Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 73–90
www.elsevier.com/locate/entcs

A Calculus of Components with Dynamic Type-Checking[1]

Sonia Fagorzi[2] and Elena Zucca[3]

DISI
University of Genova
Genova, Italy

Abstract

We present a simple module calculus modeling software composition in an open environment, where some components can be provided from the outside after execution has started. Operators for combining software components are as in previous module calculi; here, we focus on the new problems posed by the fact that components are not all available at compile time. In particular, we want to be able to statically check internal consistency of local code, by only specifying a required type for missing components, and then to perform dynamic checks which ensure that code received from the outside, which is assumed to travel with its type, can be successfully accepted, without requiring to type-check the whole code again.
We consider two alternative solutions. The former uses simple dynamic checks based on standard subtyping, that is, a component can be safely combined with local code if it provides the expected features, and all additional features are hidden, thus avoiding conflict problems. The latter preserves the semantics we would get having all components statically available, but requires a more involved type system based on constraints, where dynamic checks prevent conflicts.

Keywords: Module calculi, type systems, dynamic type-checking

1 Introduction

Component-based software systems are increasingly supporting reconfiguration features, allowing the system structure to dynamically change after starting execution of an application. Moreover, in an open scenario, some software components can become available (e.g., are received from a different site) only after execution has started. Hence, it is not possible to perform a global static analysis; still we would like to guarantee that execution will never crash.

A convenient, modular way for doing this, advocated for instance in [4] in the context of a coordination language for mobile processes that exchange object-

[1] Partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.

[2] Email: fagorzi@disi.unige.it

[3] Email: zucca@disi.unige.it

doi:10.1016/j.entcs.2006.09.032

oriented code, is by a combination of local static checks and dynamic checks, more precisely:

- Statically available code is checked and compiled by only relying on requirements on missing components (formally expressed by a type).
- Dynamically available components are equipped with their type, obtained by the previous phase.
- At execution time, when an external component is retrieved, it is accepted only if it satisfies the expected requirements; formally, this is expressed by a *subtyping* relation between the expected type and the provided type.
- The combination of static type system and dynamic checks via subtyping ensures that, if an external component is accepted, then it can be safely composed with the running application without any need of inspecting code again.

Though the schema above is clearly desirable and very abstract, that is, not bound to any specific language or system, few attempts have been made until now of formalization and investigation of related problems in a general framework for software composition. In this paper, we give a contribution in this direction by formalizing the above schema in the context of a rather general framework for software composition we have developed in previous work. In particular, our technical development here takes as starting point the R-calculus [1,7,8,9], a module calculus which improves over its direct predecessors [3,19] by allowing interleaving between execution of a module component and reconfiguration steps due to execution of module operators.

Here, in order to model an open environment, we add to the R-calculus a *receive* primitive, written $\mathsf{rcv}\,\tau$ where τ is the required type of the expression. [4] Then, we face the problem of defining, for the language extended in this way, a combination of static type system and dynamic checks guaranteeing safe composition, following the schema proposed above.

Note that, even though a $\mathsf{rcv}\,\tau$ expression can be considered as a formal parameter of type τ, this problem is not exactly the same we would have in the static case by, say, function abstraction. Indeed, in the case of dynamic retrieval, an important issue is also to reject incoming code in as few cases as possible. Hence, a possible policy for guaranteeing safe composition of a received software component with local code can be to modify its behaviour, whereas this would be non acceptable if the component was statically available.

A policy of this kind is adopted by the former solution to the problem we present, which uses simple dynamic checks based on standard subtyping. That is, a component can be safely combined with running code if it provides the expected features, and all additional features are hidden, avoiding conflict problems. The latter solution we present, instead, preserves the semantics we would get in the static case, but requires a more involved type system based on constraints, where dynamic checks

[4] Indeed, our aim here is to focus on the problem of type safe dynamic retrieval of code, hence we do not care about where this code comes from; an explicit process layer with standard send/receive primitives is considered in our subsequent paper [10].

$X, Y, Z, \ldots \in \mathsf{Name}$				**name**
$x, y, z, \ldots \in \mathsf{Var}$				**variable**
$e \in \mathsf{Exp}$	$::=$			**expression**
		$\ldots$		core expression
	$\mid$	x		variable
	$\mid$	$[\iota;\, o;\, \rho]$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset)$	basic module
	$\mid$	$[\iota;\, o;\, \rho \mid e]$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset,)$	basic configuration
	$\mid$	$e_1 + e_2$		sum
	$\mid$	$e \backslash_X$		delete
	$\mid$	$\mathsf{freeze}_X\, e$		freeze
	$\mid$	$e \downarrow_X$		run
	$\mid$	$e \uparrow$		result
ι	$:=$	$x_i{:}c\tau_i \overset{i \in I}{\mapsto} X_i$	$(X_i = X_j \Rightarrow c\tau_i = c\tau_j)$	**input** part
o	$:=$	$X_j \overset{j \in O}{\mapsto} e_j$		**output** part
ρ	$:=$	$x_l{:}c\tau_l \overset{l \in L}{\mapsto} e_l$		**local** part
$c\tau \in \mathsf{CType}$				**core type**

Fig. 1. R-calculus syntax

prevent conflicts. The two solutions, even if applied here to a fixed, though rather general, formalism, can be considered paradigmatic. We prove that both solutions are sound, in the sense that they prevent ill-formed combination of components, and formally compare the two approaches.

The paper is structured as follows. In Sect.2 we provide a brief informal introduction to the R-calculus. In Sect.3 we extend the calculus with the receive primitive, discuss informally the problem of guaranteeing safe composition in this case, and present the hiding-based solution. In Sect.4 we present the constraint-based solution and compare the two approaches. Finally, in Sect.5 we summarize the contribution of the paper and briefly discuss related and further work. To ease the reader, we have added in the Appendix the formal definition of the R-calculus, together with proofs.

2 An overview of the R-calculus

In this section we provide a brief introduction to the R-calculus [1,7,8,9] through examples. The syntax is given in Fig.1, whereas reduction and typing rules are reported in the Appendix.

The R-calculus is a parametric calculus, which can be instantiated over different *core* languages, as modeled by the first production. Terms of the calculus which are not core expressions denote either *modules* or *configurations*. For simplicity, we do not consider here higher-order modules or configurations (see the Conclusion for more comments).

A module models a software component, seen as a collection of entities whose nature depends on the underlying core language (in the examples we will just use integer expressions for simplicity), which can be either defined inside the module or *deferred*, that is, to be imported later when composing the module with others.

A *basic module* consists of three parts: a map ι from *deferred* variables to *input*

names, a map o from *output* names to expressions, and a map ρ from *local* variables to expressions. Names are used to refer to a module entity from outside (hence they are used by module operators), while variables, which are annotated with core types [5], are used to refer to an entity from inside the module.

A basic module, e.g., $[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Y \mapsto x + y;\ y \mapsto 2 + z]$, declares four kinds of entities: Z is *deferred*, since its name is only input; among entities defined inside the module, X is *virtual*, since its name is both input and output; Y is *frozen*, since its name is only output; finally, y is *local*, since it has no name, hence is only internally available.

Modules can be combined by the three operators of *sum*, *delete*, and *freeze*. Summing two modules means performing the union of deferred entities and the disjoint union of virtual, frozen and local entities. Conflicts among variables are solved by α-renaming. Deleting an output name from a module means that the corresponding definition is removed: as a consequence, a virtual entity becomes deferred, whereas a frozen entity just disappears. Freezing a virtual entity means that its input name disappears, and all variables mapped into it become local, taking as defining expression the current entity definition. As a consequence, the entity becomes frozen.

A configuration models a software component together with a running *program*, whose execution may refer to the entities declared by the component. A *basic configuration* is a pair consisting of a basic module and a core expression, modeling the program. For instance, $[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Y \mapsto x + y;\ y \mapsto 2 + z \,|\, y]$ is a basic configuration with program y.

A basic configuration can evolve by reduction steps at the core level of the program, or by replacing both local and virtual variables by their defining expressions. Moreover, it is possible to apply module simplification steps to the module part of a configuration. That is, module operators can be applied to configurations as well, and act as reconfiguration operators, in the sense that they allow to modify the context of a program during its execution.

Thanks to reconfiguration steps, a needed deferred entity can become available, as shown below:

$$
\begin{array}{l}
[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Y \mapsto x + y;\ y \mapsto 2 + z \,|\, y] \setminus_Y + [;\ Z \mapsto 3;\ ] \xrightarrow{(\text{LOCAL})} \\
[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Y \mapsto x + y;\ y \mapsto 2 + z \,|\, 2 + z] \setminus_Y + [;\ Z \mapsto 3;\ ] \xrightarrow{(\text{DEL})} \\
[x \mapsto X, z \mapsto Z;\ X \mapsto 0;\ y \mapsto 2 + z \,|\, 2 + z] + [;\ Z \mapsto 3;\ ] \xrightarrow{(\text{SUM})} \\
[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Z \mapsto 3;\ y \mapsto 2 + z \,|\, 2 + z] \xrightarrow{(\text{VIRTUAL})} \\
[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Z \mapsto 3;\ y \mapsto 2 + z \,|\, 2 + 3] \xrightarrow{(\text{CORE})} \\
[x \mapsto X, z \mapsto Z;\ X \mapsto 0, Z \mapsto 3;\ y \mapsto 2 + z \,|\, 5]
\end{array}
$$

The difference between virtual and frozen entities is that changes to their defining expressions during execution affect references to these entities only in the virtual case. For instance, assuming to replace program variable from left to right [6], the configuration

[5] We will omit type annotations when not necessary.

