

# A Lambda Calculus With Forms

Markus Lumpe

Department of Computer Science  
Iowa State University  
Ames, IA 50011, USA  
lumpe@cs.iastate.edu

**Abstract.** The need to use position-dependent parameters often hampers the definition of flexible, extensible, and reusable abstractions for software composition. This observation has led us to explore the concept of *forms*, which are first-class extensible records and that, in combination with a small set of purely asymmetric operators, provide a core language to address this issue. One interesting application of forms is the definition of contractual specifications to ensure that a component can be safely combined with other components or deployed in a new context. In fact, contractual specifications explicitly and formally state *what* a component offers without entering into the details of *how*. In this paper, we develop a formal form-based framework for the definition of contractual specifications. More precisely, we study a substitution-free variant of the lambda-calculus, called  $\lambda\mathcal{F}$ , where names are replaced with forms and parameter passing is modeled using explicit contexts and show how the  $\lambda\mathcal{F}$ -calculus can be used to define syntactic contractual specifications.

## 1 Introduction

Modern software systems are constantly growing in complexity and size. Moreover, in order to timely adapt to changing requirements those systems have to be designed in a way that software evolution becomes feasible. The component-oriented software development approach targets exactly this aspect and has become the most promising software development technology today [19, 23, 28].

A successful component-based software development approach, however, not only needs to provide abstractions to represent different component models and composition techniques, but must also provide a systematic method for constructing large software systems [4, 16]. In particular, we need a specially-designed *composition language* that allows for building applications as compositions of reusable software components [24]. This language, which should be extensible, has to provide abstractions to instantiate, coordinate, and compose components that are generally developed in different implementation language. Furthermore, to guarantee flexible, reliable, and verifiable software composition, such a composition language has to be based on a suitable formal foundation [8, 14, 20, 24]. A precise semantics is essential if we are to deal with multiple architectural styles and component models within a common, unifying framework.

There are several plausible candidates (e.g.,  $\lambda$ -calculus,  $\pi$ -calculus, or variants of them) that can serve as a computational model for component-based software development. Unfortunately, they often hamper the definition of general purpose compositional abstractions, as they impose dependence on position and arity of parameters [9, 25]. For example, in the standard  $\lambda$ -calculus the functions  $\lambda(x, y).x$  and  $\lambda(y, x).y$  are equivalent, but  $\lambda(x, y).x$  and  $\lambda(y, x).x$  are different, as position matters in  $\lambda$ -calculus. Moreover, if we use de Bruijn indices [11], then names disappear totally, as arguments to functions are uniquely identified by their positions. Thus, if we abstract from position and use instead the parameter names as keys functions like  $\lambda\langle x, y \rangle.x$  and  $\lambda\langle y, x \rangle.x$  become indistinguishable.

This observation has led us to explore the concept of *forms* [13, 15, 24]. Forms are first-class extensible records that define mappings from labels to values, which, in combination with a small set of purely asymmetric operators, provide a core language to define extensible, flexible, and robust compositional abstractions. Programmatically, forms are both compile-time and run-time entities. As compile-time entities, forms can be used to denote components, component interfaces, and component composition. At run-time, on the other hand, forms provide uniform and language-neutral access to component services and support runtime composition on demand.

Originally, forms were an integral part of the  $\pi\mathcal{L}$ -calculus [13], a conservative extension of the  $\pi$ -calculus, that already provides better support for modeling concurrent component-oriented abstractions [13, 24]. However, forms are not bound to a particular computational model. They are an environment-independent framework that has to be combined with a concrete target system like the  $\lambda$ -calculus or the  $\pi$ -calculus.

In this paper, we study a substitution-free variant of the  $\lambda$ -calculus, called  $\lambda\mathcal{F}$ , where names are replaced with forms and parameter passing is modeled using explicit contexts. Explicit contexts mimic  $\lambda$ -calculus substitutions. However, unlike  $\lambda$ -calculus in which substitutions are meta-level operations [1], explicit contexts have a syntactic representation to record named parameter bindings.

The design of the  $\lambda\mathcal{F}$ -calculus is greatly influenced by Dami's  $\lambda N$ -calculus [9, 10] that is also a calculus in which parameters are identified by names rather than positions. However, there are two significant differences. First, in the  $\lambda N$ -calculus an application is split into two different parts: an expression  $\mathbf{a}(\mathbf{l} = \mathbf{b})$ , called *bind expression*, passes the value  $\mathbf{b}$  under the name  $\mathbf{l}$  to  $\mathbf{a}$ ; an expression  $\mathbf{a}!$ , called *close expression*, ends a sequence of bind expressions and forces the evaluation of  $\mathbf{a}$ . A shortcoming of this approach is that binding expressions cannot be pooled into an additional structure or used to encode monadic communication patterns, which occur in rendezvous-based protocols like HTTP. For example, the  $\lambda N$ -term  $\mathbf{a}(\mathbf{l} = \mathbf{b})(\mathbf{m} = \mathbf{c})!$  denotes an expression  $\mathbf{a}!$  in which the parameters  $\mathbf{l}$  and  $\mathbf{m}$  are bound to their corresponding values  $\mathbf{b}$  and  $\mathbf{c}$ . Moreover,  $\mathbf{a}(\mathbf{l} = \mathbf{b})(\mathbf{m} = \mathbf{c})$  actually stands for two distinct closures  $\mathbf{a}(\mathbf{l} = \mathbf{b})$  and  $((\mathbf{a}(\mathbf{l} = \mathbf{b}))(\mathbf{m} = \mathbf{c}))$ , each requiring a separate interaction with the environment. In  $\lambda\mathcal{F}$ , on the other hand, we write  $\mathbf{a} \langle \mathbf{l} = \mathbf{b} \rangle \langle \mathbf{m} = \mathbf{c} \rangle$  instead of  $\mathbf{a}(\mathbf{l} = \mathbf{b})(\mathbf{m} = \mathbf{c})!$ . Here, the form

$\langle \rangle \langle \mathbf{l} = \mathbf{b} \rangle \langle \mathbf{m} = \mathbf{c} \rangle$  is a structured argument that can be consumed in one interaction.

Secondly, rather than using an informal meta-level operation to substitute formal parameters with actual ones, we use *explicit contexts* [1, 3]. These explicit contexts have a syntactic representation based on forms. In practice, contexts explicitly record named parameter bindings and provide an environment to resolve the occurrences of free variables in a  $\lambda\mathcal{F}$ -term. For example, the term  $\mathbf{a}[\mathbf{b}]$  denotes an expression  $\mathbf{a}$ , which is evaluated with respect to the context  $[\mathbf{b}]$ . That is, all occurrences of free variables in  $\mathbf{a}$  are resolved using form  $\mathbf{b}$ . Thus, the context  $[\mathbf{b}]$  expresses the requirements posed by the free variables of  $\mathbf{a}$  on the environment [18]. In other words, if we *close* the component  $\mathbf{a}$  by composing it with a concrete environment or component  $\mathbf{b}$ , then  $\mathbf{a}[\mathbf{b}]$  denotes a composite component, where the services *provided* by  $\mathbf{b}$  are used to satisfy the required services of  $\mathbf{a}$ .

