf{F})$ implies $f_T(local) \neq avail \vee f_{V_r}(local) = avail$ (LEGALA). Since V_r is the type of a real object, we have $f_{V_r}(local) \in \{avail, unavail\}$. Together with the fact that for target types we only allow *avail* and *optional* as possible states for the feature *local*, this means that we always have $f_T(local) \leq f_{W_r}(local)$ (POA).

Since W and W_r only differ in the state of feature *local*, in sum we have $T \leq W_r$. □

Theorem 3 (Assignment results in a subtype)**.** *Let t be a variable of a subject r with type T that references some object o. Then $T \leq \text{type}(o)$ and $T \leq \text{type}_r(o)$. Especially, if $f_T(\text{local}) = \text{avail}$, then o is owned by r.*

Proof. Let $T = (r_T, c_T, f_T, p_T)$. When t has been assigned to, the run-time system proceeded according to Algorithm 1. I.e., it has assigned $o = \text{coerce}_{r,R}(\text{unwrap}(v))$, where R is either T, $T \cap_r S$, or $T \cap_r S \cap_r V_r$ (lines 8, 12, 15, Alg. 2). In any case, we have $T \leq R$ (Theorem 1). By Theorem 2 we have $\text{coerce}_{r,R}(\text{unwrap}(v)) = \perp \vee R \leq \text{type}_r(\text{coerce}_{r,R}(\text{unwrap}(v)))$. So either the assignment failed (which is precluded by the assumption that t actually references an object), or we have $T \leq R \leq \text{type}_r(\text{coerce}_{r,R}(\text{unwrap}(v))) = \text{type}_r(o)$.

The only difference between $O = (r_O, c_O, f_O, p_O) = \text{type}(o)$ and $O_r = (r_{O_r}, c_{O_r}, f_{O_r}, p_{O_r}) = \text{type}_r(o)$ is that $f_O(local) = avail$, while $f_{O_r}(local) \in \{avail, unavail\}$. If $f_{O_r}(local) = avail$, then $O = O_r$, so $T \leq O_r$ implies $T \leq O$. Otherwise, if

$f_T(local) \neq avail$, $T \leq O_r$ still implies $T \leq O$. If $f_T(local) = avail$ and $f_{O_r}(local) = unavail$, then o is not owned by r, which means that $\text{unwrap}(v)$ is not owned by r (line 32, Algorithm 3). This makes the type check in line 13 of Algorithm 3 fail ($f_T(local) = avail$ also means that $f_R(local) = avail$, thus $\neg\text{isLegal}(R, \text{type}_r(\text{unwrap}(v)), \mathbf{F}, \mathbf{F})$ due to LEGALA), which contradicts the assumption that t actually references an object. □

In the following proofs on security properties, we just consider the direct authority provided by a single variable (reference), i.e., we only consider operations that are invoked using the variable under consideration. Since the theorems hold for all variables of a subject, this means that they also hold for the direct authority of that subject as defined in Sect. 2.

Theorem 4 (Type restricts authority)**.** *A variable s of reference type $S = (r_S, c_S, f_S, p_S)$ owned by a subject r grants the authority to use an operation x, if and only if $f_S(x) = \text{avail}$ or r can successfully assign s (directly or indirectly) to a variable t of type $T = (r_T, c_T, f_T, p_T)$, such that $f_T(x) = \text{avail}$.*

Proof. Let v be the object referenced by s and $V_r = (r_{V_r}, c_{V_r}, f_{V_r}, p_{V_r}) = \text{type}_r(v)$.

"**If**": If $f_S(x) = avail$, the run-time system will allow to use x via s. If s can successfully be assigned to t, the run-time system will allow to use x via t, since $f_T(x) = avail$. Note that in both cases $f_{V_r}(x) = avail$ must hold, i.e., v actually provides x.

