A Formal Framework for a Functional Language with Adaptable Components
Pascal Coupey, Christophe Fouqueré

To cite this version:
Pascal Coupey, Christophe Fouqueré. A Formal Framework for a Functional Language with Adaptable Components. 2010. hal-00530860

HAL Id: hal-00530860
https://hal.science/hal-00530860
Preprint submitted on 30 Oct 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A Formal Framework for a Functional Language with Adaptable Components

P. Coupey  C. Fouqueré
LIPN – UMR7030
CNRS – Université Paris 13
99 av. J-B Clément, F–93430 Villetaneuse, France
firstname.lastname@lipn.univ-paris13.fr

Abstract
We propose a component programming language called FLAC, Functional Language for Adaptable Components, on top of a functional programming language which authorizes full adaptability of components while ensuring type safety. The language is given together with a type system that offers a complete static type checking of any programs (including adaptations) to ensure error-free run-time adaptations. Dynamic adaptability and static type checking might seem at first sight paradoxical, but our approach allows it because, first, we use a single language for traditional services and control services (i.e., services for adaptations), and secondly, a specific merge operation takes care of adaptations.1

1. Introduction
Component-based programming is like a construction game: programs are assembled out of black boxes called components. The term ‘black box’ is justified by the fact that only the interfaces of components are publicly available. Such an interface characterizes the signatures of services the component offers or needs. The basic construct plug assembles a component that provides a service to a component that needs it. The domain of component-based systems (CBS) is mature enough to actually endorse the benefits of such a paradigm [12]. Safety and adaptability are the two main requirements for such systems. Safety is ensured when execution is error-free. For that purpose, most CBSs expect each required service used by a component to be satisfied by a service provided by another component (soundness) and any service offered by a component to be concretely defined (completeness) [7, 9]. Obviously, type checking must be done accordingly. Adaptable capabilities are required to fit the evolution of the environment, software or hardware. Hence it is often given as an important goal for CBSs [3]. It may be minimal as in works of Aldrich [1] or Sreedhar [11] or first works of Costa Seco et al. [8] or more developed as in works of Dowling et al. [5] or Batista et al. [2], Bruneton et al. [4] or last works of Costa Seco et al. [9, 10]. A CBS is dynamic if adaptations can be made at runtime as reactions for contextual events. If adaptations are made by an external actor (e.g., a programmer) via a declarative or procedural interface, adaptations are called external although they are called internal when the code and the prerequisites are described in the program itself. A CBS is called closed when all adaptations are internal. Current dynamic closed CBSs provide a meta-model by incorporating a specific adaptation language different from the language allowing component description and assembly. They also have mechanisms ensuring integrity and consistency of the system during dynamic reconfiguration. The metamodel manages a configuration graph of components and connections that can be inspected and modified at runtime. For example in the Fractal framework [4], component controllers allow to predict component changes at runtime (deleting or replacing a component, adding/removing functionalities, changing links with other components, . . . ) and dynamic adaptation is achieved by introspecting and reconfiguring the internal structure of components. The implementation of adaptability is frequently done by modifying the byte-code at runtime. However, in most existing CBSs, adaptability is limited if type checking is in use: the type of the new component should be a subtype of the type of the old component. The type of a component is roughly defined in current CBSs as the set of the signatures of its provided and requested services, and sub-typing is covariant with provided services and contravariant with requested services. Furthermore usual target languages, e.g., JAVA [1, 4], are not as strongly typed as one can expect as type conversion is allowed. Note also that types are not automatically inferred in these languages. Thus, if the language is not constrained, the programmer could easily override typing constraints, giving rise to unexpected results or software crashes. Costa Seco et al. [9, 10] propose a component-oriented programming language that authorizes adaptations of component objects thanks to configurators, i.e., reconfiguration scripts that describe architectural operations modifying the structure of a component. Type control is ensured by configurator type declaration using a specific language. A configurator type describes the pre and post conditions for the application of the configurator. However, first the architectural operations in the configurator description is outside the static context of the components on which they are applied, hence they are white-box operations. Consequently, the type of a configurator cannot be automatically inferred: a configurator definition requires an explicit type. Second, precondition type mismatches are not visible to the type system: the programmer has to prevent them by means of conditional structures. To summarize, two drawbacks prevent a plain use of CBSs:

- Adaptable capabilities are constrained by the way the component type is defined: adaptations cannot alter the signature of required or provided services. However, the environment may simultaneously evolve in such a way that the final system is still correctly typed.
- Casting and byte-code modification forbid static analysis of a code, hence safety of program execution.

