Template Constructors for Reusable Object Initialization

Marko Martin
Technische Universität Darmstadt
MarkoMartin@gmx.net

Mira Mezini
Technische Universität Darmstadt
mezini@cs.tu-darmstadt.de

Sebastian Erdweg
Technische Universität Darmstadt
erdweg@cs.tu-darmstadt.de

Abstract

Reuse of and abstraction over object initialization logic is not properly supported in mainstream object-oriented languages. This may result in significant amount of boilerplate code and proliferation of constructors in subclasses. It also makes it impossible for mixins to extend the initialization interface of classes they are applied to. We propose *template constructors*, which employ template parameters and pattern matching of them against signatures of superclass constructors to enable a one-to-many binding of super-calls. We demonstrate how template constructors solve the aforementioned problems. We present a formalization of the concept, a Java-based implementation, and use cases which exercise its strengths.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, inheritance; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords constructors, object initialization, reusability, mixins

1. Introduction

Reuse of and abstraction over object initialization logic is not properly supported in mainstream object-oriented languages.

First, mainstream object-oriented (OO) languages do not support constructor inheritance. In wide-spread languages like Java [2] and C# [11], to "inherit" the initialization logic encoded in some superclass constructor *BC(*T_1 p_1*, ...,* T_n p_n*)*, subclasses must declare a constructor with the same number and type of formal parameters, *SC(*T_1 p_1*, ...,* T_n p_n*)*, which just makes a *super*-call with its formal parameters as arguments. This *copy-down* pattern introduces boilerplate code and a high degree of rigidity because changes to constructors in such a hierarchy require all subclasses to adopt the changed constructor. Multiple inheritance often even worsens the problems of single-inheritance languages with respect to reusability of object initialization logic: In many built-in variants of multiple inheritance, e.g. non-virtual inheritance in C++ [8, 19], each constructor must call one constructor of each superclass, thereby establishing a hard coupling to the superclasses and to all referenced constructors. It is similar in Eiffel [14], although invocation of superclass constructors is not enforced (which may itself result in inconsistency problems). The situation is not better in Scala [17], where every class has a *primary* constructor implicitly encoded by the parameters and body of a class. All non-primary constructors must transitively call the primary constructor as first statement and only the primary constructor may call a constructor of the superclass. Consequently, even the copy-down reuse pattern is applicable to only one constructor of a superclass. Only languages that support the notion of classes as first-class objects and constructors as ordinary methods/features of these class-objects that are subject to inheritance, such as Smalltalk [10] and derivatives thereof, avoid the problems discussed so far.

Second, all languages mentioned above, including Smalltalk, do not offer mechanisms that enable a subclass to abstract over the initialization logic of the superclass. To extend the initialization logic of superclass constructors, a subclass needs to define constructors that propagate some of their parameters to superclass constructors and use the rest to initialize fields introduced by the subclass. If the superclass defines N constructors and the subclass wants to offer M different ways of initializing fields that it introduces, the subclass may end up defining $N \times M$ constructors in order to provide all possible initialization variants to clients, a phenomenon which we call *constructor explosion* in this paper.

To quantify the significance of the problem, we analyzed the Qualitas Corpus[1] [20], a carefully maintained collection of open-source Java systems. Altogether we analyzed 103 systems with a total number of 68,858 classes. We found 26,946 constructors that make a super-call with at least one argument. Of those, roughly 68% exhibit the problems outlined above: 53.3% propagate all parameters of the superclass constructor into their own parameter list, and another 14.6% do the same, but additionally declare some new parameters.

Lack of support for abstraction over object initialization logic represents an even more severe problem in languages with support for mixins [16], also called abstract subclasses [3]. Contrary to normal subclasses with statically known superclass(es), the superclass parameter of a mixin abstracts over an unlimited number of potential concrete superclasses to which the mixin applies.[2] Since the superclass is not statically known, for mixins it is not even possible to copy down the constructors of the superclass, which means that mixins have no way to extend or otherwise influence the initialization logic of their prospective superclasses.

In this paper, we propose *template constructors* – a linguistic means to address the problems outlined above. Template constructors support a powerful form of constructor inheritance, which goes beyond the normal OO inheritance of methods. Typically, a subclass inherits the public methods of its superclass and exposes their signatures to clients without change. An overriding method can

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/2517208.2517212

[1] Qualitas Corpus Version 20101126, http://qualitascorpus.com.

[2] In statically typed languages, an upper-bound can be given for the type of the superclass parameter.

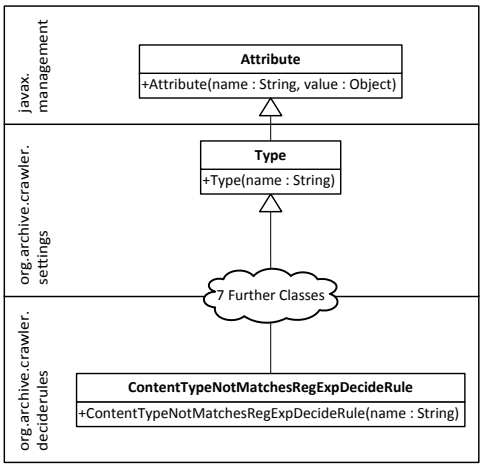

Figure 1. Copy-down Constructor Inheritance in Heritrix

call the overridden method of the superclass via a statically bound method call. For flexible constructors, this overriding technique is not sufficient for two reasons: 1. Subclasses commonly add new state that must be initialized with new constructor parameters. Yet, these new parameters and the associated initialization code should usually be an extension of *all* superclass constructors and not only of a single one. 2. For mixin constructors, the superclass constructor to be called is not known statically.

To overcome the limitations of normal OO inheritance, template constructors can employ *template parameters* to inherit/override superclass constructors based on pattern matching over constructor parameters. This way, template constructors effectively decouple *super*-calls in a subclass from the actual superclass constructors along two dimensions: A single template constructor can inherit/override multiple superclass constructors, and a template constructor can be used in a mixin to augment the initialization logic of different dynamically determined superclasses.

With this paper, we make the following contributions: 1. We motivate the need for more flexible constructors based on an empirical investigation of the the Qualitas Corpus (Sec. 2). 2. We provide an intuitive and a formal presentation of template constructors (Sec. 3 and 5) and an implementation of them as an extension of CaesarJ [1] (Sec. 6). 3. We demonstrate their usefulness by several use cases from the Qualitas Corpus and quantify potential improvements they bring to systems therein (Sec. 4).

2. Problem Statement

In this section, we demonstrate the problems of constructors in current OO languages by investigating real-world Java systems and by illustrating the problems with mixins.

2.1 Copy-down Constructor Inheritance

We exemplify the copy-down constructor inheritance phenomenon by its occurrence in a real system, quantify its frequency in the Qualitas Corpus, and discuss its problems.

Figure 1 shows a simplified version of the hierarchy of Heritrix, a Java web crawling API [15]. The class *ContentTypeNotMatchesRegExpDecideRule* is at the end of a nine-classes-deep hierarchy (the base class is *Attribute*, a subclass of *Object*). The *name* parameter of the *ContentTypeNotMatchesRegExpDecideRule* constructor is – without change – propagated by the constructors of all classes along the hierarchy up to *Attribute*, which handles and stores it.

