Types for Dynamic Reconfiguration

João Costa Seco and Luís Caires

Departamento de Informática,
Universidade Nova de Lisboa
{Joao.Seco, Luis.Caires}@di.fct.unl.pt

Abstract. We define a core language combining computational and architectural primitives, and study how static typing may be used to ensure safety properties of component composition and dynamic reconfiguration in object-based systems. We show how our language can model typed entities analogous of configuration scripts, makefiles, components, and component instances, where static typing combined with a dynamic type-directed test on the structure of objects can enforce consistency of compositions and atomicity of reconfiguration.

1 Introduction

In current object-oriented programming practice, composition-based modularization seems to have become the most common structuring mechanism, reflecting a shift of programming style from a pure, inheritance-based object-oriented style, towards the so-called "component-based programming" idioms, which favor blackbox composition.

Notwithstanding the proposal of many sophisticated type safe approaches to module and class composition [2, 5, 6, 10], the mechanism most frequently used to structure object-oriented applications in the "component-oriented" style is the ad-hoc assembly of webs of objects, where individual elements refer to each other through references. Since the code for construction of object structures is not distinguished at the programming language level from any other code, static checking of architectural consistency (of the kind found, for example, with ML functors or mix ins) is not performed during type checking, and may cause hard to correct errors to show up only at runtime. Moreover, the widespread use of sophisticated mechanisms such as dynamic loading, and mobile code, in mainstream programming frameworks adds relevance to the issue of finding expressive and safe programming constructs to dynamically build and reconfigure applications by aggregation and replacement of components and objects.

In previous work [13], we had presented a programming calculus with the aim to capture essential ingredients of object-oriented component programming styles, such as explicit context dependence, subtype polymorphism at the level of both components and objects, late composition, and avoidance of inheritance in favor of composition. A type system was also defined, with types assigned to (first-class) components and objects, thus ensuring runtime safety of compositions. However, although in such a model components may be dynamically composed, the structure of objects gets fixed once for all at instantiation time, thus excluding any possibility of dynamic reconfiguration.

In this paper, we present a new core component-oriented programming language, obtained by extending a λ-calculus with imperative records with a minimal set of architectural primitives. Moreover, we develop a type system that statically enforces, besides

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 214–229, 2006.

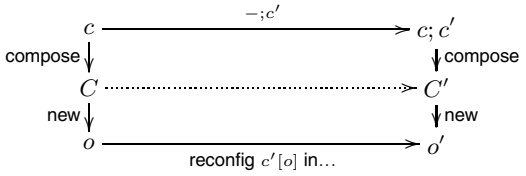

Fig. 1. Composition, Instantiation and Reconfiguration

the absence of more usual runtime errors, consistency of component compositions and atomicity of dynamic reconfiguration.

Our design is semantically motivated by considering a domain of *configurators*, *components*, and *objects*; all such entities are first-class in our model. Intuitively, configurators correspond (by analogy) to the usual notion of "makefile". Essentially, each configurator contains a series of instructions (architectural primitives) about how to assemble a component. Thus, language expressions that evaluate to configurator values may be seen as counterparts of configuration scripts, the kind of programs used in software configuration management systems to dynamically generate makefiles. Configurators which do not refer to external entities may generate components, by means of a **compose** primitive. Components are linked pieces of code (*cf.*, a class or a module), that may be further composed with other components and scripting code, in configuration scripts, or instantiated, by means of a **new** primitive, to yield objects. Methods can then be called on the appropriate ports of an object, in order to invoke its services. Additionally, configurators may also be applied to objects, by means of a **reconfig** primitive, to dynamically reconfigure their internal structure. The relation between configurators, components and objects, w.r.t. is hinted to in Figure 1. Intuitively, the object o' obtained from instantiating a component constructed from a configurator c and afterwards reconfigured by the configurator c', is *structurally* indistinguishable from the object instantiated from a component built from the composition of configurators $c; c'$ (although of course not *behaviorally* indistinguishable, since objects are stateful).

In our language, expressions denoting configurators, components and objects are distinguished at the level of typing, rather than at the syntax level, where they may be freely combined. For example, configurator types carry not only extensional but also intensional information, describing the internal architecture of the target component, while component (and object) types are purely extensional as usual, describing only the composition capabilities of a component in terms of required and provided service interfaces. Intensional information is needed to type configurator values, and ensure safety of component composition and dynamic reconfiguration.

It is expected that the soundness of any expressive notion of dynamic reconfiguration will turn out hard to ensure by purely static typing means, if one also wants to preserve object-level information hiding in the programming language. It is therefore important to explore the language design space involving combinations of static and dynamic checking, we believe to have isolated such an interesting combination. Thus, in our present proposal, type checking statically ensures good behavior of configurators, that is, that components built from well-typed configurators are architecturally consistent, and that objects instantiated from well-typed components are free from

runtime errors. Additionally, it is also ensured that objects reconfigured from well-typed configurators will always be architecturally consistent and free from runtime errors. These safety properties are crucial in our model, where both components and configurators are stateless first-class values that can be freely manipulated and composed in a language which is closed under abstraction and application. For example, it is conceivable for a software distribution system to export both a component set and a configuration script to a client, who will later on run the script, after composing it with local configuration information, to produce a certain subsystem. Such a scenario can be easily modeled in our language, in a typeful way.

We now illustrate the fundamental features of our language using a toy example; for the effect we assume to be given notions such as interfaces and method declarations with their standard meanings. Let ICounter be the interface type $\{\mathsf{tick} : \mathbf{int} \to \mathbf{int}\}$, declaring a method tick, and consider the following definition of a component Counter

```
let Counter = compose(
  provides p:ICounter;
  x[s:int=0, tick:(int→int)=fun y:int→x.s:=s+y];
  plug x into p) in ...
```

As argument of the compose operation, we find a configuration expression, namely a sequence of operations each of which introduces a particular element of a counter's architecture. First, a provided port named p, then a block of methods named x (implementing the method tick) and a state variable s, and finally a connection between the two, using the plug operation. Hence, object instances of component Counter will implement a port p conforming to the interface type ICounter. The type of Counter is $\{\} \Rightarrow \{\mathsf{p}{:}\mathsf{ICounter}\}$, meaning that it has no required services to be instantiated, and that their instances implement, at port p, the interface ICounter. Component Counter can then be instantiated, yielding an object o, by the expression

```
let o = new Counter in (o.p.tick(1); o.p.tick(1))
```