The language FLAC, Functional Language for Adaptable Components, presented in this paper allows dynamic internal adaptations without the two previous drawbacks: dynamic adaptations
may remove or add requested or provided services, while types are statically checked:

1. Adaptation constructs are first-class entities: there is no metalinguage. Hence, type specification integrates adaptations declared in the code: the type of a component is not only given by the actual signatures of requested and provided services but also by those resulting from its adaptations. To the best of our knowledge it is the first proposal where the type of a component integrates adaptation services. It means that in our language, the type of a component without adaptation capabilities is not identical to the "same" component with such adaptation properties. Indeed, how a simple car toy can be considered as having the same type than a car toy which can become a robot thanks to an adaptation property? Consequently, type checking may be completely done statically.

2. Executing an adaptation creates a new component in the memory, whereas current approaches change bytecodes in place. Hence components have only one type for the whole run of the program. A garbage collector mechanism, concretely the one given with OCaml [6], allows for freeing unused components.

3. The language is strongly-typed (as well as OCaml) ensuring error-free run-time adaptations.

In the next section we present the language FLAC. We focus on adaptations capabilities with various examples. Its operational semantics is described in section 3 and we give in section 4 a typing system that ensures safety.

2. The FLAC Language

We extend a functional programming language\(^3\) mainly by means of a Component data structure that integrates an interface (requested and provided services) and a functional part intended to define service codes (see Fig. 1). A Service name is a string. Return types of service expressions are automatically inferred. Component parameters may be any kind of expressions, including services as well as components. Services are referenced by Uniform Resource Services (URS). A URS is identified by a component followed by a service name, optionally with a signature. The syntax for expressions is augmented to take care of these new structures: call, plug and merge operations are sufficient to illustrate adaptation capability.

In Fig. 2, two components are declared: a database Server offers a service \(u\) dedicated to process database requests, a Client asks for a database service \(p\). The program creates a client \(c_1\) and a server \(s_1\) and plugs them together. The plug expression creates a call indirection from \(p\) in \(c_1\) to \(u\) in \(s_1\). Note that such a plug sends back a new component, i.e., it does not modify in place the component \(c_1\).

Adaptations in FLAC are given by control services, i.e., services that add/delete services, hence create new components. In figures, control services appear on top of components. Control services contribute to the definition of components without reference to a meta-language: not only they could be required or provided, but also they are undistinguishable from standard services. In fact, control services are those that return a component, contrarily to services that return basic data. The type system of FLAC is able to infer and check control services types. A call to a control service followed by a merge operation constructs a new component adapted from the old one. Two modes are proposed in expression (merge\(\#c_1\ c_2\ e_1\ e_2\)) add returns a new component by adding the contents of \(e_2\) to \(c_1\) while sub removes the contents of \(e_2\) from \(c_1\).

\(^3\)OCaml [6] is used as a core functional language.
trated when computing $s_3$. In the same way, it is obvious to add or to remove a requested service. For example, our server could evolve toward a securized server which needs a service $k$ to encrypt or decrypt a string. To do this, simply add a control service which returns a component including a requested encryption/decryption service $k$.

Example in Fig. 4 illustrates a succession of adaptations from a simple client-server couple plugged together to a securized client-server couple, that are plugged via both simple and administration database request services. We add to previous figures component $\text{Client}2$, a database client that can evolve toward one that needs an encryption/decryption service and/or one that requires an administration service $p$, and component $\text{Server}2$, that contains a control service $v$ able to transform it to a securized server that needs an encryption/decryption service. Component $\text{Crypt}$ proposes a service $v$ for encrypting/decrypting strings. Note the difference between the way clients $c_3$ and $c_7$ are created: client $c_3$ shares the same encryption/decryption protocol with server $s_3$ (hence also $s_4$), whereas client $c_7$ is allowed to call the service a declared in server $s_4$.

Next two sections are devoted to the formal aspects of FLAC. Next section presents an operational semantics that gives the meaning of any valid program in FLAC while section 4 highlights the type system.