The occurrences of the copy-down constructor inheritance phenomenon are significant in the Qualitas Corpus. Table 1 shows the frequencies of copy-down constructors in the Qualitas Corpus grouped by hierarchy depths: Of all 53,937 analyzed constructors with at least one parameter, 27,613 (51.2%) forward at least one parameter to another constructor with a this-call or super-call. In 9,622 cases (17.8% of all), the invoked constructor forwards the parameter again. Hierarchies with a depth of five or more are rare (0.9%), but still appear in 22 (21.4%) of 103 analyzed systems.

The copy-down constructor inheritance results in a lot of boilerplate code. Consider the case, when *Attribute* in Figure 1 has not only one but N constructors that need to be copied down the hierarchy to *ContentTypeNotMatchesRegExpDecideRule*. For each of these N constructors, there will be $9 \cdot N$ copies only for the inheritance path in Figure 1. Depending on the depth and average fan out of the hierarchy, the overall number of constructor copies becomes overwhelming.

Copy-down constructor inheritance also impairs evolution. Consider the scenario, when a new design choice requires the *Type* constructor (Figure 1) to take another parameter, say *description*. In the worst case, *description* needs to be added to the list of constructor parameters of all classes down to *ContentTypeNotMatchesRegExpDecideRule and* of classes in other inheritance paths with *Type* as base. To avoid such ripple effects, changes to constructor signatures are typically avoided; instead, setter methods are introduced for new class attributes. However, a design with setter methods is (a) error-prone because calling them is not enforced and (b) inappropriate in case of conceptually immutable objects because clients get unlimited write access to fields that should rather be set only once: during object construction.

2.2 Constructor Explosion

The constructor explosion problem refers to the phenomenon of the exploding number of initialization variants in classes inheriting from superclasses with multiple constructors. For illustration, consider an example from Quartz[3], a Java framework for job scheduling. The class *BaseCalendar* is a basic implementation of a calendar. Its subclass *DailyCalendar* implements a daily time pattern. We list their constructors in Table 2. Each row lists the parameters of one constructor, i.e., *BaseCalendar* has four constructors and *DailyCalendar* has ten. Each of the *DailyCalendar* constructors forwards as many arguments as possible to the superclass constructor. For example, *DailyCalendar(TimeZone timeZone, long startTimeInMillis, long endTimeInMillis)* calls *super(timeZone)*. Still, *DailyCalendar* adds some new state that can be initialized in different ways. By analyzing its constructors, we identify four variants: 1. String startTime, String endTime 2. int startHour, int startMinute, int startSecond, ... 3. Calendar startCalendar, Calendar endCalendar 4. long startTimeInMillis, long endTimeInMillis

For the first two *BaseCalendar* constructors in the table, all possible combinations have been declared, resulting in the first eight constructors of *DailyCalendar* given in the table. For the last two *BaseCalendar* constructors, only the last initialization variant with two longs has been implemented, resulting in the last two *DailyCalendar* constructors of the table. Yet, there is actually no reason for not having the other initialization variants for those constructors; writing them down is just tedious and error-prone.

The exploding number of constructors is not a rarity in the Qualitas Corpus. Table 3 shows how many superclass constructors are extended by how many different variants of initializing the subclass state: Most frequently, an initialization variant of the subclass extends exactly one superclass constructor. Yet, a significant number of initialization variants extend multiple constructors: In 3,321 cases, we found 2 subclass constructors that perform the same subclass initialization but call different superclass constructors, result-

[3] http://quartz-scheduler.org/

Depth	0	1	2	3	4	5	6	7	8	9
Frequency	26324	17991	6555	1886	692	358	85	30	12	4
Accumulated	53937	27613	9622	3067	1181	489	131	46	16	4
Percentage	100.0%	51.2%	17.8%	5.7%	2.2%	0.9%	0.2%	0.1%	$< 0.1\%$	$< 0.1\%$

Table 1. Frequencies of Constructor Hierarchy Depths in the Qualitas Corpus

BaseCalendar	**DailyCalendar**
(no parameters)	`String startTime, String endTime`
	`int startHour, int startMinute, int startSecond, int startMillis, int endHour, int endMinute, int endSecond, int endMillis`
	`Calendar startCalendar, Calendar endCalendar`
	`long startTimeInMillis, long endTimeInMillis`
`Calendar baseCalendar`	`Calendar baseCalendar, String startTime, String endTime`
	`Calendar baseCalendar, int startHour, int startMinute, int startSecond, int startMillis, int endHour, int endMinute, int endSecond, int endMillis`
	`Calendar baseCalendar, Calendar startCalendar, Calendar endCalendar`
	`Calendar baseCalendar, long startTimeInMillis, long endTimeInMillis`
`TimeZone timeZone`	`TimeZone timeZone, long startTimeInMillis, long endTimeInMillis`
`Calendar baseCalendar, TimeZone timeZone`	`Calendar baseCalendar, TimeZone timeZone, long startTimeInMillis, long endTimeInMillis`

Table 2. Constructors of Example Classes from the Quartz Framework

Extended Constructors	1	2	3	4	5	6	7	8	9	10	11	12	13	$\sum$
Frequency	35492	3321	349	292	26	23	9	2	4	1	0	2	1	
Possible Savings	0	3321	698	876	104	115	54	14	32	9	0	22	12	5257

Table 3. Absolute Frequency of Initialization Variants Extending a Certain Number of Constructors in the Qualitas Corpus

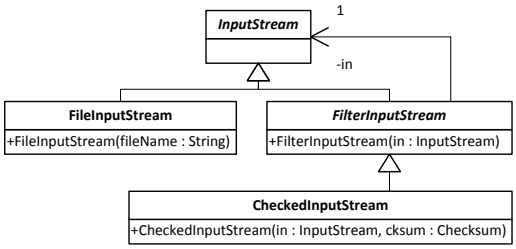

Figure 2. Extract of the *InputStream* Hierarchy from the Java API

ing in 6,642 constructors. In 349 cases, 3 superclass constructors were extended in a subclass, resulting in 1047 constructors.

The phenomenon described here has two main problems. The exploding number of constructors hampers understandability. Further, it causes the subclass-specific initialization logic to be repeated, causing boilerplate code and hampering maintainability and evolution.

2.3 Constructors in Mixins

The third problem we want to address is lack of support for mixin constructors. When a new class C' is generated by applying a mixin M to a superclass C, it is desirable to generate constructors in C' that combine the initialization logic of the superclass C with the initialization logic mixed in by M.

For illustration, consider a mixin variant of the *InputStream* hierarchy from the Java API (cf. Figure 2). *FileInputStream* is a basic implementation of *InputStream* providing access to a file. *FilterInputStream* is the basis for decorator functionality such as the *CheckedInputStream*, which calculates a checksum of the stream content according to a *Checksum* object provided as a parameter to its constructor.