Component Counter may also be used as an element to define other components, for example, a ZeroCounter component, whose instances will count all calls to tick performed with zero as argument.

```
let ZeroCounter = compose(
  provides p:{tick:int→int};
  c[Counter:{}⇒{p:ICounter}];
  x[tick:int→int =
        fun y→if y=0 then c.p.tick(1)];
  plug x into p) in
```

Here, component Counter is inserted in the architecture of ZeroCounter under the name c (in c [...]), and used in the composition context (in c.p. tick (1)). The component ZeroCounter may then be used to build other components, or instantiated as in

```
let zc = new ZeroCounter in ...
```

Now, suppose that a ZeroCounter object, such as zc, is running in a server application, and the need arises of extending it with a new service, to reset the inner counter, without shutting it down: clearly, this is a situation calling for a dynamic reconfiguration facility. Consider then the following (re)configuration script:

```
let AddReset =
 (provides r:{reset:unit→unit};
  y[reset:unit→unit =
          if c.get()>0 then c.p.tick(−1); y.reset()];
  plug y into r) in ...
```

Configurator AddReset adds a provided port r, to expose the new reset method, implemented by the method block y. Notice that the architectural operations used in the definition of AddReset refer to elements (e.g., c) which are not declared in the static context of definition (and thus may be seen as white-box operations). However, the context of use is captured at the type level, with configurator AddReset being given configurator type

$$\{c\bullet\{p:ICounter\}\} \Longrightarrow \{c\bullet\{p:ICounter\}, r\triangleright\{reset:unit \rightarrow unit\}, y\bullet\{reset:unit\rightarrow unit\}\}$$

Configurator types are of the form $K \Longrightarrow K'$, where the two bags of "resources" K and K' describe the change of internal elements in a configuration; each resource is tagged with its (object or interface) type. The type of AddReset states that the configurator may be applied in every context where an element c of (object) type {p:ICounter} is present (the $\bullet$ resource on the left hand side). It also says that, after application, c remains available, alongside with a (new) provided port r (the $\triangleright$ resource on the right hand side) and a (new) method block y. The following expression

```
reconfig zcr = AddReset[zc] in ... use of zcr... else ... use of zc ...
```

has then the effect of actually reconfigure the object zc, returning a properly typed reference zcr to the updated object that implements the reset service at a new port r. In general, a reconfiguration may not be possible, due to a mismatch between the internal structure of the object to be reconfigured (which is not visible to the type system) and the precondition of the configurator. In any case, the type system ensures the atomicity of reconfiguration, i.e. that either the reconfiguration is fully applied as specified by the configurator, and the resulting object is well defined (**in** branch), or the object is not modified (**else** branch). This property is a consequence of static typing, at the level of configurator values, and of a simple and efficient test on type information recorded inside objects, in the spirit of [1], realized at reconfiguration time.

Although not illustrated here, it is possible for a component's implementation to depend, through a required port, on some external implementations of an interface. Whenever a component with required ports is instantiated or new required ports are added through reconfiguration then both new and reconfig expressions must provide compatible implementations to each required port. This is achieved by a special with clause containing multiple assignments.

Related Work. To the best of our knowledge, the calculus presented in [13] was a first proposal to integrate computation and (first-class) architectural definition in the context of a object-oriented strongly typed programming language. Programming languages supporting first-class components have been studied by several authors [2, 10, 17], although not considering dynamic reconfiguration of instances. More related to our model are the module calculi of [3, 11, 12, 19] which also introduce composition operations for first-class modules and mixins: in these approaches the module language is

stratified on top of a core language. While relying on a different choice of primitives, inherited from our early work [13], we believe that our approach is particularly suitable as a basis for defining component-based languages where computational and configuration / reconfiguration operations may be freely combined (modulo typing constraints) at the same level. More recently, the work in [9] has extended the approach of [3] with a form of dynamic reconfiguration that allows for the interleaved execution of the module manipulation operations and the core language expressions. This seems to correspond to some form of dynamic composition, while we consider the in-place modification of the internal structure of (potentially aliased) stateful objects.

From the perspective of software evolution, several works [4, 8, 16] have addressed the operational semantics and type structure of software systems that support dynamical change of modules. In these approaches, modules implement ADTs, and the focus is on version management of values of such abstract types. In our model, components do not usually represent ADTs but rather service providers, and we concentrate on dynamic reconfiguration of architectures, rather than on individual replacement of a module's implementations.

Forms of dynamic reconfiguration for object-oriented languages involving a fixed predetermined number of future configurations, have also been considered by [7]. In this work, we aimed to model unanticipated reconfiguration using first-class typed notions of (re)configuration scripts, thus following an approach that does not seem to have been explored before. In this context, the fundamental work of [18] on meta-programming and staged programming languages also appears to bear some relation to our development here, even if our focus is on isolating first class semantic entities related to software assembly, rather than on how to express and type source (meta)level program manipulations.

Outline. The remainder of the paper is organized as follows: Section 2 formally presents the language syntax. The operational semantics is introduced in Section 3. In Section 4 we present the type system, and state the main type safety results. Finally, we conclude with some remarks on this work, and suggest possible developments.

2 The Component Calculus

In this section, we introduce λ_χ, a component-based calculus aimed at capturing the programming model motivated above. The language is a simply typed λ-calculus with mutable records enriched with primitives to build and manipulate components. The types of λ_χ are shown in Figure 2. Besides standard functional types, we include types for interfaces and mutable records, components and configurators. Not all type expressions are meaningful, for example, in a component type $\tau \Rightarrow \sigma$, τ and σ are expected to be object types, expressing the required and provided services of the component: the type system presented below will only accept meaningful type expressions.