"**Only if**": Assume that $f_S(x) \neq avail$ and r cannot successfully assign s to some variable $t : T$ with $f_T(x) = avail$. Since the run-time system will allow the use of x only via a reference where the state of x is *avail*, r cannot use x. □

Theorem 5 (Membrane restricts authority)**.** *Let s be a variable with reference type S owned by a subject r that references an object v with $\text{owner}(v) \neq r$. Then s grants the authority to use an operation x, if and only if $x \neq \text{local}$, S permits the use of x , and v provides x (i.e., $f_S(x) \notin \{\text{denied}, \text{unavail}\}$ and $f_V(x) = \text{avail}$, where $S = (r_S, c_S, f_S, p_S)$ and $V = (r_V, c_V, f_V, p_V) = \text{type}(v)$).*

Proof. We show that $f_S(x) = avail$ or s can successfully be assigned to a variable t of type $T = (r_T, c_T, f_T, p_T)$, such that $f_T(x) = avail$, if and only if $x \neq local$, $f_S(x) \notin \{denied, unavail\}$, and $f_V(x) = avail$. The proposition then follows from Theorem 4.

In the following, let $V_r = (r_{V_r}, c_{V_r}, f_{V_r}, p_{V_r}) = \text{type}_r(v)$. V_r and V are identical with the exception that $f_V(local) = avail$, while $f_{V_r}(local) = unavail$.

"**If**": Assume that $x \neq local$, $f_S(x) \notin \{denied, unavail\}$, and $f_V(x) = avail$. If $f_S(x) = avail$, the conclusion immediately follows. Otherwise, let t be a variable of type T, such that T is identical to S, except that $f_T(x) = avail$. We show that an assignment from s to t will always succeed.

Inspection of rules POA and LEGALA proves that we have $T \not\leq S$ and $\text{isLegal}(T, S, \mathbf{F}, \mathbf{T})$. This means that the static type check succeeds and the run-time system will execute Algorithm 1 to perform the assignment to t.

As $\text{owner}(v) \neq r$, $f_S(local) \neq avail$ by Theorem 3. Since T is identical to S except for $f_T(x)$ and $x \neq local$, this means that also $f_T(local) \neq avail$.

- If $v = \text{unwrap}(v)$: Let $R = T \cap_r S$. Since S and T are identical except for $f_T(x) = avail$, we have $f_R(x) = avail$ (RSUBA) and thus $T = R$. Furthermore, since $S \leq V_r$ (Theorem 3), and $f_{V_r}(x) = f_V(x) = avail$, we have $T \leq V_r$, i.e. $R \leq V_r$, which implies $\text{isLegal}(R, V_r, \mathbf{F}, \mathbf{F})$. Thus, the assignment succeeds (line 15, Alg. 3).
- If $v \neq \text{unwrap}(v)$: Let $u = \text{unwrap}(v)$ and $U_r = (r_{U_r}, c_{U_r}, f_{U_r}, p_{U_r}) = \text{type}_r(u)$. Since s references v and v is an adapter, the run-time system at some point assigned $\text{coerce}_{r,R}(u)$ with $R = S \cap_r X$ for some type X to s (c.f. Alg. 2). Thus, we must have $\text{isLegal}(R, U_r, \mathbf{F}, \mathbf{F})$ (line 13, Alg. 3), which implies $\text{isLegal}(S, U_r, \mathbf{F}, \mathbf{F})$, since $S \leq R$ (Theorem 1). Since S and T are identical except for $f_T(x) = avail$, and $f_{V_r}(x) = f_V(x) = avail$, which implies that $f_{U_r}(x) = avail$ (line 28, Alg 3), we have $\text{isLegal}(T, U_r, \mathbf{F}, \mathbf{F})$. Thus, also $\text{isLegal}(T \cap_r Y, U_r, \mathbf{F}, \mathbf{F})$ for any Y (RSUBA, LEGALA). Thus, the assignment succeeds.

"**Only if**": Assume that $x = local$ or $f_S(x) \in \{denied, unavail\}$ or $f_V(x) \neq avail$. Let t be a variable owned by r with type T, where T is an arbitrary type with $f_T(x) = avail$.