On the other hand, explicit contexts also allow for the definition of (syntactic) *contractual specification* [5]. Contractual specifications are used to ensure that a component can safely be combined with other components or deployed in a new context. Ideally, all conditions of a contract should be stated explicitly and formally as part of an interface specification [26]. The information contained in a contract should tell us *what* a component offers without entering into the details of *how* it is implemented. For example, let  $T, S$  be form-based type expressions. We write  $T[S]$  to express that we can close  $T$  by composing it with  $S$ . However, since this composition denotes a contractual specification, we must check now that the services *provided* by  $S$  satisfy the required type of  $T$ .

The remainder of this paper is organized as follows: In Section 2, we present the  $\lambda\mathcal{F}$ -calculus. We proceed by using  $\lambda\mathcal{F}$  to specify syntactic contracts in Section 3. We conclude with a summary of related and future work in Section 4.

## 2 The $\lambda\mathcal{F}$ -Calculus

The primary objective of the definition of the  $\lambda\mathcal{F}$ -calculus is to study the effect of replacing variables by forms in  $\lambda$ -calculus. So, we maintain a clear separation between the syntactic categories of *forms* and  $\lambda\mathcal{F}$ -*terms*. The linkage between expressions of both categories is modeled through a refined characterization of the set of values.

Next, the question arises how to handle best *substitution*? In the classical  $\lambda$ -calculus we write  $\mathbf{a}\{\mathbf{b}/\mathbf{x}\}$  to denote the term  $\mathbf{a}$  where all free occurrences of  $\mathbf{b}$  have been replaced with  $\mathbf{x}$ . However, substitution in  $\lambda$ -calculus is a very expensive term-rewriting operation, which actually does not belong to the calculus [1]. We address this issue by using *explicit contexts* [1, 3], which have a form-based syntactic representation. Explicit contexts are used for both forms and  $\lambda\mathcal{F}$ -terms and they provide, therefore, a uniform way to resolve occurrences of free variables.

We presuppose a countably infinite set,  $\mathcal{L}$ , of *labels*, and let  $l, m, n$  range over labels. We also presuppose a countably infinite set,  $\mathcal{V}$ , of *abstract values*, and let  $a, b, c$  range over abstract values. We think of an abstract value as a representation of any programming value like integers, objects, types, and even forms. However, we do not require any particular property except that equality and inequality be defined for elements of  $\mathcal{V}$ . We use  $F, G, H$  to range over the set of forms, and  $M, N$  to range over the set of  $\lambda\mathcal{F}$ -terms. The syntax of the  $\lambda\mathcal{F}$ -calculus is given in Figure 1.

$F, G, H ::= \langle \rangle$	<i>empty form</i>	$V ::= \mathcal{E}$	<i>empty value</i>
$X$	<i>form variable</i>	$a$	<i>abstract value</i>
$F \langle l = V \rangle$	<i>binding extension</i>	$M$	$\lambda\mathcal{F}$ – <i>value</i>
$F \cdot G$	<i>form extension</i>	$M, N ::= F$	<i>form</i>
$F \setminus G$	<i>form restriction</i>	$M.l$	<i>projection</i>
$F \rightarrow l$	<i>form dereference</i>	$\lambda(X) M$	<i>abstraction</i>
$F[G]$	<i>form context</i>	$M N$	<i>application</i>
		$M[F]$	$\lambda\mathcal{F}$ – <i>context</i>

**Fig. 1.** Syntax of the  $\lambda\mathcal{F}$ -Calculus.

Forms are used to denote both components and component composition. The services that a component offers are specified as *binding extensions*. A binding extension, written  $\langle l = \mathbf{s} \rangle$ , denotes a component's capability to perform a service  $\mathbf{s}$  that is published under the name  $l$ . For example, we write  $F.a$  to invoke the service that is bound by label  $a$  in component  $F$ .

The expressive power of forms is due to the two asymmetric operators: *form extension* and *form restriction*. Form extension, written  $F \cdot G$ , allows one to add or redefine a set of services simultaneously, whereas form restriction, written  $F \setminus G$ , can be seen as a dual operation that denotes a form, which is restricted to all bindings of  $F$  that do not occur in  $G$ . In combination, these operators provide the main building block in a fundamental concept for defining adaptable, extensible, and more robust software abstractions [13, 16, 24]. For example, suppose we want to compose two components  $F$  and  $G$ , but we want to give a specific service of  $F$  bound by label  $m$  precedence over a service bound by the same label  $m$  in  $G$ . This operation represents a *compositional style* [2] that defines a *conditional update*, which can be specified using both form extension and form restriction:  $F \cdot (G \setminus \langle m = F.m \rangle)$ . Depending on the actual services defined by the components  $F$  and  $G$ , we can distinguish three different situations covered by  $F \cdot (G \setminus \langle m = F.m \rangle)$ :

- If the label  $m$  occurs neither in  $F$  nor  $G$ , then the label  $m$  does not occur in the composition of  $F$  and  $G$ .

- If the label  $m$  does not occur in  $F$ , but in  $G$ , then  $G$ 's binding for label  $m$  occurs in the composition of  $F$  and  $G$ .
- If the label  $m$  occurs in  $F$ , then  $F$ 's binding for label  $m$  occurs in the composition of  $F$  and  $G$ .

Forms can also occur as values in binding extensions. These forms are called *nested forms* and they facilitate the specification of structured component interfaces. To extract a nested form bound by a label  $l$  in a form  $F$ , we use  $F \rightarrow l$ . Note, however, that if the binding involving label  $l$  does not actually map a nested form, then the result of  $F \rightarrow l$  is  $\langle \rangle$  – the *empty form*. The reason for this is that we want to distinguish between components, which offer a set of services, and component services themselves.

A *form context*  $F[G]$  denotes a closed form expression that is derived from  $F$  by using  $G$  as an environment to look up what would otherwise be free variables in  $F$ . We use form dereference to perform the lookup operation. That is, a free variable is reinterpreted as label. For example, if  $X$  is a free variable in  $F$  and  $[G]$  is a context, then the meaning of  $X$  in  $F$  is determined by the result of evaluating  $G \rightarrow X$ . In the case that  $G$  does not define a binding for  $X$ , the result is  $\langle \rangle$ , which effectively removes the set of provided services associated with  $X$  from  $F$ . This allows for an approach in which a sender and a receiver can communicate open form expressions. The receiver of this open form expression can use its local context to close (i.e., configure) the received form expression according to a site-specific protocol, but may also chose to ignore it (e.g., the configuration of a Web-browser to run an application associated with a specific MIME-type).

*Forms* and *projections* replace variables in  $\lambda\mathcal{F}$ . A form stands for an *explicit namespace* [3] or module [12], which can comprise an arbitrary number of provided services. The form itself can contain free variables, which will be resolved in the deployment environment or evaluation context. In other words, free variables in a form expression allow for a computational model with *late binding*.

With projections we recover variable references of  $\lambda$ -calculus. We require, however, that the subject of a projection denote a form. For example, the meaning of  $F.l$  is the value bound by label  $l$  in form  $F$ . A projection  $a.l$ , where  $a$  is not a form yields  $\mathcal{E}$ , which means “no value”.