[6] Formally, assuming core evaluation contexts of the form $\mathsf{E} + e$ and $v + \mathsf{E}$, with v integer value.

$\tau \in \mathsf{Type}$	$::=$		**type**
		$c\tau$	core type
	$\mid$	$[\pi^\iota;\, \pi^o]$	module type
	$\mid$	$[\pi^\iota;\, \pi^o;\, c\tau]$	configuration type
π	$:=$	$X_i{:}c\tau_i^{i\in I}$	signature

Fig. 2. R-calculus types

$$[x \mapsto X, z \mapsto Z;\, X \mapsto 1;\, |\, x + z + x] \setminus_X + [;\, X \mapsto 2, Z \mapsto 0;\,]$$

reduces to $[x \mapsto X, z \mapsto Z;\, X \mapsto 2, Z \mapsto 0;\, |\, 1 + 0 + 2]$. Indeed, the definition of z becomes available only by performing the delete and sum operators, that also change the definition of x. Instead, the configuration

$$(\mathsf{freeze}_X [x \mapsto X, z \mapsto Z;\, X \mapsto 1;\, |\, x + z + x]) \setminus_X + [;\, X \mapsto 2, Z \mapsto 0;\,]$$

reduces to $[z \mapsto Z;\, X \mapsto 1;\, x \mapsto 1 |\, 1 + z + x] \setminus_X + [;\, X \mapsto 2, Z \mapsto 0;\,]$

and then to $[z \mapsto Z;\, X \mapsto 2, Z \mapsto 0;\, x \mapsto 1 |\, 1 + 0 + 1]$.

The *run* operator gets a basic configuration from a (basic) module, by starting the execution of one of its output entities. For instance, the expression

$$[;\, X \mapsto x;\, y \mapsto 1, x \mapsto 2 + y] \downarrow_X$$

reduces to the basic configuration $[;\, X \mapsto x;\, y \mapsto 1, x \mapsto 2 + y |\, x]$.

The *result* operator allows to extract the program from a configuration [7], hence, to get a core value as the result of an inner computation. For instance, the expression

$$[z \mapsto Z;\, Z \mapsto ([x \mapsto X;\, ;\, |\, x] + [;\, X \mapsto 2;\,]) \uparrow;\, |\, z + 1]$$

reduces to $[z \mapsto Z;\, Z \mapsto 2;\, |\, 2 + 1]$.

Types are defined in Fig.2, and include, besides core types, *module* and *configuration types*. A module type consists of an *input signature* π^ι and an *output signature* π^o. A signature is a sequence $X_i{:}c\tau_i^{i\in I}$ of pairs consisting of an entity name and a (core) type, where order and repetitions are immaterial. In a configuration type, the first two components have the same meaning as for module types, while $c\tau$ is the (core) type of the program running in the configuration. The typing rules are given in Fig.A.3 in the Appendix. They derive judgments of shape $\Gamma \vdash e : \tau$ where a context is an assignment of well-formed (core) types to variables.

The R-calculus enjoys usual progress and subject reduction properties (see in the Appendix), under the assumption that analogous properties hold for the core calculus as well.

3 Solving dynamic clashes by hiding

The R-calculus presented until now models applications which can be dynamically reconfigured, in the sense that the software components composing the application can be manipulated after execution has started. However, these components are all available from the beginning. We now consider a different scenario, in which some

[7] When it does no longer refer to entities declared by the component.

component is not available at application compile-time, but will be provided later during execution.

We model this in a simple way by adding a *receive* primitive, written $\mathsf{rcv}\,\tau$ where τ is the required type of the expression which will be dynamically retrieved, as shown in Fig.3.

$$e \in \mathsf{Exp} ::= \ldots \mid \mathsf{rcv}\,\tau \qquad \text{receive} \ \ (\text{RCV}) \ \frac{}{\Gamma \vdash \mathsf{rcv}\,\tau : \tau}$$

Fig. 3. Extended syntax and typing rule

For instance, in the configuration $c \triangleq [x{:}\mathsf{int} \mapsto X;\ Y \mapsto 1;\ |\,x+1] + \mathsf{rcv}\ [Y{:}\mathsf{int};\ X{:}\mathsf{int}]$, the second argument of the sum is not statically available, but is required to be a module having type $[Y{:}\mathsf{int};\ X{:}\mathsf{int}]$, that is, with an input and an output entity of type int.

Intuitively, we expect the following semantics for the receive primitive: a receive expression $\mathsf{rcv}\,\tau$ can be executed if/when [8] the external environment makes available some software component, which is assumed to travel with its type. [9] We model this by a labelled reduction step $\xrightarrow{e:\tau'}$. If the reduction step is legal (that is, the external code can be safely accepted, see below), then the receive expression is replaced by the component received from the outside. For example

$$c \xrightarrow{[y:\mathsf{int}\mapsto Y;\ X\mapsto 2;\,]:[Y:\mathsf{int};\ X:\mathsf{int}]} [x{:}\mathsf{int} \mapsto X;\ Y \mapsto 1;\ |\,x+1] + [y{:}\mathsf{int} \mapsto Y;\ X \mapsto 2;\,].$$

The type annotation τ in the receive primitive allows to statically type-check the local code before execution, and then to perform a run-time check. Indeed, external code is accepted only if it provides the expected functionalities, formally expressed by the type τ. Note that in this way code is never reinspected, since dynamic checks are performed on types.

However, requiring an exact match between required and dynamically available type, as in the above example, would be a too restrictive constraint, forcing to reject many components which could be safely composed with local code. A less restrictive requirement which seems rather natural is to accept dynamically available code whose type τ' is a subtype (in the sense of the standard width/depth subtyping relation) of the required type τ. For instance, in the previous example, one could safely accept the module $[;\ X \mapsto 2, Z \mapsto 3;\,]$, containing more output and less input entities than those required, obtaining $[x{:}\mathsf{int} \mapsto X;\ Y \mapsto 1;\ |\,x+1] + [;\ X \mapsto 2, Z \mapsto 3;\,]$ which is a well-typed expression.

However, assuming the simple semantics by replacement of the receive primitive described above, a standard subtyping rule is not enough. Indeed, as it is well-known in object and module calculi [15,3,4], there is the problem of unintentional clashes. For instance, if in the example above we receive the module $[;\ X \mapsto 2, Y \mapsto 0;\,]$ of type $[;\ X{:}\mathsf{int}, Y{:}\mathsf{int}]$, which is a subtype of the required type, then we would obtain

[8] We abstract here from the details of the communication mechanism.

[9] We assume here to trust the incoming type information to be correct: a more sophisticated approach would require a *proof*, as in [14]. Moreover, we assume the retrieved expression to be ground. Formally, we assume the judgment $\emptyset \vdash e : \tau'$ to hold.

the expression $[x{:}\mathsf{int} \mapsto X;\, Y \mapsto 1;\, |\, x+1] + [;\, X \mapsto 2, Y \mapsto 0;\,]$ which cannot be reduced since there are two conflicting definitions for Y.

In this section, we present a first solution to this problem, which keeps the type system of the R-calculus and the dynamic checks based on a standard subtyping relation described above, and solves conflicts by adopting a more involved semantics of the receive primitive. That is, all entities which were not explicitly required are hidden to local code. This choice is analogous to that made in [4]. In this way more external components can be accepted, but the result is different from that we would obtain if code was statically available.

The calculus which exploits the hiding-based solution is called $R^{\mathsf{rcv}}_{\mathsf{hide}}$. The new reduction rules are given in Fig.4, where, for $H = X_1 \dots X_n$, we write $\mathsf{freeze}_H e$ for $\mathsf{freeze}_{X_1} \dots \mathsf{freeze}_{X_n} e$ and $e\backslash_H$ for $e \setminus_{X_1} \dots \setminus_{X_n}$.

$$\text{(C-RCV)}\ \frac{}{\mathsf{rcv}\ c\tau_1 \xrightarrow{e:c\tau_2} e}\ c\tau_2 \leq_{\mathsf{core}} c\tau_1$$

$$\text{(RCV)}\ \frac{}{\mathsf{rcv}\ \tau_1 \xrightarrow{e:\tau_2} (\mathsf{freeze}_H e)\backslash_H}\ \begin{array}{l}\tau_2 \leq \tau_1 \\ \tau_i \equiv [\pi^i_i;\, \pi^o_i] \vee \tau_i \equiv [\pi^i_i;\, \pi^o_i;\, c\tau_i]\,, i \in \{1,2\} \\ H = \mathsf{dom}(\pi^o_2) \backslash \mathsf{dom}(\pi^o_1)\end{array}$$

$$\text{(ERR)}\ \frac{}{\mathsf{rcv}\ \tau_1 \xrightarrow{e:\tau_2} \mathsf{err}}\ \tau_2 \not\leq \tau_1$$

$$\frac{\pi^i_2 \leq \pi^i_1 \quad \pi^o_1 \leq \pi^o_2}{[\pi^i_1;\, \pi^o_1] \leq [\pi^i_2;\, \pi^o_2]} \qquad \frac{[\pi^i_1;\, \pi^o_1] \leq [\pi^i_2;\, \pi^o_2] \quad c\tau_1 \leq_{\mathsf{core}} c\tau_2}{[\pi^i_1;\, \pi^o_1;\, c\tau_1] \leq [\pi^i_2;\, \pi^o_2;\, c\tau_2]} \qquad \frac{J \subseteq I \quad \left\{c\tau_j \leq_{\mathsf{core}} c\tau'_j \mid j \in J\right\}}{X_i{:}c\tau_i^{i\in I} \leq X_j{:}c\tau_j'^{j\in J}}$$

Fig. 4. Semantics à la MoMi for the receive primitive and subtyping relation

Rules (C-RCV) and (RCV) model successful retrieval of a core and module/configuration expression, respectively. The reduction step can be performed only if the type of the incoming expression is a subtype of the required type (side-condition in both rules) and the effect is that the receive primitive is replaced by the external code. However, in case a module/configuration is received, this replacement takes place only after all non explicitly required output names have been hidden. The hiding operator can be expressed, as usual in module calculi [3], by a combination of freeze and delete; indeed, virtual entities need to be frozen before being deleted, in order to make their definitions local [10].