Since $\text{owner}(v) \neq r$, $f_{V_r}(local) = unavail$ and $f_S(local) \neq avail$ by Theorem 3. If $f_T(local) = avail$, this implies $T \not\leq S$. If $\neg\text{isLegal}(T, S, \mathbf{F}, \mathbf{T})$, the assignment fails due to the static type check. Otherwise, Alg 2 will assign $\text{coerce}_{r,T}(\text{unwrap}(v))$. Since $\text{owner}(\text{unwrap}(v)) = \text{owner}(v) \neq r$, this assignment fails, because $\neg\text{isLegal}(T, \text{type}_r(\text{unwrap}(v)), \mathbf{F}, \mathbf{F})$ (LEGALA, line 13, Alg. 2). So for the following, we can assume $f_T(local) \neq avail$ and thus, $x \neq local$.

- If $f_S(x) \in \{denied, unavail\}$: Then $T \not\leq S$ (POA) and $\neg\text{isLegal}(T, S, \mathbf{F}, \mathbf{T})$ (LEGALT, LEGALA), so the assignment fails due to the static type check.
- If $f_S(x) \notin \{denied, unavail\}$ and $f_V(x) \neq avail$: Then also $f_{V_r}(x) \neq avail$, i.e., $f_{V_r}(x) = unavail$ as V_r is the type of a real object. Since s references v, this also implies that $f_S(x) \neq avail$ (Theorem 3), i.e., $T \not\leq S$. If $\neg\text{isLegal}(T, S, \mathbf{F}, \mathbf{T})$, the assignment fails due to the static type check. Otherwise, Alg. 1 will be executed at run-time. Let $R = T \cap_r S$. Since $f_T(x) = avail$, also $f_R(x) = avail$. This implies both $\neg\text{isLegal}(R, V_r, \mathbf{F}, \mathbf{T})$ and $\neg\text{isLegal}(R, V_r, \mathbf{F}, \mathbf{F})$ (LEGALT, LEGALA), which means that the assignment fails either in line 13 of Alg. 3 (if $v = \text{unwrap}(v)$) or in line 13 of Alg. 2 (if $v \neq \text{unwrap}(v)$).

□

Theorem 6 (No amplification of authority)**.** *Let t be a variable of a subject r with type T and s be a variable (of an arbitrary subject) with type S, referencing an object v with $\text{owner}(v) \neq r$. When s is assigned to t, t does not grant more authority than s.*

Proof. Assume that the assignment is successful, so that after the assignment t contains a reference to some object v'. Let $S = (r_S, c_S, f_S, p_S)$, $T = (r_T, c_T, f_T, p_T)$, $V = (r_V, c_V, f_V, p_V) = \text{type}(v)$, and $V' = (r_{V'}, c_{V'}, f_{V'}, p_{V'}) = \text{type}(v')$. Further assume that t grants the authority to use some operation x, but s does not. We will show that this leads to a contradiction.

If $x = \textit{local}$, then according to Theorem 4, $f_T(\textit{local}) = \textit{avail}$ or r can successfully assign t to a variable u of type $U = (r_U, c_U, f_U, p_U)$, such that $f_U(\textit{local}) = \textit{avail}$. However, since $\text{owner}(v) \neq r$, the run-time assignment defined in Algorithms 1 to 3 will always assign an object v' with $\text{owner}(v') = \text{owner}(v) \neq r$. This contradicts Theorem 3.

If $x \neq \textit{local}$, according to Theorem 5, we have $f_T(x) \notin \{\textit{denied}, \textit{unavail}\} \wedge f_{V'}(x) = \textit{avail} \wedge (f_S(x) \in \{\textit{denied}, \textit{unavail}\} \vee f_V(x) \neq \textit{avail})$.

