Type-Safe Prototype-Based Component Evolution

Matthias Zenger

École Polytechnique Fédérale de Lausanne
INR Ecublens, 1015 Lausanne, Switzerland
matthias.zenger@epfl.ch

Abstract. Component-based programming is currently carried out using mainstream object-oriented languages. These languages have to be used in a highly disciplined way to guarantee flexible component composition and extensibility. This paper investigates abstractions for component-oriented programming on the programming language level. We propose a simple prototype-based model for first-class components on top of a class-based object-oriented language. The model is formalized as an extension of *Featherweight Java*. Our calculus includes a minimal set of primitives to dynamically build, extend, and compose software components, while supporting features like explicit context dependencies, late composition, unanticipated component extensibility, and strong encapsulation. We present a type system for our calculus that ensures type-safe component definition, composition, and evolution.

1 Introduction

Component-based software development techniques gain increasing attention in industry and research. Component technology is driven by the promise of software reuse and plug-and-play programming. This promise poses high demands on the implementation platform.

Currently, component-based programming is carried out using mainstream object-oriented languages. Object-oriented languages seem to promote component-based programming well: They support encapsulation of state and behavior, inheritance and overriding enable extensibility, and subtype polymorphism and late binding allow flexible reuse of objects and classes. Unfortunately, object-oriented techniques alone are not powerful enough to provide flexible and type-safe component composition and evolution mechanisms.

Therefore, industrial component models like *CORBA* [27], *COM* [46], or *JavaBeans* [33] rely on additional concepts, namely component frameworks and meta-programming. They provide a class framework for modeling components and component interactions together with an informal set of implementation rules. Components are composed using meta-programming technology like reflection. This ad-hoc approach yields a dynamic and flexible composition mechanism, but often does not guarantee any static type security. Furthermore, the

B. Magnusson (Ed.): ECOOP 2002, LNCS 2374, pp. 470–497, 2002.

degree of extensibility depends on the framework or the meta-programming tools. In general, it has to be planned ahead, for instance by using suitable design patterns typically derived from the *AbstractFactory* pattern [22]. This lack of unanticipated extensibility hinders a smooth software evolution process substantially.

Another issue was recently pointed out by Aldrich and Chambers [2]. They observe that implementation languages are only loosely coupled to architectural descriptions. As a consequence, specifications of software architectures [44,50] formally expressed in architecture description languages [40] are often quite different from the actual object-oriented implementations. This makes it difficult to trace architectural properties in the implementation, which would allow to verify that an implementation is consistent with the corresponding architecture [2].

This is why recently various proposals have been put forward to integrate concepts known from architecture description languages into object-oriented programming languages [49,52,2]. These so-called component-oriented programming languages offer linguistic facilities for programming software components, for defining component interactions, and for composing software from components. Their promise is to do that in a type-safe way, ruling out illegal interaction patterns.

In this paper we study linguistic abstractions for component-oriented programming in the context of object-oriented programming languages. We describe the notion of prototype-based components. Our prototype-based component model is designed to support plug-and-play programming. It features lightweight components that can be dynamically manufactured and composed in a type-safe way. We emphasize the necessity for a smooth component adaption and evolution process. In particular, we allow to derive refined components from existing components without sacrificing consistency and type-safety. We present a formalization of our prototype-based component model as an extension of *Featherweight Java* [32,45]. Our typed calculus includes a minimal set of primitives to build, extend, and compose software components, while supporting principles like explicit context dependencies, late composition, unanticipated component extensibility, and strong encapsulation of component services.

We proceed by motivating the design principles of our component model. Section 2 emphasizes the importance of software adaptability, extensibility, and software evolution in general. Section 3 introduces prototype-based components by example, presenting the various component refinement primitives. A formalization of the model is presented in Section 4 in form of a core component calculus. We present a type system and prove that this system is sound with respect to the given operational semantics. A summary of the main features together with a discussion of related work is given in Section 5.2. Section 6 concludes.

2 Motivation

In this section we motivate specific design principles of our prototype-based component model. The main features of the model include:

1. Components are first-class core language abstractions,
2. composition operators enable coarse-grained component composition,
3. components can be manufactured and composed dynamically (late composition),
4. components are extensible, promoting component reuse, adaptability, and evolution.

Furthermore, our model adopts principles common among component-oriented languages, like explicit context dependencies (external linking), cyclic component linking, and strong encapsulation. Component manufacturing, composition, and refinement are type-safe. Our type system supports subtype polymorphism for components and component instances.

2.1 Language Integration

The introduction motivated already the need for specific component abstractions, directly integrated into the core of programming languages. With an explicit language construct for components, a programmer can implement architecture descriptions directly without the need for finding a suitable representation in a particular programming language.

2.2 Coarse-Grained Composition

Existing proposals for component abstractions on the programming language level like *ComponentJ* [49], *ACOEL* [52], and *ArchJava* [2] directly adopt common concepts and principles of architecture description languages. They provide constructs for manufacturing components with required and provided services. A service associates a port name with a type. Components are composed by linking ports with explicit plug instructions. The type system ensures that all ports are linked and that links are established only between compatible ports or service providers.

This approach does not scale, since for linking a component with n services, we have to issue n explicit plug instructions specifying the wiring of the component. For large-scale components with a lot of services involved, linking the component is a tedious and error-prone task. Furthermore, the sequence of plug instructions rather obscures the architecture of the system instead of making it explicit. Therefore, McDirmid, Flatt, and Hsieh argue that component systems should offer the possibility to connect many required and provided services at once [39].

We address this requirement by simplifying the interface of components and by providing means to infer the wiring of components to be linked together. Components can be composed with simple operators and without explicitly plugging ports. We also support incremental linking; i.e. we allow that components get only partially linked. For instance, components can be sent around in a distributed system and only the services available at a specific location get linked until in the end we have a fully linked component that can be instantiated.

2.3 Dynamic Manufacturing and Composition

Software component technology distinguishes two main tasks: component manufacturing and component composition. It is often explained that both tasks are separate steps being performed one after the other. But in practice, both tasks coincide when new components are built by composing other components. This form of component manufacturing is called *hierarchical component composition*.

Often it is assumed that component manufacturing is done statically before component composition takes place. Component composition itself cannot always be performed statically in cases where components are only known at runtime. Therefore component-based systems have to support some form of *dynamic linking*.

This observation implies that we also have to be able to manufacture software components dynamically, since component linking and manufacturing coincide in hierarchical component compositions. Thus, it makes no sense to assume that both manufacturing and composition are atomic tasks that are performed consecutively. In highly dynamic systems, component manufacturing and composition is rather an interleaved process in which components are created and linked incrementally.

2.4 Reuse, Adaption, Evolution and Extension

When using components from external vendors, it is quite unlikely that the interfaces of these third-party components fit to the required interfaces off-the-shelf. It is often necessary to adapt components before they can be used in a particular system [30,43]. As Section 2.3 already pointed out, components might only be supplied at runtime, therefore it is even more necessary that components can be adapted dynamically on-the-fly.

In a prototype-based component model, new components can only be created by refining an already existing component. As a consequence, we can derive two different components from a single base component. By doing this, we factor out potential reusable pieces, avoiding duplicated programming effort. In addition, this technique supports software evolution. Software evolution includes the maintenance and extension of component features and interfaces. Supporting software evolution is important, since components and component systems are architectural building blocks and as such, subject to continuous changes.

Extensibility of components [53] is not only required for a smooth component evolution. It is even more desired for enabling the development of families of software applications and product-lines in general. Traditionally, components are static black-boxes emphasizing encapsulation over extensibility. Features can be added to components only by creating a new component that forwards all existing services to the old version in addition to the new services. This is a cumbersome and error-prone procedure that duplicates programming efforts and complicates maintenance.

