Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 675–751. © 2007 Cambridge University Press
doi:10.1017/S0960129507006238 Printed in the United Kingdom

A calculus of open modules: call-by-need strategy and confluence[†]

SONIA FAGORZI and ELENA ZUCCA

Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy
Email: {fagorzi,zucca}@disi.unige.it

Received 28 October 2005; revised 14 November 2006

We present a simple module calculus where selection and execution of a component is possible on *open* modules, that is, modules that still need to import some external definitions. Hence, it provides a kernel model for a computational paradigm in which standard execution (that is, execution of a single computation described by a fragment of code) can be interleaved with operations at the meta-level, which can manipulate in various ways the context in which this computation takes place. Formally, this is achieved by introducing *configurations* as basic terms. These are, roughly speaking, pairs consisting of an (open, mutually recursive) collection of named components and a term representing a program running in the context of these components. Configurations can be manipulated by classical module/fragment operators, hence reduction steps can be either execution steps of the program or steps that perform module operations (called *reconfiguration* steps). Since configurations combine the features of lambda abstractions (first-class functions), records, environments with mutually recursive definitions and modules, the calculus extends and integrates both traditional module calculi and recursive lambda calculi. We state confluence of the calculus, and propose different ways to prevent errors arising from the lack of some required component, either by a purely static type system or by a combination of static and run-time checks. Moreover, we define a call-by-need strategy that performs module simplification *only when needed and only once*, leading to a generalisation of call-by-need lambda calculi that includes module features. We prove the soundness and completeness of this strategy using an approach based on information content, which also allows us to preserve confluence, even when local substitution rules are added to the calculus.

1. Introduction

In recent years some considerable effort has been invested in developing kernel module/fragment calculi (Ancona and Zucca 1999; Wells and Vestergaard 2000; Machkasova and Turbak 2000; Ancona and Zucca 2002; Hirschowitz and Leroy 2002; Hirschowitz *et al.* 2004; Hirschowitz 2004; Makholm and Wells 2005) providing foundations for the manipulation and combination of software components. These calculi allow us to express a large variety of existing mechanisms using a small set of operators, and hence they can

[†] This work was partially supported by APPSEM II – Thematic network IST-2001-38957 and MIUR EOS – Extensible Object Systems.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

be used as a paradigmatic example for studying modular languages, in the same spirit as the lambda calculus is used for functional programming. For instance, in *CMS*, modules are constructed from *basic modules*, which have the form $[\iota; o; \rho]$ where ι, o and ρ model input, output and local components, respectively, using just three primitive operators: *sum* (merging two modules), *freeze* (binding an input to an output component) and *reduct* (independently renaming input and output components). As shown in the literature (Ancona and Zucca 2002; Ancona and Zucca 1998; Wells and Vestergaard 2000), these operators allow us to express, for example, parameterised modules similar to ML functors, extension with overriding as in object-oriented programming, and mixin modules.

However, these calculi are based on a *static* view of software composition, in the sense that open code fragments can be combined together flexibly, though before actually starting the execution of a computation, we must have obtained a fully reduced and closed piece of code. In module calculi this is reflected formally by the fact that *selection*, denoted $e.X$, where e is a module expression and X is the name of a module component, can only be performed when e is a basic module (no module operators remain to be reduced) and, moreover, is *closed* (no components still need to be imported).

In this paper, we present a simple module calculus in which this restriction is dropped, and hence the selection and execution of a component is possible on *open* modules, which still have to import some components. Hence, we provide a kernel model for a computational paradigm in which standard execution (that is, execution of a single computation described by a possibly open fragment of code) can be interleaved with operations at the meta-level that can perform various manipulations of the context in which this computation takes place. Formally, this is achieved by defining *configurations* as basic terms of the calculus. These are, roughly speaking, pairs consisting of an (open, mutually recursive) collection of named components (formally, a basic module) and a term representing a program running in the context of these components. Configurations can be manipulated by classical module/fragment operators, such as those of *CMS*; hence, reduction steps can either be execution steps of the program or steps that perform module operators. We call the latter types of step *reconfiguration* steps, since they cause a modification in the context of the software components that may be used by the running program, and, consequently, our calculus is called the R-calculus, denoting 'reconfiguration'.

The initial motivation for the R-calculus was the quest for foundations for systems supporting dynamic reconfiguration features, in the sense that interleaving is allowed between reconfiguration steps and standard execution steps. Examples of reconfiguration features are: dynamic loading, as in Java and C#, where single code fragments are dynamically linked to an already executing program; dynamic binding and rebinding (Moreau 1998; Bierman *et al.* 2003), that is, the ability to change the meaning of names used by an application at execution time; dynamic updating (Hicks *et al.* 2001; Bierman *et al.* 2003; Stoyle *et al.* 2005; Hicks and Nettles 2005); and marshalling/unmarshalling of values or computations from one running application to another (Bierman *et al.* 2003).

However, much remains to be done before we will be able to use (a formalism based on) the R-calculus to model real systems (see Section 5.2 for a discussion and pointers to work in this direction). A more direct motivation is the desire to integrate two families of calculi into a single formalism: these are, on the one hand,

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

module calculi (Ancona and Zucca 1999; Wells and Vestergaard 2000; Machkasova and Turbak 2000; Ancona and Zucca 2002; Hirschowitz and Leroy 2002; Hirschowitz *et al.* 2004; Hirschowitz 2004; Makholm and Wells 2005), where the emphasis is on modelling operations for composing software fragments; and, on the other hand, let-rec lambda calculi (Ariola and Felleisen 1997; Ariola and Klop 1997; Ariola and Blom 1997; Ariola and Blom 2002), where the emphasis is on modelling transformations of recursive definitions of functions. Indeed, configurations combine lambda abstractions (first-class functions), records, environments with mutually recursive definitions and modules. In this paper, we focus on this aspect, looking at the R-calculus as a new computational model, and investigating its metatheory.