Both *abstraction* and *application* correspond to the notions used in  $\lambda$ -calculus. As in  $\lambda$ -calculus, the  $X$  in  $\lambda(X) a$  stands for the parameter. But unlike  $\lambda$ -calculus, we do not use substitution to replace free occurrences of this name in the body of an abstraction. Parameter passing is modeled by explicit contexts.

A  $\lambda\mathcal{F}$ -context is the counterpart of a form-context. A  $\lambda\mathcal{F}$ -context denotes a lookup environment for free variables in a  $\lambda\mathcal{F}$ -term. Moreover,  $\lambda\mathcal{F}$ -contexts provide a convenient mechanism to retain the bindings of free variables in the body of a function. For example, let  $\lambda(X) a$  be a function and  $[F]$  be a creation context for it. Then we can use  $[F]$  to build a *closure* of  $\lambda(X) a$ . A closure is a package mechanism to record the bindings of free variables of a function at the time it was created. That is, the closure of  $\lambda(X) a$  is  $\lambda(X) (a[F])$ .

As a first example, consider the Church encoding of Booleans. In the standard  $\lambda$ -calculus Booleans are encoded using position-dependent parameters:

$$\begin{aligned}\text{True} &= \lambda\text{true}.\lambda\text{false}.\text{true} \\ \text{False} &= \lambda\text{true}.\lambda\text{false}.\text{false} \\ \text{Not} &= \lambda\text{arg}.\lambda\text{true}.\lambda\text{false}.\text{arg false true}\end{aligned}$$

These encodings have the desired property that the application  $(\text{Not True})$  yields  $\text{False}$ . But they lack extensibility, that is, the possibility to add new functionality without effecting the previous behavior of the encodings [9]. On the other hand, the encodings in  $\lambda\mathcal{F}$  are extensible, as we eliminate position dependency:

$$\begin{aligned}\text{True} &= \lambda(X) X.\text{true} \\ \text{False} &= \lambda(X) X.\text{false} \\ \text{Not} &= \lambda(B) \lambda(V) B V\langle\text{true} = V.\text{false}\rangle\langle\text{false} = V.\text{true}\rangle\end{aligned}$$

These encodings are equivalent to their  $\lambda$ -calculus counterparts. However, all functions are now characterized by the arguments they are effectively using and not by the ones they declare. The application  $(\text{Not True})$  yields:

$$\text{Not True} = \lambda(V) B V\langle\text{true} = V.\text{false}\rangle\langle\text{false} = V.\text{true}\rangle [\langle\rangle\langle B = \text{True}\rangle]$$

which is equivalent to  $\text{False}$ , as illustrated in the following:

$$\begin{aligned}\text{False} &\langle\rangle\langle\text{true} = a\rangle\langle\text{false} = b\rangle \\ &= X.\text{false} [\langle\rangle\langle X = \langle\rangle\langle\text{true} = a\rangle\langle\text{false} = b\rangle\rangle] \\ &= (\langle\rangle\langle\text{true} = a\rangle\langle\text{false} = b\rangle).\text{false} \\ &= b\end{aligned}$$

Similarly, it holds that

$$\begin{aligned}(\text{Not True}) &\langle\rangle\langle\text{true} = a\rangle\langle\text{false} = b\rangle \\ &= (\lambda(V) B V\langle\text{true} = V.\text{false}\rangle\langle\text{false} = V.\text{true}\rangle [\langle\rangle\langle B = \text{True}\rangle]) \\ &\quad \langle\rangle\langle\text{true} = a\rangle\langle\text{false} = b\rangle \\ &= (\text{True } V\langle\text{true} = V.\text{false}\rangle\langle\text{false} = V.\text{true}\rangle) [\langle\rangle\langle V = \langle\rangle\langle\text{true} = a\rangle\langle\text{false} = b\rangle\rangle] \\ &= \text{True } \langle\rangle\langle\text{true} = b\rangle\langle\text{false} = a\rangle \\ &= X.\text{true} [\langle\rangle\langle X = \langle\rangle\langle\text{true} = b\rangle\langle\text{false} = a\rangle\rangle] \\ &= (\langle\rangle\langle\text{true} = b\rangle\langle\text{false} = a\rangle).\text{true} \\ &= b\end{aligned}$$

We use denotational semantics to formalize the interpretation of forms and  $\lambda\mathcal{F}$ -terms. The underlying semantic model of forms is that of interacting systems [17]. Informally, the interpretation of forms (that is, their observable behavior) is defined by an evaluation function  $\llbracket \cdot \rrbracket^F$ , which guarantees that feature access is performed from right-to-left [15]. The reader should note, however, that in contrast to standard records, a given binding may not be observable<sup>1</sup> in a form and, therefore, may not be used to redefine or hide an existing one.

<sup>1</sup> A binding is not observable if it cannot be distinguished from  $\mathcal{E}$  or  $\langle\rangle$ . For example, the forms  $\langle\rangle\langle m = \mathcal{E}\rangle$ ,  $\langle\rangle\langle m = \langle\rangle\rangle$ , and  $\langle\rangle$  are all considered equivalent.

Form composition may yield form expressions in which many of their bindings have become inaccessible due to extension or restriction. Those bindings can be garbage collected. Garbage collecting inaccessible bindings of a form  $F$  yields a so-called *normalized form*  $\overline{F}$  containing solely observable binding extensions. In other words, form normalization yields an expression that is isomorphic to a classical record. However, we still maintain position independency. That is,  $\langle \rangle \langle 1 = \mathbf{a} \rangle \langle \mathbf{m} = \mathbf{b} \rangle$  and  $\langle \rangle \langle \mathbf{m} = \mathbf{b} \rangle \langle 1 = \mathbf{a} \rangle$  are behaviorally equivalent. Moreover, it holds that for every form  $F$  that there exists a normalized form  $\overline{F}$ , such that  $F$  is behaviorally equivalent to  $\overline{F}$ , written  $F \approx \overline{F}$ . We use  $\overline{F}, \overline{G}, \overline{H}$  to range over the set,  $\overline{\mathcal{F}}$ , of normalized forms, which is the smallest set that satisfies the specification given in Figure 2. There exists an algorithm, called **normalize**, that can generate a behaviorally equivalent normalized form  $\overline{F}$  for every given form  $F$  [15].