Rule (ERR) raises an error if the type of the incoming expression is not a subtype of the required type. We omit error propagation rules through one hole contexts, which are standard. Moreover, we assume that standard contextual closure (rule (E) in Fig.A.1 in the Appendix) also holds for labelled steps.

The subtyping relation is defined assuming a subtyping relation $\leq_{\mathsf{core}}$ on core types [11], and allows to replace a module or a configuration by another having less input and more output entities, with covariant subtyping for output and contravariant subtyping for input. For instance, in the configuration

$$c_1 \triangleq [x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\, Y \mapsto 1, Z \mapsto 2;\, |\, y + x] + \mathsf{rcv}\ [Y{:}\mathsf{int}, W{:}\mathsf{int};\, X{:}\mathsf{int}]$$

[10] The freeze operator has no effect on entities which are already frozen.

[11] We assume that $\leq_{\mathsf{core}}$ satisfies subsumption property, that is, $\Gamma \vdash e : c\tau_1$ and $c\tau_1 \leq_{\mathsf{core}} c\tau_2$ implies $\Gamma \vdash e : c\tau_2$.

the module $m \triangleq [y{:}\mathsf{int} \mapsto Y;\ X \mapsto 3+y, Y \mapsto 4, Z \mapsto 5;\ ]$ can be safely received, leading to:

$$c_1 \xrightarrow{m:[Y:\mathsf{int};\ X:\mathsf{int}, Y:\mathsf{int}, Z:\mathsf{int}]} [x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\ Y \mapsto 1, Z \mapsto 2;\ |\, y+x] + (\mathsf{freeze}_{Y,Z}\, m)\backslash_{Y,Z} \xrightarrow{(\text{FREEZE})} \xrightarrow{(\text{DEL})}$$
$$[x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\ Y \mapsto 1, Z \mapsto 2;\ |\, y+x] + [;\ X \mapsto 3+y;\ y \mapsto 4] \xrightarrow{(\text{SUM})}$$
$$[x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\ Y \mapsto 1, Z \mapsto 2, X \mapsto 3+y';\ y' \mapsto 4 |\, y+x] \longrightarrow \ldots$$

Note that, in case a deferred (that is, only input) entity is required, and a virtual entity is provided (as happens for Y in the example), since unexpected output entities are hidden, the virtual entity becomes local. This reflects the intuition that local code wants to supply a definition for Y ($Y \mapsto 1$ in the example) but, since external code already has its own definition ($Y \mapsto 4$), the latter takes the precedence. An alternative semantics giving precedence to local code would be obtained by replacing, in rule (RCV), $(\mathsf{freeze}_H e)\backslash_H$ by $e\backslash_H$. The example above would become:

$$c_1 \xrightarrow{m:[Y:\mathsf{int};\ X:\mathsf{int}, Y:\mathsf{int}, Z:\mathsf{int}]} [x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\ Y \mapsto 1, Z \mapsto 2;\ |\, y+x] + m\backslash_{Y,Z} \xrightarrow{(\text{DEL})}$$
$$[x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\ Y \mapsto 1, Z \mapsto 2;\ |\, y+x] + [y{:}\mathsf{int} \mapsto Y;\ X \mapsto 3+y;\ ] \xrightarrow{(\text{SUM})}$$
$$[x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y, y'{:}\mathsf{int} \mapsto Y;\ Y \mapsto 1, Z \mapsto 2, X \mapsto 3+y';\ |\, y+x] \longrightarrow \ldots$$

Analogously, a frozen entity can be provided when a virtual is required.

As the example shows, $R^{\mathsf{rcv}}_{\mathsf{hide}}$ does not preserve the semantics of the R-calculus. That is, when dynamically retrieving an expression the behaviour is different from that we would get if the expression was statically available (see Theorem 4.5 in the next section).

Soundness of the $R^{\mathsf{rcv}}_{\mathsf{hide}}$-calculus is guaranteed by soundness of the R-calculus (Theorems A.2 and A.3 in the Appendix), together with the theorem below which states that dynamic retrieval of an expression leads to a well-typed term.

Theorem 3.1 (Subject Reduction) *If* $\Gamma \vdash e_1 : \tau_1$ *and* $e_1 \xrightarrow{e:\tau} e_2$, *then* $\Gamma \vdash e_2 : \tau_2$, *for some* $\tau_2 \leq \tau_1$.

4 Preventing dynamic clashes by constraints checking

In this section we consider a different solution to the problem of unintentional clashes, which keeps a simple semantics by replacement for the receive primitive by means of a more sophisticated type system which prevents dynamic conflicts. The resulting new calculus is called $R^{\mathsf{rcv}}_{\mathsf{cstr}}$. Before giving the formal definitions, we illustrate the approach on a simple example. Consider the following configuration

$$c_2 \triangleq [x{:}\mathsf{int} \mapsto X, y{:}\mathsf{int} \mapsto Y;\ Z \mapsto 2;\ |\, y+x] + \mathsf{rcv}\ [W{:}\mathsf{int};\ X{:}\mathsf{int}]$$

In order to avoid conflicts with local code, the type of the expression to be retrieved, besides being a subtype of $[W{:}\mathsf{int};\ X{:}\mathsf{int}]$, should satisfy the following informal constraints:

- no output entity named Z should be present, since this would cause a conflict with the definition in c_2;
- in case an output entity named Y is present, its type should be int (otherwise we would get an ill-formed module type where the input type of a virtual entity is different from its output type).

In order to formally express these constraints, we assign to the receive subexpression a polymorphic (module) type of shape $[W{:}\mathsf{int};\ X{:}\mathsf{int}, r]$, where r is a *row variable* [18] which models the unknown additional features of code which will be retrieved at run time. Consequently, the type assigned to c_2 is the following *constrained polymorphic (configuration) type*:

$$Z{:}\mathsf{int}\#r, Y{:}\mathsf{int}\|r \Rightarrow [W{:}\mathsf{int}, Y{:}\mathsf{int};\ Z{:}\mathsf{int}, X{:}\mathsf{int}, r;\ \mathsf{int}]$$

where $Z{:}\mathsf{int}\#r$ and $Y{:}\mathsf{int}\|r$ mean that $Z{:}\mathsf{int}$ and r have disjoint domain[12] and that r assigns type int to Y entity, if any, respectively.

The row variable r will be replaced during execution by the actual unexpected output signature of the received module. For instance, if the module $[;\ X \mapsto 2, Y \mapsto 3;\ ]$ is dynamically retrieved, then r is replaced by $Y{:}\mathsf{int}$. This replacement is safe since the ground constraints $Y{:}\mathsf{int}\#Z{:}\mathsf{int}$ and $Y{:}\mathsf{int}\|Y{:}\mathsf{int}$ are valid. On the contrary, the module $[;\ X \mapsto 2, Z \mapsto 2;\ ]$ of type $[;\ X{:}\mathsf{int}, Z{:}\mathsf{int}]$ is rejected, since $Z{:}\mathsf{int}$ violates the first constraint.

We now formally define the $R^{\mathsf{rcv}}_{\mathsf{cstr}}$-calculus.

Type system

First, we introduce constrained polymorphic types in Fig.5. They are pairs

$\mathcal{C} \Rightarrow \hat{\tau}$		**constrained polymorphic type**
$\hat{\tau} \in \mathsf{Type}$	$::= c\tau \mid [\pi^{\iota};\ \hat{\pi}^{o}] \mid [\pi^{\iota};\ \hat{\pi}^{o};\ c\tau]$	**polymorphic type**
$\hat{\pi}$	$::= \pi, \mathcal{R}$	**polymorphic signature**
$\mathcal{R}$	$::= r_i^{i \in I}$	**row variables**
$\mathcal{C}$	$::= c_i^{i \in I}$	**constraints**
c	$::= \pi \| \hat{\pi} \mid \hat{\pi}_1 \# \hat{\pi}_2 \ \ (\mathsf{RVar}(\hat{\pi}_1) \cap \mathsf{RVar}(\hat{\pi}_2) = \emptyset)$	**constraint**

Fig. 5. $R^{\mathsf{rcv}}_{\mathsf{cstr}}$ constrained polymorphic types

consisting of a sequence of constraints $\mathcal{C}$ and a polymorphic type $\hat{\tau}$. Polymorphic types are a generalization of R-calculus types (as in Fig.2) where output signatures $\hat{\pi}^{o}$ may contain row variables (one for each receive subexpression). We conventionally write row variables at the end, and denote by $\mathsf{RVar}(\hat{\pi})$ the set of row variables inside $\hat{\pi}$, defined in the obvious way, and analogously for $\mathcal{C}$. Standard R-calculus types (types of terms which do not contain receive subexpressions) are obtained by taking empty constraint and row variable sequences. As explained above, the former constraint requires a non-polymorphic (input) signature to be compatible with a polymorphic (output) signature, whereas the latter requires two polymorphic (output) signatures to have disjoint domains. The condition $\mathsf{RVar}(\hat{\pi}^{o}_1) \cap \mathsf{RVar}(\hat{\pi}^{o}_2) = \emptyset$ prevents row variable clashes.