Configurator types describe the effects of configurators on compositions, expressed in the form of required and provided *resources* (do not confuse with required and provided service *ports*). A resource is represented by a combination of a tag, a name, and a type. The possible tags are: $\circ$ (open), meaning that the resource is unsatisfied, for instance, that a provided port is not connected; $\bullet$, meaning that the resource is available

$$
\begin{array}{lll}
\tau, \sigma ::= & & \text{types} \\
& \tau \to \sigma & \text{function} \\
\mid & \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\} & \text{record} \\
\mid & \{\ell_i : \tau_i\ ^{i\in 1..n}\} & \text{interface} \\
\mid & \tau \Rightarrow \sigma & \text{component} \\
\mid & \{r_i\ ^{i\in 1..n}\} \Longrightarrow \{r_i\ ^{i\in 1..m}\} & \text{configurator} \\
r \quad ::= & \pi \circ \tau \mid \pi \bullet \tau \mid \pi \triangleright \tau \mid \pi \triangleleft \tau & \text{resources}
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & x \mid \lambda x : \tau.e \mid e(e) \mid [\ell_i : \tau_i = e_i\ ^{i\in 1..n}] \mid e.\ell \mid e.\ell := e \\
& \mid & \mathsf{compose}\ e \mid \mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i\ ^{i\in 1..n} \\
& \mid & \mathsf{reconfig}\ x = e[e]\ \mathsf{with}\ \ell_i := e_i\ ^{i\in 1..n}\ \mathsf{in}\ e\ \mathsf{else}\ e \\
& \mid & c \\
c & ::= & e; e \mid \mathsf{requires}\ \ell : \tau \mid \mathsf{provides}\ \ell : \tau \mid \mathsf{plug}\ \pi : \tau\ \mathsf{into}\ \pi : \tau \\
& \mid & x[e : \tau] \mid x_K[\ell_i : \tau_i = \lambda x_i : \tau_i.e_i\ ^{i\in 1..n}] \\
\pi & :: = & x \mid \ell \mid x.\ell
\end{array}
$$

Fig. 2. Types and terms for λ_χ

for connection, for instance a certain method block or inner component is present; $\triangleright$ denotes that a provided port is present, and $\triangleleft$ denotes that a required port is present. Typically, at the level of typing, composition operation rewrites a bag of resources into another bag of resources, reflecting the internal change that takes place in the component architecture. In general, we use K to denote resource sets. We also define K_* to be the interface type containing all resources tagged with $*$ in K where $*$ may be $\circ$, $\bullet$, $\triangleright$, or $\triangleleft$. For example, $K_\bullet$ is $\{\ell_i : \tau_i\ ^{i\in 1..n}\}$ where $\ell_i \bullet \tau_i$ for $i = 1..n$ are all the $\bullet$-tagged elements in K. We use I, J for interface types and R, P for object types, i.e. interfaces of the form $\{\ell_i : I_i\ ^{i\in 1..n}\}$. We denote by $- \oplus -$ the concatenation operation on interfaces, and by $-\#-$ the disjointness predicate for interfaces and resource sets.

We define the syntax of λ_χ on Figure 2, based on a standard formulation for an imperative λ-calculus, enriched with three new imperative expressions of interest, compose, new, and reconfig. Additionally, a set of primitive composition operations are defined (under syntactic category c), each being a canonical configuration script represented at runtime by a configurator. These configurators are typed stateless values programmed to produce a specific structural effect on an architecture, either in the construction of a component or in the reconfiguration of an instance. They are combined under a *white-box* discipline by the composition operation $(e_1; e_2)$, which means that any element introduced by e_1 can be referred and connected to elements introduced by e_2.

The typed and named ports of a component are declared by (requires $\ell : \tau$) to import a service and by (provides $\ell : \tau$) to declare a port exporting a service; $(x[e : \tau])$ to introduce in the architecture a component, resulting from evaluating e. Such an element is referred in the composition context by the local name x. Basic building blocks containing method implementations are introduced by $(y_K[\ell_i : \tau_i = \lambda x_i : \tau_i.e_i\ ^{i\in 1..n}])$ and referred by the local name y. Notice that the set of resources K declares explicit architectural dependencies from other elements at the same compositional level, allowing references to them to be made inside the expressions of the methods. Connections between elements in architectures are created by (plug $\pi_1 : \tau_1$ into $\pi_2 : \tau_2$) expressions, declaring that method invocations at port π_2 should be redirected to port π_1.

Given an expression e denoting a configurator with no required resources, compose e yields a component value which "freezes" the configurator's architecture in a component value in such a way that it can only be further composed using black-box operations (by means of a composition operation $x[c : \tau]$). Component values can be instantiated, with new, to yield objects. Notice that these objects will be fully operational only if all of their required ports get actually linked to compatible implementations. Such dependencies may either be satisfied in a composition context, or by *plug-assignments* (with clause) in a instantiation expression.

In a reconfiguration expression reconfig $x = e_1[e_2]$ with $\ell_i := e'_i \ ^{i\in 1..n}$ in e_3 else e_4, the distinguished occurrence of x is binding, with scope e_3 and e_4. If a reconfiguration is successful, the e_3 branch will be executed, with x denoting the reconfigured instance (at the "new" type), otherwise the fail branch e_4 will be chosen. Moreover, since the configurator e_1 may add new required ports to the instance new values must be assigned to them by *plug-assignments*.

3 Operational Semantics

In this section, we present the semantics of our language. Technically, this will be accomplished with big-step operational semantics, using judgments of the form $e; S \downarrow v; S'$, where e is an expression and S a heap, v is the value of e and S' is the resulting heap. An heap S is an assignment of values v to locations l from a set of locations $\mathtt{Loc}$, along standard lines. The values of λ_χ are listed in Figure 3.

$$v ::= \lambda x : \tau.e \mid r \mid \mathsf{conf}(\tau,c) \mid \mathsf{comp}(c) \mid (r,r,r)_\Gamma \mid l \mid \mathsf{nil}$$
$$r ::= \{\ell_i \mapsto l_i \ ^{i\in 1..n}\}$$

Fig. 3. Evaluation results

As expected, the more basic values are *abstractions*, and mutable *records*, which are in our current context finite mappings from labels to locations. A *configurator* value, of the form $\mathsf{conf}(\tau, c)$, is a pair that packs the runtime representation of a sequence of instructions to construct or change the architecture of a component, with a configurator type τ that specifies a precondition on its application. Thus, configurators embed some type information at runtime, to be used in a dynamic check during the evaluation of reconfiguration expressions. A *component* value, of the form $\mathsf{comp}(c)$, is the runtime representation of a sequence of instructions to construct or change the architecture of a component. Notice that configurator and component values are pure values, while object instances are (of course) stateful entities, constructed as specified by their generating component. An object value (component instance), is a triple of records of the form $(r, e, p)_\Gamma$ where the labels in r refer to its required ports, the labels in e refer to its inner elements, and the labels in p their provided ports. Γ is a local typing environment assigning types to the object's internal elements, useful for checking the precondition of a configurator. For the sake of simplicity we sometimes refer to an object value s by the single record obtained by concatenating the three records r, e and p in s. Let $s = (r, e, p)_\Gamma$ be an object, we write $s_\triangleleft$ to denote the record r containing the required

ports in s, Γ_s to denote Γ in s and $s_\triangleleft \oplus s_\bullet$ to denote the concatenation of the records r and e. We write $\Gamma(s_\triangleright.\ell) = \tau$ for $s_\triangleright = \{\ldots \ell \mapsto \mathtt{l} \ldots\}$ and $\Gamma(\mathtt{l}) = \tau$.