---


$$\overline{F} ::= \begin{cases} \langle \rangle \\ \langle \rangle \langle l_1 = v_1 \rangle \langle l_2 = v_2 \rangle \dots \langle l_n = v_n \rangle \quad n > 0 \end{cases}$$

$$\forall i, j \in \{1 \dots n\} \wedge i \neq j : l_i \neq l_j \quad \text{and} \quad \forall i \in \{1 \dots n\} : v_i \neq \mathcal{E} \wedge v_i \neq \langle \rangle$$


---

**Fig. 2.** Normalized Forms.

The meaning of a  $\lambda\mathcal{F}$ -term depends on its deployment context. A deployment context is represented by a normalized form. We write  $\llbracket \mathbf{a} \rrbracket^{LF} [\overline{H}]$  to evaluate the  $\lambda\mathcal{F}$ -expression  $\mathbf{a}$  in a deployment context  $\overline{H}$ . For example, we can use the encoding of Booleans as shown above to build a deployment context  $\overline{B}$ . This context defines three bindings: **True**, **False**, and **Not**:

$$\begin{aligned} \overline{B} = & \langle \rangle \langle \mathbf{True} = \lambda(X) X.\mathbf{true} \rangle \\ & \langle \mathbf{False} = \lambda(X) X.\mathbf{false} \rangle \\ & \langle \mathbf{Not} = \lambda(B) \lambda(V) B V \langle \mathbf{true} = V.\mathbf{false} \rangle \langle \mathbf{false} = V.\mathbf{true} \rangle \rangle \end{aligned}$$

We can use  $\overline{B}$  to evaluate  $(\mathbf{Not} \mathbf{True}) \langle \rangle \langle \mathbf{true} = \mathbf{a} \rangle \langle \mathbf{false} = \mathbf{b} \rangle$ , written

$$\llbracket (\mathbf{Not} \mathbf{True}) \langle \rangle \langle \mathbf{true} = \mathbf{a} \rangle \langle \mathbf{false} = \mathbf{b} \rangle \rrbracket^{LF} [\overline{B}],$$

which yields

$$\begin{aligned} & \mathbf{apply} (\llbracket (\mathbf{Not} \mathbf{True}) \rrbracket^{LF} [\overline{B}], \overline{B}) \langle \rangle \langle \mathbf{true} = \mathbf{a} \rangle \langle \mathbf{false} = \mathbf{b} \rangle \\ & = \mathbf{b}. \end{aligned}$$

The evaluation rules for forms and  $\lambda\mathcal{F}$ -terms are shown in Figure 3.

---

Form evaluation:

$$\begin{aligned}
\llbracket \langle \rangle \rrbracket^F [\bar{H}] &= \langle \rangle && \text{(F-EMPTY)} \\
\llbracket X \rrbracket^F [\bar{H}] &= \langle \langle \bar{H} \rightarrow X \rangle \rangle && \text{(F-VAR)} \\
\llbracket F(l = V) \rrbracket^F [\bar{H}] &= (\llbracket F \rrbracket^F [\bar{H}]) \langle l = \llbracket V \rrbracket^V [\bar{H}] \rangle && \text{(F-BIND)} \\
\llbracket F \cdot G \rrbracket^F [\bar{H}] &= (\llbracket F \rrbracket^F [\bar{H}]) \cdot (\llbracket G \rrbracket^F [\bar{H}]) && \text{(F-PBIND)} \\
\llbracket F \setminus G \rrbracket^F [\bar{H}] &= (\llbracket F \rrbracket^F [\bar{H}]) \setminus (\llbracket G \rrbracket^F [\bar{H}]) && \text{(F-PRES)} \\
\llbracket F \rightarrow l \rrbracket^F [\bar{H}] &= \langle \langle (\llbracket F \rrbracket^F [\bar{H}]) \rightarrow l \rangle \rangle && \text{(F-DEREF)} \\
\llbracket F[G] \rrbracket^F [\bar{H}] &= \llbracket F \rrbracket^F [\mathbf{normalize} ((\llbracket G \rrbracket^F [\bar{H}]) \cdot \bar{H})] && \text{(F-CONTEXT)}
\end{aligned}$$

Value evaluation:

$$\llbracket \mathcal{E} \rrbracket^V [\bar{H}] = \mathcal{E} \qquad \llbracket a \rrbracket^V [\bar{H}] = a \qquad \llbracket M \rrbracket^V [\bar{H}] = \llbracket M \rrbracket^{LF} [\bar{H}]$$

Form normalization:

$$\llbracket F \rrbracket^{\bar{F}} [\bar{H}] = \mathbf{normalize} (\llbracket F \rrbracket^F [\bar{H}]) \qquad \text{(F-NORM)}$$

$\lambda\mathcal{F}$ -evaluation:

$$\begin{aligned}
\llbracket F \rrbracket^{LF} [\bar{H}] &= \begin{cases} v & \text{if } F \equiv X \wedge v = \llbracket \bar{H}.X \rrbracket \neq \mathcal{E} \\ \llbracket F \rrbracket^{\bar{F}} [\bar{H}] & \text{otherwise} \end{cases} && \text{(LF-FORM)} \\
\llbracket M.l \rrbracket^{LF} [\bar{H}] &= \llbracket (\llbracket M \rrbracket^{LF} [\bar{H}]).l \rrbracket && \text{(LF-PROJ)} \\
\llbracket \lambda(X) M \rrbracket^{LF} [\bar{H}] &= \lambda(X) (\llbracket M \rrbracket^{LF} [\bar{H}]) && \text{(LF-ABS)} \\
\llbracket M N \rrbracket^{LF} [\bar{H}] &= \mathbf{apply} M N \bar{H} && \text{(LF-APP)} \\
\llbracket M[F] \rrbracket^{LF} [\bar{H}] &= \llbracket M \rrbracket^{LF} [\mathbf{normalize} ((\llbracket F \rrbracket^F [\bar{H}]) \cdot \bar{H})] && \text{(LF-CONTEXT)}
\end{aligned}$$

Projection evaluation ( $\hat{F} = \llbracket F \rrbracket^F [\bar{H}]$  and  $\hat{G} = \llbracket G \rrbracket^F [\bar{H}]$  for some  $\bar{H}$ ):

$$\begin{aligned}
&\left. \begin{aligned} &\llbracket (\langle \rangle).l \rrbracket, \llbracket a.l \rrbracket, \llbracket \mathcal{E}.l \rrbracket \\ &\llbracket (\lambda(X) (M[\bar{H}])).l \rrbracket \end{aligned} \right\} = \mathcal{E} \\
&\llbracket (\hat{F} \langle m = \hat{V} \rangle).l \rrbracket = \llbracket \hat{F}.l \rrbracket \quad \text{if } m \neq l \\
&\llbracket (\hat{F} \langle l = \hat{V} \rangle).l \rrbracket = \begin{cases} \hat{V} & \text{if } \hat{V} \in \mathcal{V} \\ \mathcal{E} & \text{otherwise} \end{cases} \\
&\llbracket (\hat{F} \cdot \hat{G}).l \rrbracket = \begin{cases} \llbracket \hat{G}.l \rrbracket & \text{if } \llbracket \hat{G}.l \rrbracket \neq \mathcal{E} \vee \langle \langle \hat{G} \rightarrow l \rangle \rangle \neq \langle \rangle \\ \llbracket \hat{F}.l \rrbracket & \text{otherwise} \end{cases} \\
&\llbracket (\hat{F} \setminus \hat{G}).l \rrbracket = \begin{cases} \mathcal{E} & \text{if } \llbracket \hat{G}.l \rrbracket \neq \mathcal{E} \vee \langle \langle \hat{G} \rightarrow l \rangle \rangle \neq \langle \rangle \\ \llbracket \hat{F}.l \rrbracket & \text{otherwise} \end{cases}
\end{aligned}$$