The type system is in Fig.6. It derives judgments of the form $\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow \hat{\tau}$, where a context $\hat{\Gamma}$ is an assignment of constrained polymorphic (core) types to variables. We assume to associate to each receive expression a fresh row variable. That is, an expression $\mathsf{rcv}\ [\pi^{\iota};\ \pi^{o}]$ becomes $\mathsf{rcv}\ [\pi^{\iota};\ \pi^{o}, r]$, with r fresh.

[12] Even though this form of constraint does not depend on the type of entities, we write a signature rather than just a set of names for uniformity with the other form.

We use the following abbreviations: $\pi_1 \| \pi_2$ for $\forall X \in \mathsf{dom}(\pi_1) \cap \mathsf{dom}(\pi_2).\, \pi_1(X) = \pi_2(X)$, that is, the two signatures agree on their common domain; $\pi_1^o \# \pi_2^o$ for $\mathsf{dom}(\pi_1^o) \cap \mathsf{dom}(\pi_2^o) = \emptyset$, that is, the two (output) signatures have disjoint domains. Typing rules are

$\ldots$ (typing rules for core operators) (VAR) $\dfrac{}{\hat{\Gamma} \vdash x : \hat{\Gamma}(x)}$

(M-BASIC) $$\frac{\left\{\hat{\Gamma}, x_i{:}\mathcal{C}_i \Rightarrow c\tau_i^{i \in I \cup L} \vdash e_h : \mathcal{C}_h \Rightarrow c\tau_h \mid h \in O \cup L\right\}}{\hat{\Gamma} \vdash \left[x_i{:}c\tau_i \overset{i \in I}{\mapsto} X_i;\ X_j \overset{j \in O}{\mapsto} e_j;\ x_l{:}c\tau_l \overset{l \in L}{\mapsto} e_l\right] : \mathcal{C}_k^{k \in O \cup L} \Rightarrow \left[X_i{:}c\tau_i^{i \in I};\ X_j{:}c\tau_j^{j \in O}\right]} \; X_i{:}c\tau_i^{i \in I} \| X_j{:}c\tau_j^{j \in O}$$

(M-SUM) $$\frac{\hat{\Gamma} \vdash e_1 : \mathcal{C}_1 \Rightarrow [\pi_1^\iota;\ \pi_1^o, \mathcal{R}_1] \quad \hat{\Gamma} \vdash e_2 : \mathcal{C}_2 \Rightarrow [\pi_2^\iota;\ \pi_2^o, \mathcal{R}_2]}{\hat{\Gamma} \vdash e_1 + e_2 : \mathcal{C}_1, \mathcal{C}_2, \pi_2^\iota \| \mathcal{R}_1, \pi_1^\iota \| \mathcal{R}_2, \pi_1^o \# \mathcal{R}_2, \pi_2^o \# \mathcal{R}_1, \mathcal{R}_1 \# \mathcal{R}_2 \Rightarrow [\pi_1^\iota, \pi_2^\iota;\ \pi_1^o, \pi_2^o, \mathcal{R}_1, \mathcal{R}_2]} \begin{array}{l} \pi_1^\iota \| \pi_2^\iota \\ \pi_1^o \# \pi_2^o \\ \pi_1^\iota \| \pi_2^o, \pi_2^\iota \| \pi_1^o \end{array}$$

(M-DEL) $$\frac{\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow [\pi^\iota;\ \pi^o, \mathcal{R}]}{\hat{\Gamma} \vdash e \backslash_X : \mathcal{C} \Rightarrow [\pi^\iota;\ \pi^o \backslash X, \mathcal{R}]} \; X \in \mathsf{dom}(\pi^o)$$

(M-FREEZE) $$\frac{\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow [\pi^\iota;\ \pi^o, \mathcal{R}]}{\hat{\Gamma} \vdash \mathsf{freeze}_X e : \mathcal{C} \Rightarrow [\pi^\iota \backslash X;\ \pi^o, \mathcal{R}]} \; X \in \mathsf{dom}(\pi^\iota) \Rightarrow X \in \mathsf{dom}(\pi^o)$$

(BASIC) $$\frac{\hat{\Gamma}, x_i{:}\mathcal{C}_i \Rightarrow c\tau_i^{i \in I \cup L} \vdash e : \mathcal{C} \Rightarrow c\tau \quad \left\{\hat{\Gamma}, x_i{:}\mathcal{C}_i \Rightarrow c\tau_i^{i \in I \cup L} \vdash e_h : \mathcal{C}_h \Rightarrow c\tau_h \mid h \in O \cup L\right\}}{\hat{\Gamma} \vdash \left[x_i{:}c\tau_i \overset{i \in I}{\mapsto} X_i;\ X_j \overset{j \in O}{\mapsto} e_j;\ x_l{:}c\tau_l \overset{l \in L}{\mapsto} e_l \mid e\right] : \mathcal{C}_k^{k \in O \cup L}, \mathcal{C} \Rightarrow \left[X_i{:}c\tau_i^{i \in I};\ X_j{:}c\tau_j^{j \in O};\ c\tau\right]} \; X_i{:}c\tau_i^{i \in I} \| X_h{:}c\tau_h^{h \in O}$$

(SUM) $$\frac{\hat{\Gamma} \vdash e_1 : \mathcal{C}_1 \Rightarrow [\pi_1^\iota;\ \pi_1^o, \mathcal{R}_1;\ c\tau] \quad \hat{\Gamma} \vdash e_2 : \mathcal{C}_2 \Rightarrow [\pi_2^\iota;\ \pi_2^o, \mathcal{R}_2]}{\hat{\Gamma} \vdash e_1 + e_2 : \mathcal{C}_1, \mathcal{C}_2, \pi_2^\iota \| \mathcal{R}_1, \pi_1^\iota \| \mathcal{R}_2, \pi_1^o \# \mathcal{R}_2, \pi_2^o \# \mathcal{R}_1, \mathcal{R}_1 \# \mathcal{R}_2 \Rightarrow [\pi_1^\iota, \pi_2^\iota;\ \pi_1^o, \pi_2^o, \mathcal{R}_1, \mathcal{R}_2;\ c\tau]} \begin{array}{l} \pi_1^\iota \| \pi_2^\iota \\ \pi_1^o \# \pi_2^o \\ \pi_1^\iota \| \pi_2^o, \pi_2^\iota \| \pi_1^o \end{array}$$

(DEL) $$\frac{\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow [\pi^\iota;\ \pi^o, \mathcal{R};\ c\tau]}{\hat{\Gamma} \vdash e \backslash_X : \mathcal{C} \Rightarrow [\pi^\iota;\ \pi^o \backslash X, \mathcal{R};\ c\tau]} \; X \in \mathsf{dom}(\pi^o)$$

(FREEZE) $$\frac{\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow [\pi^\iota;\ \pi^o, \mathcal{R};\ c\tau]}{\hat{\Gamma} \vdash \mathsf{freeze}_X e : \mathcal{C} \Rightarrow [\pi^\iota \backslash X;\ \pi^o, \mathcal{R};\ c\tau]} \; X \in \mathsf{dom}(\pi^\iota) \Rightarrow X \in \mathsf{dom}(\pi^o)$$

(RUN) $\dfrac{\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow [\pi^\iota;\ \hat{\pi}^o]}{\hat{\Gamma} \vdash e \downarrow_X : \mathcal{C} \Rightarrow [\pi^\iota;\ \hat{\pi}^o;\ \hat{\pi}^o(X)]}$ (RES) $\dfrac{\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow [\emptyset;\ \hat{\pi}^o;\ c\tau]}{\hat{\Gamma} \vdash e \Uparrow : \mathcal{C} \Rightarrow c\tau}$ (RCV) $\dfrac{}{\hat{\Gamma} \vdash \mathsf{rcv}\ \hat{\tau} : \emptyset \Rightarrow \hat{\tau}} \vdash \hat{\tau}$

Fig. 6. $R^{\mathsf{rcv}}_{\mathsf{cstr}}$ typing rules

analogous to those given for the R-calculus (see Fig.A.3 in the Appendix), with the difference that types assigned to expressions with receive subexpressions contain row variables in the output signature and may contain constraints. In particular, row variables are generated by the (RCV) rule, that assigns to a receive expression $\mathsf{rcv}\ \hat{\tau}$ the polymorphic type $\hat{\tau}$, which must be well-formed (side-condition), where well-formed polymorphic types are defined in Fig.8. Then, constraints are generated by rules (M-SUM) and (SUM), to guarantee that statically unknown output entities of an argument will be type compatible with deferred (only input) [13] entities of the other (first two constraints), and there are no conflicts between statically unknown output entities of an argument and output entities (both statically known and unknown) of the other (last three constraints). All other rules only propagate constraints.