We write $\{\mathtt{l}_i \mapsto v_i \ ^{i\in 1..n}\}$ to define a heap, $S(\mathtt{l})$ to denote the value associated to $\mathtt{l}$ in S, $S[\mathtt{l} \mapsto v]$ to denote a heap S updated with a new relation, and $Dom(S)$ to denote the domain set of S. We say that a heap S is *closed* if all locations occurring in S are elements of $Dom(S)$. We define $\mathrm{new}(S) \triangleq \mathtt{l}$ such that $\mathtt{l} \in \mathtt{Loc} \backslash Dom(S)$, and use it in the operational semantics to denote a fresh memory location. Notice that since locations are values, cyclic chains of locations may potentially exist in a heap, leading from a location to itself after a number of indirections. We say that a location participating in such a cycle is *undefined*. Such cyclic reference chains may only be introduced if components with vacuous connections, connecting a provided port to a required port, are defined.

The rules defining the operational semantics of λ_χ are listed in Figures 4, 6, and 7. The "main" judgment form is mutually dependent on a second judgment form, $s; c; S \Downarrow s'; S'$. This defines the application of a composition operation c to a

$$v; S \downarrow v; S \quad \text{(Eval Value)}$$

$$\frac{e_1; S \downarrow \lambda x:\tau.e; S' \ e_2; S' \downarrow v; S'' \ e[x \leftarrow v]; S'' \downarrow v'; S'''}{e_1(e_2); S \downarrow v'; S'''} \quad \text{(Eval Call)}$$

$$\frac{e_i; S_{i-1} \downarrow v_i; S_i \ \forall_{i\in 1..n}}{[\ell_i = e_i \ ^{i\in 1..n}]; S_0 \downarrow \mathtt{l}; S^n[\mathtt{l} \mapsto \{\ell_i \mapsto \mathtt{l}_i \ ^{i\in 1..n}\}][\mathtt{l}_i \mapsto v_i \ ^{i\in 1..n}]} \quad \text{(Eval Record)} \quad (\mathtt{l}, \mathtt{l}_i = \mathrm{new}(S) \ \ \forall_{i\in 1..n})$$

$$\frac{e_1; S \downarrow \mathtt{l}; S' \quad \mathtt{l}' = \mathrm{deref}_{S'}(\mathtt{l}) \quad S'(\mathtt{l}') = \{\ldots, \ell \mapsto \mathtt{l}'', \ldots\} \quad e_2; S' \downarrow v; S''}{e_1.\ell := e_2; S \downarrow v; S''[\mathtt{l}'' \mapsto v]} \quad \text{(Eval Assign)}$$

$$\frac{e; S \downarrow \mathtt{l}; S' \quad \mathtt{l}' = \mathrm{deref}_S(\mathtt{l}) \quad S'(\mathtt{l}') = \{\ldots, \ell \mapsto \mathtt{l}'', \ldots\}}{e.\ell; S \downarrow S'(\mathtt{l}''); S'} \quad \text{(Eval Select)}$$

$$\frac{e; S \downarrow \mathsf{comp}(\tau, c); S'}{\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'} \quad \text{(Eval Compose)}$$

$$\frac{e; S \downarrow \mathsf{comp}(c); S' \quad \mathbf{0}; c; S' \Downarrow s; S_0 \quad e_i; S_{i-1} \downarrow v_i; S_i \ \forall_{i\in 1..n}}{\mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i \ ^{i\in 1..n}; S \downarrow \mathtt{l}; S_n[\mathtt{l} \mapsto s][\mathtt{l}_i \mapsto v_i \ ^{i\in 1..n}]} \quad \text{(Eval New)} \quad (s_\triangleleft = \{\ell_i \mapsto \mathtt{l}_i \ ^{i\in 1..n}\}, \ \ \mathtt{l} = \mathrm{new}(S))$$

$$\frac{\begin{array}{l} e_1; S \downarrow \mathsf{conf}(K \Longrightarrow K', c); S' \\ e_2; S' \downarrow \mathtt{l}; S_0 \quad s = S_0(\mathtt{l}) \quad s /\!\!/ K \\ s; c; S_n \Downarrow s'; S_{n+1} \\ f_i; S_{i-1} \downarrow v_i; S_i \quad \forall_{i\in 1..n} \\ e_3[x \leftarrow \mathtt{l}']; S_{n+1}[\mathtt{l}' \mapsto s'][\mathtt{l}_i \mapsto v_i \ ^{i\in 1..n}] \downarrow v; S'''' \end{array}}{\left(\begin{array}{c}\mathsf{reconfig}\ x = e_1[e_2] \\ \mathsf{with}\ \ell_i := f_i \ ^{i\in 1..n} \\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4\end{array}\right); S \downarrow v; S'''} \quad \text{(Eval Reconfig)} \quad (s'_\triangleleft = s_\triangleleft \oplus \{\ell_i \mapsto \mathtt{l}_i \ ^{i\in 1..n}\})$$

$$\frac{\begin{array}{l} e_1; S \downarrow \mathsf{conf}(K \Longrightarrow K', c); S' \\ e_2; S' \downarrow \mathtt{l}; S'' \quad s = S''(\mathtt{l}) \quad \neg s /\!\!/ K \\ e_4[x \leftarrow \mathtt{l}]; S'' \downarrow v; S''' \end{array}}{\left(\begin{array}{c}\mathsf{reconfig}\ x = e_1[e_2] \\ \mathsf{with}\ \ell_i := f_i \ ^{i\in 1..n} \\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4\end{array}\right); S \downarrow v; S'''} \quad \text{(Eval Reconfig Else)}$$