Form dereference evaluation ( $\hat{F} = \llbracket F \rrbracket^F [\bar{H}]$  and  $\hat{G} = \llbracket G \rrbracket^F [\bar{H}]$  for some  $\bar{H}$ ):

$$\begin{aligned}
\langle \langle \rangle \rightarrow l \rangle \rangle &= \langle \rangle \\
\langle \langle (\hat{F} \langle m = \hat{V} \rangle) \rightarrow l \rangle \rangle &= \langle \langle \hat{F} \rightarrow l \rangle \rangle \quad \text{if } m \neq l \\
\langle \langle (\hat{F} \langle l = \hat{V} \rangle) \rightarrow l \rangle \rangle &= \begin{cases} \langle \rangle & \text{if } \hat{V} \in \mathcal{V} \\ \mathbf{normalize} \hat{V} & \text{otherwise} \end{cases} \\
\langle \langle (\hat{F} \cdot \hat{G}) \rightarrow l \rangle \rangle &= \begin{cases} \langle \langle \hat{G} \rightarrow l \rangle \rangle & \text{if } \llbracket \hat{G}.l \rrbracket \neq \mathcal{E} \vee \langle \langle \hat{G} \rightarrow l \rangle \rangle \neq \langle \rangle \\ \langle \langle \hat{F} \rightarrow l \rangle \rangle & \text{otherwise} \end{cases} \\
\langle \langle (\hat{F} \setminus \hat{G}) \rightarrow l \rangle \rangle &= \begin{cases} \langle \rangle & \text{if } \llbracket \hat{G}.l \rrbracket \neq \mathcal{E} \vee \langle \langle \hat{G} \rightarrow l \rangle \rangle \neq \langle \rangle \\ \langle \langle \hat{F} \rightarrow l \rangle \rangle & \text{otherwise} \end{cases}
\end{aligned}$$


---

**Fig. 3.** Evaluation Rules.

The operator **apply** is the heart of the  $\lambda\mathcal{F}$ -evaluation process. It actually implements a *lazy evaluation* mechanism. The reason for this is that the first argument (i.e., the operator) may not yield a closure.

The  $\lambda\mathcal{F}$ -operator **apply** is defined as follows:

$$\begin{aligned}
\mathbf{apply} \ M \ N \ \overline{F} &= \\
\mathbf{cases} \ \llbracket M \rrbracket^{LF}[\overline{F}] \ \mathbf{of} & \\
\overline{G} &: \llbracket N \rrbracket^{LF}[\llbracket \overline{F} \cdot \overline{G} \rrbracket^{\overline{F}}] & \text{(c1)} \\
a &: a \ (\llbracket N \rrbracket^{LF}[\overline{F}]) & \text{(c2)} \\
\mathcal{E} &: \mathcal{E} & \text{(c3)} \\
\lambda(X) \ (M' \ \overline{H}) &: \llbracket M' \ \overline{H} \rrbracket^{LF}[\langle \langle X = \llbracket N \rrbracket^{LF}[\overline{F}] \rangle \rangle] & \text{(c4)} \\
\mathbf{end} &
\end{aligned}$$

The first rule states that if the operator, denoted by  $M$ , evaluates to a (normal) form expression  $\overline{G}$ , we evaluate the argument  $N$  in a new extended context  $\llbracket \overline{F} \cdot \overline{G} \rrbracket^{\overline{F}}$ , where  $\overline{G}$  defines a local refinement of  $\overline{F}$ . Consider, for example, the following evaluation

$$\begin{aligned}
&\langle \langle \mathbf{True} = \lambda(X) \ X.\mathbf{true} \rangle \rangle (\mathbf{True} \ \langle \langle \mathbf{true} = a \rangle \langle \mathbf{false} = b \rangle \rangle) \\
&= (\mathbf{True} \ \langle \langle \mathbf{true} = a \rangle \langle \mathbf{false} = b \rangle \rangle) [\langle \langle \mathbf{True} = \lambda(X) \ X.\mathbf{true} \rangle \rangle] \\
&= (\lambda(X) \ X.\mathbf{true}) \ (\langle \langle \mathbf{true} = a \rangle \langle \mathbf{false} = b \rangle \rangle) \\
&= (\langle \langle \mathbf{true} = a \rangle \langle \mathbf{false} = b \rangle \rangle).\mathbf{true} \\
&= a
\end{aligned}$$

Here, the form  $\langle \langle \mathbf{True} = \lambda(X) \ X.\mathbf{true} \rangle \rangle$  provides a proper environment that allows for the evaluation of the expression  $(\mathbf{True} \ \langle \langle \mathbf{true} = a \rangle \langle \mathbf{false} = b \rangle \rangle)$  in a meaningful way. This approach is similar to way the so-called sandbox expressions are handled in the PICCOLA-calculus [2]. When evaluating a sandbox expression  $A;B$  the term left to the semicolon defines a *root* context or *controlled environment* for the right-hand side agent. However,  $A$  in  $A;B$  may not evaluate to a form. In this case the agent  $A;B$  is *stuck* and identified with  $\mathcal{E}$ .

The second rule defines the evaluation of a system-depended expression, that is, when the operator  $M$  evaluates to an abstract value  $a$ . The actual meaning of  $a$  lies outside the  $\lambda\mathcal{F}$ -calculus. Therefore, the target system is responsible for the proper handling of the expression  $a \ (\llbracket N \rrbracket^{LF}[\overline{F}])$ . We have chosen this approach, rather than using  $\perp$  (i.e., undefined), because the meaning of  $a \ (\llbracket N \rrbracket^{LF}[\overline{F}])$  is not really undefined, but merely our knowledge about it is incomplete.

The third rule defines *error propagation*. If the operator (i.e.,  $M$ ) evaluates to “no value”, the whole expression has no value. It will simply be discarded.

The fourth rule states that if operator  $M$  evaluates in context  $\overline{F}$  to a closure  $\lambda(X) \ (M' \ \overline{H})$ , then the body of the closure (i.e.,  $(M' \ \overline{H})$ ) is being evaluated in a new context  $[\langle \langle X = \llbracket N \rrbracket^{LF}[\overline{F}] \rangle \rangle]$ . Thus, actual parameters are passed to a functions as bindings in the evaluation context.

To illustrate the  $\lambda\mathcal{F}$ -evaluation process, consider the following example. To simplify the presentation, we assume that both  $F_1$  and  $F_2$  do not contain any free variables:

$$\begin{aligned}
& \llbracket \mathbf{F}_1 (\lambda(\mathbf{X}) (\lambda(\mathbf{Y}) \mathbf{Y}) \mathbf{X}) \mathbf{F}_2 \rrbracket^{LF} [\langle \rangle] \\
= & \mathbf{apply} \ \mathbf{F}_1 \ ((\lambda(\mathbf{X}) (\lambda(\mathbf{Y}) \mathbf{Y}) \mathbf{X}) \mathbf{F}_2) \ \langle \rangle && \text{(LF-APP)} \\
= & \llbracket (\lambda(\mathbf{X}) (\lambda(\mathbf{Y}) \mathbf{Y}) \mathbf{X}) \mathbf{F}_2 \rrbracket^{LF} [\llbracket \langle \rangle \cdot \mathbf{F}_1 \rrbracket^{\overline{F}}] && \text{(C1)} \\
= & \llbracket (\lambda(\mathbf{X}) (\lambda(\mathbf{Y}) \mathbf{Y}) \mathbf{X}) \mathbf{F}_2 \rrbracket^{LF} [\mathbf{F}_1] && \text{(F-NORM)} \\
= & \mathbf{apply} \ (\lambda(\mathbf{X}) (\lambda(\mathbf{Y}) \mathbf{Y}) \mathbf{X}) \ \mathbf{F}_2 \ \mathbf{F}_1 && \text{(LF-APP)} \\
= & \llbracket (\lambda(\mathbf{Y}) \mathbf{Y}) \ \mathbf{X} \ [\mathbf{F}_1] \rrbracket^{LF} [\langle \rangle \langle \mathbf{X} = \llbracket \mathbf{F}_2 \rrbracket^{LF} [\mathbf{F}_1] \rangle] && \text{(C4)} \\
= & \llbracket (\lambda(\mathbf{Y}) \mathbf{Y}) \ \mathbf{X} \ [\mathbf{F}_1] \rrbracket^{LF} [\langle \rangle \langle \mathbf{X} = \mathbf{F}_2 \rangle] && \text{(LF-FORM)} \\
= & \llbracket (\lambda(\mathbf{Y}) \mathbf{Y}) \ \mathbf{X} \rrbracket^{LF} [\llbracket (\llbracket \mathbf{F}_1 \rrbracket^F [\langle \rangle \langle \mathbf{X} = \mathbf{F}_2 \rangle]) \cdot \langle \rangle \langle \mathbf{X} = \mathbf{F}_2 \rangle \rrbracket^{\overline{F}}] && \text{(LF-CONTEXT)} \\
= & \llbracket (\lambda(\mathbf{Y}) \mathbf{Y}) \ \mathbf{X} \rrbracket^{LF} [\llbracket \mathbf{F}_1 \cdot \langle \rangle \langle \mathbf{X} = \mathbf{F}_2 \rangle \rrbracket^{\overline{F}}] \\
= & \llbracket (\lambda(\mathbf{Y}) \mathbf{Y}) \ \mathbf{X} \rrbracket^{LF} [\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle] && \text{(F-NORM)} \\
= & \mathbf{apply} \ (\lambda(\mathbf{Y}) \mathbf{Y}) \ \mathbf{X} \ (\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle) && \text{(LF-APP)} \\
= & \llbracket \mathbf{Y} \ [\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle] \rrbracket^{LF} [\langle \rangle \langle \mathbf{Y} = (\llbracket \mathbf{X} \rrbracket^{LF} [\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle]) \rangle] && \text{(C4)} \\
= & \llbracket \mathbf{Y} \ [\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle] \rrbracket^{LF} [\langle \rangle \langle \mathbf{Y} = \mathbf{F}_2 \rangle] && \text{(LF-FORM)} \\
= & \llbracket \mathbf{Y} \rrbracket^{LF} [\llbracket (\llbracket \mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle \rrbracket^F [\langle \rangle \langle \mathbf{Y} = \mathbf{F}_2 \rangle]) \cdot \langle \rangle \langle \mathbf{Y} = \mathbf{F}_2 \rangle \rrbracket^{\overline{F}}] && \text{(LF-CONTEXT)} \\
= & \llbracket \mathbf{Y} \rrbracket^{LF} [\llbracket (\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle) \cdot \langle \rangle \langle \mathbf{Y} = \mathbf{F}_2 \rangle \rrbracket^{\overline{F}}] \\
= & \llbracket \mathbf{Y} \rrbracket^{LF} [(\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle) \langle \mathbf{Y} = \mathbf{F}_2 \rangle] && \text{(F-NORM)} \\
= & \llbracket \mathbf{Y} \rrbracket^{\overline{F}} [(\mathbf{F}_1 \langle \mathbf{X} = \mathbf{F}_2 \rangle) \langle \mathbf{Y} = \mathbf{F}_2 \rangle] && \text{(LF-FORM)} \\
= & \mathbf{F}_2 && \text{(F-NORM)}
\end{aligned}$$

The above example illustrates that keyword-based parameter-passing can effectively be modeled with form-based explicit contexts (e.g.,  $\llbracket \langle \rangle \langle \mathbf{X} = \mathbf{F}_2 \rangle \rrbracket$ ). Actual function arguments are encoded as bindings in the form that represents the current context.

In  $\lambda\mathcal{F}$ , forms take the role of  $\lambda$ -calculus variables. But is it possible to embed the  $\lambda$ -calculus in  $\lambda\mathcal{F}$  itself? Assume, for example, a closed  $\lambda$ -calculus term  $M$ . Then the embedding of  $M$  into  $\lambda\mathcal{F}$  is given by the translation  $\llbracket M \rrbracket$  as specified below:

$$\begin{aligned}
\llbracket x \rrbracket &= x.arg \\
\llbracket \lambda x. M \rrbracket &= \lambda(x) \llbracket M \rrbracket \\
\llbracket M N \rrbracket &= (\llbracket M \rrbracket \ \langle \rangle \langle arg = \llbracket N \rrbracket \rangle)
\end{aligned}$$

Here,  $\lambda$ -calculus variables are encoded as projections, which extract the actual value. The encoding of abstractions maps a position-dependent function to a position-independent functions, whereas the encoding of application builds a form expression for the  $\lambda$ -term in argument position.

Unfortunately, a simple translation of a  $\lambda$ -calculus term does not necessarily yield a position-independent  $\lambda\mathcal{F}$ -term. Suppose we want to specify a recursive function definition (e.g., the length function for lists) of the form  $\mathbf{f} = \lambda x. \{ \mathit{body} \ \mathit{containing} \ \mathbf{f} \}$ . That is, we want to write a function where the term on the right-hand side of the  $=$  uses the very function that we are defining. To solve this problem, we define a function  $\mathbf{g} = \lambda \mathbf{f}. \lambda x. \{ \mathit{body} \ \mathit{containing} \ \mathbf{f} \}$  and a function  $\mathbf{h} = (\mathbf{fix} \ \mathbf{g})$ , where  $\mathbf{fix}$  is the applicative-order fixed-point combinator

$$\mathbf{fix} = \lambda \mathbf{f}. ((\lambda x. \mathbf{f} \ (\lambda y. (x \ x) \ y)) \ (\lambda x. \mathbf{f} \ (\lambda y. (x \ x) \ y)))$$

Now, we can translate  $(\mathbf{fix} \ g)$  into  $\lambda\mathcal{F}$ , but the result would still be position-dependent, since the sub-term  $(\lambda x.f \ (\lambda y.(x \ x) \ y))$  forces a position-dependent order on  $g$ 's arguments. In particular,  $\llbracket g \rrbracket$  is a  $\lambda\mathcal{F}$ -function that has to be applied to a “self-replicator” (i.e., the fixed-point) before it can consume any additional arguments. Further analysis reveals that in order to achieve a position-independent encoding of  $(\mathbf{fix} \ g)$ , we need to be able to convert  $g$  into an equivalent “uncurried” function, which can consume the self-replicator and any additional arguments at the same time. But the application of  $g$  in term  $(\lambda x.f \ (\lambda y.(x \ x) \ y))\{f/g\}$  does not allow this. We solve this problem by applying  $f$  later. That is, we move the application of  $f$  underneath the innermost abstraction:  $\lambda x.\lambda y.(f \ (x \ x) \ y)$ . We observe now that  $f$  has become a function that takes two arguments. Replacing the sequence of arguments with a structured argument (i.e., uncurrying  $f$ ) yields the desired effect that the self-replicator and any additional arguments are consumed at the same time. We call a fixed-point operator with this property *late applicative-order fixed-point combinator*, which is defined as

$$\mathbf{fix}_L = \lambda f.((\lambda x.\lambda y.(f \ (x \ x) \ y)) \ (\lambda x.\lambda y.(f \ (x \ x) \ y)))$$