An equivalent implementation with mixins would be to define *CheckedInputStream* as a mixin that can be applied to any *InputStream*. When applying it to *FileInputStream*, it would be desirable that the resulting new class has a constructor with two parameters: (a) *String fileName* to initialize the superclass part from *FileInputStream*, and (b) *Checksum cksum* to initialize the mixin part from *CheckedInputStream*. This constructor mixes the initialization logic from the superclass with the initialization logic from the mixin.

To the best of our knowledge, no OO language currently supports mixin constructors in this fashion. Instead, existing OO languages rely on implicit policies for the initialization of a mixin (for example, by calling setter methods) after the superclass initialization has completed.

3. Template Constructors in a Nutshell

A template constructor denotes a family of traditional Java-like constructors. Template constructors feature template parameters that abstract over any specific parameter list. By passing template arguments to the super-call, a single template constructor can extend many superclass constructors simultaneously. The targets of a super-call are determined based on pattern matching the super-call arguments against the parameter lists of superclass constructors. This abstraction over superclass constructors enables a single template constructor to implement an initialization variant of the subclass independently of the initialization variants of the superclass.

Table 4 compares Java constructors and template constructors with regard to parameters and arguments to super-calls. Like Java constructors, the declaration of a template constructor contains a comma-separated, ordered list of type/identifier pairs. Correspondingly, a super-call takes an ordered list of expressions as arguments, each with a certain type. The types of argument expressions are matched against the types of the formal parameter list of the superclass constructors in the order in which arguments appear.

	Java Constructor	Template Constructor
Parameter definition	identifier and type	identifier and type
		template parameter
Argument for super-call	expression	expression
		named expression
		template argument
Order of arguments	relevant	relevant (except for named expressions)

Table 4. Java Constructors versus Template Constructors

```
class DailyCalendar extends BaseCalender {
   private Calendar start;
   private Calendar end;
   public ? DailyCalendar(p*, Calendar start,
        Calendar end) {
      super(p*);
      this.start = start;
      this.end = end;
   }
}
```

Listing 1. Template Constructor for *DailyCalender*

3.1 Template Parameters and Arguments

In addition to type/identifier parameters, template constructor declarations may specify *template parameters*. A template parameter consists of an identifier followed by an asterisk (*). In addition to expression arguments, template constructors can use template arguments – the counterpart to template parameters – as arguments to a super-call. Each template parameter must appear as a template argument in the super-call and vice versa. The template argument matches an arbitrary list of parameters of a superclass constructor and propagates these into the parameter list of the constructor which contains the super-call. As a consequence, a super-call may match multiple superclass constructors – thereby enabling the desired one-to-many binding of constructor super-calls.

For illustration, consider the template constructor of the class *DailyCalender* in Listing 1. This constructor uses a template parameter *p** in its list of formal parameters and uses a template argument *p** in the super-call in its body. Accordingly, the template constructor accepts each list of arguments *p** followed by two objects of type *Calendar*, such that there is a superclass constructor accepting the list *p**. The template constructor uses the other two arguments to initialize two fields that the class *DailyCalender* introduces. The question mark in the declaration of template constructors clarifies at a glance that there exist as many versions of the constructor as there are superclass constructors that match the template constructor's super-call. Since *super(p*)* matches all superclass constructors, one *DailyCalender* constructor is generated for each constructor of *BaseCalender*.

3.2 Named Expression Arguments

Template constructors also support *named expression arguments* of the form *<name> : <expression>* in super-calls. A named expression argument only matches a formal parameter with exactly the same name and, in contrast to unnamed expressions, the positions of the named expression and the matched parameter are irrelevant. The motivation for this feature is as follows.

Template parameters/arguments expose all parameters of superclass constructors as constructor parameters in subclasses. Sometimes, however, a subclass may want to set a certain parameter of a superclass constructor and stop delegating the initialization responsibility to further subclasses or even to clients. For illustration, consider again Figure 1. The base class *Attribute* has a constructor parameter named *value*, which does not reappear in subclass constructors because it is set to a certain value by the *Type* constructor. To preserve this feature, template constructors must be able to set certain superclass parameters, while abstracting over the others.

```
class Type extends Attribute {
  public ? Type(p*) { super(p*, value : null); }
}
```

Listing 2. Template Constructor for *Type* with Named Expression

In the example, the subclass wants to set the last parameter of the superclass constructor. The subclass can influence the initialization of the superclass by simply passing an expression argument in the last position of the super-call. For example, `super(p*, null);` would set the last parameter to `null` and forward the others. However, given that template constructors are made to cope with later changes of superclass constructors, assumptions about the position of a parameter in the super-constructor signature may render the design fragile in the presence of evolution. If the parameter list of *Attribute*'s constructor is modified by removing, adding, or reordering parameters, the subclass constructor setting only the *value* parameter should not break as long as the *value* parameter is still present in the modified list; instead, it should set the *value* parameter as before and forward the other – possibly new – parameters to subclass constructors. With a named expression argument, a parameter can be set regardless of its position in the superclass constructor. Listing 2 demonstrates this for the example of the *value* parameter, which is set to `null`.

It should be mentioned that named-based parameter matching, of course, requires stability of parameter names in superclass constructors. This means, in contrast to Java, programmers do not have to be careful with changes concerning the order of parameters but with changes concerning their names.

3.3 Instantiating Template Constructors

We call constructors that can be executed by a Java virtual machine *concrete* to distinguish them from template constructors. Template constructors are written to the binary class files during compilation. When the containing class is loaded, they are converted to concrete constructors. This two-step procedure allows a class C to adopt initialization variants of superclass C' without recompilation, which is particularly relevant if the source code of C is not available. We call the process of converting template to concrete constructors *constructor generation*; it is specified by Algorithm 1 and executed when loading a class C with superclass C'. *match(Con, Con')* matches the super-call arguments of a constructor *Con* against the formal parameter list of a concrete superclass constructor *Con'* and produces a matching σ as an output, if one exists. σ maps template arguments in the super-call of *Con* to parameters of *Con'*.

We formalize the matching semantics in Sec. 5. For an initial intuition, here we discuss some examples for matching super-calls against formal parameters of a superclass constructor (cf. Table 5). The first two columns denote the input to the matching procedure: The *Arguments* column shows the arguments of the super-call; the *Parameters* column shows the formal parameters the super-call is matched against. The column *Matching* shows the output of the matching procedure: either the possibly empty ($\emptyset$) mapping from template arguments to parameters, or "–" if the matching fails.