Note that in rule (M-BASIC) and (BASIC) constraints inferred for module entities and

[13] Compatibility with input entities which are output as well (that is, are virtual) is implied by the stronger third and fourth constraints.

$$(\text{RCV})\ \frac{}{\langle \mathsf{rcv}\ \hat{\tau}_1, \mathcal{C}_1\rangle \xrightarrow{e:\mathcal{C}_2\Rightarrow\hat{\tau}_2} \langle e, \mathcal{C}, \mathcal{C}_2\rangle} \quad \begin{array}{l}\mathsf{RVar}(\mathcal{C}_1\Rightarrow\hat{\tau}_1)\cap\mathsf{RVar}(\mathcal{C}_2\Rightarrow\hat{\tau}_2)=\emptyset\\ \mathcal{C},\mathcal{C}_2\Rightarrow\hat{\tau}_2\leq\mathcal{C}_1\Rightarrow\hat{\tau}_1\end{array}$$

$$(\text{ERR})\ \frac{}{\langle \mathsf{rcv}\ \hat{\tau}_1, \mathcal{C}_1\rangle \xrightarrow{e:\mathcal{C}_2\Rightarrow\hat{\tau}_2} \mathsf{err}} \quad \begin{array}{l}\mathsf{RVar}(\mathcal{C}_1\Rightarrow\hat{\tau}_1)\cap\mathsf{RVar}(\mathcal{C}_2\Rightarrow\hat{\tau}_2)=\emptyset\\ \mathcal{C},\mathcal{C}_2\Rightarrow\hat{\tau}_2\not\leq\mathcal{C}_1\Rightarrow\hat{\tau}_1\end{array}$$

$$\mathcal{C}\Rightarrow\hat{\tau}\leq\mathcal{C}'\Rightarrow\hat{\tau}' \quad \text{iff} \quad \mathcal{C}\vdash\mathcal{C}'\{\rho\} \text{ and } \hat{\tau}\leq_\rho\hat{\tau}' \text{ (for some } \rho)$$

$$\frac{\mathcal{C}_1'\vdash\mathcal{C}_1 \quad \mathcal{C}_2'\vdash\mathcal{C}_2}{\mathcal{C}_1',\mathcal{C}_2'\vdash\mathcal{C}_1,\mathcal{C}_2} \qquad \frac{}{c\vdash\emptyset} \qquad c ::= \pi^\iota\|\mathcal{R} \mid \pi^o\#\mathcal{R} \mid \mathcal{R}_1\#\mathcal{R}_2 \qquad \frac{}{\pi^\iota\|\mathcal{R}\vdash\pi^\iota\|\pi^o,\mathcal{R}} \quad \pi^\iota\|\pi^o$$

$$\frac{}{\pi_1^o\#\mathcal{R}_2,\pi_2^o\#\mathcal{R}_1,\mathcal{R}_1\#\mathcal{R}_2\vdash\pi_1^o,\mathcal{R}_1\#\pi_2^o,\mathcal{R}_2} \quad \pi_1^o\#\pi_2^o$$

$$\frac{\pi_2^\iota\leq\pi_1^\iota \quad \pi_1^o\leq\pi_2^o}{[\pi_1^\iota;\pi_1^o,\hat{\pi}^o]\leq_\rho[\pi_2^\iota;\pi_2^o,\mathcal{R}]} \quad \mathsf{dom}(\pi_1^o)=\mathsf{dom}(\pi_2^o) \quad \rho(\mathcal{R})=\hat{\pi}^o \qquad \frac{[\pi_1^\iota;\hat{\pi}_1^o]\leq_\rho[\pi_2^\iota;\hat{\pi}_2^o] \quad c\tau_1\leq_{\mathsf{core}}c\tau_2}{[\pi_1^\iota;\hat{\pi}_1^o,c\tau_1]\leq_\rho[\pi_2^\iota;\hat{\pi}_2^o,c\tau_2]}$$

Fig. 7. Semantics with dynamic checks for the receive primitive and subtyping relation

program are propagated to the whole module/configuration. For instance, the configuration

$$[x{:}\,\mathsf{int}\mapsto X;\ X\mapsto(\mathsf{rcv}\ [;\ Y{:}\,\mathsf{bool},r]+[y{:}\,\mathsf{bool}\mapsto Y;\ P\mapsto\mathsf{if}\ y\ \mathsf{then}\ 0\ \mathsf{else}\ 1;\ ]\downarrow_P)\uparrow;\ |\,x].$$

can be safely reduced only under the constraint $P{:}\,\mathsf{int}\#r$.

Semantics

The semantics is given on pairs of the form $\langle e,\mathcal{C}\rangle$, where $\mathcal{C}$ is assumed to contain all constraints inferred during the type-checking of e (see Theorem 4.2 below). Fig.7 contains reduction rules for the receive primitive. Other rules are omitted since they are analogous to those of the R-calculus (see Fig.A.1 in the Appendix) with the constraints component unchanged. Note that, since this also holds for contextual closure, in reduction rules for receive expressions in Fig.7 constraints are *global*, that is, are those of the enclosing top-level expression.

Rule (RCV) deals with the dynamic retrieval of an expression. The former side-condition prevents row variable clashes between local and external code and can always be satisfied by an appropriate α-conversion. The latter side-condition allows the reduction step only if type $\hat{\tau}_2$ is a subtype of the required type $\hat{\tau}_1$ modulo substitution of the row variable with the unexpected output signature, and, moreover, all required constraints are valid by applying this substitution. This is formally modeled by the subtyping relation in Fig.7.

For instance, the expression $\mathsf{rcv}\ [;\ X{:}\,\mathsf{int},r']+[x{:}\,\mathsf{int}\mapsto X;\ P\mapsto x+1;\ ]$ requires the constraint $P{:}\,\mathsf{int}\#r'$. Reduction under this constraint can proceed by dynamically retrieving the module $\mathsf{freeze}_Z([z{:}\,\mathsf{int}\mapsto Z;\ X\mapsto z*2,W\mapsto 1;\ ]+\mathsf{rcv}\ [;\ Z{:}\,\mathsf{int},r])$ of type $X{:}\,\mathsf{int},W{:}\,\mathsf{int}\#r\Rightarrow[;\ X{:}\,\mathsf{int},W{:}\,\mathsf{int},Z{:}\,\mathsf{int},r]$ (r' is replaced by $W{:}\,\mathsf{int},Z{:}\,\mathsf{int},r$ and the ground constraint $P{:}\,\mathsf{int}\#W{:}\,\mathsf{int},Z{:}\,\mathsf{int}$ is satisfied). After the (RCV) step, we get an expression requiring the constraints $P{:}\,\mathsf{int}\#r,X{:}\,\mathsf{int},W{:}\,\mathsf{int}\#r$ which, indeed, have been added in the current constraint sequence.

Rule (ERR) raises an error err if the required subtyping relation does not hold. We

omit error propagation rules through one hole contexts, which are standard.

Note that constraints are solved incrementally, as shown by the following example. Consider the configuration $c_3 \triangleq \mathsf{rcv}\ [X{:}\mathsf{int};\ Y{:}\mathsf{int}, r_1] + \mathsf{rcv}\ [Y{:}\mathsf{int};\ Z{:}\mathsf{int}, r_2]$, for which constraints $\mathcal{C} = X{:}\mathsf{int} \| r_2, Z{:}\mathsf{int} \# r_1, Y{:}\mathsf{int} \# r_2, r_1 \# r_2$ are inferred. Assume to reduce sum arguments from left to right and to first receive a module $m = [x{:}\mathsf{int} \mapsto X;\ Y \mapsto 0, W \mapsto 1;\ ]$; then, since $Z{:}\mathsf{int} \# W{:}\mathsf{int}$, constraints can be successfully simplified as follows:

$$\langle c_3, \mathcal{C} \rangle \xrightarrow{m\,:\,\emptyset \Rightarrow [X{:}\mathsf{int};\ Y{:}\mathsf{int}, W{:}\mathsf{int}]}$$
$$\langle [x{:}\mathsf{int} \mapsto X;\ Y \mapsto 0, W \mapsto 1;\ ] + \mathsf{rcv}\ [Y{:}\mathsf{int};\ Z{:}\mathsf{int}, r_2]\ , X{:}\mathsf{int} \| r_2, Y{:}\mathsf{int} \# r_2, W{:}\mathsf{int} \# r_2 \rangle.$$

Assume to receive now the module $[;\ Z \mapsto 2, W \mapsto 1;\ ]$, whose type is a subtype of the required type. Constraints are not satisfied, hence we get an error.

Results

First of all, types inferred by the type system are well-formed, in the sense that their non polymorphic parts are well-formed R-calculus types (see Fig.8). Well-formedness of R-calculus types is defined in Fig.A.2 in the Appendix, and means that signatures π^ι and π^o together define a map from entity names into well-formed (core) types (this implies in particular that the input and output signature agree on the type of a virtual entity). Moreover, constraints inferred by the type system are *in normal form*, in the sense that they do not contain ground constraints whose validity could be checked. This is formalized by the proposition below.

$$\frac{\vdash [\pi^\iota;\ \pi^o]}{\vdash [\pi^\iota;\ \pi^o, \mathcal{R}]} \qquad \frac{\vdash [\pi^\iota;\ \pi^o;\ c\tau]}{\vdash [\pi^\iota;\ \pi^o, \mathcal{R};\ c\tau]}$$

Fig. 8. $R^{\mathsf{rcv}}_{\mathsf{cstr}}$-calculus well-formed types

Proposition 4.1 *If* $\hat{\Gamma} \vdash e : \mathcal{C} \Rightarrow \hat{\tau}$, *then* $\mathcal{C} \vdash \mathcal{C}$ *and* $\vdash \hat{\tau}$.