3 Introduction to Prototype-Based Components

In this section we describe prototype-based components in the context of a small, statically typed, object-oriented *Java*-like base language. Our component model relies on a nominal type system [45] of the base language. In nominal type systems, two types with the same structure but a different name are considered to be different, as opposed to structural type systems that match the structure and not the name. Prototype-based components do not rely on other base language features like inheritance or even classes, even though we present them here in a class-based context. Therefore it should be straightforward to add prototype-based components to other object-oriented languages with nominal object types.

3.1 Components and Component Instances

In our model, a component is a unit of computation that can be accessed through a well-defined interface. A component is a first-class citizen. Its interface specifies the services it provides to allow other components to interact with it. The interface also specifies the services a component requires from other components to be able to provide the own services.

Our component model is prototype-based; i.e. the only way to create a new component is by refining an already existing prototypical component. For bootstrapping purposes, we have a single predefined component that does not provide or require any services. This empty component is denoted by the keyword `component`.

We strictly distinguish components from component instances. A component describes a template for possibly multiple component instances. It is the component instances that provide the actual services. Services are described by object types, e.g. types defined by classes or interfaces. Objects serve as service providers. They usually get created at component instantiation time. Therefore, components can be seen as organizational units with well-defined interfaces that structure object interdependencies. Components have neither a unique identity, nor an observable state. They come to life through objects at the time they get instantiated.

In the remainder of this section we introduce prototype-based components by example. We derive some simple software components that could be used, for instance, in online retail stores to manage stock and clients.

3.2 Service Provision

We start by manufacturing a software component that provides access to a customer database. We want every customer to have a unique client number. A service that maps customer names to client numbers could be described by the following interface definition:

```
interface CustomerIDs {
    int lookupId(String name);
}
```

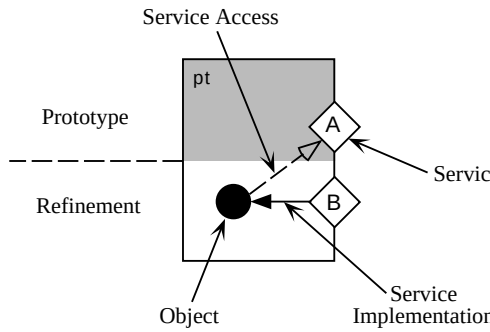

Fig. 1. Schematic notation for prototype-based components

The *CustomerIDs* interface consists of a single method *lookupId*. Given a customer's name, this method tries to find the corresponding client number. If there is no client number yet for this customer, a new number will be issued and returned by *lookupId*. Imagine we have the following implementation of the *CustomerIDs* interface:

```
class MyCustomerIDs implements CustomerIDs {
   MyCustomerIDs() { ... }
   int lookupId(String name) { ... }
   ...
}
```

With this implementation we are able to manufacture a software component that provides a *CustomerIDs* service. Since we can only create new components by refining existing ones, we have to take the empty component as a prototype and refine it such that it provides a *CustomerIDs* service. In our calculus, this is done with the provides primitive:

```
c0 =  component
      provides CustomerIDs as This with new MyCustomerIDs();
```

The clause d provides $\overline{C}$ as x with e returns a new component that refines component d by providing some possibly new services $\overline{C}$. These services are implemented by an object specified with expression e. Note that we are extending a component here. Therefore, expression e only gets evaluated at component instantiation time. x is a variable that gets bound to the own component instance. In object-oriented languages this self reference corresponds to variable *this* or *self* referring to the own object. Only expression e is in the scope of x. Typically, expression e refers to other services of the own component instance via x.

We use a graphical notation to illustrate the structure of components. Figure 1 gives an overview. Here, a component is represented by a box. The gray part corresponds to the prototype of the component, the white part specifies the refinement. In our graphical notation, services are symbolized by diamonds. Objects are simply black dots. An arrow from a service to an object expresses

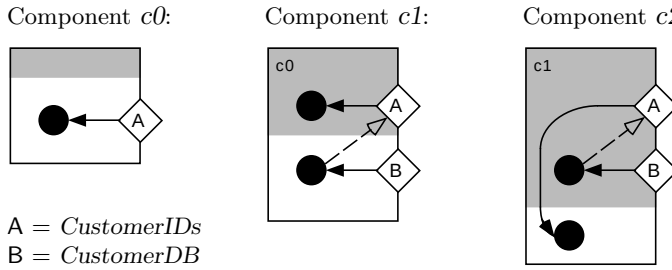

Fig. 2. Component evolution

that this object implements the service. We also have outlined arrows that depict service dependencies. These dependencies are not explicit in our calculus. If an object refers to other services, for instance via the self reference, then every such dependency is specified with an outlined arrow. Figure 2 shows the structure of our previously defined component *c0*.

3.3 Component Instantiation

We already pointed out that components have to be instantiated before services can be accessed. In our component calculus, a component gets instantiated with the new primitive.

i0 = ***new*** *c0;*

The services of a component instance like *i0* get accessed via the service selection operator ::. The expression $e :: C$ selects a service C from component instance e. C is a type name that identifies a service and at the same time describes the service's interface. Other component models refer to services via named ports. In these models it is possible to have two distinct ports with the same interface type but different port names. In programming languages with nominal type systems like *Java* [26] or $C^{\#}$ [28], types do not only define structural object properties like available methods or fields. They also stand for semantic specifications [14], and as such, they are well-suited for specifying roles. In those type systems it is possible to have two distinct types with the same interface description but different type names. Therefore, it is no restriction to describe a service only by its type without having a port name in addition. This simplifies the definition of components and the service access in general significantly. It also acts as a standardization of port names. One only has to know a service's type in order to access it from a component instance. It is not necessary to lookup the port name in the component specification. We will see later in Section 3.7 that this standardization of component port names has another advantage: it promotes automatic composition mechanisms. Of course, in the few cases where two ports could share a type, we have to create new type names and in the worst case use wrappers to adapt existing objects.

Here is an example demonstrating the usage of the component *i0*. In this example we call the *lookupId* method of the *CustomerIDs* service provided by component instance *i0*.

```
i0 :: CustomerIDs.lookupId("John_Smith");
```

3.4 Component Refinement

Now imagine the requirements for our customer administration component *c0* are changing and we also need the capability to store customer names and addresses. We can describe this new database service with the following interface:

```
interface CustomerDB {
    void enter(String name, String address);
    String lookupName(int id);
    String lookupAddr(int id);
}
```

Method *enter* stores a new address in the database. Whenever a new customer is entered, a new client number will automatically be assigned to this new customer. The methods *lookupName* and *lookupAddr* find a name or address for a given client number. The following class implements *CustomerDB*. It depends on a component instance that provides a *CustomerIDs* service. This component instance is passed as a parameter to the constructor. Following [49], we use the notation $[S_1, ..., S_n]$ to specify the type for component instances supporting at least the services S_1 to S_n.

```
class MyCustomerDB implements CustomerDB {
    [CustomerIDs] This;
    MyCustomerDB([CustomerIDs] This) {
        this.This = This;
    }
     ... This::CustomerIDs.lookupId(name) ...
}
```

We already mentioned that prototype-based components offer a smooth component evolution mechanism. For creating an extended version of a component, we just have to interpret the old component as a prototype. In our example, the new refined component evolves out of the old one simply by an application of the provides primitive. The following code refines component *c0* by additionally providing the service *CustomerDB*.

```
c1 = c0 provides CustomerDB as This with new MyCustomerDB(This);
```

The provides primitive can also be used to refine a component by defining a new service implementation for an already provided service. In this case we *override* the old implementation. Here is the definition of component *c2* that refines *c1* by using, for instance, a more efficient client numbering service.

```
c2 = c1 provides CustomerIDs as This with new EfficientCustomerIDs();
```

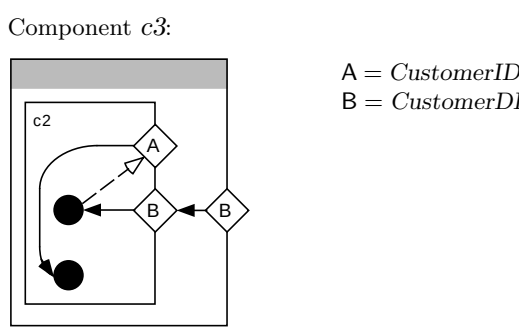

Fig. 3. Service forwarding

The service implementation for *CustomerDB*, specified already in the prototype of *c2*, now automatically refers to this new numbering service implementation. A graphical illustration of components *c1* and *c2* can be found in Figure 2.