Rows 1 and 2 in Table 5 show examples that are equivalent to method parameter matching in Java. Examples with a named argument are given in rows 3 (matching is successful but empty because

Algorithm 1 ConstructorGeneration(C, superclass C')

```
if C' contains template constructors then
    call ConstructorGeneration(C', superclass(C'))
end if
for all template constructors Con in C do
    for all concrete constructors Con' in C' do
        σ ← match (Con, Con')
        if matching succeeded then
            CCon ← instantiate (Con, Con', σ)
            Add CCon to C.
        end if
    end for
    Remove Con from C.
end for
```

#	Arguments	Parameters	Matching
1	`5, 0`	`int x, int y`	∅
2	`5, 0`	`int x`	–
3	`x:5, 0`	`int x, int y`	∅
4	`x:5, 0`	`int a, int b`	–
5	`p*`	`int x, int y`	`p*/(x, y)`
6	`p*, y:0`	`int x, int y`	`p*/(x)`
7	`p*, y:"s"`	`int x, int y`	–
8	`p*, c:4, 3`	`String a, int b, int c`	`p*/(a)`

Table 5. Examples of Constructor Matching

there are no template arguments to be mapped) and 4 (matching fails due to wrong name). Rows 5 to 8 illustrate template arguments: The super-call argument *<name>** as in row 5 generally matches every possible parameter list completely. The pattern in row 6 uses one named argument (*y*) so that all other parameters (*x*) are mapped to the template argument *p**. The matching in row 7 fails because the pattern requires parameter *y* to have type *String*, which is not true. Row 8 demonstrates the irrelevant position of named expressions. As the value 4 is assigned to parameter *c*, the residual argument list *p*, 3* is matched against the residual parameter list *String a, int b*; hence, *p** is mapped to *a*.

If the matching is successful, *instantiate(Con, Con', σ)* (cf. Algorithm 1) instantiates a template constructor *Con* with a matching σ and a referenced superclass constructor *Con'* to a concrete constructor *CCon* in the following manner: 1. Replace each template parameter $p*$ of Con by the formal parameter list $\sigma\,(p)$. 2. For each template argument $p*$ of Con, add instructions which load the arguments provided for the parameters $\sigma\,(p)$. 3. Bind the super-call to Con'.

If there is no superclass constructor matching the super-call of a template constructor, the template constructor is just removed. If there are multiple matching superclass constructors, one concrete constructor is generated for each of them.

4. Template Constructors in Action

In this section, we discuss the benefits of template constructors on the design of programs.

4.1 Avoiding Copy-down Constructor Inheritance

In Sec. 2.1, we discussed two problems with the design of programs that employ copy-down constructor inheritance: Abundance of boilerplate code and fragility in the presence of evolution, because a change to a constructor signature may entail many cascading changes to subclass constructors. We first discuss how the application of template constructors fosters evolution.

For illustration, consider a version of the Heritrix example from Sec. 2.1, where all direct and indirect subclasses of *Type* (Figure 1) use template constructors. For example, the template constructor for *ContentTypeNotMatchesRegExpDecideRule* looks as follows:

```
public ? ContentTypeNotMatchesRegExpDecideRule(p*) {
    super(p*);
}
```

As a result, signature changes of *Type*'s constructors automatically propagate to the subclasses, which do not need to be adapted manually any more. Only clients of these classes that call the constructors still have to be adapted to provide the correct arguments to the constructor calls. This is unavoidable, given that the interface of the class used by these clients has changed. Compile-time errors can indicate instantiation sites in the code where modifications are necessary.

In the example of the *ContentTypeNotMatchesRegExpDecideRule* hierarchy, only one constructor is defined in the base class *Type* and entailed through the whole hierarchy. If there are more than one, template constructors are even more effective: Only one template constructor – analogous to the one above – has to be defined per class in order to expose all constructors of the superclass.

In the current design, a subclass developer has to define the trivial template constructor manually to enable constructor inheritance (see, e.g., the template constructor *ContentTypeNotMatchesRegExpDecideRule* again). An alternative design is to generate such a "trivial" template constructor by default. The former alternative resembles Java's strategy not to inherit constructors by default. The latter resembles Smalltalk's strategy to do so. There are tradeoffs related to these alternatives: The explicit template constructors strategy enables subclass developers to decide on whether to expose superclass constructors in the instantiation interface of the subclass or not. When generating the trivial template constructor by default, the developer of the subclass does not have such control and the initialization interface may become wide, bearing the risk that clients are more exposed to constructor changes in the hierarchy.

There is another aspect of object instantiation in the presence of template constructors that needs closer consideration. In Java, the signatures of available constructors are explicitly specified in a class. Without dedicated tool support for template constructors, the programmer of a client class has to look at super-calls in the implementation of constructors in order to know which concrete constructors will be generated. While type safety is ensured by compiler checks at class instance expressions (cf. Sec. 6), lack of an explicit instantiation interface may impair understandability and modular reasoning. We believe these problems can be resolved by presenting a view of the concrete constructors to the programmer at class instantiation sites in an IDE.

4.2 Avoiding Constructor Explosion

Template constructors enable a subclass to define initialization variants for the fields it introduces as deltas that apply to all constructors (initialization variants) of its superclass. Four template constructors – two of them shown in the listing below – are, thus, sufficient to preserve all possible initialization variants of *DailyCalendar* from Sec. 2.2 in the combination with the superclass *BaseCalendar*: When *DailyCalendar* is loaded, the constructor generation process produces all 16 possible constructor variants automatically.

```
public ? DailyCalendar(p*, String start, String end) {
    super(p*);
    // initialize with Strings start and end
}
public ? DailyCalendar(p*, Calendar start, Calendar end){
    super(p*);
    // initialize with Calendars start and end
}
// similarly for the other two initialization variants
```

The example demonstrates how template constructors avoid an exploding number of constructors and the need for adapting constructors in subclasses to modifications of constructors in superclasses. To quantify the potential of template constructors to reduce the number of constructors in a larger scale, reconsider Table 3. The last row shows the number of constructors that could have been saved if the systems in the Qualitas Corpus used template constructors. For example, half of the constructors that combine the initialization logic of a subclass with two superclass constructors can be saved (3,321) because one template constructor can capture them both. In total, 5,257 constructors could have been saved, i.e., 11.7% of all 44,779 constructors in classes with a non-*Object* superclass in the analyzed systems of the Qualitas Corpus.

Template constructors also enable *DailyCalendar* to automatically adapt to changes in the constructor interface of the superclass *BaseCalendar*. If only a new constructor is added to *BaseCalendar*, no changes are required neither to *DailyCalendar* nor to its clients; yet, the new constructor is automatically inherited by *DailyCalendar* in all four combinations with its initialization variants. If the signatures of existing constructors change, only places in code, where *DailyCalendar* is instantiated, possibly have to be adapted to the new constructor signatures inherited from *BaseCalendar*.

Template constructors can also be used to reduce the number of constructors within one class with this-calls. Traditionally, a class that has n independent, optional parameters, requires 2^n distinct constructors. With template constructors, we can reduce the number of required constructors to $n + 1$: We need one constructor for the default instantiation that takes no optional arguments, and then one template constructor for each optional argument. For example, for *BaseCalendar* we define:

```
public BaseCalendar() { /* empty constructor */ }
public ? BaseCalendar(p*, Calendar baseCalendar) {
   this(p*);
   setBaseCalendar(baseCalendar);
}
public ? BaseCalendar(p*, TimeZone timeZone) {
   this(p*);
   setTimeZone(timeZone);
}
```

Constructor generation will produce four concrete constructors for *BaseCalendar*: no arguments, single *Calendar* argument, single *TimeZone* argument, two arguments *Calendar* and *TimeZone*. If we add one more optional class argument to *BaseCalendar*, we get four more constructors. For generated constructors that involve multiple arguments, the order of constructor arguments is defined by the order of template constructors. If two optional class arguments have the same type, constructor generation favors the argument whose template constructor precedes the other.

