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

$e \in E$	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 1. Syntax and notation for constructor matching

1. Formalization of Constructor Matching

1.1 Definitions and Notations

Figure 1 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 $v*$.

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 $\{x_1/v_1, ..., x_n/v_n\}$.

1.2 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 2 displays the inference rules of the constructor matching calculus. Rule *type* (1) describes the 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 as long as the labels of these parameters are mutually different $(S1)$. 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.

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

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

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

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

$$\begin{array}{l} (S1) : \forall i, j \in \{1, ..., n\} : i \neq j \Rightarrow v_i \neq v_j \text{ where } p_x = v_x : t_x \\ (S2) : \forall i \in \{k+1, ..., n\} : \forall \sigma'_1, \sigma'_2 \in V \rightharpoonup V^* : \\ \qquad \left(\begin{array}{l} ((a_1 \doteq p_1, ..., p_i), \sigma'_1) \notin \bar{\mathcal{L}} \vee \\ ((a_2, ..., a_m \doteq p_{i+1}, ..., p_n), \sigma'_2) \notin \bar{\mathcal{L}} \end{array} \right) \\ (S3) : \mathcal{D}(\sigma_1) \cap \mathcal{D}(\sigma_2) = \emptyset \end{array}$$

Figure 2. Rules of the Constructor Matching Calculus

1.3 Properties of the Matching Calculus

1.3.1 Theorems

The following two theorems lay down two key properties of the constructor matching calculus.

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.*

1.3.2 Proofs

Auxiliary Definitions. In the following, we assume $p_x = v_x : t_x$. Further, we define the application of a matching σ to an argument a as follows:

$$\begin{array}{rl} \sigma : & A \to E^* \\ a \mapsto & \begin{cases} e & \text{if } a = e \in E \\ e & \text{if } a = v : e \\ \sigma(v) & \text{if } a = v* \end{cases} \end{array}$$

Also, we define the application of a matching to an argument list recursively as follows:

$$\sigma(a_1, A_{\text{rest}}) = (\sigma(a_1), \sigma(A_{\text{rest}}))$$

The result must be interpreted as a non-nested tuple of expressions.

DEFINITION 1 (Validity of an argument list). *We first formalize what validity of an argument list means with respect to a formal parameter list: $a_1, \ldots, a_n$ is valid for $p_1, \ldots, p_k$ if and only if $n = k$ and*

$$\forall i \in \{1, \ldots, n\} : \exists e \in E : a_i = e \wedge \tau(e) <: t_i$$

In other words: The arguments a_i must be resolved expressions and not template arguments and they must have valid types. Note that $\tau(v_i)$ can be resolved to t_i in the context of a parameter list which contains $p_i = v_i : t_i$.

LEMMA 1 (Postponability of rule name). *If $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ is derivable, there is a derivation tree for it in which all applications of rule* name *are sequentially located at the end of the tree.*

Proof. The only rule which could interrupt *name* applications is *compos* because other rules do not have preconditions. Yet, it is always possible to move up or remove an application of the *compos* rule if the *name* rule produces one of its preconditions:

If the *name* rule produces the first precondition, then it is $a_1 = v : e$ and $p_1, \ldots, p_k = v : t$ for some variable name v and type t. The conclusion of the *compos* rule is thus $v : e, a_2, \ldots, a_m \doteq v : t, p_{k+1}, \ldots, p_n \mapsto \sigma_1 \cup \sigma_2$. We can obtain the same conclusion by eliminating the *compos* rule so that its former second precondition becomes the second precondition of the *name* rule. For application of the *name* rule, we choose $q = 1, k = 1$ and, thus, obtain the same conclusion as before with *compos*.

If the *name* rule produces the second precondition, we distinguish two cases: If, for the *name* rule application, it is $m = 1$, the *compos* application can be eliminated similarly to the above case, now with the first precondition of *compos* becoming the second precondition of *name* and choosing $q = 2$ and $k = k' + 1$ for the *name* application where k' is the value of k in the application of the *compos* rule. If m is not 1, we exchange the *compos* and the *name* application so that the *compos* rule produces the second precondition of the *name* rule. In the *compos* rule application, m and n will be decreased by 1 because the new named expression and the corresponding parameter are not yet contained in $a_2, \ldots, a_m$ and $p_{k+1}, \ldots, p_n$, respectively. In the *name* application, we increase q by 1 and k by k' (k' is the value of k in the former *compos* application, again) in order to account for the argument a_1 and the parameters $p_1, \ldots, p_k$ inserted by *compos*. This finally results in the same conclusion of the *name* rule that was previously achieved with the *compos* rule.