Fig. 4. Evaluation of computational expressions

$$\text{(Match Provides)}\quad \frac{\Gamma_s(s_{\triangleright}.\ell) = \tau \quad s /\!\!/ K}{s /\!\!/ \ell \triangleright \tau, K} \qquad \text{(Match Requires)}\quad \frac{\Gamma_s(s_{\triangleleft}.\ell) = \tau \quad s /\!\!/ K}{s /\!\!/ \ell \triangleleft \tau, K} \qquad \text{(Match Element)}\quad \frac{\Gamma_s((s_{\triangleleft} \oplus s_{\bullet}).\ell) = \tau \quad s /\!\!/ K}{s /\!\!/ \ell \bullet \tau, K}$$

$$\text{(Match Element Port)}\quad \frac{S(s_{\bullet}.x) = s' \;\; \Gamma_{s'}(s'_{\triangleright}.\ell) = \tau \;\; s /\!\!/ K}{s /\!\!/ x.\ell \bullet \tau, K} \qquad \text{(Match Unsatisfied)}\quad \frac{\Gamma_s(s_{\triangleright}.\ell) = \tau \;\; s /\!\!/ K}{s /\!\!/ \ell \circ \tau, K} \qquad \text{(Match Unsatisfied Port)}\quad \frac{S(s_{\bullet}.x) = s' \;\; \Gamma_{s'}(s'_{\triangleleft}.\ell) = \tau \;\; s /\!\!/ K}{s /\!\!/ x.\ell \circ \tau, K}$$

Fig. 5. Rules for matching

$$\text{(Eval Requires)}\;\; (\sigma = \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \triangleleft \tau\}) \qquad \text{(Eval Provides)}\;\; (\sigma = \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\})$$
$$\mathsf{requires}\ \ell : \tau; S \downarrow \mathsf{conf}(\sigma, \mathsf{requires}\ \ell : \tau); S \qquad \mathsf{provides}\ \ell : \tau; S \downarrow \mathsf{conf}(\sigma, \mathsf{provides}\ \ell : \tau); S$$

$$\text{(Eval Plug)} \qquad (\sigma = \{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\})$$
$$\mathsf{plug}\ \pi_1 : \tau\ \mathsf{into}\ \pi_2 : \tau; S \downarrow\ \mathsf{conf}(\sigma, \mathsf{plug}\ \pi_1 : \tau\ \mathsf{into}\ \pi_2 : \tau); S$$

$$\text{(Eval Sequence)}\quad \frac{e_1; S \downarrow \mathsf{conf}((K \Longrightarrow K', K_c), c_1); S' \qquad e_2; S \downarrow \mathsf{conf}((K_c, K'' \Longrightarrow K'''), c_2); S'}{(e_1; e_2); S \downarrow \mathsf{conf}((K, K'' \Longrightarrow K', K'''), (c_1; c_2)); S'}$$

$$\text{(Eval Uses)}\quad \frac{\begin{pmatrix}\tau = \{\ell_i^r : \tau_i\ ^{i \in 1..n}\},\ \sigma = \{\ell_j^p : \sigma_j\ ^{j \in 1..m}\} \\ K = \{x \bullet \sigma, x.\ell_i^r \circ \tau_i\ ^{i \in 1..n}, x.\ell_j^p \bullet \sigma_j\ ^{j \in 1..m}\}\end{pmatrix} \quad e; S \downarrow v; S'}{x[e : \tau \Rightarrow \sigma]; S \downarrow \mathsf{conf}(\emptyset \Longrightarrow K, x[v : \tau \Rightarrow \sigma]); S'}$$

$$\text{(Eval Method Block)} \qquad (\sigma = K \Longrightarrow K, x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\})$$
$$x_K[\ell_i : \tau_i = v_i\ ^{i \in 1..n}]; S_0 \downarrow\ \mathsf{conf}(\sigma, x_K[\ell_i : \tau_i = v_i\ ^{i \in 1..n}]); S$$

Fig. 6. Evaluation of Composition Operations

composition context s with relation to a heap S, and resulting in a modified object instance s' and heap S'. s is a partially built instance where the effects of composition operations get accumulated during composition. To dereference a chains of locations in the heap to its target value we introduce the auxiliary function $\mathrm{deref}_S(\mathit{l})$ that denotes the last location of a chain starting with l. $\mathrm{deref}_S(-)$ is useful to make the mapping from locations to values independent of the number of heap indirections, created during **plug** operations.

We now discuss some key aspects of the operational semantics. For simplicity, in the rules of Figure 6, the basic composition operations provides, requires, and plug, evaluate to themselves, and are directly stored in configurator values along with appropriate intensional type information. Since the fields of method blocks (code) are, by definition values of the language, they may also directly stored inside a configurator value. In the case of the introduction of an inner component ($x[e:\tau]$) the resulting value depends on the evaluation of the inner expression e to a component which then forms a composition operation ($x[v:\tau]$) where v is again a value, then stored in a configurator. The combination of two operations, Rule (Eval Sequence), produces a configurator containing the composition of the two operands. Notice that the new type information

in the sequential composition is obtained from the manifest information of both parts. In general, the type annotations in configurators are constructed in a mechanical way, we will later show that in well-typed programs this computations always succeed.

The evaluation of a compose, Rule (Eval Compose), simply tags a configurator as being a closed architecture by means of a comp constructor, enclosing its composition operation. Such a configurator may not be further combined by composition with other configurators, but only to instantiate objects.

The evaluation of the instantiation expression **new** uses the configuration instruction stored in the component value, applying them to an "empty" object instance, written $\mathbf{0}$. This is expressed in Rule (Eval New) by the premise $\mathbf{0}; c; S \Downarrow s; S'$. Required ports left open by the application are satisfied by the values given by the plug-assignments.

A reconfiguration depends on a (runtime) test, to check that a configurator is in fact compatible with the structure of a given instance. Formally, we specify that a configurator with precondition type K is applicable to s if the matching $s /\!\!/ K$ test (defined in Figure 5) holds. Intuitively, an instance $s = (r, e, p)_\Gamma$ matches a set of resources K if each one of the resources in K can be found, with compatible types, in r, e, or p.

The evaluation of reconfig expression, is thus defined by two rules (Eval Reconfig) and (Eval Reconfig Else), that consider the two possible outcomes of a matching test. Rule (Eval Reconfig) is applicable if the test $s /\!\!/ K$ succeeds and the composition operation c, taken from the configurator yield by e_1 is applied to s, the instance obtained from e_2. The final result comes from evaluating e_3. Rule (Eval Reconfig Else) is applicable otherwise, it skips the application of the composition application and follows by evaluating the else branch. Notice that only the required resources (in the precondition) in the runtime type information are used to test the instance. The added resources (in the type's post condition) are nevertheless important in the process of building the type information (see (Eval Sequence)).