In particular, we extend the following two important results for let-rec lambda calculi to the R-calculus (and hence to a calculus that includes modularity features):

— The definition of an efficient strategy, called *call-by-need* (Ariola and Felleisen 1997), that performs term evaluation (in our case, also including module simplification) *only when needed and only once*. We provide reduction rules that correspond to such a strategy, and prove its soundness and completeness using an approach introduced in Ariola and Blom (1997) and Ariola and Blom (2002) that is based on the notion of *information content*.

— A proof of *confluence* (again, up to information content) of the calculus, even when local substitution rules[†] are added. In previous work on module calculi, confluence was obtained by either forbidding local substitution (Ancona and Zucca 2002) or restricting the interaction between local components (Ariola and Klop 1997; Wells and Vestergaard 2000). However, this approach is too restrictive since it forbids pairs of infinite reduction sequences that never commute but, intuitively, do not correspond to a real situation of non-determinism. In Ariola and Blom (1997) and Ariola and Blom (2002), and here, the problem is solved in a more abstract way by relaxing the notion of confluence.

The organisation of the paper is as follows. In Section 2, we introduce the main features of the R-calculus using some examples, give its formal syntax and semantics, and prove its confluence (in the standard sense) in the absence of local substitution rules. In Section 3, we propose different ways to prevent errors arising from the lack of some required component using either a purely static type system or a combination of static and run-time checks. In Section 4, we define the call-by-need strategy and prove its soundness and completeness; moreover, we prove confluence (up to information content) of an extended version of the calculus that includes local substitution rules. Finally, in Section 5 we begin by summarising the contribution of the paper, and discussing the role of the R-calculus compared with other basic computational models. We then consider issues related to reconfiguration features in real systems and give some pointers to how one might proceed in this direction. Finally, we give a brief review of related work and indications of further research directions.

[†] That is, rules allowing us replace a variable with its definition inside a group of mutually recursive definitions: it is well known that these rules break confluence – see Section 4.5.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$X, Y, Z \in \mathsf{Name}$	$::=$		**name**
$x, y, z \in \mathsf{Var}$	$::=$		**variable**
$e \in \mathsf{Exp}$	$::=$		**expression**
	$\mid \quad x$		variable
	$\mid \quad [\iota; o; \rho]$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \varnothing)$	basic module
	$\mid \quad \left[\iota; o; \rho \mid e\right]$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \varnothing)$	basic configuration
	$\mid \quad e_1 + e_2$		sum
	$\mid \quad {}_{\sigma^\iota}\lvert e \rvert_{\sigma^o}$		reduct
	$\mid \quad \mathsf{freeze}_\sigma e$		freeze
	$\mid \quad e{\downarrow_X}$		run
	$\mid \quad e{\uparrow}$		result
ι	$::= x_i \overset{i \in I}{\mapsto} X_i$		**input** assignment
o	$::= X_i \overset{i \in I}{\mapsto} e_i$		**output** assignment
ρ	$::= x_i \overset{i \in I}{\mapsto} e_i$		**local** assignment
σ	$::= X_i \overset{i \in I}{\mapsto} Y_i, _ \overset{j \in J}{\mapsto} Y_j$		**renaming**

Fig. 1. *R*-calculus: syntax

A preliminary version of the work developed in this paper was presented in Fagorzi and Zucca (2005).

2. The *R*-calculus

In this section we provide an introduction to the *R*-calculus. We begin by defining the basic notation that we will use throughout the paper, and then provide an illustration of the module fragment of the calculus, which is a variant of a module calculus like those in Ancona and Zucca (1999), Wells and Vestergaard (2000) and Ancona and Zucca (2002). Then we introduce configurations and illustrate the interleaving of program execution and reconfiguration. Finally, we discuss higher-order features of the calculus, showing how reconfiguration can take place at many levels.

The formal syntax is defined in Figure 1, and the reduction rules are given in Figures 3 and 4. We assume two infinite disjoint sets Name of *component names* and Var of *variables*. Rule $(\mathscr{E})$ is the usual contextual closure, where one-hole contexts, denoted by the meta-variable $\mathscr{E}$, are defined at the top of Figure 3.

2.1. *Notation*

We write $f : A \to B$ to denote a map f from A, written $\mathsf{dom}(f)$, to B, written $\mathsf{cod}(f)$.

We use $\varnothing$ to denote the empty map. For I a set of indices (I possibly empty), $a_i \overset{i \in I}{\mapsto} b_i$ is the unique surjective map f such that $\mathsf{dom}(f) = \{a_i \mid i \in I\}$, $\mathsf{cod}(f) = \{b_i \mid i \in I\}$ and $f(a_i) = b_i$ for $i \in I$. The expression is well formed only if for all i_1 and i_2 in I with $i_1 \neq i_2$, we have that $a_{i_1} \neq a_{i_2}$. Moreover, for I and J sets of indices (I and J possibly empty), $a_i \overset{i \in I}{\mapsto} b_i, _ \overset{j \in J}{\mapsto} b_j$ is the unique map f such that $\mathsf{dom}(f) = \{a_i \mid i \in I\}$,

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$\mathsf{cod}(f) = \{b_i \mid i \in I \cup J\}$ and $f(a_i) = b_i$ for $i \in I$. The expression is well formed only if $\{b_i \mid i \in I\}$ and $\{b_j \mid j \in J\}$ are disjoint sets and for any i_1 and i_2 in I, if $i_1 \neq i_2$, then $a_{i_1} \neq a_{i_2}$ and, similarly, for any j_1 and j_2 in J, if $j_1 \neq j_2$, then $b_{j_1} \neq b_{j_2}$.