Soundness of the $R^{\mathsf{rcv}}_{\mathsf{cstr}}$-calculus is guaranteed by soundness of the R-calculus, together with the theorem below which states that dynamic retrieval of an expression leads to a well-typed term.

Theorem 4.2 (Subject Reduction) *If* $\hat{\Gamma} \vdash e_1 : \mathcal{C}_1 \Rightarrow \hat{\tau}_1$ *and* $\langle e_1, \mathcal{C}_1 \rangle \xrightarrow{e:\mathcal{C} \Rightarrow \hat{\tau}} \langle e_2, \mathcal{C}_2 \rangle$, *then* $\hat{\Gamma} \vdash e_2 : \mathcal{C}_2 \Rightarrow \hat{\tau}_2$, *with* $\mathcal{C}_2 \Rightarrow \hat{\tau}_2 \leq \mathcal{C}_1 \Rightarrow \hat{\tau}_1$.

Finally, we state some results of comparison between the approach based on hiding and that based on constraint checking. To this end, we denote by $||\_||$ an erasure function forgetting row variables in expressions and types; moreover, we add an index in reductions and type judgments, to make clear to which approach they are related.

The following proposition states that the constraint-based system derives types which are an extension with row variables and constraints of those derived by the hiding-based system.

Proposition 4.3 $\hat{\Gamma} \overset{\mathsf{cstr}}{\vdash} e : \mathcal{C} \Rightarrow \hat{\tau}$ *iff* $||\hat{\Gamma}|| \overset{\mathsf{hide}}{\vdash} ||e|| : ||\hat{\tau}||$.

The following theorem states that the hiding-based approach is strictly more permissive than the constraint-based, in the sense that more external code is accepted.

Theorem 4.4

- *If* $\hat{\Gamma} \overset{\mathsf{cstr}}{\vdash} e_1 : \mathcal{C}_1 \Rightarrow \hat{\tau}_1$, *and* $\langle e_1, \mathcal{C}_1 \rangle \xrightarrow[\mathsf{cstr}]{e:\mathcal{C}\Rightarrow\hat{\tau}} \langle e_2, \mathcal{C}_2 \rangle$, *then* $||e_1|| \xrightarrow[\mathsf{hide}]{e:||\hat{\tau}||} e_3$, *for some* e_3.
- *The converse does not hold.*

Finally, the following theorem states that the constraint-based approach preserves the R-calculus semantics, in the sense that a term containing receive subexpressions reduces, by dynamically retrieving code, to the same term we would get if code was statically available. This property does not hold for the $R^{\mathsf{rcv}}_{\mathsf{hide}}$-calculus (see for instance the reduction sequence for c_1 in Sect.3). Here, we denote by $\mathsf{E}[\,]_1 \ldots [\,]_n$ a context with n holes, and we abbreviate by $\xrightarrow{e_1:\tau_1 \ldots e_n:\tau_n}$ a reduction sequence $\xrightarrow{*} \xrightarrow{e_1:\tau_1} \xrightarrow{*} \ldots \xrightarrow{*} \xrightarrow{e_n:\tau_n} \xrightarrow{*}$.

Theorem 4.5 *Given* $e = \mathsf{E}[\mathsf{rcv}\,\tau_1]_1 \ldots [\mathsf{rcv}\,\tau_n]_n$ *s.t.* $\hat{\Gamma} \overset{\mathsf{cstr}}{\underset{\mathsf{hide}}{\vdash}} e : \mathcal{C} \Rightarrow \hat{\tau}$ *holds (or, equivalently by Prop.4.3,* $||\hat{\Gamma}|| \overset{\mathsf{cstr}}{\underset{\mathsf{hide}}{\vdash}} ||e|| : ||\hat{\tau}||$ *),*

- *if* $\langle e, \mathcal{C} \rangle \xrightarrow[\mathsf{cstr}]{e_1:\tau_1 \ldots e_n:\tau_n} \langle e', \mathcal{C}' \rangle$, *then* $\mathsf{E}[e_1]_1 \ldots [e_n]_n \xrightarrow{*} e'$.
- *if* $e \xrightarrow[\mathsf{hide}]{e_1:\tau_1 \ldots e_n:\tau_n} e'$, *then* $\mathsf{E}[e_1]_1 \ldots [e_n]_n \xrightarrow{*} e'$ *does not hold.*

5 Conclusion

The main contribution of the paper is to illustrate, in a simple and expressive formal framework for software composition, two different approaches to the problem of guaranteeing safe composition of software components retrieved at run-time with local code.

Both approaches are based on run-time checks which compare requirements statically made on external code with functionalities actually provided at run-time, and model this comparison by a relation between types. The former keeps dynamic checks simple, and solves interference problems by hiding functionalities which were not explicitly required. As already mentioned, this approach is advocated in [4] with the motivation that, whereas static name clashes should be considered as errors, dynamic name clashes should be allowed since they represent only an homonymy problem to be solved by static bindings. However, a drawback of this solution is that it does not preserve the semantics we would get if code would have been statically available. The latter solution, on the contrary, preserves the semantics we would get in the static case, by means of a more involved type system based on constraints, where dynamic checks prevents conflicts.

Our aim here is not to defend either choice, but rather to show that both can be modeled in a rather simple way, proved to be sound, and formally compared, in our calculus. In particular, it is worth to note that in the general and powerful frame-

work for software composition provided by mixin modules the operation consisting in solving clashes by renaming can be nicely internalized in terms of the primitive language operators, whereas in previous work [11,4] this operation can be only defined at a metatheory level and sometimes in a rather tricky way. However, by a preliminary analysis there seems to be no trivial way to generalize this approach to the higher-order case, since we would need an hiding operator propagating to subcomponents as well, which can be hardly be expressed as a derived module operator. This can be seen as another drawback of the hiding-based approach versus the constraint-based, which can be immediately generalized to higher-order modules.

An hybrid solution could adopt the constraint-based type system given for $R^{\mathsf{rcv}}_{\mathsf{cstr}}$, but with a different treatment of unexpected output components violating constraints, which could be hidden as in $R^{\mathsf{rcv}}_{\mathsf{hide}}$ rather than just rejecting external code.

As final remark, it is worth to mention that the problem of name clashes is, of course, not significant if components are just put side by side as black boxes, and ambiguity solved by using qualified names; however, in this case references to other components are *hard links*. Instead, modularity and software reuse can be greatly improved if a component's code abstracts from used components, by just specifying the services they should provide, hence can be possibly instantiated in many contexts, as, e.g., in the component frameworks for Java-like languages proposed in [2].

As already mentioned, the schema we adopt here, based on a combination of static typechecking of local code and dynamic checks via subtyping is the same as in MoMi [4]. However, there are many differences with this work. First, in MoMi exchanged code is object-oriented (more precisely, classes or mixin classes), whereas here we are interested in a more general framework for software composition where entities which are exchanged are mixin modules on an arbitrary core language. Note, in particular, that this allows symmetric composition (by sum) of local and external code, differently from MoMi where composition is always in one direction (instantiation of a mixin class on a parent). Moreover, in MoMi there is an explicit language for the process layer, different from the language used for the object-oriented code; here instead for simplicity we have just a receive primitive integrated in the language, since this addition was enough in order to study the problem of dynamic checks.

Other work which has directly inspired this paper is that on dynamic software updating in, e.g., [5,17,12]. However, our approach here is in the context of module/fragment calculi rather than lambda-calculi, and we take as basic primitive a receive primitive for retrieving external code on demand, whereas in [5,17,12] the basic primitive is an *update* primitive which when performed changes some part of the local code in a less controlled way.

The type system based on constraints used in the latter solution is similar to that defined in [13] for getting principal typings for mixin modules. However, here more simple constraints are enough since we consider an explicitly typed module calculus, where the only source of polymorphism is the receive primitive which is,

however, also annotated with the required components. A precise comparison with the type system defined in [13] is subject of further work.

Finally, other works on formalization of components are, e.g., [6,16].

Other directions for further work, are the integration of our approach with an explicit language for the process layer, as in MoMi, and, as mentioned above, the extension to a higher-order calculus. Finally, even though the framework for software composition offered by the R-calculus is rather general and powerful, we are bound to a fixed arbitrary set of operators; hence, we would like to investigate an even more abstract framework for expressing and studying safe combination of local and retrieved code via dynamic checks. All these directions have already been partly investigated in [10].

References

[1] Ancona, D., S. Fagorzi and E. Zucca, *Mixin modules for dynamic rebinding*, in: R. D. Nicola and D. Sangiorgi, editors, *TGC 2005 - Trustworthy Global Computing, International Symposium, Revised Selected Papers*, number 3705 in Lecture Notes in Computer Science (2005), pp. 279–298.

[2] Ancona, D., G. Lagorio and E. Zucca, *Flexible type-safe linking of components for Java-like languages*, in: *JMLC'06 - Joint Modular Languages Conference*, Lecture Notes in Computer Science (2006), to appear.

[3] Ancona, D. and E. Zucca, *A calculus of module systems*, Journ. of Functional Programming **12** (2002), pp. 91–132.

[4] Bettini, L., B. Venneri and V. Bono, *MOMI: a calculus for mobile mixins*, Acta Informatica **42** (2005), pp. 143–190.

[5] Bierman, G., M. W. Hicks, P. Sewell, G. Stoyle and K. Wansbrough, *Dynamic rebinding for marshalling and update, with destruct-time* λ, in: C. Runciman and O. Shivers, editors, *Intl. Conf. on Functional Programming 2003* (2003), pp. 99–110.