4.3 Template Constructors and Mixins

Template constructors are particularly suitable for mixins. While mixins abstract over a concrete superclass, template constructors abstract over both a concrete superclass and the constructors of the superclass. For example, we can define a template constructor for the mixin *CheckedInputStream* (cf. Sec. 2.3) in order to extend the constructor signature of the class that the mixin is applied to. For example, we can add an additional parameter *chksum*:

```
public ? CheckedInputStream(p*, Checksum chksum) {
   super(p*);
   this.chksum = chksum;
}
```

The main application of template constructors in the context of mixins is to propagate and extend parameter lists of superclass constructors. We do not allow mixins to set parameters of the superclass constructor. That is, template constructors of mixins always have the following form:

$e \in E$	expressions
$v \in V$	variable names
$t \in T$	types
$<: \subseteq T \times T$	subtyping relation
$\tau : E \to T$	mapping to most specific type
$p \in P ::= v : t$	formal parameters
$a \in A ::= e$	expression argument
$\mid v : e$	named expression argument
$\mid v*$	template argument

Figure 3. Syntax and notation for constructor matching

```
public ? C(<NewParams1>, p*, <NewParams2>)
{  super(p*); /* ... */ }
```

The super-call contains one single template argument, which propagates the constructor parameters of the unknown superclass. *<NewParams1>* are added before existing constructor parameters and *<NewParams2>* are added after them.

The design decision not to allow template constructors to set parameters of superclass constructors is motivated by the following two reasons:

1. A mixin does not know the semantics of superclass constructor parameters because it does not know its actual superclass. Hence, guarantees regarding positions and/or names of constructor parameters as required for setting super-constructor parameters (see Sec. 3) are hard to enforce for all classes that a mixin is possibly applied to.
2. When combining multiple mixins, it may quickly become confusing for the programmer which parameter is set by which mixin and which parameters are actually left for being set. Composabilty is fostered if mixins are only allowed to extend constructors of superclasses with new parameters but not to remove parameters by setting them.

5. Formalization of Constructor Matching

We presented some examples of constructor matching in Table 5. In this section, we formalize constructor matching with a calculus.

5.1 Definitions and Notations

Figure 3 introduces syntax relevant for constructor matching. We denote formal parameters P of a concrete constructor by their name and type $v : t$. For constructor arguments A, we distinguish simple expression arguments e, named expression arguments $v : e$, and template arguments $v*$ as illustrated in Section 3.

Based on these definitions, we define an *instance of the constructor matching problem* as a list of arguments to be matched against a list of formal parameters:

$$a_1, \ldots, a_n \doteq p_1, \ldots, p_k$$

The result of the constructor matching problem, if successful, is a matching function $\sigma \in V \rightharpoonup V^*$ which provides a mapping from names of the template arguments in $a_1, ..., a_n$ to names of matched formal parameters in $p_1, ..., p_k$. Note that we write M^* to denote the free monoid on a set M, thus the mapping retains the order of matched parameters V^*. We write ε for the empty sequence.

In addition, we require the following auxiliary definitions. We define the union of partial functions $\mathrm{f}, \mathrm{g} : A \rightharpoonup B$ as follows:

$$\mathrm{f} \cup \mathrm{g} : A \rightharpoonup B : x \mapsto \begin{cases} \mathrm{f}(x) & \text{if } x \in \mathcal{D}(\mathrm{f}) \\ \mathrm{g}(x) & \text{if } x \in \mathcal{D}(\mathrm{g}), x \notin \mathcal{D}(\mathrm{f}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\mathcal{D}(\mathrm{f})$ denotes the set of values for which the partial function f is defined. For the empty partial function f with $\mathcal{D}(\mathrm{f}) = \emptyset$, we write $\emptyset$. For the partial function f with finite $\mathcal{D}(\mathrm{f}) = \{x_1, ..., x_n\}$ and values $v_i = \mathrm{f}(x_i)$, we write $\{x_1/v_1, ..., x_n/v_n\}$.

5.2 Constructor Matching Calculus

We present constructor matching calculus through inference rules that relate a constructor matching problem to a matching function:

$$\mathcal{L} \quad ::= \quad a_1, \ldots, a_n \doteq p_1, \ldots, p_k \quad \mapsto \quad \sigma$$

We define $\bar{\mathcal{L}}$ as the subset of words in $\mathcal{L}$ that are derivable by the matching calculus.

Figure 4 displays the inference rules of the constructor matching calculus. Rule *type* (1) describes usual method parameter matching: Expression e can be used for a parameter with type t if the type of e is t or a subtype thereof. Since the mapping resulting from constructor matching maps template argument names to the corresponding constructor parameters, the reverse mapping of parameters to expressions is not recorded in the mapping. Rule *empty* (2) defines that the empty argument list matches the empty parameter list. Rule *template* (3) expresses that a single template argument matches an arbitrary list of formal parameters with mutually different labels $(S1)$. Rule *name* (4) matches named expression arguments. A named expression argument can occur anywhere in the argument list and can match a constructor parameter of the expected name at any position. We delegate the type checking of the named expression to rule type. Rule *compos* (5) allows lists of arguments to match lists of formal parameters whose labels must be disjunct again. The side condition $(S2)$ ensures that the first argument pattern always matches the longest possible sequence of formal parameters, and $(S3)$ essentially expresses that each template argument may occur only once in a pattern list.

The calculus defines a matching which has a slight similiarity to unification: It aims at finding a substitution for template arguments to match the formal parameters. Yet, it is more than unification because it mixes matching with and without respecting the order of parameters – the order is ignored for named parameters whereas it is important for the others. Also, contrary to pure unification, the calculus respects types and names.

5.3 Properties of the Matching Calculus

The following two theorems lay down two key properties of the constructor matching calculus. Their proofs are available online.[4]

THEOREM 1 (Matching correctness). *If for an instance* $\mathcal{E} = a_1, \ldots, a_n \doteq p_1, \ldots, p_k$ *of the constructor matching problem* $\mathcal{E} \mapsto \sigma$ *is derivable, then applying* σ *to the template arguments in* $a_1, \ldots, a_n$ *and aligning the named expression arguments in* $a_1, \ldots, a_n$ *with the corresponding constructor parameters yields a valid argument list for the formal parameters* $p_1, \ldots, p_k$.

THEOREM 2 (Matching uniqueness). *If for an instance* $\mathcal{E}$ *of the constructor matching problem* $\mathcal{E} \mapsto \sigma$ *is derivable, then the matching function* σ *is unique, i.e., for all* σ' *with* $\sigma' \neq \sigma$, $\mathcal{E} \mapsto \sigma'$ *is not derivable.*

5.4 Solving the Constructor Matching Problem

Now that we have defined the constructor matching calculus, how can we apply it to the constructor matching problem? In formal terms, the matching procedure is specified by the sub-language $\bar{\mathcal{L}}$, consisting of the subset of words in $\mathcal{L}$ which can be derived with the constructor matching calculus. The theorems about the properties of the calculus show that the matching function σ is correct and uniquely determined by a constructor matching problem instance $\mathcal{E}$. Moreover, it is easy to obtain an algorithmic implementation, for example, by resolving all named expressions first.