With these operations, we can move down applications of the *name* rule until all of them are sequentially located at the end of the derivation tree. As each single operation does not change the conclusion of the modified part of the tree, the derived statement $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ does not change, either.

LEMMA 2 (Eliminability of named expressions). *If $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ is derivable, then $a'_1, \ldots, a'_{n'} \doteq p'_1, \ldots, p'_{k'} \mapsto \sigma$ is derivable, too, where $a'_1, \ldots, a'_{n'}$ denotes the – equally ordered – subset of $a_1, \ldots, a_n$ which does not contain named expressions and $p'_1, \ldots, p'_{k'}$ denotes the subset of $p_1, \ldots, p_k$ from which all parameters with a correspondingly named expression in $a_1, \ldots, a_n$ have been removed.*

Proof. Due to Lemma 1, there is a derivation tree for $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ in which all applications of the *name* rule are at the end of the derivation. We can now, one after the other, remove the very last application, thereby removing the inserted named expression and the corresponding parameter. However, this does not change the matching σ because the *name* rule just copies it from the second precondition. After having removed all *name* applications, we have found the derivable statement $a'_1, \ldots, a'_{n'} \doteq p'_1, \ldots, p'_{k'} \mapsto \sigma$ as desired.

LEMMA 3 (Matching correctness without named expressions). *If $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ is derivable and there are no named expressions in $a_1, \ldots, a_n$, then $\sigma(a_1, \ldots, a_n)$ is a valid argument list for the formal parameters $p_1, \ldots, p_k$.*

We proof the lemma with induction over the derivation of $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$:

Induction hypothesis: For each premise $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ of the last rule application in the derivation, $\sigma(a_1, \ldots, a_n)$ is valid for $p_1, \ldots, p_k$.

Base step: The base step covers all derivations consisting of only one rule application.
type: It is $\sigma = \emptyset$; hence, $\sigma(e) = e$ and e is a valid argument list for $v : t$ because $\tau(e) <: t$ by the side condition of the rule.
empty: The empty argument list is trivially valid for the empty parameter list.
template: $\sigma(v) = (v_1, \ldots, v_n)$ is a valid argument list for $v_1 : t_1, \ldots, v_n : t_n$ because $\forall i \in \{1, \ldots, n\} : v_i$ is an expression and $\tau(v_i) = t_i <: t_i$.

Induction step: In the induction step, we show that the lemma is valid for the conclusion of the last rule application in a derivation consisting of at least two rule applications, thereby building on the induction hypothesis. We can exclude rule *name* because it produces a named expression in the conclusion which contradicts the assumption of the lemma. We show validity for rule *compos*. Application of $\sigma_1 \cup \sigma_2$ to the argument list yields:

$$\begin{aligned}(\sigma_1 \cup \sigma_2)(a_1, \ldots, a_m) &= ((\sigma_1 \cup \sigma_2)(a_1), \\ &\qquad (\sigma_1 \cup \sigma_2)(a_2, \ldots, a_m)) \\ &\overset{(*)}{=} (\sigma_1(a_1), \sigma_2(a_2, \ldots, a_m))\end{aligned}$$

This is a valid argument list for $p_1, \ldots, p_n$ because, by induction hypothesis, $\sigma_1(a_1)$ is valid for $p_1, \ldots, p_k$ and $\sigma_2(a_2, \ldots, a_m)$ is valid for $p_{k+1}, \ldots, p_n$. $(*)$ is valid because $\sigma_1(a_1) = (\sigma_1 \cup \sigma_2)(a_1)$ and $\sigma_2(a_2, \ldots, a_m) = (\sigma_1 \cup \sigma_2)(a_2, \ldots, a_m)$. This is a consequence of side condition (S3): If $a_1 = v*$, then σ_1 is defined for v and, thus, σ_2 is not defined for v. Unifying σ_1 with σ_2 does therefore not change the value which is assigned to v. Vice versa, the same is true of template argument names contained in $a_2, \ldots, a_m$.

Proof of Matching Correctness.
Let $a_1, \ldots, a_n \doteq p_1, \ldots, p_k \mapsto \sigma$ be derivable. If we remove all named expressions from $a_1, \ldots, a_n$, obtaining $a'_1, \ldots, a'_{n'}$, and all corresponding named parameters from $p_1, \ldots, p_k$, obtaining $\hat{P} := p'_1, \ldots, p'_{k'}$, we know from Lemma 2 that $a'_1, \ldots, a'_{n'} \doteq \hat{P} \mapsto \sigma$ is derivable.

As there are no named expressions in $a'_1, \ldots, a'_{n'}$, we can infer from Lemma 3 that $\hat{A} := \sigma(a'_1, \ldots, a'_{n'})$ is a valid argument list for $\hat{P}$.