[6] Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani, *An open component model and its support in Java*, in: *CBSE'04 - Component-Based Software Engineering*, 2004, pp. 7–22.

[7] Fagorzi, S., "Module Calculi for Dynamic Reconfiguration," Ph.D. thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova (2005).

[8] Fagorzi, S. and E. Zucca, *A calculus for reconfiguration*, Electronic Notes in Theoretical Computer Science **135** (2005), DCM 2005 - International Workshop on Developments in Computational Models.

[9] Fagorzi, S. and E. Zucca, *A calculus for reconfiguration*, Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova (2005), extended version of [8], submitted for journal publication.

[10] Fagorzi, S. and E. Zucca, *A framework for type safe exchange of mobile code*, Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova (2006), submitted for publication.

[11] Flatt, M., S. Krishnamurthi and M. Felleisen, *Classes and mixins*, in: *ACM Symp. on Principles of Programming Languages 1998* (1998), pp. 171–183.

[12] Hicks, M. W. and S. Nettles, *Dynamic software updating.*, ACM Transactions on Programming Languages and Systems **27** (2005), pp. 1049–1096.

[13] Makholm, H. and J. B. Wells, *Type inference, principal typings, and let-polymorphism for first-class mixin modules*, in: O. Danvy and B. Pierce, editors, *Intl. Conf. on Functional Programming 2005* (2005), pp. 156–167.

[14] Necula, G. C., *Proof-carrying code.*, in: *ACM Symp. on Principles of Programming Languages 1997* (1997), pp. 106–119.

[15] Riecke, J. G. and C. A. Stone, *Privacy via subsumption*, Information and Computation **172** (2002), pp. 2–28.

[16] Schmitt, A. and J.-B. Stefani, *The kell calculus: A family of higher-order distributed process calculi*, in: C. Priami and P. Quaglia, editors, *GC2004 - Global Computing IST/FET International Workshop - Revised Selected Papers*, Lecture Notes in Computer Science **3267** (2005), pp. 146 – 178.

[17] Stoyle, G., M. W. Hicks, G. Bierman, P. Sewell and I. Neamtiu, *Mutatis mutandis: safe and predictable dynamic software updating*, in: *ACM Symp. on Principles of Programming Languages 2005* (2005), pp. 183–194.

[18] Wand, M., *Complete type inference for simple objects*, in: *Proc. IEEE Symp. on Logic in Computer Science 1987*, 1987, pp. 37–44, a corrigendum appeared at LICS 1988.

[19] Wells, J. B. and R. Vestergaard, *Confluent equational reasoning for linking with first-class primitive modules*, in: *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science (2000), pp. 412–428.

A Formal definition of the R-calculus

We report semantics and type system of the R-calculus in Fig.A.1-A.2-A.3-A.4, and state related results. We refer to [1,7,8,9] for the formal (standard) definitions of free variables of an expression $\mathsf{FV}(e)$, hole binders of a context $\mathsf{HB}(\mathsf{E})$, capture avoiding hole substitution $\mathsf{E}\{e\}$ and proofs.

One hole contexts

$\mathsf{E} ::= \Box \mid \overset{\text{(core contexts)}}{\cdots} \mid [\iota;\, X \mapsto \mathsf{E}, o;\, \rho] \mid [\iota;\, o;\, x \mapsto \mathsf{E}, \rho] \mid [\iota;\, X \mapsto \mathsf{E}, o;\, \rho \mid e] \mid [\iota;\, o;\, \rho \mid e] \mid [\iota;\, o;\, \rho \mid \mathsf{E}] \mid \mathsf{E} + e \mid e + \mathsf{E} \mid \mathsf{E} \setminus_X \mid \mathsf{freeze}_X \mathsf{E} \mid \mathsf{E}{\downarrow_X} \mid \mathsf{E}{\uparrow}$

$$(\mathsf{E})\ \frac{e \longrightarrow e'}{\mathsf{E}[e] \longrightarrow \mathsf{E}[e']} \quad \ldots \text{ (rules for core terms)}$$

Module simplification

$$(\text{M-SUM})\ \frac{}{[\iota_1;\, o_1;\, \rho_1] + [\iota_2;\, o_2;\, \rho_2] \longrightarrow [\iota_1, \iota_2;\, o_1, o_2;\, \rho_1, \rho_2]} \quad \begin{array}{l}\mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{FV}([\iota_2;\, o_2;\, \rho_2]) = \emptyset \\ \mathsf{dom}(\iota_2, \rho_2) \cap \mathsf{FV}([\iota_1;\, o_1;\, \rho_1]) = \emptyset\end{array}$$

$$(\text{M-DEL})\ \frac{}{[\iota;\, o;\, \rho] \setminus_X \longrightarrow [\iota;\, o \setminus X;\, \rho]} \quad X \in \mathsf{dom}(o)$$

$$(\text{M-FREEZE})\ \frac{}{\mathsf{freeze}_X [\iota;\, o;\, \rho] \longrightarrow \left[\iota \setminus F;\, o;\, \rho, x \overset{x \in F}{\mapsto} o(X)\right]} \quad \begin{array}{l} F = \{x \mid \iota(x) = X\} \\ F \neq \emptyset \Rightarrow X \in \mathsf{dom}(o)\end{array}$$

Reconfiguration

$$(\text{SUM})\ \frac{}{[\iota_1;\, o_1;\, \rho_1 \mid e] + [\iota_2;\, o_2;\, \rho_2] \longrightarrow [\iota_1, \iota_2;\, o_1, o_2;\, \rho_1, \rho_2 \mid e]} \quad \begin{array}{l}\mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{FV}([\iota_2;\, o_2;\, \rho_2]) = \emptyset \\ \mathsf{dom}(\iota_2, \rho_2) \cap \mathsf{FV}([\iota_1;\, o_1;\, \rho_1]) = \emptyset\end{array}$$

$$(\text{DEL})\ \frac{}{[\iota;\, o;\, \rho \mid e] \setminus_X \longrightarrow [\iota;\, o \setminus X;\, \rho \mid e]} \quad X \in \mathsf{dom}(o)$$

$$(\text{FREEZE})\ \frac{}{\mathsf{freeze}_X [\iota;\, o;\, \rho \mid e] \longrightarrow \left[\iota \setminus F;\, o;\, \rho, x \overset{x \in F}{\mapsto} o(X) \mid e\right]} \quad \begin{array}{l} F = \{x \mid \iota(x) = X\} \\ F \neq \emptyset \Rightarrow X \in \mathsf{dom}(o)\end{array}$$

$$(\text{LOCAL})\ \frac{}{[\iota;\, o;\, \rho \mid \mathsf{E}[x]] \longrightarrow [\iota;\, o;\, \rho \mid \mathsf{E}\{\rho(x)\}]} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathsf{E}) \\ x \in \mathsf{dom}(\rho)\end{array}$$

$$(\text{VIRTUAL})\ \frac{}{[\iota;\, o;\, \rho \mid \mathsf{E}[x]] \longrightarrow [\iota;\, o;\, \rho \mid \mathsf{E}\{o(\iota(x))\}]} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathsf{E}) \\ x \in \mathsf{dom}(\iota) \wedge \iota(x) \in \mathsf{dom}(o)\end{array}$$

Run and result

$$(\text{RUN})\ \frac{}{[\iota;\, o;\, \rho]{\downarrow_X} \longrightarrow [\iota;\, o;\, \rho \mid o(X)]} \qquad (\text{RES})\ \frac{}{(\mathsf{R}[\iota;\, o;\, \rho \mid e]){\uparrow} \longrightarrow e} \quad \mathsf{FV}(e) \cap \mathsf{dom}(\iota, \rho) = \emptyset$$

Reconfiguration contexts

$\mathsf{R} ::= \Box \mid \mathsf{R} + e \mid \mathsf{R} \setminus_X \mid \mathsf{freeze}_X \mathsf{R}$

Fig. A.1. R-calculus reduction rules

$$\frac{\vdash \pi^\iota, \pi^o \quad \overset{\text{core}}{\vdash}\ c\tau}{\vdash [\pi^\iota;\ \pi^o;\ c\tau]} \qquad \frac{\vdash \pi^\iota, \pi^o}{\vdash [\pi^\iota;\ \pi^o]} \qquad \frac{\left\{\overset{\text{core}}{\vdash}\ c\tau_i \mid i \in I\right\}}{\vdash X_i : c\tau_i^{i\in I}}\ \forall h,k \in I. X_h = X_k \Rightarrow c\tau_h = c\tau_k$$