We identify all expressions representing the same map. Two maps f and g are *compatible* if $f(a) = g(a)$ for all $a \in \mathsf{dom}(f) \cap \mathsf{dom}(g)$.

We use the following operators on maps:

— $f \cup g : \mathsf{dom}(f) \cup \mathsf{dom}(g) \to \mathsf{cod}(f) \cup \mathsf{cod}(g)$,
$(f \cup g)(x) = f(x)$ if $x \in \mathsf{dom}(f)$, $(f \cup g)(x) = g(x)$ if $x \in \mathsf{dom}(g)$
(hence $f \cup g$ is well formed only if f, g compatible).
— f, g is well formed if $\mathsf{dom}(f) \cap \mathsf{dom}(g) = \varnothing$,
$f, g : \mathsf{dom}(f) \cup \mathsf{dom}(g) \to \mathsf{cod}(f) \cup \mathsf{cod}(g)$,
$(f, g)(x) = (f \cup g)(x)$.
— $f|_C : \mathsf{dom}(f) \cap C \to \mathsf{cod}(f)$,
$f|_C(x) = f(x)$ for $x \in \mathsf{dom}(f) \cap C$.
— $f \setminus C : \mathsf{dom}(f) \setminus C \to \mathsf{cod}(f)$,
$(f \setminus C)(x) = f(x)$ for $x \in \mathsf{dom}(f) \setminus C$.
— $g \circ f$ is well formed if $\mathsf{cod}(f) \subseteq \mathsf{dom}(g)$,
$g \circ f : \mathsf{dom}(f) \to \mathsf{cod}(g)$,
$(g \circ f)(x) = g(f(x))$.

Finally, we write $f \subseteq g$ if $\mathsf{dom}(f) \subseteq \mathsf{dom}(g)$ and f, g are compatible.

2.2. *Module operators*

The terms of the calculus denote either *modules* (collections of *components* that are *input*, *output* or *local*) or *configurations* (pairs consisting of a module and a *program* running in the context of the components offered by the module).

Module expressions are constructed on top of *basic modules* by the use of three operators: *sum*, *reduct* and *freeze*. The semantics of these operators is a slight variant of that in *CMS* (Ancona and Zucca 2002) – the difference is that we relax the applicability of the reduct and freeze operators. Apart from this, the formal and informal descriptions below are largely taken from Ancona and Zucca (2002).

In Figure 2 we define free variables, capture-avoiding parallel substitution for expressions and assignments, and α-congruence on expressions. The relation $\widetilde{=}$ over well-formed terms is defined as the closure by congruence of the relation inductively defined by the given rules.

As usual, in the following we will identify any pair of terms e_1 and e_2 such that $e_1 \widetilde{=} e_2$; more precisely, we will feel free to α-convert any term in order to avoid variable captures and clashes while applying either substitutions or reduction steps.

A **basic module** has the shape $[\iota; o; \rho]$ where:

— ι is a map from *deferred* variables to *input* names.
— o is a map from *output* names to expressions.
— ρ is a map from *local* variables to expressions.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Free variables

$$
\begin{aligned}
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}([\iota;\, o;\, \rho]) &= \mathsf{FV}(o) \cup \mathsf{FV}(\rho) \setminus \mathsf{dom}(\iota, \rho) \\
\mathsf{FV}([\iota;\, o;\, \rho \mid e]) &= \mathsf{FV}(o) \cup \mathsf{FV}(\rho) \cup \mathsf{FV}(e) \setminus \mathsf{dom}(\iota, \rho) \\
\mathsf{FV}(e_1 + e_2) &= \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}({}_{\sigma^\iota|}e_{|\sigma^o}) &= \mathsf{FV}(e) \\
\mathsf{FV}(\mathsf{freeze}_\sigma e) &= \mathsf{FV}(e) \\
\mathsf{FV}(e{\downarrow_X}) &= \mathsf{FV}(e) \\
\mathsf{FV}(e{\uparrow}) &= \mathsf{FV}(e) \\
\mathsf{FV}(x_i \overset{i\in I}{\mapsto} e_i) &= \cup_{i\in I}\mathsf{FV}(e_i) \\
\mathsf{FV}(X_i \overset{i\in I}{\mapsto} e_i) &= \cup_{i\in I}\mathsf{FV}(e_i)
\end{aligned}
$$

Capture-avoiding parallel substitution

$$
\begin{aligned}
x\,\{\rho\} &= \rho(x) && \text{if } x \in \mathsf{dom}(\rho) \\
x\,\{\rho\} &= x && \text{if } x \notin \mathsf{dom}(\rho) \\
[\iota;\, o;\, \rho]\,\{\rho'\} &= [\iota;\, o\,\{\rho'\};\, \rho\,\{\rho'\}] && \text{if } \mathsf{dom}(\iota, \rho) \cap (\mathsf{dom}(\rho') \cup \mathsf{FV}(\rho')) = \varnothing \\
[\iota;\, o;\, \rho \mid e]\,\{\rho'\} &= [\iota;\, o\,\{\rho'\};\, \rho\,\{\rho'\} \mid e\,\{\rho'\}] && \text{if } \mathsf{dom}(\iota, \rho) \cap (\mathsf{dom}(\rho') \cup \mathsf{FV}(\rho')) = \varnothing \\
(e_1 + e_2)\,\{\rho\} &= e_1\,\{\rho\} + e_2\,\{\rho\} \\
{}_{\sigma^\iota|}e_{|\sigma^o}\,\{\rho\} &= {}_{\sigma^\iota|}e\,\{\rho\}_{|\sigma^o} \\
(\mathsf{freeze}_\sigma e)\,\{\rho\} &= \mathsf{freeze}_\sigma e\,\{\rho\} \\
e{\downarrow_X}\,\{\rho\} &= e\,\{\rho\}{\downarrow_X} \\
e{\uparrow}\,\{\rho\} &= e\,\{\rho\}{\uparrow} \\
(x_i \overset{i\in I}{\mapsto} e_i)\,\{\rho\} &= x_i \overset{i\in I}{\mapsto} e_i\,\{\rho\} \\
(X_i \overset{i\in I}{\mapsto} e_i)\,\{\rho\} &= X_i \overset{i\in I}{\mapsto} e_i\,\{\rho\}
\end{aligned}
$$