3.5 Service Forwarding

Until now, we are only able to develop new components by adding new services or by overriding existing service implementations of a prototypical component. Every service we add gets exported automatically; i.e. it can be accessed from outside the component. This *white-box approach* is necessary to keep the component extensible, because it allows us to override service implementations and to add new service implementations that refer to already existing services. But often we do not want to publish internally used services. Being able to hide internal interfaces is an important feature of component-oriented programming. Our component calculus supports this form of encapsulation with the component projection operator forwards. The clause d forwards $\overline{C}$ as x to e extends component prototype d with the services $\overline{C}$. The new component forwards accesses of these services to the component instance e. Expression e can refer to other services of the own component instance via the self reference x. This primitive is primarily used for hierarchical component compositions. In the following example it is specifically used to hide services and service interconnections. Thus, it turns a "white-box" into a "black-box" by wrapping the original component.

```
c3 =  component
      forwards CustomerDB as This to new c2;
```

In this example we create a new component *c3* that only provides a single service *CustomerDB* by forwarding calls to a component instance of *c2*. Thus, we hide the *CustomerIDs* service of component *c2*. We say, an instance of *c2* is nested inside every instance of component *c3*. We call the hidden *CustomerIDs* service an internal service of component *c3*. An illustration of *c3* instances can be found in Figure 3. Here, the instance of component *c2* that is contained in *c3* is depicted by a nested box. Service implementations are now arrows pointing from external services to internal services of nested component instances.

3.6 Service Abstraction

The previous sections showed how to evolve a component by incrementally adding new services either by a new service implementation or by forwarding services to a nested component instance. In both cases we introduced new services and implementations for these services at the same time. This approach does not allow us to write components that depend on services provided by other components. Furthermore, we are not even able to define two services where service implementations depend mutually on each other, because we introduce services linearly, one after the other.

We tackle both problems with a service abstraction facility. Before going into detail, we proceed by manufacturing a new component for handling orders of a shop. The service for placing orders is described by the following interface:

```
interface OrderDB {
    void order(int id, String article, int num);
}
```

With method *order*, new orders can be placed. Orders consists of a client number, an article descriptor and the number of items to deliver. If possible, this method tries to execute the order immediately. Therefore it needs access to a stock database service specified by the following interface:

```
interface StockDB {
    void enter(String article, int num);
    void remove(String article, int num);
    int available(String article);
}
```

Method *order* checks if the articles are available. If this is the case, it removes them from the stock database and sends the articles to the customer's address. Therefore, service implementations of *OrderDB* like *MyOrderDB* also need access to the *CustomerDB* service. Thus, the constructor of the following class expects a component instance providing *StockDB* and *CustomerDB* services.

```
class MyOrderDB implements OrderDB {
    [StockDB, CustomerDB] This;
    MyOrderDB([StockDB, CustomerDB] This) {
        this.This = This;
    }
    ...
}
```

Since we do not want our order system component to already commit to a specific service implementation for the *StockDB* and the *CustomerDB* service, we have to factor out these two services. In order to make use of the component later, we then either have to provide the missing service implementations from outside at composition time, or we further refine the component and provide service implementations from inside the component.

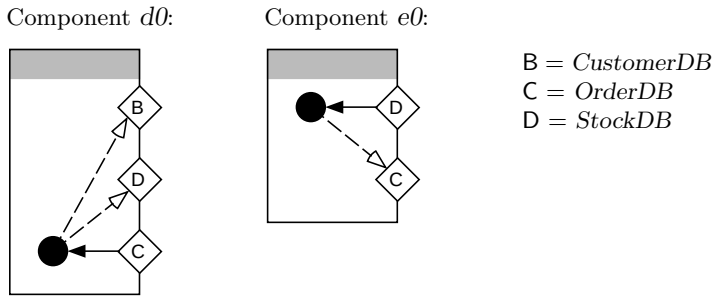

Fig. 4. Service abstraction

In our component calculus, services are factored out with the service abstraction primitive requires. The requires primitive allows to define services that are required for implementing other services without the need for specifying a concrete service implementation. We make use of this abstraction facility in the following implementation of component *d0* which requires two services *CustomerDB* and *StockDB* and provides a *OrderDB* service. Figure 4 contains an illustration of component *d0*.

```
d0 =  component
      requires CustomerDB
      requires StockDB
      provides OrderDB as This with new MyOrderDB(This);
```

The expression d requires C takes a prototypical component d and returns a refined version with a service C that has to be provided before the component can be instantiated. Other service implementations can refer to this service, even though there is no implementation known yet. This is why in the example above, self reference *This* has type [*CustomerDB, StockDB, OrderDB*] and thus is a legal parameter for the constructor of *MyOrderDB*. Components have a type of the form $(R_1, \ldots, R_n \Rightarrow P_1, \ldots, P_m)$ where R_1 to R_n are services required by the component, and P_1 to P_m are the provided services. Thus, the type of component *d0* is (*CustomerDB, StockDB* $\Rightarrow$ *OrderDB*). As already mentioned before, component *d0* cannot be instantiated, since not all service provisions are resolved yet. We first have to derive a new component that specifies implementations for all required services before we can actually create component instances.

We continue in our example by defining a new component *e0* that provides an implementation for a *StockDB* service.

```
e0 =  component
      requires OrderDB
      provides StockDB as This with new MyStockDB(This);
```

The implementation of service *StockDB* makes use of an externally supplied *OrderDB* service. This is, because in cases where new stock arrives and orders are still pending, it would trigger the process of sending out the articles. The type of component *e0* is (*OrderDB* $\Rightarrow$ *StockDB*).

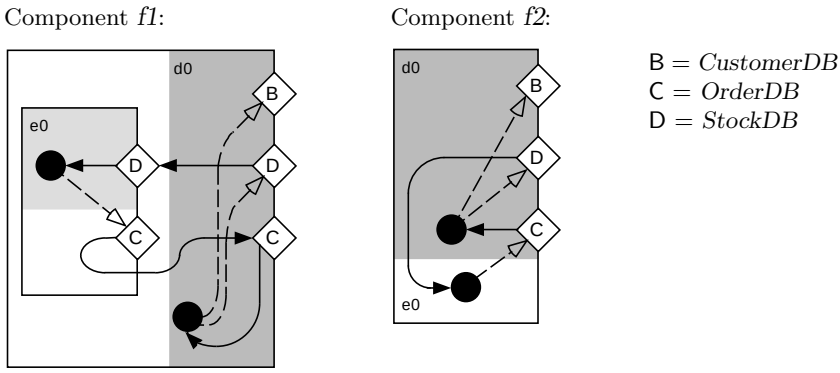

Fig. 5. Component composition

3.7 Component Composition

In the previous section we defined two components *d0* and *e0* that mutually refer to each other; i.e. the service provided by one component is required by the other one. We would now like to link these two components together yielding a component which only requires a *CustomerDB* service and provides both a *OrderDB* and a *StockDB* service. The simplest way to achieve this is to refine component *d0* with an implementation for service *StockDB*. This service is provided by a refined version of *e0* that refers back to the *OrderDB* service provided by the enclosing *d0* prototype.