Fig. A.2. R-calculus well-formed types

... (typing rules for core operators)

$$(\text{VAR})\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$(\text{M-BASIC})\ \frac{\left\{\Gamma, x_i : c\tau_i^{i\in I\cup L} \vdash e_h : c\tau_h \mid h \in O \cup L\right\}}{\Gamma \vdash \left[x_i : c\tau_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in O}{\mapsto} e_j;\ x_l : c\tau_l \overset{l\in L}{\mapsto} e_l\right] : \left[X_i : c\tau_i^{i\in I};\ X_j : c\tau_j^{j\in O}\right]}\ X_i : c\tau_i^{i\in I} \| X_j : c\tau_j^{j\in O}$$

$$(\text{M-SUM})\ \frac{\Gamma \vdash e_1 : [\pi_1^\iota;\ \pi_1^o] \quad \Gamma \vdash e_2 : [\pi_2^\iota;\ \pi_2^o]}{\Gamma \vdash e_1 + e_2 : [\pi_1^\iota, \pi_2^\iota;\ \pi_1^o, \pi_2^o]}\ \begin{array}{l}\pi_1^\iota \| \pi_2^\iota \\ \pi_1^o \# \pi_2^o \\ \pi_1^\iota \| \pi_2^o, \pi_2^\iota \| \pi_1^o\end{array} \qquad (\text{M-DEL})\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o]}{\Gamma \vdash e \backslash_X : [\pi^\iota;\ \pi^o \backslash X]}\ X \in \mathsf{dom}(\pi^o)$$

$$(\text{M-FREEZE})\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o]}{\Gamma \vdash \mathsf{freeze}_X e : [\pi^\iota \backslash X;\ \pi^o]}\ X \in \mathsf{dom}(\pi^\iota) \Rightarrow X \in \mathsf{dom}(\pi^o)$$

$$(\text{BASIC})\ \frac{\left\{\Gamma, x_i : c\tau_i^{i\in I\cup L} \vdash e_h : c\tau_h \mid h \in O \cup L\right\} \quad \Gamma, x_i : c\tau_i^{i\in I\cup L} \vdash e : c\tau}{\Gamma \vdash \left[x_i : c\tau_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in O}{\mapsto} e_j;\ x_l : c\tau_l \overset{l\in L}{\mapsto} e_l \mid e\right] : \left[X_i : c\tau_i^{i\in I};\ X_j : c\tau_j^{j\in O};\ c\tau\right]}\ X_i : c\tau_i^{i\in I} \| X_j : c\tau_j^{j\in O}$$

$$(\text{SUM})\ \frac{\Gamma \vdash e_1 : [\pi_1^\iota;\ \pi_1^o;\ c\tau] \quad \Gamma \vdash e_2 : [\pi_2^\iota;\ \pi_2^o]}{\Gamma \vdash e_1 + e_2 : [\pi_1^\iota, \pi_2^\iota;\ \pi_1^o, \pi_2^o;\ c\tau]}\ \begin{array}{l}\pi_1^\iota \| \pi_2^\iota \\ \pi_1^o \# \pi_2^o \\ \pi_1^\iota \| \pi_2^o, \pi_2^\iota \| \pi_1^o\end{array} \qquad (\text{DEL})\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o;\ c\tau]}{\Gamma \vdash e \backslash_X : [\pi^\iota;\ \pi^o \backslash X;\ c\tau]}\ X \in \mathsf{dom}(\pi^o)$$

$$(\text{FREEZE})\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o;\ c\tau]}{\Gamma \vdash \mathsf{freeze}_X e : [\pi_2^\iota \backslash X;\ \pi^o;\ c\tau]}\ \begin{array}{l}X \in \mathsf{dom}(\pi^\iota) \Rightarrow \\ X \in \mathsf{dom}(\pi^o)\end{array} \qquad (\text{RUN})\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o]}{\Gamma \vdash e \downarrow_X : [\pi^\iota;\ \pi^o;\ \pi^o(X)]}$$

$$(\text{RES})\ \frac{\Gamma \vdash e : [\emptyset;\ \pi^o;\ c\tau]}{\Gamma \vdash e \uparrow : c\tau}$$

Fig. A.3. R-calculus typing rules

$v \in \mathsf{Val}$	::=	**value**
	...	core values
	$\mid\ [\iota;\ o;\ \rho]$	basic module
	$\mid\ [\iota;\ o;\ \rho \mid v]$	basic configuration

Fig. A.4. R-calculus values

In typing rules we use the abbreviations $\pi_1 \| \pi_2$ and $\pi_1^o \# \pi_2^o$ introduced in Sect.4.

The following proposition states that the type system only derives well-formed types (defined in Fig.A.2).

Proposition A.1 *If* $\Gamma \vdash e : \tau$, *then* $\vdash \tau$.

Theorem A.2 (Progress) *If* $\emptyset \vdash e : \tau$, *then either* $e \in \mathsf{Val}$, *or* $e \longrightarrow e'$, *for some* e'.

Theorem A.3 (Subject Reduction) *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : \tau$.

B Proofs

Proof of Theorem 3.1. Induction on reduction rules. We only show the two base cases (C-RCV) and (RCV).

(C-RCV) In this case we have $\mathsf{rcv}\, c\tau_1 \xrightarrow{e:c\tau} e$, with $c\tau \leq_{\mathsf{core}} c\tau_1$, and by hypothesis $\Gamma \vdash \mathsf{rcv}\, c\tau_1 : c\tau_1'$. This last judgment can only be derived by using typing rule (RCV), hence, it must be $c\tau_1' = c\tau_1$. Moreover, we assume $\emptyset \vdash e : c\tau$ (see footnote at page 6) and thus (by applying a weakening lemma) we get $\Gamma \vdash e : c\tau$.

(RCV) In this case we have $\mathsf{rcv}\, \tau_1 \xrightarrow{e:\tau_2} (\mathsf{freeze}_H e)\backslash_H$, with $\tau_2 \leq \tau_1$ (*). We consider the case $\tau_i \equiv [\pi_i^\iota; \pi_i^o]$ (the other one is similar). By hypothesis we have $\Gamma \vdash \mathsf{rcv}\, [\pi_1^\iota; \pi_1^o] : \tau_1'$. To derive this last judgment, the only applicable rule is (RCV), hence it must be $\tau_1' \equiv [\pi_1^\iota; \pi_1^o]$. Moreover, we assume $\emptyset \vdash e : [\pi_2^\iota; \pi_2^o]$ (see footnote at page 6) and thus $\emptyset \vdash (\mathsf{freeze}_H e)\backslash_H : [\pi_2^\iota\backslash_H; \pi_2^o\backslash_H]$, with $H = \mathsf{dom}(\pi_2^o)\backslash\mathsf{dom}(\pi_1^o)$. By applying a weakening lemma we have that $\Gamma \vdash (\mathsf{freeze}_H e)\backslash_H : [\pi_2^\iota\backslash_H; \pi_2^o\backslash_H]$. We can now conclude by observing that $\mathsf{dom}(\pi_2^o\backslash_H) = \mathsf{dom}(\pi_1^o)$ and thus $\pi_1^\iota \leq \pi_2^\iota\backslash_H$ and $\pi_2^o\backslash_H \leq \pi_1^o$ (from (*)); hence, $[\pi_2^\iota\backslash_H; \pi_2^o\backslash_H] \leq [\pi_1^\iota; \pi_1^o]$.

Proof of Proposition 4.1. Induction on typing rules.

Proof of Theorem 4.2. We consider the following more general property:

If $\hat{\Gamma} \vdash e_1 : \mathcal{C}_1' \Rightarrow \hat{\tau}_1$ and $\langle e_1, \mathcal{C}_1\rangle \xrightarrow{e:\mathcal{C}\Rightarrow\hat{\tau}} \langle e_2, \mathcal{C}_2\rangle$ with $\mathcal{C}_1' \subseteq \mathcal{C}_1$, then $\hat{\Gamma} \vdash e_2 : \mathcal{C}_2' \Rightarrow \hat{\tau}_2$ with $\mathcal{C}_2' \subseteq \mathcal{C}_2$ and $\mathcal{C}_2 \Rightarrow \hat{\tau}_2 \leq \mathcal{C}_1 \Rightarrow \hat{\tau}_1$.

proved by induction on reduction rules. We illustrate the base case (RCV). We have $\langle \mathsf{rcv}\, \hat{\tau}_1, \mathcal{C}_1\rangle \xrightarrow{e:\mathcal{C}_2\Rightarrow\hat{\tau}_2} \langle e, \mathcal{C}, \mathcal{C}_2\rangle$, with $\mathcal{C}, \mathcal{C}_2 \Rightarrow \hat{\tau}_2 \leq \mathcal{C}_1 \Rightarrow \hat{\tau}_1$, and, by hypothesis, $\hat{\Gamma} \vdash \mathsf{rcv}\, \hat{\tau}_1 : \mathcal{C}_1' \Rightarrow \hat{\tau}_1'$, with $\mathcal{C}_1' \subseteq \mathcal{C}_1$. This typing judgment can only be derived by using rule (RCV), hence, it must be $\mathcal{C}_1' = \emptyset$ and $\hat{\tau}_1' = \hat{\tau}_1$. Moreover, we assume that $\emptyset \vdash e : \mathcal{C}_2 \Rightarrow \hat{\tau}_2$, and thus (by applying a weakening lemma) we have $\hat{\Gamma} \vdash e : \mathcal{C}_2 \Rightarrow \hat{\tau}_2$. We can now conclude by observing that $\mathcal{C}_2 \subseteq \mathcal{C}, \mathcal{C}_2$.

Proof of Proposition 4.3. Induction on typing rules.

Proof of Theorem 4.4. Case analysis on reduction rules used to derive $\langle e_1, \mathcal{C}_1\rangle \xrightarrow[\mathsf{cstr}]{e:\mathcal{C}\Rightarrow\hat{\tau}} \langle e_2, \mathcal{C}_2\rangle$ (the only applicable rules are (C-RCV) and (RCV)).

Proof of Theorem 4.5. Induction on the length n of the reduction sequence $\xrightarrow{e_1:\tau_1 \ldots e_n:\tau_n}$ and induction on typing rules used to derive $\hat{\Gamma} \overset{\mathsf{cstr}}{\vdash} e : \mathcal{C} \Rightarrow \hat{\tau}$.