α-congruence

$$
(\iota\text{-M-BASIC})\ \frac{}{[x \mapsto X, \iota;\, o;\, \rho] \cong [x' \mapsto X, \iota;\, o\{\rho_x\};\, \rho\{\rho_x\}]}\ x' \text{ fresh}
$$

$$
(\rho\text{-M-BASIC})\ \frac{}{[\iota;\, o;\, x \mapsto e, \rho] \cong [\iota;\, o\{\rho_x\};\, x' \mapsto e\{\rho_x\}, \rho\{\rho_x\}]}\ x' \text{ fresh}
$$

$$
(\iota\text{-BASIC})\ \frac{}{[x \mapsto X, \iota;\, o;\, \rho \mid e] \cong [x' \mapsto X, \iota;\, o\{\rho_x\};\, \rho\{\rho_x\} \mid e\{\rho_x\}]}\ x' \text{ fresh}
$$

$$
(\rho\text{-BASIC})\ \frac{}{[\iota;\, o;\, x \mapsto e_1, \rho \mid e_2] \cong [\iota;\, o\{\rho_x\};\, x' \mapsto e_1\{\rho_x\}, \rho\{\rho_x\} \mid e_2\{\rho_x\}]}\ x' \text{ fresh}
$$

where $\rho_x = x \mapsto x'$

Fig. 2. *R*-calculus: free variables, capture-avoiding parallel substitution and α-congruence

Names are used to refer to a component from outside the module (hence they are used by module operators), while variables are used to refer to a component from the inside. For instance, using $e[x_1, \ldots, x_n]$ to denote an expression possibly containing $x_1, \ldots, x_n$ as free variables,

$$
e_0 \triangleq [x \mapsto X, z \mapsto Z\,;\; Y \mapsto e_1[x, z, y]\,;\; y \mapsto e_2[x, z, y]]
$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

One-hole contexts and contextual closure

$$\mathscr{E} ::= \Box \mid [\iota; X \mapsto \mathscr{E}, o; \rho] \mid [\iota; o; x \mapsto \mathscr{E}, \rho] \mid \left[\iota; X \mapsto \mathscr{E}, o; \rho \mid e\right] \mid \left[\iota; o; x \mapsto \mathscr{E}, \rho \mid e\right]$$
$$\mid \left[\iota; o; \rho \mid \mathscr{E}\right] \mid \mathscr{E} + e \mid e + \mathscr{E} \mid {}_{\sigma^\iota|}\mathscr{E}_{|\sigma^o} \mid \mathsf{freeze}_\sigma \mathscr{E} \mid \mathscr{E}\downarrow_X \mid \mathscr{E}\uparrow$$

$$(\mathscr{E})\ \frac{e \longrightarrow e'}{\mathscr{E}[e] \longrightarrow \mathscr{E}[e']}$$

Module simplification

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

$$(\text{M-REDUCT})\ \frac{}{{}_{\sigma^\iota|}[\iota_1, \iota_2; o; \rho]_{|\sigma^o} \longrightarrow [(\sigma^\iota \circ \iota_1), \iota_2; o \circ \sigma^o; \rho]}\ \mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma^\iota) = \varnothing$$

$$(\text{M-FREEZE})\ \frac{}{\mathsf{freeze}_\sigma \left[x_i \overset{i\in I}{\mapsto} X_i, \iota; o; \rho\right] \longrightarrow \left[\iota; o; \rho, x_i \overset{i\in I}{\mapsto} o(\sigma(X_i))\right]}\ \mathsf{cod}(\iota) \cap \mathsf{dom}(\sigma) = \varnothing$$

Fig. 3. *R*-calculus: contextual closure and reduction rules for modules

is a basic module with two inputs, one output and one local component. Local components can be mutually recursive.

The **sum** operator has the effect of gluing together two modules – see rule (M-SUM).

Side-conditions prevent undesired captures of free variables, which are defined in Figure 2. Furthermore, since reduction is defined over well-formed terms only, we have the implicit side-condition $\mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$. These (explicit and implicit) side-conditions can always be satisfied by an appropriate α-conversion.

Analogously, since o_1, o_2 must be well formed, we have the implicit side-condition $\mathsf{dom}(o_1) \cap \mathsf{dom}(o_2) = \varnothing$; however, in this case the reduction gets stuck since this conflict cannot be resolved by an α-conversion. The only way to solve this problem is to rename the output components explicitly in an appropriate way by means of the reduct operator (see below), thus changing the term.

Note that, since ι goes 'backwards', that is, from variables into names, the fact that ι_1, ι_2 must be well formed does not prevent it from sharing input names, but only, as can be seen, from sharing deferred variables (which can be avoided by α-conversion).