f0 = *d0* ***provides*** *StockDB* ***as*** *This* ***with***
(***new*** (*e0* ***provides*** *OrderDB* ***as*** *Me* ***with*** *This::OrderDB*))*::StockDB*

This technique does not work for components where more than two services depend mutual recursively on each other. For such cases we have to use the forwards primitive in order to link the components together. A graphical illustration of the resulting component *f1* can be found in Figure 5.

f1 = *d0* ***forwards*** *StockDB* ***as*** *This* ***to***
new (*e0* ***provides*** *OrderDB* ***as*** *Me* ***with*** *This::OrderDB*)

The previously discussed composition schemes use service forwarding where the nested component instance refers back to services provided by the enclosing component being defined. Our component calculus offers an alternative to this rather complicated composition pattern. With the mixin operator it is possible to create a new component by mixing in the services provided by another component. The expression e mixin d refines the prototypical component e with component d; i.e. e gets refined by including all the services provided by component d. Services that are already present in e are automatically overridden by the corresponding services of d. This operation identifies the self references of both components e and d by binding it to the resulting merged component. The resulting component requires services that are either required by e or d and that

are not provided by any of the two components. It provides all the services that are provided by either *e* or *d*. Thus, the following expression yields a component *f2* of type (*CustomerDB* ⇒ *OrderDB, StockDB*).

f2 = d0 ***mixin*** *e0*

When using such a mixin-based composition scheme, one has to be aware that for the expression above, all services *e0* provides get mixed in, no matter what static type *e0* has in this context. Thus, we might accidentally override services provided by *d0*. Sometimes this is desired, for instance, when we want to express that *e0* has got the more recent or more trustworthy service implementations than *d0*. For cases where we want to define explicitly what services to override, we have to use a forwarding-based composition scheme instead. For instance, we could write *d0 forwards StockDB as This to new (e0 forwards OrderDB as Me to This)*.

All three components defined in this section are equivalent in the sense that they provide and require the same services and that services are implemented by the same objects. Though, Figure 5 reveals that the internal structure of components manufactured using the forwarding and the mixin technique are quite different. Therefore, they may behave differently when it comes to refinements of both components. In the given example, this is not the case. But one might imagine a bigger nested component instance where overriding a service of the enclosing component does not have any effect on the formerly forwarded service of the nested component, while it would have an effect on the mixin-based approach.

We finish this section by manufacturing a component that permits access to customer related services only; i.e. *CustomerDB* and *OrderDB*. We do this by first linking together the customer management component *c2* and the stock management component *f2*. The linked component *c2* mixin *f2* provides all the various services introduced in this section. Since we want to restrict the access to customer related services, we have to project the resulting component to a new component *g0* offering only the desired services.

g0 = ***component***
forwards *CustomerDB, OrderDB* ***as*** *This* ***to new*** *(c2* ***mixin*** *f2)*

g0 has type (⇒ *CustomerDB, OrderDB*); thus, it is possible to instantiate this component. The structure of an instance of our final component *g0* is presented in Figure 6. Leaving out some intermediate steps, we could have composed *g0* out of three essential components: *c2* which administers clients, *d0* which handles orders, and *e0* which manages the stock.

g0 = ***component***
forwards *CustomerDB, OrderDB* ***as*** *This* ***to new*** *(c2* ***mixin*** *d0* ***mixin*** *e0)*

This short expression demonstrates how concise component manufacturing and linking is in our model. Furthermore it outlines how components are typically deployed. The sub-expression *c2 mixin d0 mixin e0*[1] first links components *c2*,

[1] Please note that the mixin operator is associative.

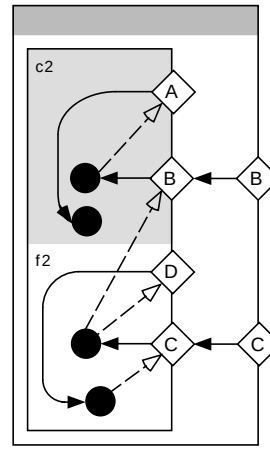

Fig. 6. The final component *g0*

d0, and *e0*, yielding a single extensible component. This component exposes internal interfaces. We might want that, for instance to use this component as a basis for further refinements. But before instantiating (or even selling) it, we should hide the internals by wrapping the component in a black-box only offering specific functionality with restricted support for extensibility. In the example above, this is done using the component projection primitive forwards.

4 Component Calculus

In this section we present a formalization of our prototype-based component model for a functional subset of *Java*. Our calculus is built on top of *Featherweight Java* (*FJ*) [32]. We omit type casts from the original calculus since type casts are irrelevant for our application and complicate the formal treatment unnecessarily.

4.1 Syntax

The syntax of the calculus is presented in Figure 7. Like in *FJ*, a program consists of a collection of class declarations plus an expression to be evaluated. The syntax of classes, constructors, and methods is identical to *FJ*. We only extend the set of expressions with the primitives introduced in Section 3. In particular, we add an empty component, a service abstraction and implementation primitive, a component projection primitive as well as a component mixin operator. In addition, we have a construct for instantiating components and a service selection operator for accessing services from a component instance. In our calculus, a service is characterized by a class name.

Opposed to the presentation in Section 3.2, the calculus only supports a provides primitive that introduces a single service. This is no restriction since we

Program	$P = \overline{L}\ ;\ e$	*program*
Class	$L = \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\ \overline{T}\ \overline{f}\ ;\ K\ ;\ \overline{M}\ \}$	*class declaration*
Constructor	$K = C(\overline{T}\ \overline{f})\ \{\ \mathsf{super}(\overline{f});\ \mathsf{this}.\overline{f} = \overline{f};\ \}$	*constructor declaration*
Method	$M = T\ m(\overline{T}\ \overline{x})\ \{\ \mathsf{return}\ e;\ \}$	*method declaration*
Expressions	$e = x$	*variable*
	$\mid e.f$	*field selection*
	$\mid e.m(\overline{e})$	*method invocation*
	$\mid \mathsf{new}\ C(\overline{e})$	*object creation*
	$\mid \mathsf{component}$	*empty component*
	$\mid e\ \mathsf{requires}\ C$	*service abstraction*
	$\mid e\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ e$	*service implementation*
	$\mid e\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ e$	*component projection*
	$\mid e\ \mathsf{mixin}\ e$	*component mixin*
	$\mid \mathsf{new}\ e$	*component instantiation*
	$\mid e :: C$	*service selection*
Types	$T = C$	*object type*
	$\mid \overline{C}\ \Rightarrow\ \overline{C}$	*component type*
	$\mid [\overline{C}]$	*component instance type*

Fig. 7. Syntax

can easily model the former semantics by using the more general forwards construct in combination with a nested component that implements several services with a single object.

FJ's types only consist of class names. For simplicity, *Java*'s interface types are not modeled. For working with components and component instances we also need syntactical forms for expressing component and component instance types. Please note that compared to the explanations in Section 3.6, we use a slightly simplified syntax for component types without enclosing parenthesis. As in *FJ*, we write $\overline{T}$ as a shortcut for $T_1, \ldots, T_n$. We use similar shorthands for sequences like $\overline{C}, \overline{f}, \overline{e}$, etc. as well as for pairs of sequences like $\overline{T}\ \overline{f}$. Such a pair of sequences is a shorthand for $T_1\ f_1, \ldots, T_n\ f_n$.

We assume that sequences of field declarations, parameter names, and method declarations do not contain duplicate names. Furthermore, the service implementation and the component projection operators always introduce fresh names for their self reference variable. For the presentation of the operational semantics in the next section we assume to apply alpha-renaming whenever necessary to avoid name capture.