Finally, we can invert the removal of named expressions and the corresponding parameters, thereby restoring $\hat{P}$ to the parameter list $p_1, \ldots, p_n$ and pertaining validity of $\hat{A}$ for $\hat{P}$. For that purpose, we add each removed parameter $v : t$ to $\hat{P}$ at the original position of $v : t$ in $p_1, ..., p_k$, and we insert the correspondingly named expression e at the same position into $\hat{A}$. This operation does not sacrifice validity of $\hat{A}$ with respect to $\hat{P}$ because we pertain the property of equal lengths of $\hat{A}$ and $\hat{P}$ and each single named expression argument is valid for the parallel parameter. The latter can be concluded from rule *name*, the only one which can have produced the named expression in the derivation: Its first premise requires $e \doteq v : t \mapsto \emptyset$ to be derivable which in turn ensures $\tau(e) <: t$ as required for validity.

After restoring $\hat{P}$ to $p_1, \ldots, p_k$, we have also found the desired $\hat{A}$ that is valid for $p_1, \ldots, p_k$, and $\hat{A}$ can be seen as produced as required by the theorem: σ was applied to $a'_1, \ldots, a'_{n'}$ – the subset of $a_1, \ldots, a_n$ which does not contain named expressions – and named expressions were aligned according to the formal parameters by re-inserting them at the positions of the corresponding parameters.

LEMMA 4 (Unique derivation without named expressions). *If $\mathcal{E} \mapsto \sigma$ is derivable and $\mathcal{E}$ does not contain named expressions, the order of rule applications in the derivation tree for $\mathcal{E} \mapsto \sigma$ is uniquely determined by $\mathcal{E}$.*

Proof. Let $\mathcal{E} \mapsto \sigma$ be derivable and $\mathcal{E}$ not contain named expressions. We show the uniqueness of rule applications with a case distinction over the possible forms of $\mathcal{E}$:

$e \doteq v : t$**:** Only rule *type* is applicable.

$\varepsilon \doteq \varepsilon$**:** Only rule *empty* is applicable.

$v* \doteq P$ **for some parameter list** P**:** Only rule *template* is applicable.

$a_1, \ldots, a_m \doteq P$ **with** $m > 1$ **for some parameter list** P**:** Only rule *compos* is applicable because we can exclude the possiblity of named expressions existing in $a_1, \ldots, a_m$.

In the first three cases, there are no preconditions; hence, the derivation tree must consist of only one rule application. In the last case, the preconditions are uniquely determined by the matching problem $\mathcal{E}$ and the side condition (S2) of the rule. The above case distinction can be applied to the preconditions with induction over the number of arguments in the matching problem instance yielding the statement that the rules to be applied and their order in the derivation tree are always uniquely determined by the instance of the matching problem.

Proof of Matching Uniqueness. Let $\mathcal{E} \mapsto \sigma$ be derivable. Then we know from Lemma 2, that $\mathcal{E}' \mapsto \sigma$ is derivable where $\mathcal{E}'$ is $\mathcal{E}$ after removing named expressions and corresponding parameters. From Lemma 4 we know that the rules and their order in the derivation tree of $\mathcal{E}' \mapsto \sigma$ are uniquely determined by $\mathcal{E}'$. We show with induction over the derivation of $\mathcal{E}' \mapsto \sigma$ that σ is uniquely determined by $\mathcal{E}'$ together with the order of rule applications and consequently by $\mathcal{E}$.

Induction hypothesis: For each premise $\hat{\mathcal{E}} \mapsto \hat{\sigma}$ of the last rule application in the derivation, $\hat{\sigma}$ is uniquely determined by $\hat{\mathcal{E}}$.

Base step: The base step covers all derivations consisting of only one rule application.
type, empty: The produced matching is constant ($\emptyset$) and thus uniquely determined trivially.
template: The produced matching is $\sigma(v) = (v_1, \ldots, v_n)$ and thus uniquely determined by the matching problem instance $v* \doteq v_1 : t_1, \ldots, v_n : t_n$.

Induction step: The induction step covers all rule applications with preconditions. As *name* does not need to be considered – because the matching problem $\mathcal{E}'$ does not contain named expressions –, uniqueness of σ has only to be shown for *compos*: The matching problem instances in the precondition are uniquely determined by the one in the conclusion and the side conditions. By induction hypothesis, we know that these in turn uniquely determine the matchings in the preconditions. Finally, the construction of the matching in the conclusion is uniquely determined as the union of the matchings in the precondition. Hence, the matching is uniquely determined by the matching problem in the conclusion.

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

Figure 3. Applying the Constructor Matching Calculus

1.4 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 3 gives an example for the application of the calculus to the following instance of the constructor matching problem:

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

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. Hence, 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*.