For instance, since the sets of output names are disjoint, we can perform the sum of the basic module e_0 above with

$$e_5 \triangleq [y \mapsto Y, w \mapsto Z; X \mapsto e_3[y, w, x]; x \mapsto e_4[y, w, x]]$$

to get the basic module

$$e_6 \triangleq \begin{array}{l}[x \mapsto X, z \mapsto Z, y' \mapsto Y, w \mapsto Z; \\ \quad Y \mapsto e_1[x, z, y], X \mapsto e_3[y', w, x']; \\ \quad y \mapsto e_2[x, z, y], x' \mapsto e_4[y', w, x']]\end{array}$$

where the input name Z is shared by the two variables z and w, while variables y and x in e_5 are α-renamed to y' and x', respectively, since they conflict with those in e_0.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

The sum operator supports cross-module recursion, in the sense that definitions needed by one module, through some input name, can be provided by the other, through an output name, and conversely. For instance, in the previous example, the definition of Y is needed by e_5 and provided by e_0, whereas the definition of X is needed by e_0 and provided by e_5.

The **reduct** operator allows us to perform a renaming of component names, where input and output are renamed independently by specifying two renamings σ^i and σ^o, respectively, see rule (M-REDUCT).

The input renaming σ^i is a map whose domain and codomain are old and new input names, respectively, whereas the output renaming σ^o is a map whose domain and codomain are new and old output names, respectively. The two renamings are contravariant for the same reason that a function from A to B can be converted into a function from A' to B' whenever two conversion functions from A' to A and from B to B' are provided.

For instance,

$$_{X_1\mapsto X,X_2\mapsto X,Z\mapsto W}|[x_1 \mapsto X_1, x_2 \mapsto X_2, x_3 \mapsto X_3;\ Y \mapsto e_1, Z \mapsto e_2;\ x \mapsto e_3]_{|Y_1\mapsto Y,Y_2\mapsto Y}$$

reduces to

$$[x_1 \mapsto X, x_2 \mapsto X, x_3 \mapsto X_3, w \mapsto W;\ Y_1 \mapsto e_1, Y_2 \mapsto e_1;\ x \mapsto e_3]\,.$$

A non-injective input renaming allows the merging of two input names (such as X_1 and X_2 into X in the example), whereas a non-surjective one is used to add dummy input names (such as W in the example), which are bound to fresh variables (w in the example). A non-injective output renaming allows the duplication of definitions (in the example, the definition of Y is used for both Y_1 and Y_2), whereas a non-surjective one is used to delete output components (Z in the example).

Moreover, the input renaming can be defined on an arbitrary set of names; names that are actually renamed ($\mathsf{cod}(\iota_1)$) are those both in the input of the module and in $\mathsf{dom}(\sigma^i)$. Since map composition must be well defined, see Section 2.1, we have the implicit side-conditions $\mathsf{cod}(\iota_1) \subseteq \mathsf{dom}(\sigma^i)$ and $\mathsf{cod}(\sigma^o) \subseteq \mathsf{dom}(o)$. This semantics is more liberal than the original formulation in Ancona and Zucca (2002), where $\mathsf{dom}(\sigma^i)$ was required to contain all the module's input names. The explicit side-condition just ensures that $\mathsf{cod}(\iota_1)$ actually contains all the input names that have to be renamed. Input names that are not renamed (represented by $\mathsf{cod}(\iota_2)$) are not affected.

In the following we will use the abbreviation $e \setminus X$ to denote the application to the module e of a reduct operator where the input renaming is the identity and the output renaming is the embedding of all output names except X.

The **freeze** operator allows the linking of input and output names inside a module, see rule (M-FREEZE). That is, this operator resolves input names, so that input components are transformed into local ones. For instance, the expression

$$\mathsf{freeze}_{X\mapsto X,Z\mapsto Y}\, e_6$$

reduces to

$$\begin{array}{l}[\,y' \mapsto Y\,;\\ Y \mapsto e_1[x,z,y], X \mapsto e_3[y',w,x']\,;\\ y \mapsto e_2[x,z,y], x' \mapsto e_4[y',w,x'], x \mapsto e_3[y',w,x'], z \mapsto e_1[x,z,y], w \mapsto e_1[x,z,y]]\end{array}$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Reconfiguration contexts

$$\mathscr{R} \;::=\; \Box \mid \mathscr{R} + e \mid {}_{\sigma^I|}\mathscr{R}_{|\sigma^O} \mid \mathsf{freeze}_\sigma \mathscr{R}$$

Reconfiguration and substitution

$$\text{(MOD-PROP)}\ \frac{\mathscr{R}[\iota;\, o;\, \rho] \underset{\alpha}{\longrightarrow} \mathscr{R}'[\iota';\, o';\, \rho']}{\mathscr{R}\big[\iota;\, o;\, \rho \mid e\big] \longrightarrow \mathscr{R}'\big[\iota';\, o';\, \rho' \mid \alpha(e)\big]}$$

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

Run and result

$$\text{(RUN)}\ \frac{}{[\iota;\, o;\, \rho]{\downarrow_X} \longrightarrow \big[\iota;\, o;\, \rho \mid o(X)\big]}$$

$$\text{(RES)}\ \frac{}{\big(\mathscr{R}\big[\iota;\, o;\, \rho \mid e\big]\big)\uparrow \longrightarrow e}\ \mathsf{FV}(e) \cap \mathsf{dom}(\iota, \rho) = \varnothing$$