Figure 5 gives an example for the application of the calculus to the following instance of the constructor matching problem:

$$a*, z : 4, 3 \quad \doteq \quad x : \mathrm{String}, y : \mathrm{int}, z : \mathrm{int}$$

Since a derivation exists, a matching is possible, namely with the matching function in the root of the derivation tree: $\{a/(x)\}$.

The example illustrates that the position of named expressions is indeed arbitrary: With rule *name* in the last step of the tree, the argument *z : 4* is inserted between the other arguments whereas the formal parameter *z : int* is located at the last position. The unnamed expression *3* is assigned to the last formal parameter *except for z*, namely *y*, and the template argument *a** covers the rest, namely *x*.

6. Implementation

We have implemented template constructors in CaesarJ [1], a Java extension, which among other features also supports mixin-based inheritance. More specifically, the implementation extends the static type checking and the Java byte code generation phases of the JastAdd compiler [7] of CaesarJ. We exploit a custom class loader and the ASM bytecode toolkit [4] for applying transformations specific to template constructors. The implementation is available at GitHub; its main building blocks are detailed below.

Type Checking. The integration of template constructors affects the static type checking in two places of the abstract syntax tree generated by JastAdd: (a) class instance expressions (`new C(...)`) and (b) super-calls within constructors.

At *class instance expressions*, the implementation checks whether there is an appropriate constructor available for the class to be instantiated. To perform the check, the implementation partially executes the constructor generation process (cf. Sec. 3) only for the signatures of the template constructors. The check succeeds if one of the generated signatures or one of the non-template constructors contained in the class match the instance expression; otherwise, a compiler error is produced.

At *super-calls within constructors*, the implementation distinguishes between super-calls contained (a) in a template constructor and (b) in a non-template constructor. For super-calls of non-template constructors, the implementation checks whether they refer to an existing non-template constructor in the superclass. If this is not the case, a compiler error is produced. In contrast, super-calls of template constructors are not checked immediately, but rather at class instance expressions (see above).

Bytecode Generation. Figure 6 visualizes the process of compiling template constructors and loading them from a class file by means of the example in Listing 2. The compiled class file is saved with a special suffix that is recognized during class loading. The process of bytecode generation for template constructors (left part of the figure) deviates from that for normal Java constructors in following ways. First, the method signature does not declare any parameters. Second, the method code does not contain a super-call[5]. However, the super-call is prepared by loading the expression arguments of the super-call onto the stack. In the example, an instruction for loading null onto the stack is written to the code attribute (marked as 1 in Figure 6) because null is assigned to the parameter *value* in the super-call. The name *value* of the named argument expression is ignored in the code attribute as well as template arguments are. Third, a special template constructor attribute (marked

[4] https://github.com/tud-stg-lang/caesar-jastadd/blob/mixin-constructors/doc/TemplateConstructors%20-%20Matching%20Calculus.pdf?raw=true

[5] In standard bytecode, the code for super-calls is located between argument loading instructions and remaining constructor instructions.

$$\text{type}\ \frac{}{e \doteq v : t \mapsto \emptyset}\ \tau(e) \leq t \quad (1) \qquad \text{empty}\ \frac{}{\varepsilon \doteq \varepsilon \mapsto \emptyset} \quad (2)$$

$$\text{template}\ \frac{}{v* \doteq v_1 : t_1, ..., v_n : t_n \mapsto \{v/(v_1, ..., v_n)\}}\ \begin{array}{l} n \geq 0 \\ (S1) \end{array} \quad (3)$$

$$\text{name}\ \frac{e \doteq v_k : t_k \mapsto \emptyset \qquad a_1, ..., a_{q-1}, a_{q+1}, ..., a_m \doteq p_1, ..., p_{k-1}, p_{k+1}, ..., p_n \mapsto \sigma}{a_1, ..., a_{q-1}, v_k : e, a_{q+1}, ..., a_m \doteq p_1, ..., p_{k-1}, v_k : t_k, p_{k+1}, ..., p_n \mapsto \sigma}\ \begin{array}{l} m, n \geq 0 \\ 1 \leq q \leq m \\ 1 \leq k \leq n \\ (S1) \end{array} \quad (4)$$

$$\text{compos}\ \frac{a_1 \doteq p_1, ..., p_k \mapsto \sigma_1 \qquad a_2, ..., a_m \doteq p_{k+1}, ..., p_n \mapsto \sigma_2}{a_1, ..., a_m \doteq p_1, ..., p_n \mapsto \sigma_1 \cup \sigma_2}\ \begin{array}{l} m \geq 2 \\ n \geq k \geq 0 \\ (S1), (S2), (S3) \end{array} \quad (5)$$

$$(S1) : \forall i, j \in \{1, ..., n\} : i \neq j \Rightarrow v_i \neq v_j \text{ where } p_x = v_x : t_x$$
$$(S2) : \forall i \in \{k+1, ..., n\} : \forall \sigma_1', \sigma_2' \in V \rightharpoonup V^* :$$
$$\left(\begin{array}{l} ((\quad a_1 \quad \doteq\ p_1, ..., p_i), \sigma_1') \notin \bar{\mathcal{L}} \vee \\ ((a_2, ..., a_m \doteq p_{i+1}, .., p_n), \sigma_2') \notin \bar{\mathcal{L}} \end{array}\right)$$
$$(S3) : \mathcal{D}(\sigma_1) \cap \mathcal{D}(\sigma_2) = \emptyset$$

Figure 4. Rules of the Constructor Matching Calculus

$$\dfrac{\dfrac{(4)\ \dfrac{}{(a* \doteq x : \text{String}), \{a/(x)\}} \quad (1)\ \dfrac{}{(3 \doteq y : \text{int}), \emptyset}}{(1)\ \dfrac{}{(4 \doteq z : \text{int}), \emptyset} \quad (5)\ \dfrac{}{(a*, 3 \doteq x : \text{String}, y : \text{int}), \{a/(x)\}}}\ (2)}{(a*, z : 4, 3 \doteq x : \text{String}, y : \text{int}, z : \text{int}), \{a/(x)\}}$$

Figure 5. Applying the Constructor Matching Calculus

as 2 in Figure 6) is defined, which contains all the template constructor information that cannot be expressed in the code attribute: the parameters of the template constructor (*p** in the example), including identifiers of parameters and template parameters, and a complete list of the super-call arguments, which contains for each argument the type signature if it is a named or unnamed expression and the name if it is a named expression or a template argument.