4.2 Semantics

The semantics of our calculus are formalized in Figure 8 as a small-step operational semantics. The reduction relation has the form $e \longrightarrow e'$ which expresses that expression e evaluates to expression e' in a single step.

$$\text{(R-Fld)}\ \frac{\text{fields}(C) = \overline{T}\,\overline{f}}{\mathsf{new}\ C(\overline{e}).f_i \longrightarrow e_i} \qquad \text{(R-Serv)}\ \frac{\text{service}(\mathsf{new}\ e, e, C) = e'}{\mathsf{new}\ e :: C \longrightarrow e'}$$

$$\text{(R-Inv)}\ \frac{\text{mbody}(m, C) = (\overline{x}, e_0)}{\mathsf{new}\ C(\overline{e}).m(\overline{d}) \longrightarrow [\overline{d}/\overline{x}, \mathsf{new}\ C(\overline{e})/\mathsf{this}]\, e_0}$$

$$\text{(R-Req)}\ \ e\ \mathsf{requires}\ C \longrightarrow e \qquad \text{(R-MixC)}\ \ e\ \mathsf{mixin\ component} \longrightarrow e$$

$$\text{(R-MixP)}\ \ e\ \mathsf{mixin}\ (e_0\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d) \longrightarrow (e\ \mathsf{mixin}\ e_0)\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d$$

$$\text{(R-MixF)}\ \ e\ \mathsf{mixin}\ (e_0\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d) \longrightarrow (e\ \mathsf{mixin}\ e_0)\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d$$

$$\text{(RC-Fld)}\ \frac{e \longrightarrow e'}{e.f \longrightarrow e'.f} \qquad \text{(RC-InvR)}\ \frac{e \longrightarrow e'}{e.m(\overline{d}) \longrightarrow e'.m(\overline{d})}$$

$$\text{(RC-InvA)}\ \frac{e_i \longrightarrow e_i'}{d.m(\ldots, e_i, \ldots) \longrightarrow d.m(\ldots, e_i', \ldots)}$$

$$\text{(RC-NewA)}\ \frac{e_i \longrightarrow e_i'}{\mathsf{new}\ C(\ldots, e_i, \ldots) \longrightarrow \mathsf{new}\ C(\ldots, e_i', \ldots)}$$

$$\text{(RC-Inst)}\ \frac{e \longrightarrow e'}{\mathsf{new}\ e \longrightarrow \mathsf{new}\ e'} \qquad \text{(RC-Serv)}\ \frac{e \longrightarrow e'}{e :: C \longrightarrow e' :: C}$$

$$\text{(RC-Prv)}\ \frac{e \longrightarrow e'}{e\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d \longrightarrow e'\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d}$$

$$\text{(RC-Fwd)}\ \frac{e \longrightarrow e'}{e\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d \longrightarrow e'\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d}$$

$$\text{(RC-MixL)}\ \frac{e \longrightarrow e'}{e\ \mathsf{mixin}\ d \longrightarrow e'\ \mathsf{mixin}\ d} \qquad \text{(RC-MixR)}\ \frac{d \longrightarrow d'}{e\ \mathsf{mixin}\ d \longrightarrow e\ \mathsf{mixin}\ d'}$$

Fig. 8. Operational semantics

We adopt all reduction rules from *FJ* and define various new rules for our new syntactical constructs. Service abstractions simply reduce to the prototype component, so they do not have any computational effect. The semantics of mixins are described by three reduction rules, depending on the form of the right operand. Mixing in the empty component results in the same component. For service implementations and component projections we mix the prototype of the right operand into the left operand and apply the component refinement on that new component. Thus, we incrementally combine the two operands into a single component where service definitions of the right operand override definitions of the left operand.

The reduction rule for service selections relies on an auxiliary function $\text{service}(e', e, C)$ which searches the component definition e of component instance

Field lookup

$$\text{fields}(\mathsf{Object}) = \emptyset$$

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{\,\overline{T}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad \text{fields}(D) = \overline{U}\,\overline{g}}{\text{fields}(C) = \overline{U}\,\overline{g}, \overline{T}\,\overline{f}}$$

Method body lookup

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{\,\overline{U}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad T'\ m(\overline{T}\ \overline{x})\,\{\,\mathsf{return}\ e;\,\}\ \in \overline{M}}{\text{mbody}(m, C) = (\overline{x}, e)}$$

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{\,\overline{T}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad m \text{ not defined in } \overline{M}}{\text{mbody}(m, C) = \text{mbody}(m, D)}$$

Service lookup

$$\text{service}(e,\ e_0\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d, C) = [e/x]\,d$$

$$\text{service}(e,\ e_0\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d, C_i) = [e/x]\,d :: C_i$$

$$\frac{D \neq C}{\text{service}(e,\ e_0\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d, D) = \text{service}(e, e_0, D)}$$

$$\frac{D \notin \overline{C}}{\text{service}(e,\ e_0\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d, D) = \text{service}(e, e_0, D)}$$

Fig. 9. Auxiliary definitions for evaluation

e' for a service C. Note that the service lookup performed by service(e', e, C) is only defined on service implementation and component projection terms. Thus, even for cases where e provides a service C, evaluation of service(e', e, C) may not be well-defined if e has not been evaluated far enough. In such a case, we first have to apply rules (RC-Inst) and (RC-Serv) to further evaluate the component before making use of the actual service selection rule (R-Serv). An overview of all auxiliary definitions used by the operational semantics of Figure 8 are given in Figure 9.

4.3 Type System

We have three different forms of types: object types, component types and component instance types. An object type is simply denoted by a class name C. An object type is well-formed if the class name appears in the domain of the class table CT. The class table is a mapping from class names to class declarations. As in the presentation of *FJ*, we assume that we have a fixed class table to simplify the notation. Otherwise we would have to parameterize all typing rules with CT. It is assumed that CT satisfies some sanity conditions: $\mathit{Object} \notin \text{dom}(CT)$, all

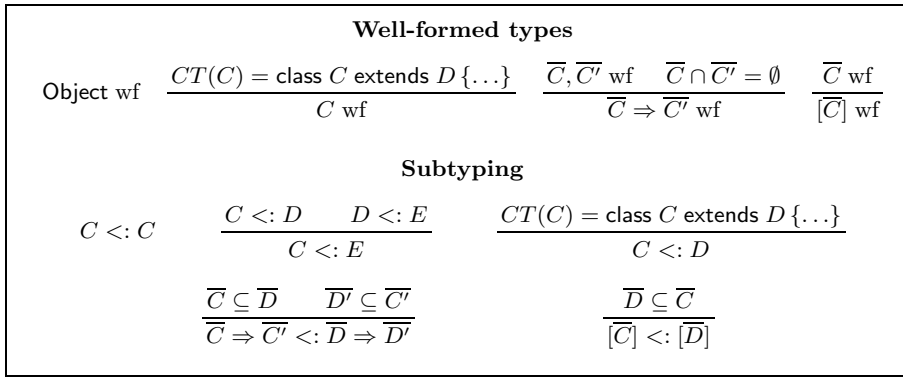

Fig. 10. Well-formed types and subtyping

types appearing explicitly in CT are well-formed, and there are no cycles in the subtype relation induced by CT.

Component types have the form $\overline{C} \Rightarrow \overline{C'}$ where $\overline{C}$ specifies the services required by the component and $\overline{C'}$ specifies the provided services. Services are described by object types. A component type is only well-formed if the sets of the provided and required services are disjoint. $[\overline{C}]$ types a component instance that provides the services $\overline{C}$. Figure 10 summarizes the well-formedness criteria on types.

Method types cannot be written explicitly. In the type system, we use the notation $\overline{T} \to T'$ for a method with the argument types $\overline{T}$ and the result type T'. Note that depending on the context, $\overline{T}$ denotes either a sequence of types $(T_1, \ldots, T_n)$ or a set of types $\{T_1, \ldots, T_n\}$. We use shorthands of the form $\overline{C} \cup D$ for expressing $\overline{C} \cup \{D\}$.