The rules in Figure 7, for the judgement form $s; c; S \Downarrow s'; S'$, interpret the application of a composition operation c to a partially built instance s, with relation to a heap S, and incrementally build an object instance.

As expected, Rule (App Sequence) sequentially applies the two parts of the operation thus causing the combined effect of both. (App Requires) and (App Provides) both create nil initialised references (empty placeholders) in the heap for ports, and establish the corresponding connections in the records r or p.

The integration of a inner component instance inside an instance depends on the (recursive) construction of the inner instance, and corresponding introduction as an inner element of the instance in record e. Similarly, (App Method Block) takes the field values and builds a record associated to its local name. Here, $v_i[(r, e, p)_\Gamma]$ denotes the substitution of the object's labels by their locations in the fields and therefore give access to the elements already in the instance and that $[x \leftarrow \mathit{1}]$ introduces the "self" reference of the method block itself in the field expressions.

Finally, the application of a plug expression connects plug sources to target ports by simply forming a chain between the two locations, Rule (App Plug). We use the function $\mathrm{select}_S(o, \pi)$ to denote the location corresponding to port π. Notice how the runtime type annotation of an object is progressively built on each rule, and added to the Γ component. The resulting object is then a structured web of ports, other objects, and

$$\text{(App Requires)} \qquad (\mathtt{l} = \text{new}(S))$$
$$(r, e, p)_\Gamma; (\mathsf{requires}\ \ell : \tau); S \Downarrow (r \oplus \{\ell \mapsto \mathtt{l}\}, e, p)_{\Gamma, \mathtt{l}:\tau}; S[\mathtt{l} \mapsto \mathsf{nil}]$$

$$\text{(App Provides)} \qquad (\mathtt{l} = \text{new}(S))$$
$$(r, e, p)_\Gamma; (\mathsf{provides}\ \ell : \tau); S \Downarrow (r, e, p \oplus \{\ell \mapsto \mathtt{l}\})_{\Gamma, \mathtt{l}:\tau}; S[\mathtt{l} \mapsto \mathsf{nil}]$$

$$\text{(App Uses)} \qquad \frac{\mathsf{new}\ v; S \downarrow \mathtt{l}; S'}{(r, e, p)_\Gamma; x[v : \tau]; S \Downarrow (r, e \oplus \{x \mapsto \mathtt{l}\}, p)_{\Gamma, \mathtt{l}:\tau}; S'}$$

$$\text{(App Sequence)} \qquad \frac{s; c_1; S \Downarrow s'; S' \quad s'; c_2; S' \Downarrow s''; S''}{s; (c_1; c_2); S \Downarrow s''; S''}$$

$$\text{(App Method Block)}$$
$$(r, e, p)_\Gamma; x_K[\ell_i{:}\tau_i = v_i\ ^{i\in 1..n}]; S \Downarrow (r, e \oplus \{x \mapsto \mathtt{l}\}, p)_{\Gamma, \mathtt{l}:\{\!|\ell_i:\tau_i\ ^{i\in 1..n}|\!\}}; S'$$
$$\begin{pmatrix} \mathtt{l}, \mathtt{l}_i\ ^{i\in 1..n} = \text{new}(S), \\ S' = S[\mathtt{l} \mapsto \{\ell_i \mapsto \mathtt{l}_i\ ^{i\in 1..n}\}][\mathtt{l}_i \mapsto v_i[(r, e, p)_\Gamma][x \leftarrow \mathtt{l}]\ ^{i\in 1..n}] \end{pmatrix}$$

$$\text{(App Plug)}$$
$$s; \mathsf{plug}\ \pi_1{:}\tau_1\ \mathsf{into}\ \pi_2{:}\tau_2; S \Downarrow s; S[\text{select}_S(s, \pi_2) \mapsto \text{select}_S(s, \pi_1)]$$

Fig. 7. Application of Configurators

records containing variables and methods. Methods can be accessed through provided ports, that lead to appropriate implementations as specified by the object's architecture.

4 Type System

In this section we present a type system for λ_χ. Well-typed programs are ensured to be well behaved, in the sense motivated in the introduction, and made precise below.

Our type system includes rules for typing computational expressions (Figure 8), and rules for typing compositional expressions (Figure 9). Typing environments (Δ, Γ) assign types to variables, as usual, and also to locations (this is only useful for stating our subject reduction result). The rules for the λ-calculus and imperative records are standard. Rule (Val Interface) allows us to coerce a record type to an interface type.

The architectural soundness of configurators, components and instances is ensured by the combination of the typing of composition operations in Figure 9 together with the typing of the compose, new and reconfig. On one hand, the typing of composition operations intentionally describes and combines their effect, on the other hand, the typing of computational expressions uses that information in three levels of visibility. Rule (Val Compose) ensures that components are only produced given a completed architecture, i.e. from configurators that do not depend on any existing resource ($\emptyset \Longrightarrow K$) and leave no unsatisfied resources left open ($K_\circ = \emptyset$). The resulting component type $K_\triangleleft \Rightarrow K_\triangleright$ reveals only the required and provided service types, hiding the remaining intensional information about the component internal structure. Thus, a component is indistinguishable from any other with the same type. Rule (Val New) types a new instance with the object type containing the provided ports of its generator component and checks for the proper satisfaction of all required ports, if there are any. Notice that once again some type information gets hidden: here, the existing required ports are not

$$\text{(Val Var)}\quad \frac{x:\tau \in \Delta}{\Delta \vdash x:\tau} \qquad \text{(Val Abstraction)}\quad \frac{\Delta, x:\tau \vdash e:\sigma}{\Delta \vdash \lambda x:\tau.e:\tau\to\sigma} \qquad \text{(Val Application)}\quad \frac{\Delta \vdash e_1:\tau\to\sigma \quad \Delta\vdash e_2:\tau}{\Delta\vdash e_1(e_2):\sigma} \qquad \text{(Val Interface)}\ (m\le n)\quad \frac{\Delta\vdash e:\{\!|\ell_i:\tau_i\ ^{i\in 1..n}|\!\}}{\Delta\vdash e:\{\ell_i:\tau_i\ ^{i\in 1..m}\}}$$