Lift

$$\text{(SUM-LIFT)}\ \frac{}{\big[\iota_1;\, o_1;\, \rho_1 \mid e_1\big]\uparrow + e_2 \longrightarrow \big[\iota_1;\, o_1;\, \rho_1 \mid e_1 + e_2\big]\uparrow}\ \mathsf{FV}(e_2) \cap \mathsf{dom}(\iota_1, \rho_1) = \varnothing$$

$$\text{(R-SUM-LIFT)}\ \frac{}{e_1 + \big[\iota_2;\, o_2;\, \rho_2 \mid e_2\big]\uparrow \longrightarrow \big[\iota_2;\, o_2;\, \rho_2 \mid e_1 + e_2\big]\uparrow}\ \mathsf{FV}(e_1) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$$

$$\text{(REDUCT-LIFT)}\ \frac{}{{}_{\sigma^I|}\big[\iota;\, o;\, \rho \mid e\big]\uparrow_{|\sigma^O} \longrightarrow \big[\iota;\, o;\, \rho \mid {}_{\sigma^I|}e_{|\sigma^O}\big]\uparrow}$$

$$\text{(FREEZE-LIFT)}\ \frac{}{\mathsf{freeze}_\sigma\big(\big[\iota;\, o;\, \rho \mid e\big]\uparrow\big) \longrightarrow \big[\iota;\, o;\, \rho \mid \mathsf{freeze}_\sigma e\big]\uparrow}$$

$$\text{(RUN-LIFT)}\ \frac{}{\big[\iota;\, o;\, \rho \mid e\big]\uparrow{\downarrow_X} \longrightarrow \big[\iota;\, o;\, \rho \mid e{\downarrow_X}\big]\uparrow}$$

$$\text{(RES-LIFT)}\ \frac{}{\big[\iota;\, o;\, \rho \mid e\big]\uparrow\uparrow \longrightarrow \big[\iota;\, o;\, \rho \mid e\uparrow\big]\uparrow}$$

Fig. 4. *R*-calculus: reduction rules for configurations

Input components with names X and Z are transformed into local components associated with variables x and z, w, and with the definitions of the output components X and Y, respectively. Note that resolution can be partial (Y is not resolved in the example). Moreover, unlike the case with the original formulation in Ancona and Zucca (2002), the freeze operator can specify associations (which simply have no effect) for names that are not present in the module; the only constraint is that an existing input name must be mapped into an existing output name (implicit side-condition).

2.3. *Configurations*

A **basic configuration** is a pair $\big[\iota;\, o;\, \rho \mid e\big]$, consisting of a basic module and an expression, called a *program.*

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Hole binders

$$
\begin{aligned}
\mathsf{HB}(\Box) &= \varnothing \\
\mathsf{HB}([\iota; X \mapsto \mathscr{E}, o; \rho]) &= \mathsf{dom}(\iota, \rho) \cup \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}([\iota; o; x \mapsto \mathscr{E}, \rho]) &= \mathsf{dom}(\iota, \rho) \cup \{x\} \cup \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}\left(\left[\iota; X \mapsto \mathscr{E}, o; \rho \mid e\right]\right) &= \mathsf{dom}(\iota, \rho) \cup \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}\left(\left[\iota; o; x \mapsto \mathscr{E}, \rho \mid e\right]\right) &= \mathsf{dom}(\iota, \rho) \cup \{x\} \cup \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}\left(\left[\iota; o; \rho \mid \mathscr{E}\right]\right) &= \mathsf{dom}(\iota, \rho) \cup \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}(\mathscr{E} + e) &= \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}(e + \mathscr{E}) &= \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}\left({}_{\sigma^I}|\mathscr{E}|_{\sigma^O}\right) &= \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}(\mathsf{freeze}_{\sigma}\mathscr{E}) &= \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}(\mathscr{E}\downarrow_X) &= \mathsf{HB}(\mathscr{E}) \\
\mathsf{HB}(\mathscr{E}\uparrow) &= \mathsf{HB}(\mathscr{E})
\end{aligned}
$$

Capture-avoiding hole substitution

$$\mathscr{E}\{e\} = \mathscr{E}[e] \quad \text{if } \mathsf{HB}(\mathscr{E}) \cap \mathsf{FV}(e) = \varnothing$$

Fig. 5. *R*-calculus: hole binders and capture-avoiding hole substitution

A basic configuration can evolve by reduction steps of the program, by applying rule $_{(\mathscr{E})}$; moreover, local variables can be replaced by their defining expressions. This is modelled by rule (PRG-SUBST), which can be applied only if the occurrence of the variable x in the position denoted by the hole of the context $\mathscr{E}$ is free (that is, not captured by any binder around the hole), as required by the side-condition $x \notin \mathsf{HB}(\mathscr{E})$ (hole binders $\mathsf{HB}(\mathscr{E})$ of a context $\mathscr{E}$ are defined in Figure 5). In this case, the program execution can proceed by replacing the variable with its defining expression. We use $\mathscr{E}\{e\}$ to denote the capture-avoiding substitution of the hole in context $\mathscr{E}$ by the expression e, defined in Figure 5.

This is illustrated by the following reduction sequence[†]:

$$
\begin{aligned}
\left[; X \mapsto x; y \mapsto 1, x \mapsto 2 + y \mid x\right] &\xrightarrow{\text{(PRG-SUBST)}} \left[; X \mapsto x; y \mapsto 1, x \mapsto 2 + y \mid 2 + y\right] \\
&\xrightarrow{\text{(PRG-SUBST)}} \left[; X \mapsto x; y \mapsto 1, x \mapsto 2 + y \mid 2 + 1\right] \\
&\longrightarrow \left[; X \mapsto x; y \mapsto 1, x \mapsto 2 + y \mid 3\right].
\end{aligned}
$$

Rule (MOD-PROP) allows us to propagate the module simplification steps described in the previous section to the module part of a configuration. That is, module operators can also be applied to configurations, and act as *reconfiguration* operators, in the sense that they allow us to modify the context of a program during its execution.