3. Operational Semantics

The operational semantics follow standard functional programming operational semantics: it is given as an evaluation judgment on programs and expressions to be computed with respect to a given environment. An environment is an evaluation environment together with a handler environment. An evaluation environment is a partial function from the set of variable names and component locations to values, either ground values or handlers to such values (supposing a domain of handlers). The evaluation environment has a special variable name '_self' whose value is a handler, it is supposed to be the handler of the component defined in the current context. A handler environment is a partial function from the set of handlers to values. Handlers are used to denote component values. The evaluation judgment for expressions is of the following form:

$$ E, H \vdash e \downarrow v, H' $$

read as: the expression evaluation $e$ in an evaluation environment $E$ with a handler environment $H$ leads to a value $v$ together with a handler environment $H'$. A handler value is given as a function over the domain \{req, prv, serv\}. Values for req and prv selectors are sets of requested or provided services, i.e., signatures of services. The value of serv selector is a map from service names to functional closures. We do not present the operational semantics of the functional part of the language. We extend a domain of basic (functional and data) values by the following kinds of values:

- $h$, value of a component, i.e., a handler value.
- $(h, s, t)$, value of an URS where $h$ is the value of a component, $s$ is a service name, i.e., a string, $t$ is a type (for discriminating services with the same name, it may be empty).

The operational semantics of Components is given in Fig. 5. The semantics corresponding to a Component declaration is straightforwardly given by lists of requested or provided services, and a closure similar to the treatment of functions: the rule is nothing else but a new value given for the reference. The lists of requested or provided services may be used if one extends the language by expressions asking for the interface of components. The semantics of plugging a service is similar to

$$ \text{Component} \quad \text{Services} \quad \{ s_1(p) \Rightarrow (\text{call} c_2?p); \} \quad \text{Cend} $$

except that '_self' should refer to the component given in the first argument. The operation noted $\downarrow^e$, resp. $\downarrow^c$, adds (resp. deletes) from its first argument the second argument if present. The semantics of an URS follows its syntactical structure. Rules are given in Fig. 6. A call to a service stipulates a service name and possibly a type, and as usual parameter values if needed.

There are various ways of merging components in FLAC. This gives the user full control over the model of components to have at the end and full modularity with respect to specification. Informally, two merge modes are given here:
4. Typing System

One adds, to a standard language of functional types, types \( t_c \) for component expressions. Such a type \( t_c \) is a triple of sequences of service types. A service type is a mapping from a service name to a type of the service code. The type of the service code may not be given: this case corresponds to a requested service. Otherwise it is a mapping from a pattern type \( t_p \) to a type. The system we present below is a type checker, however it requires only standard methods to derive from it a type inference system. The type environment is a partial function from a set of variable names and a special name self to types. Typing judgments are of one of the following forms:\(^4\)

- \( \Delta \vdash e : t \) where \( e \) is an expression
- \( \Delta \vdash s : e \) where \( e \) is a service name (string)
- \( \Delta \vdash \text{usr} : (s, t_p) \) where \( e \) is an URS
- \( \Delta \vdash \text{iserve} : e : t \) where \( e \) is a list of services
- \( \Delta \vdash p : t_p \) where \( p \) is a pattern

As components are not modified in-place, there is no specific difficulties in the typing system as soon as one considers that a component is a partially defined object. Hence its expected services are marked as partially defined function types, and a plug operation completes accordingly the type of the provided service in the component. Conversely, an unplug operation partially undefines part of the component type. Finally, as in the operational semantics, in a call operation, the special name self is defined to be the type of the first argument to be reused during the typing of the second argument of the call operation if the component is omitted. Note that this special name self is not part of the (user) type language.

Types for service names and URS follow the data structure (in URS, \( t_p \) is empty if \( t \) is not given):

\[
\begin{align*}
\text{s string} & \quad t_c = (s, t_p) \quad \Delta \vdash s : e \quad \Delta \vdash e : t \\
\Delta \vdash s : s & \quad \Delta \vdash e : t \\
\Delta \vdash \text{usr} : (s, t_p) & \quad \Delta \vdash \text{iserve} : e : t \\
\Delta \vdash \text{iserve} : e : t & \quad \Delta \vdash \text{iserve} : e : t \\
\Delta \vdash p : t_p & \quad \Delta \vdash p : t_p \\
\end{align*}
\]

\(^4\) We omit type judgments and type rules for the functional language.
\(^5\) We do not consider subtyping in this paper.
Provided and requested service declarations are typed with \( \Gamma_{\text{mlist}} \) judgment rules. Note that requested services are given service types without code types.

\[
\Delta \vdash s : s \ [t \text{ a type}] \\
\Delta \vdash \Gamma_{\text{mlist}} s_{\text{mlist}} : \rho \\
\rho' = \rho \lor (s \rightarrow t) \\
\Delta \vdash \Gamma_{\text{mlist}} s[(t)] ; s_{\text{mlist}} : \rho'
\]

A component type merges together service types as declared in the three parts of a component definition.