Class Loading. Loading classes with template constructors is done by our custom classloader (cf. right-hand side of Figure 6). The template constructor attribute (2) contains all the information needed to perform constructor matching against superclass constructors at class load time. The constructor generation procedure (cf. Sec. 3) is now completely executed, resulting in zero, one or more concrete constructors for each template constructor. The process of matching a template constructor against a superclass constructor is as follows: First, the constructor signature (in the example *name : String*) is created using the full parameter list (*p**) contained in the template constructor attribute and the mapping of template arguments to the matched parameters of the superclass constructor ($\{p^*/(\text{name})\}$). Second, instructions are created to load the arguments to be forwarded to the superclass constructor (3); these instructions are interleaved with instructions for loading expression arguments already generated in the bytecode (1). Finally, instructions for invoking the matched superclass constructor are inserted (4) – the binding step of the constructor generation process. The remaining instructions are retained from the bytecode (5).

Since a template constructor can match multiple superclass constructors, the code generated for it is replicated and each replica is provided with specific super-call invocation instructions for each matching superclass constructor.

7. Related Work

Parameterized inheritance in C++ (PI) [18, 21] is a method to support mixins in C++: The mixin is defined as a subclass, which is parameterized by the superclass by using C++ templates. This approach has considerable restrictions regarding abstraction over initialization logic [6]. Mixins have to pass all initialization parameters of the original superclass as well as parameters of previously applied mixins. Hence, mixin constructors hard-code assumptions about constructors of their yet unknown superclasses/super-mixins. This restricts their applicability: One would have to define different constructors for different sets of possible superclasses and different orders of composition. Hence, constructor reusability is not properly supported, if at all only at the cost of a very complex design.

PI with virtual inheritance [6] is proposed as a countermeasure to the above problem with PI. Changing the inheritance mode of mixins to virtual inheritance makes it possible to abstract over the order in which mixins are applied (since, with virtual inheritance, the implementing subclass – instead of the mixin – is responsible for initializing them all). The drawback is that an implementing subclass together with all necessary calls to superclass constructors has to be written explicitly for every desired combination of mixins, yielding high design complexity. Also, the solution inherits the general weaknesses of virtual inheritance: Constructor invocations in mixins are ignored; particularly, mixins do not have any possibility to modify arguments passed to a superclass.

PI with argument class [19] is another workaround for the problems of the PI approach. The constructors of classes/mixins, which may be part of a (multiple) inheritance hierarchy, take an instance of an argument class as a parameter. The latter encapsulates data needed in any class/mixin, enabling a uniform constructor interface. Each mixin constructor can select the data it needs from the argument object and pass it through to the next superclass. This approach achieves flexibility at the cost of increasing design complexity and losing declarative expression of design intent. When many mixins exist, which may eventually not all be used in a certain hierarchy, the size of the argument class may constitute a significant overhead in memory. Also, the argument class must be extended for every new feature which is needed by any constructor of any mixin, completely defeating the open-closed principle [14].

The typed argument list approach [12] – like the argument class approach – is based on a standardized constructor interface; however, the type of the expected constructor parameter is now a heterogeneous value list. Based on C++ templates, the type of each list is generated automatically by the compiler, so that it defines the number and types of expected list elements according to the expected parameter types of the mixin and the inherited parameter types from other mixins. Constructor reusability is now achieved for mixins because changing the superclass of a mixin will not

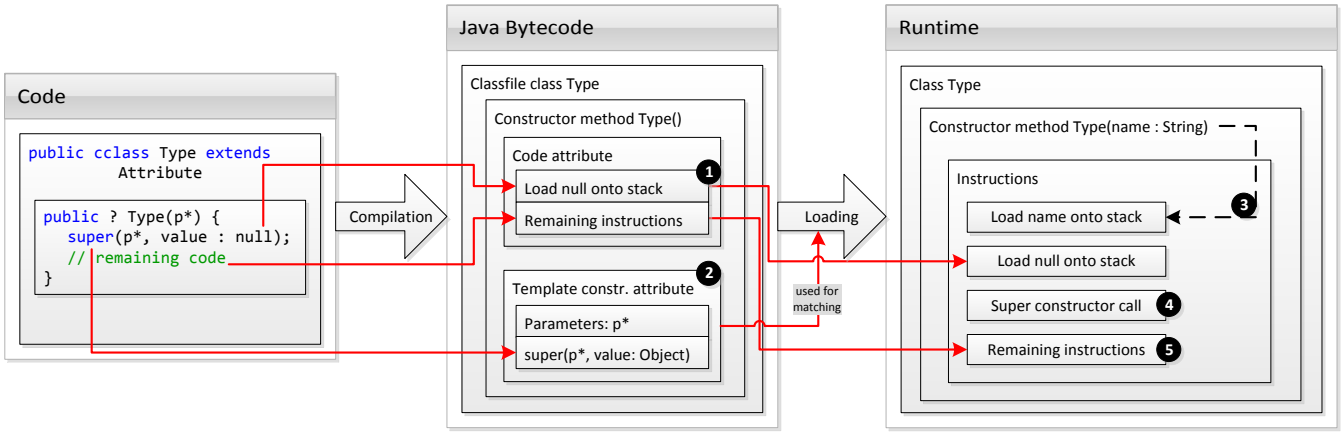

Figure 6. Process of Compiling and Loading the Template Constructor of the Example in Listing 2

render its constructor useless: The correct type of a heterogeneous value list is automatically generated for each possible hierarchy of mixins, which also makes the approach type-safe. The drawback is, however, the overhead for defining the mixins and the infrastructure needed to use the heterogeneous value lists. Design complexity remains high and design intent is still implicit.

Scala traits can have a parameterless default constructor encoding arbitrary initialization logic in the body of the trait after the opening bracket {. The order of trait constructor invocations is defined by the order in which traits are mixed in. The key disadvantage is that traits cannot be initialized with constructor parameters. Hence, initialization methods have to be used when creating trait-based objects. Also, traits cannot initialize their superclass, since *super* constructor calls are not allowed. Like with virtual inheritance in C++, if a trait extends a class without a default constructor, subclasses of the trait must explicitly invoke the superclass constructor. Unlike virtual inheritance, constructor reusability is, at least partially, achieved: A composition of a class with several traits can be created with one line of code, thereby reusing constructors of the class and parameterless constructors of mixins.

Object initializers in C# are a convenient notation to initialize object properties directly after object creation. An object initializer can set any combination of class properties without the necessity of defining a constructor for each combination. However, it is only syntactic sugar because it can be equivalently replaced by setting the properties on the object reference after creating the object: The constructor is entirely executed before the object initializer and can, thus, not perform initialization actions based on the values set in the object initializer. Therefore, object initializers do not bring advantages concerning constructor flexibility compared to Java.

CZ extends Java with multiple inheritance, but avoids the problems of diamond hierarchies [13]. In addition to *extends*, it introduces a new subtyping relationship, *requires*. Classes requiring other classes are abstract: They are not allowed to invoke constructors of required classes, but they can use their features as if they were subclasses. If a class *A* extends *B* and *B* requires *C*, *A* must require or extend *C* or a subclass of *C*. This ensures that *B* can rely on the existence of *C* in a concrete implementation. Superclass decoupling is simulated by only requiring, but not extending a certain class. E.g., *B* in the example above can be composed with any direct or indirect subclass of *C*. A weakness is that combining multiple classes always requires defining a new one, which must explicitly invoke constructors of all extended classes: There may be many of them if many classes requiring some other classes are extended. As with Scala traits, the restriction that constructors of required classes may not be called limits expressiveness of constructors.