The encoding<sup>2</sup> of  $\mathbf{fix}_L$  into  $\lambda\mathcal{F}$  is as follows:

$$\begin{aligned} \llbracket \mathbf{fix}_L \rrbracket &= \lambda(f) \llbracket (\lambda x.\lambda y.(f \ (x \ x) \ y)) \ (\lambda x.\lambda y.(f \ (x \ x) \ y)) \rrbracket \\ &= \lambda(f) \llbracket (\lambda x.\lambda y.(f \ (x \ x) \ y)) \ \langle \rangle \langle \mathbf{arg} = \llbracket (\lambda x.\lambda y.(f \ (x \ x) \ y)) \rrbracket \rangle \rrbracket \\ &= \lambda(f) \ (H \ \langle \rangle \langle \mathbf{arg} = H \rangle) \\ &\quad \text{where } H = \lambda(x) \ \lambda(y) \ (f.\mathbf{arg} \ \langle \rangle \langle \mathbf{arg} = (x.\mathbf{arg} \ \langle \rangle \langle \mathbf{arg} = x.\mathbf{arg} \rangle) \rangle) \\ &\quad \quad \langle \rangle \langle \mathbf{arg} = y.\mathbf{arg} \rangle) \end{aligned}$$

By further analyzing the expression  $H$ , we notice that a form expression of the kind  $\langle \rangle \langle \mathbf{arg} = X.\mathbf{arg} \rangle$  is the same as  $X$ . Thus, we can rewrite  $H$  as follows:

$$H = \lambda(x) \ \lambda(y) \ (f.\mathbf{arg} \ \langle \rangle \langle \mathbf{arg} = (x.\mathbf{arg} \ x) \rangle) \ y)$$

In the next step, we use the fact that  $f.\mathbf{arg}$  actually has to denote a position-independent function, which can consume all arguments in any order at once:

$$H = \lambda(x) \ \lambda(y) \ (f.\mathbf{arg} \ y \langle \mathbf{arg} = (x.\mathbf{arg} \ x) \rangle)$$

Thus, a position-independent applicative-order fixed-point combinator in  $\lambda\mathcal{F}$ , written  $\mathbf{FIX}$ , can be defined as follows:

$$\begin{aligned} \mathbf{FIX} &= \lambda(\mathbf{Fun}) \ (H \ \langle \rangle \langle \mathbf{self} = H \rangle) \\ &\quad [\langle \rangle \langle H = \lambda(\mathbf{Fix}) \ \lambda(\mathbf{Args}) \ (\mathbf{Fun}.f \ \mathbf{Args} \langle \mathbf{self} = (\mathbf{Fix}.\mathbf{self} \ \mathbf{Fix}) \rangle) \rangle] \end{aligned}$$

The ability to define the fixed-point operator  $\mathbf{FIX}$  in  $\lambda\mathcal{F}$  suggests that it should be possible to embed arbitrary  $\lambda$ -terms in the  $\lambda\mathcal{F}$ -calculus. However, a simple translation of a  $\lambda$ -term into a  $\lambda\mathcal{F}$ -term does not guarantee a position-independent result. A more detailed study of embedding  $\lambda$ -calculus into  $\lambda\mathcal{F}$  is part of future work.

<sup>2</sup> In the encoding of  $\mathbf{fix}_L$ , the term  $H$  is still position-dependent, but the application of the recursive function, denoted by  $\mathbf{Fun}.f$ , is not.

### 3 Contractual Specifications

The contractual specification of component interfaces has to guarantee that a component can be safely combined with other components or deployed in a new context. Ideally, all conditions of a contract should be stated explicitly and formally as part of an interface specification. The information contained in a contract should tell us *what* a component offers without entering into the details of *how* it is provided. Beugnard et al [5] have identified four levels of component contracts:

- Syntactic contracts to specify data type compatibility,
- Behavioral contracts to specify pre- and postconditions invariants,
- Synchronization contracts to specify constraints in concurrent contexts, and
- Quality-of-service contracts to specify quantitative properties like maximum response time.

Each of these four levels is important, but in this paper we focus only on syntactic contracts.

Consider, for example, the following code written in C#-like language:

```
using System;
public class OneMinute : MarshalByRefObject {
    public override ILease InitializeLifetimeService() {
        ILease lease = base.InitializeLifetimeService();
        lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
        return lease;
    }
}
```

This code defines a class *OneMinute*, which can be used to instantiate .NET Remoting [21] objects with a lease time of one minute. *OneMinute* is derived from *MarshalByRefObject*, a .NET class that enables access to objects across application domain boundaries. The .NET Remoting infrastructure is an abstract approach to support interprocess communication. The .NET lifetime service associates a lease with each remotely activated object and after the lease expires, the object is removed from the system. The system assigns a default lifetime to each Remoting object, but the user can redefine the default by overriding the method `InitializeLifetimeService`. In the case of *OneMinute*, the lease will expire after one minute.

Now, we can represent the class *OneMinute* as a  $\lambda\mathcal{F}$ -term using the approach of Lumpe and Schneider [16]. The behavior of the class *OneMinute* can be captured by  $\Delta_{OneMinute}$  denoting the incremental modification defined by this class:

$$\begin{aligned} \Delta_{OneMinute} = & \\ & \lambda(I)\langle \langle \text{InitializeLifetimeService} = \\ & \quad \lambda() ((I \rightarrow \text{super}).\text{InitializeLifetimeService} \langle \rangle) \\ & \quad \langle \text{InitialLeaseTime} = (\text{TimeSpan.FromMinutes} (\langle \rangle \langle \text{value} = 1 \rangle)) \rangle \rangle \end{aligned}$$

The term  $(I \rightarrow \text{super}).\text{InitializeLifetimeService} \langle \rangle$  implements the call to the inherited method `InitializeLifetimeService`, which returns a `lease` form. The binding `InitialLeaseTime` is then overridden to set the lease time to one minute. To create the class `OneMinute`, we use the abstraction `Class` that when applied to an appropriate model generator for C# yields a class builder for C#-classes. We use `CSharpClass` to denote this class builder and apply it to both the super class `MarshalByRefObject` and the incremental modification  $\Delta_{OneMinute}$  to construct `OneMinute`:

$$OneMinute = \text{CSharpClass} (\text{MarshalByRefObject} \langle \Delta = \Delta_{OneMinute} \rangle)$$