Figure 10 also defines a subtype relation $T <: T'$ between two types T and T'. Subtyping of object types is identical to *FJ*. A component instance type is a subtype of another component instance type if the services provided by the supertype constitute a subset of the subtype's provided services. A component type $\tau_1 = \overline{C} \Rightarrow \overline{C'}$ is a subtype of component type $\tau_2 = \overline{D} \Rightarrow \overline{D'}$, if τ_1 requires less and provides more services than τ_2; i.e. $\overline{C} \subseteq \overline{D}$ and $\overline{D'} \subseteq \overline{C'}$. This corresponds to the typical co/contravariant subtyping rule for function types [17] adopted already by related approaches to component subtyping [20,49,25]. In Section 4.6 we discuss an alternative subtyping rule.

The type system is presented in Figure 11. We have three different typing judgment forms. The one for classes has the form "L ok" meaning that class declaration L is type correct. The judgment for method declarations has the form "M ok in C", expressing that the method declaration M typechecks as a declaration of class C. Both rules are directly taken from *FJ*. The judgment for expressions $\Gamma \vdash e : T$ relates a type T to an expression e. Most typing rules for expressions are straightforward. (T-Prv) and (T-Fwd) are among the interesting rules. Here, the service provision expression is typed under an extended

Expression typing

$$\text{(T-Var)}\ \ \Gamma \vdash x : \Gamma(x) \qquad \text{(T-Fld)}\ \frac{\Gamma \vdash e : C \qquad \text{fields}(C) = \overline{T}\ \overline{f}}{\Gamma \vdash e.f_i : T_i}$$

$$\text{(T-Inv)}\ \frac{\Gamma \vdash d : C \qquad \text{mtype}(m, C) = \overline{T} \to T' \qquad \Gamma \vdash \overline{e} : \overline{U} \qquad \overline{U} <: \overline{T}}{\Gamma \vdash d.m(\overline{e}) : T'}$$

$$\text{(T-New)}\ \frac{\text{fields}(C) = \overline{T}\ \overline{f} \qquad \Gamma \vdash \overline{e} : \overline{U} \qquad \overline{U} <: \overline{T}}{\Gamma \vdash \mathsf{new}\ C(\overline{e}) : C}$$

$$\text{(T-Inst)}\ \frac{\Gamma \vdash e : \emptyset \Rightarrow \overline{C}}{\Gamma \vdash \mathsf{new}\ e : [\overline{C}]} \qquad \text{(T-Serv)}\ \frac{\Gamma \vdash e : [\overline{C}]}{\Gamma \vdash e :: C_i : C_i}$$

$$\text{(T-Com)}\ \ \Gamma \vdash \mathsf{component} : \emptyset \Rightarrow \emptyset$$

$$\text{(T-Mix)}\ \frac{\Gamma \vdash e : \overline{C} \Rightarrow \overline{C'} \qquad \Gamma \vdash d : \overline{D} \Rightarrow \overline{D'}}{\Gamma \vdash e\ \mathsf{mixin}\ d : (\overline{C} \cup \overline{D}) \backslash (\overline{C'} \cup \overline{D'}) \Rightarrow \overline{C'} \cup \overline{D'}}$$

$$\text{(T-Req)}\ \frac{C\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'}}{\Gamma \vdash e\ \mathsf{requires}\ C : \overline{D} \cup C \Rightarrow \overline{D'} \backslash C}$$

$$\text{(T-Prv)}\ \frac{C\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D} \cup \overline{D'} \cup C] \vdash d : B \qquad B <: C}{\Gamma \vdash e\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d : \overline{D} \backslash C \Rightarrow \overline{D'} \cup C}$$

$$\text{(T-Fwd)}\ \frac{\overline{C}\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D} \cup \overline{D'} \cup \overline{C}] \vdash d : [\overline{B}] \qquad \overline{C} \subseteq \overline{B}}{\Gamma \vdash e\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d : \overline{D} \backslash \overline{C} \Rightarrow \overline{D'} \cup \overline{C}}$$

Method and class typing

$$\text{(T-Meth)}\ \frac{\begin{array}{c}\overline{T}\ \text{wf} \qquad T'\ \text{wf} \qquad \overline{x} : \overline{T}, \mathsf{this} : C \vdash e : U \qquad U <: T' \\ CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{\ldots\} \qquad \text{override}(m, D, \overline{T} \to T')\end{array}}{T' m(\overline{T}\ \overline{x})\,\{\,\mathsf{return}\ e;\,\}\ \text{ok in } C}$$

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

Fig. 11. Type system

environment, including the self reference to the own component instance. We assume that the type of the self reference variable is a component instance type offering both, the services that are required and provided by the component being refined. The auxiliary definitions used for typing field and method selections as well as object creations are directly adopted from *FJ* and summarized in Figure 12.

Method type lookup

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{\,\overline{U}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad T'm(\overline{T}\ \overline{x})\,\{\,\mathsf{return}\ e;\,\} \in \overline{M}}{\mathrm{mtype}(m, C) = \overline{T} \to T'}$$

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{\,\overline{T}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad m \text{ not defined in } \overline{M}}{\mathrm{mtype}(m, C) = \mathrm{mtype}(m, D)}$$

Valid method overriding

$$\frac{\mathrm{mtype}(m, C) = \overline{U} \to U' \ \text{ implies } \ \overline{U} = \overline{T} \text{ and } U' = T'}{\mathrm{override}(m, C, \overline{T} \to T')}$$

Fig. 12. Auxiliary definitions for typing

4.4 Type Soundness

For proving type soundness, we weaken the typing rules for provides and forwards terms. We use the following two rules (T-Prv') and (T-Fwd') instead:

$$\text{(T-Prv')}\ \frac{C \text{ wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D''}] \vdash d : B \qquad B <: C}{\Gamma \vdash e\ \mathsf{provides}\ C\ \mathsf{as}\ x\ \mathsf{with}\ d : (\overline{D} \cup \overline{D''}) \backslash (\overline{D'} \cup C) \Rightarrow \overline{D'} \cup C}$$

$$\text{(T-Fwd')}\ \frac{\overline{C} \text{ wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D''}] \vdash d : [\overline{B}] \qquad \overline{C} \subseteq \overline{B}}{\Gamma \vdash e\ \mathsf{forwards}\ \overline{C}\ \mathsf{as}\ x\ \mathsf{to}\ d : (\overline{D} \cup \overline{D''}) \backslash (\overline{D'} \cup \overline{C}) \Rightarrow \overline{D'} \cup \overline{C}}$$

In this weaker system we allow that provides and forwards primitives introduce service abstractions in a non-deterministic way. We show type soundness for this weaker type system. As a consequence, the type system with the stronger typing rules, presented in Figure 11, is sound as well. This system has the advantage that typings are deterministic. Furthermore, its design follows the principle that service abstractions have to be declared explicitly. Weakening the type system was necessary for subject reduction to hold. We present the type soundness results for our weaker type system in the style of Wright and Felleisen [56]. The proof can be found in [57].

Theorem 4.1 (Subject reduction) If all types in Γ are well-formed, $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T'$ for some $T' <: T$.

For a well-typed term which can be reduced to a second term, Theorem 4.1 states that this second term is also well-typed. Furthermore, the type of the second term is a subtype of the type of the first term.

In addition to that we can show that the evaluation of every well-typed term does not get stuck. To formalize this, we introduce a term subset denoting values.