$$\text{(Val Record)}\quad \frac{\Delta\vdash e_i:\tau_i \quad \forall i\in 1..n}{\Delta\vdash [\ell_i:\tau_i = e_i\ ^{i\in 1..n}]:\{\!|\ell_i:\tau_i\ ^{i\in 1..n}|\!\}} \qquad \text{(Val Select)}\quad \frac{\Delta\vdash e:\{\ldots,\ell:\tau,\ldots\}}{\Delta\vdash e.\ell:\tau} \qquad \text{(Val Assign)}\quad \frac{\Delta\vdash e_1:\{\!|\ldots,\ell:\tau,\ldots|\!\} \quad \Delta\vdash e_2:\tau}{\Delta\vdash e_1.\ell:=e_2:\tau}$$

$$\text{(Val Compose)}\quad (K_\circ=\emptyset)\quad \frac{\Delta\vdash e:\emptyset\Longrightarrow K}{\Delta\vdash \mathsf{compose}\ e:K_\triangleleft\Rightarrow K_\triangleright} \qquad \text{(Val New)}\quad \frac{\Delta\vdash e:\{\ell_i:\tau_i\ ^{i\in 1..n}\}\Rightarrow\sigma \quad \Delta\vdash e_i:\tau_i\ \ \forall_{i\in 1..n}}{\Delta\vdash \mathsf{new}\ e\ \mathsf{with}\ \ell_i:=e_i\ ^{i\in 1..n}:\sigma}$$

$$\text{(Val Reconfig)}\quad \big(K'_\circ=\emptyset,\ K'_\triangleright \# I,\ K'_\triangleleft=\{\ell_i:\sigma_i\ ^{i\in 1..n}\}\big)$$
$$\frac{\Delta\vdash e_1:K\Longrightarrow K' \quad \Delta\vdash e_2:I \quad \Delta\vdash e'_i:\sigma_i\ \ \forall_{i\in 1..n} \quad \Delta,x:I\oplus K'_\triangleright\vdash e_3:\delta \quad \Delta,x:I\vdash e_4:\delta}{\Delta\vdash \mathsf{reconfig}\ x=e_1[e_2]\ \mathsf{with}\ \ell_i:=e'_i\ ^{i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4:\delta}$$

Fig. 8. Typing Rules for Computational Expressions

included in the instance type. Hence, the type system does not distinguish instances of components providing the same ports.

Despite the dependence of reconfiguration on a runtime check, some basic conformance between the configurator type ($K \Longrightarrow K'$) and the type of the target object (τ) is tested statically in the (Val Reconfig) rule. We basically use K' to ensure that the continuations of reconfigurations are well-typed, in particular: that no dependencies are left open after the application ($K'_\circ = \emptyset$); that the configurator does not override the object's ports ($K'_\triangleright \# I$); and that all new requirements must be correctly satisfied by the existing plug assignments, ($K'_\triangleleft - K_\triangleleft = \{\ell^r_i : \sigma_i\ ^{i\in 1..m}\}$).

The rule system of Figure 9 assigns a type of the form $K \Longrightarrow K'$ to each composition operation to denote the required (K) and provided (K') resources. Basic composition operations get natural configurator types, which are then elaborated by means of composition. For instance the type of (provides $\ell:\tau$) indicates the providing of an unsatisfied resource ($\ell \circ \tau$), i.e. a resource that must be satisfied before this configurator is used to make a component or reconfigure an instance, and of a new provided port ($\ell \triangleright \tau$). Symmetrically, (requires $\ell:\tau$) adds a new required port ($\ell \triangleleft \tau$) and an available resource ($\ell \bullet \tau$) to a composition context. The typing of ($x[e:\tau]$) indicate that it adds available resources corresponding to an instance of the inner component ($x \bullet \{\ell^p_j : \sigma_j\ ^{j\in 1..m}\}$) and its provided ports ($x.\ell^p_j \bullet \sigma_j\ ^{j\in 1..m}$), and unsatisfied resources that denote the internally required ports ($x.\ell^r_i \circ \tau_i\ ^{i\in 1..n}$). Similar type information is associated with method blocks but using the required set of resources K. Notice the restricted typing environment $|\Delta|$ in the premises of Rule (Comp Method Block), and (Comp Uses). $|\Delta|$ denotes the typing environment retaining the type assignments in Δ that have component or configurator type. This forbids any reference to the heap to be made from well-typed method blocks, and therefore ensures that configurators are closed values.

$$\text{(Comp Requires)} \qquad \Delta \vdash (\mathsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \triangleleft \tau\}$$

$$\text{(Comp Provides)} \qquad \Delta \vdash (\mathsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\}$$

$$\text{(Comp Plug)} \qquad \Delta \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\})$$

$$\text{(Comp Sequencing)} \qquad \frac{\Delta \vdash e_1 : K \Longrightarrow K', K_c \qquad \Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''}{\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''} \quad (K' \# K'', K' \# K''')$$

$$\text{(Comp Uses)} \qquad \frac{|\Delta| \vdash e : \tau \Rightarrow \sigma}{\Delta \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow \left\{x \bullet \sigma, x.\ell_i \circ \tau_i\ ^{i \in 1..n}, x.\ell'_j \bullet \sigma_j\ ^{j \in 1..m}\right\}} \quad (\tau = \{\ell_i : \tau_i\ ^{i \in 1..n}\},\ \sigma = \{\ell'_j : \sigma_j\ ^{j \in 1..m}\})$$

$$\text{(Comp Method Block)} \qquad \frac{|\Delta|, x : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\}, K_\bullet \vdash e_i : \tau_i \quad \forall_{i \in 1..n}}{\Delta \vdash x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}] : K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\}\}}$$

Fig. 9. Typing Rules for Composition Expressions

Rule (Comp Sequencing) combines the effect of two expressions. This rule shows the propagation of resources (K_c) from e_1 to e_2, meaning that e_2 handles these resources either by keeping them in K''' or by consuming them.

For a sequence of composition operations to be accepted in a compose expression it must denote a complete architecture, in particular the set of unsatisfied resources must be empty ($K_\circ = \emptyset$). The elimination of these resources from the types is captured by the typing of plug operations: they are typed as having a required resource ($\pi_2 \circ \tau$) that is not propagated to the set of provided resources. This denotes the satisfaction of internal dependencies in a composition. We now state and characterize type safety in λ_χ.