This rule is a more compact way of expressing three different rules analogous to (M-SUM), (M-REDUCT), (M-FREEZE), but having as (first) argument a basic configuration. In the premise of the

[†] Here, and elsewhere, to make the examples more concise and familiar, we assume that terms of the calculus include integers with the standard reduction rules. However, the formal development in the paper does not assume any underlying term calculus, and all examples can be easily rephrased in the pure *R*-calculus. Note, incidentally, that module calculi, and hence the R-calculus, are Turing complete formalisms since lambda calculus can be encoded in them (Wells and Vestergaard 2000; Ancona and Zucca 2002).

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

rule, we use the notation $\xrightarrow[\alpha]{}$ to mean that the α-renamings that may have happened during the module evaluation step must be propagated to the program as well. To describe all the configurations obtained by applying to a basic configuration the three module operators (that is, sum with a module, reduct and freeze), we introduce *reconfiguration contexts* $\mathscr{R}$, defined at the top of Figure 4. These are a subset of one-hole contexts, so, in principle, they could be filled with an arbitrary term of the calculus. However, in this paper holes will always be filled with a basic configuration†. Note that, while the sum of two modules and the sum of a configuration and a module (conventionally taken in this order) are handled by rule (M-SUM) and (MOD-PROP), respectively, there is no rule for the sum of two configurations, which, therefore, gets stuck (and will be rejected by the type systems in Section 3). This corresponds to the fact that we are considering a sequential calculus, in which there is only one executing program at a given configuration level (see below). For the same reason, we do not include the case $e + \mathscr{R}$ in reconfiguration contexts.

Thanks to reconfiguration steps, a required input component can become available, as in the following example:

$$
\begin{aligned}
&\mathsf{freeze}_{Z\mapsto Z}(\,[z\mapsto Z\,;\,;\,y\mapsto 1, x\mapsto 2+z\mid x]+[;Z\mapsto 3;\,]\,)\\
&\quad\xrightarrow{\text{(PRG-SUBST)}}\quad \mathsf{freeze}_{Z\mapsto Z}(\,[z\mapsto Z\,;\,;\,y\mapsto 1, x\mapsto 2+z\mid 2+z]+[;Z\mapsto 3;\,]\,)\\
&\quad\xrightarrow{\text{(MOD-PROP)}+\text{(M-SUM)}}\quad \mathsf{freeze}_{Z\mapsto Z}\,[Z\mapsto z; Z\mapsto 3;\, y\mapsto 1, x\mapsto 2+z\mid 2+z]\\
&\quad\xrightarrow{\text{(MOD-PROP)}+\text{(M-FREEZE)}}\quad [;Z\mapsto 3;\, y\mapsto 1, x\mapsto 2+z, z\mapsto 3\mid 2+z]\\
&\quad\xrightarrow{\text{(PRG-SUBST)}}\quad [;Z\mapsto 3;\, y\mapsto 1, x\mapsto 2+z, z\mapsto 3\mid 2+3]\\
&\quad\longrightarrow\quad [;Z\mapsto 3;\, y\mapsto 1, x\mapsto 2+z, z\mapsto 3\mid 5].
\end{aligned}
$$

2.4. *Higher-order features and configuration levels*

Since a program can, in turn, be a configuration, both substitution and reconfiguration steps can take place at an outer configuration level (for substitution, the innermost where the required variable is bound). This is illustrated by the following reduction sequence:

$$
\begin{aligned}
&\mathsf{freeze}_{Z\mapsto Z}\,([z\mapsto Z\,;\,;\,y\mapsto 2\mid [;X\mapsto x; x\mapsto y+z\mid x]]+[;Z\mapsto 3;\,])\\
&\quad\xrightarrow{\text{(PRG-SUBST)}}\quad \mathsf{freeze}_{Z\mapsto Z}\,([z\mapsto Z\,;\,;\,y\mapsto 2\mid [;X\mapsto x; x\mapsto y+z\mid y+z]]+[;Z\mapsto 3;\,])\\
&\quad\xrightarrow{\text{(PRG-SUBST)}}\quad \mathsf{freeze}_{Z\mapsto Z}\,([z\mapsto Z\,;\,;\,y\mapsto 2\mid [;X\mapsto x; x\mapsto y+z\mid 2+z]]+[;Z\mapsto 3;\,])\\
&\quad\xrightarrow{\text{(MOD-PROP)}+\text{(M-SUM)}}\quad \mathsf{freeze}_{Z\mapsto Z}\,[z\mapsto Z\,;Z\mapsto 3;\, y\mapsto 2\mid [;X\mapsto x; x\mapsto 2+z\mid 2+z]]\\
&\quad\xrightarrow{\text{(MOD-PROP)}+\text{(M-FREEZE)}}\quad [;Z\mapsto 3;\, y\mapsto 2, z\mapsto 3\mid [;X\mapsto x; x\mapsto 2+z\mid 2+z]]\\
&\quad\xrightarrow{\text{(PRG-SUBST)}}\quad [;Z\mapsto 3;\, y\mapsto 2, z\mapsto 3\mid [;X\mapsto x; x\mapsto 2+z\mid 2+3]]\\
&\quad\longrightarrow\quad [;Z\mapsto 3;\, y\mapsto 2, z\mapsto 3\mid [;X\mapsto x; x\mapsto 2+z\mid 5]]\,.
\end{aligned}
$$

The **run** operator allows us to obtain a basic configuration from a (basic) module by starting the execution of one of its output components, see rule (RUN). For instance, the

† To simplify the notation, we will omit one pair of square brackets when a hole is filled by a basic module/configuration.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

expression

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

reduces in one step to the basic configuration $\left[; X \mapsto x;\ y \mapsto 1, x \mapsto 2 + y \mid x\right]$ shown earlier. Since the run operator yields a configuration, which can still be combined with other modules, the R-calculus also allows the execution of module operators after the evaluation of an output component has begun. In other words, there can be *interleaving* between execution steps of the running program and *reconfiguration steps*, that is, the application of reconfiguration operators (starting from the innermost) to the basic configuration.