- If $f_S(x) = \textit{denied}$, then $T \not\leq S$ (POA). Thus, the run-time system has assigned a value v' according to Algorithms 1 to 3, where $v' = \text{coerce}_{r,R}(\text{unwrap}(v))$ and $R = T \cap_r S$ or $R = T \cap_r S \cap_r \text{type}_r(v)$. As $f_T(x) \notin \{\textit{denied}, \textit{unavail}\}$, we have $f_R(x) = \textit{unavail}$ in both cases (RSUBA). Let $u = \text{unwrap}(v)$ and $U_r = (r_{U_r}, c_{U_r}, f_{U_r}, p_{U_r}) = \textit{type}_r(u)$. If $R \leq U_r$, then $v' = u$ (line 15, Alg. 3). Since then U_r and V' are identical except the status of feature *avail*, $R \leq U_r$ implies $f_{V'}(x) = \textit{unavail}$ (POA). Otherwise, $v' = \text{wrap}_R(u)$. $f_R(x) = \textit{unavail}$ again implies $f_{V'}(x) = \textit{unavail}$ (line 28, Alg. 3).
- If $f_S(x) = \textit{unavail}$, then $f_V(x) = \textit{unavail}$ by Theorem 3 and POA. The same is true, if $f_V(x) \neq \textit{avail}$, since V is the type of a real object.
  The run-time system will assign either v or a value v' according to Algorithms 1 to 3. In the first case, we get an immediate contradiction, since $v' = v$. In the second case, $v' = \text{coerce}_{r,R}(\text{unwrap}(v))$ and either $v = \text{unwrap}(v) \wedge R = T \cap_r S$ or $v \neq \text{unwrap}(v) \wedge R = T \cap_r S \cap_r \text{type}_r(v)$. If $v = \text{unwrap}(v)$, v' will be either v or $\text{wrap}_R(v)$ (lines 16, 23 of Alg. 3). In both cases, we get $f_{V'}(x) = \textit{unavail}$ (c.f. line 28 of Alg. 3). Otherwise, we have $R = T \cap_r S \cap_r \text{type}_r(v)$, which implies $f_R(x) = \textit{unavail}$ (RSUBA). With the same argumentation as in the item above, we again get $f_{V'}(x) = \textit{unavail}$.

Thus, we get a contradiction in all cases. □

Theorem 7 (No adapter for local class type)**.** *Let t be a variable of type $T = (r_T, c_T, f_T, p_T)$ with $C_{\text{class}} \leq c_T$ and $f_T(\text{local}) = \text{avail}$. A legal assignment to t will never result in the generation of an adapter.*

Proof. We assume that t is owned by a subject r and is being successfully assigned from a variable that references some object o (or from the result of executing the `new` operator creating some object o). Since $f_T(\textit{local}) = \textit{avail}$, the value $\text{coerce}_{r,T}(\text{unwrap}(o))$ will be assigned to t at run-time (line 8, Alg. 2).

Let $v = \text{unwrap}(o)$, $V = (r_V, c_V, f_V, p_V) = \text{type}(v)$, and $V_r = (r_{V_r}, c_{V_r}, f_{V_r}, p_{V_r}) = \text{type}_r(v)$. Since v is not an adapter, but a real object (which ultimately has been created by the `new` operator), we must have $C_{class} \leq c_V$ and $V = \text{type}(c_V)$. As we assume that the assignment is successful, we must have $\text{isLegal}(T, V_r, \mathbf{F}, \mathbf{F})$ (line 13, Alg. 3). Since $c_{V_r} = c_V$, this implies $c_T \leq c_V$ (LEGALT). Further, $f_T(\textit{local}) = \textit{avail}$ implies that $f_{V_r}(\textit{local}) = \textit{avail}$ (LEGALA), i.e., $V_r = V$.

Now, the requirements stated in Sect. 4.5 imply $T \leq \text{type}(c_T)$, and (since $c_T \leq c_V$) also $\text{type}(c_T) \leq \text{type}(c_V)$. I.e., $T \leq \text{type}(c_V) = V = V_r$. Thus, v will be directly assigned to t (line 16, Alg. 3). □