Value	$v = c$
	$\mid$ new c
	$\mid$ new $C(\overline{v})$
Component value	$c =$ component
	$\mid$ c provides C as x with e
	$\mid$ c forwards $\overline{C}$ as x to e

A value is either a component, a component instance or an object. For component values we have three different constructors. One denotes the empty component, one adds a new service to an existing component value, and a third one adds services by forwarding them to another component instance. Note that during evaluation, service abstractions are eliminated in expressions with reduction rule (R-Req). Therefore, the definition of component values does not include the requires primitive.

Theorem 4.2 states that every well-typed term is either a value or it can be reduced to another term. In other words, evaluation does not get stuck for well-typed terms.

Theorem 4.2 (Progress) If $\vdash e : T$ then e is either a value or $e \longrightarrow e'$ for some e'.

4.5 Component Instantiation Evaluation

The operational semantics presented in Figure 8 formalize an evaluation strategy that does not allow to reduce service implementation expressions inside of component instances. At component instantiation time, in fact none of these terms get evaluated. A term specifying a service implementation, for example in provides or forwards primitives, only gets evaluated when the service is accessed via the :: operator. Evaluating a service implementation expression more than once does no cause any problems in our calculus, since we only have functional objects without any side-effects. In real-world systems, this form of *lazy* evaluation can be efficiently implemented using a memoization technique, so that for multiple accesses to the same service, the service implementation expression will be evaluated only once.

We decided to have this restriction in our calculus for several reasons. First, it keeps the calculus simple. But lazy evaluation also constitutes a reasonable evaluation strategy for service implementations. A *strict* evaluation order would be difficult to define. For instance we could evaluate the service implementations in the order the component evolution primitives introduce a service. But this would be a completely arbitrary choice, since services can be introduced using the requires primitive in any order, not implying any dependencies.

With any fixed strict evaluation order one risks to access a not yet initialized service from the service implementation that is currently being evaluated. With a lazy service evaluation strategy one still faces this problem, but only for recursive service references. With our operational semantics, such recursive dependencies could possibly lead to infinite computations. We avoided this problem in the examples of the previous sections by not accessing services of the own component

instance in service provision expressions directly. Instead, objects that implement a service access other services of the same component instance only at the time a method of the other service actually has to be called, which happens typically after the component got instantiated.

In [57] we present an extension of the operational semantics that supports the reduction of service provision expressions at component instantiation time.

4.6 Component Subtyping

The subtyping rule presented so far only supports *width*-subtyping for component types; i.e. subtypes provide more and require less services. We could relax this rules easily by additionally supporting a form of *depth*-subtyping which incorporates subtyping of service interface types. Here, $\tau_1 <: \tau_2$ would hold for two component types τ_1 and τ_2, if the required service types of τ_1 are supertypes of the required service types of τ_2. Similarly, the provided service types of τ_1 are supposed to be subtypes of the provided service types of τ_2. Exactly this is expressed by the following alternative subtyping rule:

$$\frac{\forall i \exists j : D_j <: C_i \qquad \forall i \exists j : C'_j <: D'_i}{\overline{C} \Rightarrow \overline{C'} <: \overline{D} \Rightarrow \overline{D'}}$$

To make use of such a rule in our type system, we would also have to update the subtyping rule for component instances together with the typing rules (T-Mix), (T-Req), (T-Prv), and (T-Fwd). Furthermore, the service lookup function would have to be modified to reflect the fact that we can now override a service by introducing a new service with a refined type.

5 Discussion and Related Work

Before concluding, we finally review the main ingredients of our component model, explain design decisions, and compare the constructs with related work.

5.1 Prototype-Based Components Revisited

In our model, components are first-class abstractions that have neither state nor identity. Components define the structure of component instances in the same way as classes define the structure of objects. In most class-based languages, classes are either not first-class, or they are specified using meta-classes. For simplicity, and in order to avoid such a meta-regress [55], our first-class components are prototype-based [1]. Thus, instead of instantiating components from meta-component descriptions, new components are derived from prototypical components by a set of refinement primitives. Since components are stateless, we do not need a cloning operation known from object-based programming languages [18,55]. This approach emphasizes the reuse of components in the creation

of new, extended components by refinement. In fact, even component composition, which is mostly regarded as the only form of component reuse, is explained in terms of component refinement.

Components specify implementations for a set of provided services. These implementations may rely on services provided by other components. Thus, component types are characterized by a set of required and provided services. Services are described by nominal object types. In Section 3.3 we explained already why this approach does not constitute a restriction compared to component models with named ports [49,52,2]. Our service abstraction does not only allow us to conveniently refer to an aggregate of functionality, opposed to individual methods, for instance. It also facilitates to override an aggregate of functionality consistently and promotes distinct, non-interfering views of components. Service specifications that are solely based on nominal object types were inspired by *COM* [46,31].

Services are added to a component using the service abstraction and service implementation primitives. For composing components, two mechanisms are supported: forwarding and mixin-based composition. Forwarding delegates the implementation of a set of services to another, possibly nested component instance. The significance of the forwarding primitive is two-fold: On the one hand it enables hierarchical component compositions, on the other hand, it is used to hide internal services of encapsulated components.

Opposed to forwarding, the mixin-based approach merges two components by refining one component with the services provided by another component and by rebinding the self reference to the merged component. Compared with the approach based on forwarding where the services of the nested component cannot be overridden and are therefore statically linked, component composition based on mixins yields a fully extensible component where it is possible to redefine service implementations by overriding. On the other hand, forwarding allows us to specify exactly what services to include, opposed to the mixin-based approach which always mixes in all provided services. As mentioned already in Section 3.7, this may lead to accidental overrides. This weakness of our type system could be addressed, for example, by making overriding explicit and by including negative information in component types. Discussions about forwarding versus delegation (object-based inheritance), which can be seen as an implementation technique for mixins, can be found, for instance, in [54,34,15]. Support for dynamic object-based inheritance in a class-based context is provided by Büchi's and Weck's *generic wrappers* [15] and Kniesel's object model *Darwin* [34].

Mixins were first identified as linguistic abstractions for generalizing inheritance by Bracha and Cook [11]. It was also Bracha who observed that inheritance can be seen as a mechanism for modular program composition [13]. With his work on the programming language *Jigsaw* [10], he lifts the notion of class-based inheritance and overriding to the level of modules.

A formal account of mixins and mixin-based inheritance is given in [9,21,4]. In particular, Bono, Patel, and Shmatikov's calculus of first-class classes and mixins is similar to our work [9]. Bono's mixins correspond to components in our model.

Classes correspond roughly to components without required services. Based on the same framework, Bettini, Bono, and Venneri recently showed that mixins are a suitable abstraction for mobile software components [8]. Opposed to the work by Bono *et al.*, the programming language *Scala* [42] does not distinguish between classes and mixins. It only has the notion of classes that are interpreted as mixins when used in mixin-based class compositions (inheritance). This is identical to the way we interpret components. Scala's mixins were inspired by *Strongtalk* [12], an extension of the programming language *Smalltalk*.

5.2 Related Work

Our work is strongly related to alternative proposals for component abstractions on the level of programming languages. Seco and Caires describe *ComponentJ*, a simple typed imperative core calculus for first-class components in the context of inheritance-free object-oriented programming [49]. *ComponentJ* completely avoids inheritance in favor of object composition. Components are closed black-boxes that can be dynamically composed.

ACOEL has a similar component model [52]. Interaction points of *ACOEL* components are in- and out-ports. The language is class-based and supports a restricted form of inheritance. Like in *ComponentJ*, ports are connected explicitly. Opposed to *ComponentJ*, the design of *ACOEL* does not allow to check that all ports are connected. *ACOEL* supports a richer form of component subtyping, including other constraints, specified in *CORAL*, a language for abstracting and specifying *ACOEL* components [51].