Object initialization in Common Lisp Object System (CLOS) [5] shares some similarity with template constructors. Basic constructors for setting object fields by name can be generated automatically. *:before*, *:after*, or *:around* method qualifiers – applied to the initialization function *initialize-instance* – can define additional constructor logic to be executed before, after, or around basic object initialization; in the latter case the arguments can also be modified. With the *&key* keyword, arguments can be extracted by name, and with the *&rest* keyword, other arguments can be forwarded to the wrapped initialization method called with the *call-next-method* keyword. Similarly to template parameters of template constructors, Lisp's *&rest* keyword enables forwarding some parts of the argument list to the next method, whereby the *&rest* concept applies to any method.[6] Its disadvantages in the context of object initialization are the quite complex syntax for forwarding and modifying arguments and the inherent lack of safety: With the *&key* keyword, argument names are conceptually contained in the list of arguments. Setting the argument of a method call is thus equivalent to adding the name of the argument to be set and its value to the list of arguments. A simple typo may leave the desired argument undefined. Also, other – possibly unknown – *:around*-qualified methods of a generic function may silently change the argument list. Hence, there is no safe way for a subclass-specific object initializer to fill an object slot with a certain value.

The Racket class system [9] circumvents many constructor pitfalls with its design to force exactly one initialization variant per class or mixin; unlike Java, multiple variants cannot be encoded in the form of multiple constructors. This is also the greatest drawback because it limits expressiveness of class definitions: For example, a class cannot create a file when a file name is provided for construction, and initialize an output stream when a stream is provided.

Instead, a class defines exactly one list of initialization arguments that must be provided during initialization. (Arguments with default values can be left out.) The arguments can be provided either by subclasses when calling the superclass initialization (`super -new`), or by clients when instantiating the class, which means that initialization arguments are aggregated down the inheritance hierarchy until they are set. As a consequence, subclasses and mixins

[6] In our approach, we restricted template parameters to constructors because we do not have evidence from existing systems that their main use case – abstracting over possibly multiple other constructors – is transferrable to normal methods. Nevertheless, further investigation of template parameters for normal methods might be an interesting aspect for future research.

Solution	Constr. inheritance	Constr. abstraction	No supercl. coupling	Expressiveness	Safety	Performance	Ease of use
Java, C#	–	–	–	+	+	+	0
Smalltalk	+	–	–	+	+	+	+
C++ default inherit.	–	–	–	+	+	+	0
C++ virtual inherit.	–	–	–	0	0	+	0
PI default inherit.	–	–	+	+	+	+	0
PI virtual inherit.	–	–	+	0	0	+	0
PI argument class	–	–	+	+	–	–	0
PI heter. val. lists	+	+	+	0	+	+	–
Scala classes	–	–	–	+	+	+	0
Scala traits	–	–	+	–	0	+	+
CZ *extends* relation	–	–	–	+	+	+	0
CZ *requires* relation	–	–	+	0	+	+	+
CLOS	+	+	+	+	0	+	0
Racket	+	+	+	0	+	+	+
Template constr.	+	+	+	+	+	+	0

Table 6. Comparison of Object Initialization Solutions

do not need to be aware of the initialization arguments of the superclass; if they do not set them, clients or subclasses will do.

Summary. Table 6 summarizes the advantages and weaknesses of various solutions for object initialization. "Constructor inheritance" refers to the ability of classes to inherit constructors from the superclass and to expose them to instantiating clients and subclasses. "Constructor abstraction" pertains to abstracting from superclass constructors in a way which allows to extend them with new parameters. "No superclass coupling" denotes the possibility to write constructors which are not restricted for application to a particular superclass. "Expressiveness" denotes the possibilities – even of mixins – to have constructors with parameters, to influence the initialization of the superclass, and to define multiple initialization variants per class. "Safety" indicates that super-calls are not ignored as for example with virtual inheritance and that initialization methods need not to be called separately because this is unsafe in the sense that it is not enforced by the compiler. "Performance" generally refers to a good runtime performance and requires that the solution does not impose a significant memory overhead. "Ease of use" indicates that boilerplate code is avoided and the particular solution is comfortable to use and does not require a large infrastructure to be applied. A "+" in the table indicates that the respective criterion is fulfilled (nearly) perfectly, "0" indicates partial validity, and "–" indicates that the criterion does not apply. Template constructors are rated with "0" concerning ease of use because the trivial template constructor, which just emulates constructor inheritance, must be written manually by the programmer. The reason was discussed in Sec. 4.1.

8. Summary

In this paper, we presented template constructors as a means to address the problems of object initialization in object-oriented programming languages. Template constructors abstract over concrete superclass constructors by employing named expressions and template arguments for use in super-calls besides unnamed expressions that are used in super-calls in mainstream OO languages such as Java and C++. Template parameters/arguments support matching super-calls against superclass constructors and abolish the need for static coupling super-calls to superclass constructors. A formal foundation of constructor matching was given and applicability and usefulness was shown by several use cases. The discussion in Sec.7 (cf. Tab. 6) indicates that from all existing solutions for object initialization, template constructors provide the best reusability without sacrificing other properties, such as safety and performance.

Acknowledgments

We would like to thank Vaidas Gasiunas for discussion on template constructors, and the anonymous reviewers for their helpful feedback.

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, LNCS, pages 135–173. Springer, 2006.

[2] K. Arnold, J. Gosling, and D. Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311. ACM, 1990.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[5] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *ECOOP*, pages 151–170. Springer, 1987.

[6] U. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *GCSE'2000 Workshop on C++ Template Programming*, 2000.

[7] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[8] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[9] M. Flatt, R. B. Findler, and M. Felleisen. Scheme with classes, mixins, and traits. In *APLAS*, pages 270–289, 2006.

[10] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[11] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *C# Programming Language*. Addison-Wesley Professional, 4th edition, 2010.

[12] J. Järvi. Tuples and multiple return values in C++. Technical report, Turku Centre for Computer Science, 1999.

[13] D. Malayeri and J. Aldrich. CZ: multiple inheritance without diamonds. In *OOPSLA*, pages 21–40, 2009.

[14] B. Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[15] G. Mohr, M. Kimpton, M. Stack, and I. Ranitovic. Introduction to Heritrix, an archival quality web crawler. In *International Web Archiving Workshop*, 2004.

[16] D. A. Moon. Object-oriented programming with flavors. In *OOPLSA*, pages 1–8. ACM, 1986.

[17] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[18] Y. Smaragdakis and D. S. Batory. Mixin-based programming in C++. In *GCSE*, pages 163–177. Springer, 2001.

[19] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., 1994.

[20] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *APSEC*, 2010.

[21] M. VanHilst and D. Notkin. Using role components in implement collaboration-based designs. In *OOPSLA*, pages 359–369. ACM, 1996.