Type safety is a corollary of our main theorem (Theorem 1) that, as a side effect of the traditional progress and type preservation properties, implies the architectural soundness of configurators, components and instances (before and after a reconfiguration action). First, our language is extended with a distinguished value, wrong, to which an expression evaluates whenever a runtime error occurs. A runtime error is defined to occur whenever an operation is undefined, this includes usual cases such as: application of a value which is not an abstraction, assignment to a value which is not a location, selection of a field on a value which is not a record or does not possess the relevant label (notice that this case includes calling a method on a null reference), and so on. Essentially, we include all situations in which the operational semantics is undefined.

In order to prove subject reduction for expression evaluation, we rely on a auxiliary lemma that establishes subject reduction with respect to the application of composition operations, and is used when analysing the cases of the new and reconfig expressions, where composition and reconfiguration steps take place. The proof of this lemma relies on certain special type annotations, of the form $[\![\tau \Rightarrow \sigma]\!]$, that keep track, during the process of constructing an object instance from its generating component, of its unsatisfied required ports (τ) on one hand, and of the declared provided ports on the other hand (σ). The final type of the object being built is σ. The type τ is used to verify the

satisfaction of the required ports. We also need to define a notion of conformance between the structure of object instances and resource sets, in order to specify an invariant of the reconfiguration process, and relate such invariant with the result of the runtime matching test performed by the reconfig operation. All these ingredients are presented with full details in [15], and combined to prove Theorem 1.We write $\Gamma \vdash S$ to denote that Γ types heap S.

Theorem 1 (Subject Reduction). *Let $e \in \lambda_\chi \backslash \{\mathsf{nil}\}$ and S be a heap, such that e is a closed expression in S, $\mathsf{nil}(S) = \emptyset$. Let $\Gamma \vdash S$ and $\Gamma \vdash e : \tau$. If $(e; S \downarrow v; S')$ then*

a) There is a typing environment Γ' such that $\Gamma \vdash S'$ and $\Gamma' \vdash v : \tau$;
b) The value v is either an abstraction, a component, a configurator, or a location that maps to a record or an object, and
c) $\mathsf{nil}(S') = \emptyset$.

Notice that the addition of required and provided ports to an instance introduces nil values in the heap (*e.g.*, Rule App Requires). As expected, in well-formed architectures all such ports must become plugged to compatible implementations. Thus, from the theorem's assumption that nil does not occur in the source program, from the invariant $\mathsf{nil}(S) = \emptyset$, and a correct typing of the heap, we conclude that all instances must be structurally well-formed. From the fact that nil is not admissible as a result of an evaluation, we also conclude that "nil dereferencing errors" cannot occur.

5 Concluding Remarks

We have presented a small object-oriented component programming language, by adding to a λ-calculus with imperative records a arguably minimal set of language constructs to express component definition, using method blocks, other components, and connection operations as basic ingredients. Both configuration of components and reconfiguration of objects are uniformly represented at the semantic level by configurators, typed values that represent architectural change and play a role similar to makefiles or project templates in software development support systems. However, configurators and components are first-class values, that can constructed and manipulated dynamically. Therefore, our language is expressive enough to model many sophisticated software management operations, typical of component-based systems, involving dynamic composition, configuration and reconfiguration, even if the well-typed architectural programming fragment is computationally incomplete, for expected reasons. Our main result is a type system enforcing that well-typed programs do not go wrong; in our setting this implies not only absence of "method not implemented" errors, but also architectural consistency of dynamic composition and reconfiguration processes, as made precise by Theorem 1.

Although defined from rather standard language constructs (a λ-calculus with imperative records), our language does not seem straightforwardly encodable, in a type preserving way, in such a canonical language, due to the presence of intensional information at the level of types, to the particular notion of "staging" involved, related with architectural manipulations rather than with source level program manipulation, and to

the particular combination of static and dynamic type checking used. In particular, configurator values carry type information (*e.g.*, as Java class files do) and evaluation of configurator operations (Fig. 6) involve computing with intensional type information in order to ensure soundness of configurator composition and reconfiguration.

It would be interesting to investigate more flexible typing relations, involving subtyping and polymorphism, along the lines of [14], and particularly challenging to identify a natural and useful notion of subtyping for configurator values, given the intensional character of configurator types. At the level of the basic language, it is also conceivable, in principle, to extend the application of reconfiguration not only to objects, but also to component values, we refrained from pursuing that, because that does not seem to increase the expressiveness of our language, and lacks pragmatical motivation.

This work is partially supported by IST-3-016004-IP-09 Sensoria and by Microsoft Research Grant 2002-73. We also acknowledge many useful comments by the reviewers.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. on Progr. Languages and Systems*, 13(2), April 1991.
2. Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 2002.
3. Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.
4. Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proc. of the Int. Workshop on Unanticipated Software Evolution*, 2003.
5. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 1999.
6. Luca Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages*. ACM Press, 1997.
7. Sophia Drossopoulou, Ferruccio Damiani, Mariangolia Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *Proc. of the European Conf. on Object-Oriented Programming*, 2001.
8. Dominic Duggan. Type-based hot swapping of running modules. In *Proc. of the Int. Conf. on Functional Programming*, 2001.
9. S. Fagorzi and E. Zucca. A calculus for reconfiguration. In *On-line Proc. of the Int. Workshop Developments in Computational Models at ICALP*, 2005.
10. Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1998.
11. Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *Proc. of the Euro. Symp. on Programming*, 2002.
12. Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Proc. of the Euro. Symp. on Programming*, 2004.
13. João Costa Seco and Luís Caires. A basic model of typed components. In *Proc. of the European Conf. on Object-Oriented Programming*, Cannes, France, 2000. Springer-Verlag.
14. João Costa Seco and Luís Caires. Subtyping First-Class Polymorphic Components. In *Proc. of the Euro. Symp. on Programming*, Edinburgh, 2005. Springer-Verlag.
15. João Costa Seco and Luís Caires. Types for dynamic reconfiguration. Technical Report UNL-DI-1-2006, FCT-UNL, 2006.

16. Peter Sewell. Modules, abstract types, and distributed versioning. In *ACM Symp. on Principles of Programming Languages*, pages 236–247, New York, NY, USA, 2001. ACM Press.
17. Vugranam C. Sreedhar. Mixin'up components. In *Proceedings of the International Conference on Software Engineering*. ACM Press, 2002.
18. Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.
19. Joe Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *Proc. of the Euro. Symp. on Programming*, 1999.