\[
\Delta \vdash_{\text{provided}} s_{\text{list}} : \rho_p \quad \Delta \vdash_{\text{provided}} s_{\text{list}} : \rho_p \quad \Delta \vdash_{\text{provided}} s_{\text{list}} : \rho_p
\]

Component

<table>
<thead>
<tr>
<th>Requested</th>
<th>Provided</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>{s_{\text{list}}}</td>
<td>{\rho_p, \rho_p, \rho_p}</td>
<td>{s_{\text{list}}}</td>
</tr>
</tbody>
</table>

For typing the plug operation, a virtual component is created that serves as a go-between to send the call from a component to another one. The unplug operation is typed accordingly: a virtual component is created containing only an undefined service type for what is unplugged.

\[
\Delta \vdash_{\text{unplug}} e_1 : (t_{c1}, s_1, t_{p1}) \quad t_{c1} = (\rho_{p1}, \rho_{p1}, \rho_{s1}) \\
\Delta \vdash_{\text{unplug}} e_2 : (t_{c2}, s_2, t_{p2}) \\
\Delta, v \mapsto t_{p1} \vdash \text{(call } e_2? t): t_2 \\
\rho'_1 = \rho_{p1} \leftarrow s_1 \rightarrow t_{p1} \quad \rho'_2 = \rho_{p2} \leftarrow s_1 \rightarrow t_{p2} \rightarrow t_2 \\
\Delta \vdash (\text{plug } e_1 \ e_2) : (\rho'_1, \rho_{p1}, \rho'_2)
\]

\[
\Delta \vdash_{\text{unplug}} e_1 : (t_{c1}, s_1, t_{p1}) \quad t_{c1} = (\rho_{p1}, \rho_{p1}, \rho_{s1}) \\
\rho'_1 = \rho_{p1} \leftarrow s_1 \rightarrow t_{p1} \\
\rho'_2 = \rho_{p2} \leftarrow s_1 \rightarrow t_{p2} \rightarrow t_2 \\
\Delta \vdash (\text{unplug } e_1) : (\rho'_1, \rho_{p1}, \rho'_2)
\]

The call result type is the result type of the service.

\[
\Delta \vdash_{\text{unplug}} e_1 : (t_{c1}, s_1, t_{p1}) \quad t_{c1} = (\rho_{p1}, \rho_{p1}, \rho_{s1}) \\
\Delta \vdash \mu \vdash e_2 : t_3 \\
\Delta, v \mapsto t_3, t_3 \vdash (\text{call } e_1[t_2]): t'
\]

The type language is extended by a mode value \( \mu \subset \{\text{add, sub}\} \). The type of a component expression is a sequence of service types. A service type is a mapping from a service name to a set of modes and a code type (the type of the service code). The set of modes declares the available ways to call the service. Finally, as in object functional programming, component locations appear as types. This generates a finite set of types for a finite program. The merge operation merges the two component types with respect to the mode. In rule below, \( m = \text{add} \) if \( e_3 \) is empty.

\[
\Delta \vdash e_1 : t_{c1} \quad \Delta, \text{ self } \mapsto t_{c1} \vdash e_2 : t_{c2} \quad [\Delta \vdash \mu \vdash e_3 : m] \\
\Delta \vdash (\text{merge} \#e_3 \ e_1 \ e_2) : t_{c1} \leftarrow m \ t_{c2}
\]

Type safety follows from verifying standard type preservation properties. Hence well-typed expressions are evaluable, i.e., there cannot be evaluation errors (provided for the functional language part an operational semantics safe with respect to a classic typing). E.g., if expressions typable in the underlying functional language are always reducible to values:

**Theorem 1.** Let \( e \) be an expression of the language, if \( \vdash e : t \) is provable, then there exist \( v, H \) such that \( \vdash e \Downarrow v, H \) is provable.

The proof is standard and the more general property is checked: let \( e \) be an expression of the language, if \( \Delta \vdash e : t \) is provable, then there exist \( v, H, H' \) such that \( E, H \vdash e \Downarrow v, H' \) is provable, furthermore if \( t \) is a component type then \( v \) is a handler with value in \( H' \) such that the structure of this value follows the structure of \( t \), finally if \( \Delta(x) \) is defined then also \( E(x) \) and if \( E(x) \) is a handler then \( H(E(x)) \) has a value. Similar properties for other type inference systems, i.e., \( \Gamma_{\text{mlist}} \ldots \). The proof is straightforward if we notice that there is no in-place adaptation and the underlying programming language is supposed to be functional, thus adaptations do not lead to type changes. Note that there is no necessity for a component to have values for all its required services: the typing system ensures that a call is correct as soon as what is needed for the call to be executed is present in the component, and only that. However, in case of distributed systems or web services, a dynamic type-checking has to be added as one cannot be sure that requests are well-formed with respect to the program.