Suppose now that we would like to verify the correctness of this definition without excessive dependencies on the features imported from the `System` namespace. Based on a static analysis of this code, we might express what it *provides* as the following form, where values are type expressions:

$$P = \langle \rangle \langle \text{OneMinute} = () \rightarrow \langle \rangle \langle \text{InitializeLifetimeService} = () \rightarrow ILease \rangle \rangle$$

that is, a default constructor called `OneMinute`, which yields an instance of `OneMinute` that understands at least the `InitializeLifetimeService` method, which returns a lease object that implements the interface `ILease`. We can, however, say even more about the assumptions this definition places on its environment. In particular, instances of `OneMinute` will safely provide their services if and only if the environment (i.e., the namespace `System`) satisfies the following requirement:

$$\begin{aligned} \bar{R} = & \langle \rangle \langle \text{MarshalByRefObject} = () \rightarrow \langle \rangle \langle \text{InitializeLifetimeService} = () \rightarrow ILease \rangle \\ & \langle ILease = \langle \rangle \langle \text{InitialLeaseTime} = \text{TimeSpan} \rangle \rangle \\ & \langle \text{TimeSpan} = \langle \rangle \langle \text{FromMinutes} = (\langle \rangle \langle \text{value} = \text{Integer} \rangle) \rightarrow \text{TimeSpan} \rangle \rangle \end{aligned}$$

that is, the environment `System`, denoted by  $\bar{R}$ , has to provide suitable definitions for the types `MarshalByRefObject`, `ILease`, and `TimeSpan`. In particular, the class `MarshalByRefObject` must define a default constructor and has to provide a method `InitializeLifetimeService`, which returns a value that implements the `ILease` interface. Furthermore, it is required that the `ILease` interface must define a virtual field `InitialLeaseTime` that can be assigned a `TimeSpan` structure, which has at least a suitable `FromMinutes` method.

Thus, we can say that `System` denotes the required type, which expresses the requirements posed by the free variables of class `OneMinute`. If we close `OneMinute` by composing it with an environment or component like `System`, written  $P[\bar{R}]$ , we must then check whether the services provided by `System` satisfy the required type of `OneMinute`.

## 4 Conclusion and Future Work

In this paper, we have presented the  $\lambda\mathcal{F}$ -calculus, a substitution-free variant of the  $\lambda$ -calculus in which names are replaced with forms and parameter passing is modeled using explicit contexts. The  $\lambda\mathcal{F}$ -calculus, like Dami's  $\lambda N$ , is a calculus in which parameters are identified by names rather than positions. Position-independent parameter specification allows for the development of extensible, flexible, and reliable component-based application. The resulting flexibility of a form-based programming model can also be seen, for example, in XML/HTML forms [30], where fields are encoded as named (rather than positional) parameters, in Python [29] and Common Lisp [27], where functions can be defined to take arguments by keywords, and in Perl [31] where it is a common technique to pass a map of name/value pairs as arguments to a function or method.

The definition of explicit contexts is inspired by the work of Achermann's PICCOLA-calculus [2] and the work on *explicit substitutions* by Abadi et al [1]. Substitution, as used in the classical  $\lambda$ -calculus, is actually a meta-level concept and not part of the language. By making it part of the language, Abadi et al argue that we can achieve a better correspondence between the language theory and its implementation.

However, Abadi et al do not address the problem of position dependency. In fact, explicit substitutions use de Bruijn indices [11] to correctly map parameters to their actual values. The resulting operational semantics of substitutions is not trivial and makes it cumbersome to trace the actual effects of them. On the other hand, forms provide a convenient way to record parameter bindings in a small and expressive framework.

We can use the  $\lambda\mathcal{F}$ -calculus to define (syntactic) contractual specifications. Contractual specifications raise explicitly the confidence level in the development of applications involving third-party components. The fundamental purpose of a contractual specifications is to prevent the occurrence of *run-time errors* while executing a component-based program, that is, contractual specifications should impose a well-balanced set of constraints to enforce the correctness of a component-based application. However, the verification process of contractual specifications must be defined in a way such that the programmer can easily predict whether a contract is satisfiable or not [6]. That is, a contract should be defined in a manner that the reasons why the verification of it has failed are self-evident.

The key challenge of the design of contractual specifications will be the proper characterization of incomplete system knowledge and the definition of the verification rules for both form extension and form restriction. The form extension operator is similar to *asymmetric record concatenation* [7, 22, 32] and takes two forms and returns a new form in which the bindings of the argument forms are merged. The specific nature of this operation makes it hard to find a suitable type assignment. Some type systems [7, 32] cannot assign a type to this operator at all. In type systems that incorporate a *subsumption rule* the form extension

operator requires an additional set of constraints that limits the number of applicable subtypes. Early results from recent work in this area [13, 18] indicate that the definition of a suitable component type system will go beyond “traditional” type theories.

**Acknowledgement.** The author thanks Jean-Guy Schneider, Oscar Nierstrasz, and the anonymous reviewers for their valuable comments and discussions.

## References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit Substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
2. Franz Achermann. *Forms, Agents and Channels: Defining Composition Abstraction with Style*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 2002.
3. Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, LNCS 1897, pages 77–89. Springer, September 2000.
4. Uwe Assmann. *Invasive Software Composition*. Springer, 2003.
5. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.
6. Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.
7. Luca Cardelli and John C. Mitchell. Operations on Records. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994. Also appeared as SRC Research Report 48, and in *Mathematical Structures in Computer Science*, 1(1):3–48, March 1991.
8. Francisco Curbera, Sanjiva Weerawarana, and Matthew J. Duftler. On Component Composition Languages. Proceedings of ECOOP 2000 Workshop on Component-Oriented Programming, June 2000.
9. Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d’Informatique, University of Geneva, CH, 1994.
10. Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, February 1998.
11. Nikolas G. de Bruijn. Lambda Calculus Notation with Nameless Dummies. *Indagationes Mathematicae*, 34:381–392, 1972.
12. Frank DeRemer and Hans H. Kron. Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.
13. Markus Lumpe. *A  $\pi$ -Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.
14. Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 4, pages 69–90. Cambridge University Press, March 2000.

15. Markus Lumpe and Jean-Guy Schneider. Form-based Software Composition. In Mike Barnett, Steve Edwards, Dimitra Giannakopoulou, and Gary T. Leavens, editors, *Proceedings of ESEC '03 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '03)*, pages 58–65, Helsinki, Finland, September 2003.
16. Markus Lumpe and Jean-Guy Schneider. A Form-based Metamodel for Software Composition. *Science of Computer Programming*, 56:59–78, April 2005.
17. Robin Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press, 1999.
18. Oscar Nierstrasz and Franz Achermann. A Calculus for Modeling Software Components. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Proceedings of First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, pages 339–360, Leiden, The Netherlands, 2003. Springer.
19. Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.
20. Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.
21. Ingo Rammer. *Advanced .NET Remoting*. APress, 2002.
22. Didier Rémy. Typing Record Concatenation for Free. Technical Report RR-1739, INRIA Rocquencourt, August 1992.
23. Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
24. Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.
25. Jean-Guy Schneider and Markus Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Languages et Modèles à Objets '97*, pages 61–76, Roscoff, October 1997. Hermes.
26. João Costa Seco and Luís Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, *Proceedings of ECOOP 2000*, LNCS 1850, pages 108–128. Springer, 2000.
27. Guy L. Steele. *Common Lisp the Language*. Digital Press, Thinking Machines, Inc., 2nd edition, 1990.
28. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002.
29. Guido van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.
30. W3C. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation, February 2004. <http://www.w3.org/TR/REC-xml>.
31. Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, September 1996.
32. Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93:1–15, 1991. Preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), pp. 92–97.