ArchJava is an extension of *Java* that tries to unify the software architecture of a system with its implementation [2]. It introduces direct support for components, connections and ports. Components are implemented using extensible component classes. *ArchJava* does not distinguish between required and provided ports. Instead, a port declares required and provided methods. Ports are again connected explicitly. Like the previous two languages, *ArchJava* allows component composition only via nesting of subcomponents. A distinct feature of the *ArchJava* type system is to guarantee communication integrity [41].

Ibrahim formalizes *COM* by introducing a small programming language *COMEL* [31]. Similar to our approach, *COMEL* does not have named ports. Services are specified solely by type names. In the spirit of *COM*, *COMEL* emphasizes aggregation and does not support implementation inheritance. *COMEL* components have to be self-contained, not having any context dependencies. This is a severe restriction that contradicts the aim to modularize software into small components that have to depend on their deployment context in order to be flexibly reusable.

Most concepts of component-oriented programming languages originate from notions of architectural description languages (ADLs) like *ACME* [24], *Aesop* [23], *Darwin* [37], *Rapide* [35], *Wright* [3] etc. ADLs are used to specify a software architecture formally. A software architecture describes the organizational structure of a software system in terms of a collection of components and relationships among them [44,50]. Typically, a specification of a software archi-

tecture contains information about the participating software components, the connections between these components and constraints on the interactions [54]. By using ADLs, the details of a design get explicit and more precise, enabling formal analysis techniques. Furthermore, they can help in understanding the structure of a system, its implementation and reuse. A comparison of ADLs is given in [40].

Advanced module linking [16,25] and component systems that are built on top of a programming language can be used to model component systems as well. Module systems with external linking facilities include *SML*'s *functors* [36] and *MzScheme*'s *units* [20]. Opposed to our components, *SML functors* are neither first-class nor higher-order. Consequently, they cannot be used to dynamically manufacture modules. Furthermore, they are not extensible, which makes it difficult to perform adaptations. An extension of *SML* with first-class modules was recently proposed by Russo [47,48].

Unlike *SML* modules, *units* offer better support for component-oriented programming [20,19]. They provide first-class module abstractions and linking facilities to compose modules hierarchically. Like all the component-oriented languages mentioned before, *units* are linked by explicitly connecting provided with required ports. Since port descriptions of *units* are relatively fine-grained — they are, in fact, just variable definitions —, this can be a tedious task. For this reason, *MzScheme* supports *signed units* that support bundles of variables, called signatures, being connected in one step [19]. Even thought superficially similar to services in our component model, signatures are merely syntactic sugar and are flattened to a linear list of variables. *Jiazzi* [38] is a working enhancement of *Java* with support for large-scale software components based on *MzScheme*'s *units*. *Jiazzi*'s units are conceptually containers of compiled *Java* classes with support for well-defined connections, specified by a number of imported and exported classes.

A comparable module system for *Java*-like programming languages was proposed by Ancona and Zucca [7]. This system is based on *CMS* [5,6], a simple but expressive calculus of modules which can be instantiated over an arbitrary core calculus. The calculus supports a large variety of module composition mechanisms including mixin module composition with overriding. Recently, Hirschowitz and Leroy adapted the type system of *CMS* to a call-by-value setting [29].

6 Conclusion

In this paper, we presented a component model that was designed to support the implementation and evolution of lightweight, extensible components in object-oriented programming languages. The model supports dynamic component manufacturing and composition in a type-safe way through a minimal set of component refinement primitives. Opposed to other approaches, we do not need to link services of components explicitly. Instead, components are composed using high-level composition operators. We formalized the component model as an extension

of *Featherweight Java* and prove our type system to be sound with respect to the operational semantics. Currently, we are investigating how to integrate our component model into a full programming language.

Acknowledgments

I am grateful to Martin Odersky for valuable discussions about related topics. I would also like to thank Christoph Zenger and Martin Sulzmann for their comments about the type soundness proof.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996.
2. J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.
3. R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1997.
4. D. Ancona and E. Zucca. A theory of mixin modules: basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
5. D. Ancona and E. Zucca. A primitive calculus for module systems. In *Principles and Practice of Declarative Programming*, LNCS 1702. Springer-Verlag, 1999.
6. D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 2001.
7. D. Ancona and E. Zucca. True modules for Java-like languages. In *Proceedings of European Conference on Object-Oriented Programming*, LNCS 2072. Springer-Verlag, 2001.
8. L. Bettini, V. Bono, and B. Venneri. Coordinating mobile object-oriented code. In *Proceedings of Coordination 2002*, York, UK, April 2002.
9. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 43–66, Lisbon, Portugal, 1999.
10. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
11. G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
12. G. Bracha and D. Griswold. Extending Smalltalk with mixins. In *OOPSLA '96 Workshop on Extending the Smalltalk Language*, April 1996.
13. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.
14. M. Büchi and W. Weck. Compound types for Java. In *Proceedings of OOPSLA 1998*, pages 362–373, October 1998.
15. M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 201–225, June 2000.

16. L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997.
17. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
18. C. Chambers and C. Team. The Cecil language, specification and rationale, December 1998.
19. M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, June 1999.
20. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
21. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 1998.
22. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
23. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT '94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.
24. D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON '97*, November 1997.
25. N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Conference Record of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, Texas, 1999.
26. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series, Sun Microsystems, second edition, 2000. ISBN 0-201-31008-2.
27. O. M. Group. The Common Object Request Broker: Architecture and specification, revision 2.0, February 1997.
28. A. Hejlsberg and S. Wiltamuth. C# language specification. Microsoft Corporation, 2000.
29. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proceedings of the European Symposium on Programming*, Grenoble, France, April 2002.
30. U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 36–56, 1993.
31. R. Ibrahim. COMEL: A formal model for COM. Technical report, Queensland University of Technology, Brisbane, Australia, 1998.
32. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, volume 34(10), pages 132–146, 1999.
33. JavaSoft. JavaBeans™. `http://java.sun.com/beans`, December 1996.
34. G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, Lisbon, Portugal, 1999.
35. D. Luckham, L. Augustin, J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. In *IEEE Transactions on Software Engineering*, April 1995.
36. D. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, New York, August 1984.

37. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, Barcelona, Spain, September 1995.
38. S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, October 2001.
39. S. McDirmid, M. Flatt, and W. C. Hsieh. Mixing COP and OOP. In *OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 29–32. Technical Report NU-CCS-01-06, Northeastern University, Boston, MA, October 2001.
40. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, pages 70–93, January 2000.
41. M. Moriconi, X. Quian, and A. Riemenschneider. Correct architecture refinement. In *IEEE Transactions on Software Engineering*, volume 21, April 1995.
42. M. Odersky. Report on the programming language Scala. École Polytechnique Fédérale de Lausanne, Switzerland, 2002. `http://lamp.epfl.ch/~odersky/scala`.
43. M. Odersky. Objects + views = components? In *Proceedings of Abstract State Machines 2000*, March 2000.
44. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 40–52, October 1992.
45. B. C. Pierce. *Types and Programming Languages*. MIT Press, February 2002. ISBN 0-262-16209-1.
46. D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
47. C. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.
48. C. Russo. First-class structures for Standard ML. In *Proceedings of the 9th European Symposium on Programming*, pages 336–350, Berlin, Germany, 2000.
49. J. C. Seco and L. Caires. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, 2000.
50. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
51. V. C. Sreedhar. ACOEL on CORAL: A component requirement and abstraction language. In *OOPSLA Workshop on Specification and Verification of Component-Based Systems*, October 2001.
52. V. C. Sreedhar. Programming software components using ACOEL. Unpublished manuscript, IBM T.J. Watson Research Center, 2002.
53. C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.
54. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley / ACM Press, New York, 1998. ISBN 0-201-17888-5.
55. D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, March 1991.
56. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994.
57. M. Zenger. Type-safe prototype-based component evolution. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, April 2002.