5. Conclusion

The FLAC programming language deals with adaptable components. Its main feature concerns dynamic internal adaptations in a strongly-typed language. It is facilitated by the fact that adaptations are described in the same language as the component description language. This language may be developed in several directions we currently study. Among them, structuring components is in practice highly expected as it increases the modularity of the language. This may be done by adding named parameters to component definitions. Such named parameters not only allow assignments of (sub)components but they may be used for denoting them. It is then possible, for the programmer, to write control services such that the part of the component not involved in the evolution is automatically rebuilt. This may be implemented by abstractly manipulating the structure of components, i.e., addressing each subcomponent by its logical named path.

**References**


[7] Piriquito Margarida. The FLAC programming language deals with adaptable components. Its main feature concerns dynamic internal adaptations in a strongly-typed language. It is facilitated by the fact that adaptations are described in the same language as the component description language. This language may be developed in several directions we currently study. Among them, structuring components is in practice highly expected as it increases the modularity of the language. This may be done by adding named parameters to component definitions. Such named parameters not only allow assignments of (sub)components but they may be used for denoting them. It is then possible, for the programmer, to write control services such that the part of the component not involved in the evolution is automatically rebuilt. This may be implemented by abstractly manipulating the structure of components, i.e., addressing each subcomponent by its logical named path.


6. Appendix

Theorem 1  Provided the underlying functional language evaluates typable expressions, let e be an expression of the language, if e : t is provable, then there exist v, H such that e ⊢ v, H is provable.

Proof. We prove the following: Let e be an expression of the language, if e : t is provable, then there exist v, E, H, t′ such that E, H ⊢ e ⊣ v, H′ is provable, furthermore if t is a component type then v is a handler with value in H′ such that the structure of this value follows the structure of t, finally if (Δ(x)) is defined then also E(x) and if E(x) is a handler then H(E(x)) has a value. Similar properties for other type inference systems, i.e., Π-contexts. ...The proof is done by considering each rule of the typing system.

\[
\Delta \vdash s : s \quad \text{then for all } H, \quad E, H \vdash s \downarrow s, H
\]

\[
\Delta \vdash (\text{self}) \quad \Delta \vdash e_2 : s \quad \Delta \vdash p \vdash t : p
\]

\[
\Delta \vdash \#e_2(t_p) : (t_3, s, t_p)
\]

then, given the hypothesis of the typing rule, there exist E, H, h, t, t_1, t_2, that justify the hypothesis of the operational rule:

\[
\Delta \vdash e_1 : t_1 \quad \Delta \vdash s : s \quad \Delta \vdash p : p
\]

\[
\Delta \vdash \#e_3(t_p) : (t_3, s, t_p)
\]

then, given the hypothesis of the typing rule, there exist E, H, h, t, t_1, t_2, that justify the hypothesis of the operational rule:

\[
\Delta \vdash \oplus_{\text{list}} e_1 : t_1 \quad \Delta \vdash \oplus_{\text{list}} e_2 : (t_3, s, t_p, s, H_3)
\]

\[
\Delta \vdash e_1 : t_1 \quad \Delta \vdash s : s \quad \Delta \vdash p : p \quad \Delta \vdash e : t
\]

\[
\Delta \vdash \oplus_{\text{list}} S\_list : \rho \quad [\rho' \vdash s \rightarrow t_p \rightarrow t]
\]

\[
\Delta \vdash \sigma_{\text{list}} S\_list : \rho'
\]

then, given the hypothesis of the typing rule, there exist E, H, v, L, that justify the hypothesis of the operational rules:

\[
\Delta \vdash \sigma_{\text{list}} [e] S\_list(t_p) : \rho \quad \epsilon : S\_list \rightarrow \rho'
\]

This rule as well as the following one are used to present a list of available or requested services. The operational semantics is nothing else but the list itself.

\[
\Delta \vdash s : s \quad [t \text{ a type}]
\]

\[
\Delta \vdash \sigma_{\text{list}} S\_mllist : \rho
\]

\[
\rho' = \rho \lor \sigma (s \rightarrow t) \rightarrow \rho
\]

\[
\Delta \vdash \sigma_{\text{list}} S\_mllist[t][t]
\]