Note that, as seen above, in some cases reconfigurations steps are required by the running program in order to proceed. This will be exploited in Section 4, where we will define a call-by-need reduction strategy for the calculus that performs reconfiguration steps only when necessary.

The **result** operator allows us to extract the program from a configuration level if it does not refer to variables bound at that level, see rule (RES). Indeed, in this case the program execution can no longer be affected by reconfiguration steps.

For instance, the expression

$$\left(\mathsf{freeze}_{Z \mapsto Z}\left(\left[z \mapsto Z;\ ;\ \mid \left[x \mapsto X;\ ;\ \mid x\right]\right] + [; Z \mapsto 3;]\right)\right) \uparrow + [; X \mapsto 0;]$$

reduces to

$$\left[x \mapsto X;\ ;\ \mid x\right] + [; X \mapsto 0;]\,.$$

Formally, a configuration level is modelled by an expression of the form $\mathscr{R}\left[\iota; o; \rho \mid e\right]$ (recall that $\mathscr{R}$ is a context consisting of module operators only).

The side-condition prevents scope extrusion of variables. For instance, in the expression

$$\left(\mathsf{freeze}_{Z \mapsto Z}\left(\left[z \mapsto Z;\ ;\ \mid \left[; X \mapsto x; x \mapsto 2 + z \mid x\right] \uparrow\right] + [; Z \mapsto 3;]\right)\right) \uparrow$$

the innermost result operator cannot be executed since the program refers to the variable x bound in the enclosing basic configuration. Hence, we first need to resolve x, obtaining

$$\left(\mathsf{freeze}_{Z \mapsto Z}\left(\left[z \mapsto Z;\ ;\ \mid \left[; X \mapsto x; x \mapsto 2 + z \mid 2 + z\right] \uparrow\right] + [; Z \mapsto 3;]\right)\right) \uparrow .$$

At this point, we can either apply reconfiguration steps, or the innermost result operator. If we choose the latter, we get the configuration

$$\left(\mathsf{freeze}_{Z \mapsto Z}\left(\left[z \mapsto Z;\ ;\ \mid 2 + z\right] + [; Z \mapsto 3;]\right)\right) \uparrow,$$

which reduces after many steps to 5.

The final block of rules (the *lifting* rules) allows us to move a module operator inside a basic configuration to which a result operator is applied. They are inspired by analogous rules in let-rec calculi (Ariola and Felleisen 1997; Ariola and Blom 1997; Ariola and Blom 2002).

These rules are useful for handling cases in which the program in a certain configuration level cannot be extracted, since it either depends on local variables that are mutually recursive, or on deferred variables, but could be safely extracted after performing some module operators applied in the outer configuration level. For instance, in the following

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$v \in \mathsf{Val} \ ::= \ [\iota; o; \rho] \ \mid \ \lceil \iota; o; \rho \mid v \rceil \ \mid \ \lceil \iota; o; \rho \mid v \rceil \uparrow$$

Fig. 6. *R*-calculus: values

example

$$\lceil ; ; x \mapsto y, y \mapsto x \mid [; Z \mapsto x, W \mapsto 1;] \rceil \uparrow \backslash Z$$

the program, which is a basic module, cannot be extracted since the definition of component Z refers to the variable x, which is bound in the enclosing basic configuration, and cannot be eliminated. However, by applying the (REDUCT-LIFT) rule, it becomes possible to delete the component Z from this basic module, hence the program can then be safely extracted, as shown below:

$$\begin{aligned} \lceil ; ; x \mapsto y, y \mapsto x \mid [; Z \mapsto x, W \mapsto 1;] \rceil \uparrow \backslash Z & \\ \xrightarrow{\text{(REDUCT-LIFT)}} & \lceil ; ; x \mapsto y, y \mapsto x \mid [; Z \mapsto x, W \mapsto 1;] \backslash Z \rceil \uparrow \\ \xrightarrow{\text{(REDUCT)}} & \lceil ; ; x \mapsto y, y \mapsto x \mid [; W \mapsto 1;] \rceil \uparrow \\ \xrightarrow{\text{(RES)}} & [; W \mapsto 1;] \, . \end{aligned}$$

Note that for the sum we need two different lifting rules: one corresponding to the case in which the left-hand argument of a sum is the result of a basic configuration; and the other for when it is the right-hand argument. In both rules, e_1 can be either a configuration or a module, whereas e_2 must be a module (since, as we mentioned earlier, we only allow the sum of two modules or of a configuration and a module).

2.5. *Values*

The set Val of values of the *R*-calculus, ranged over by v, is defined in Figure 6.

They are basic modules and basic configurations (possibly enclosed by a result operator) where the program is a value. Note that, in general, a value is not a normal form; indeed, a value can reduce to another value either by reducing a component, or by extracting the program, or by substitution steps. Intuitively, all these values are equivalent (as will be formalised in Section 4.2 using the notion of *information content*).

2.6. *Confluence*

The *R*-calculus enjoys the Church–Rosser (CR) property[†]. In order to prove this result, we first introduce some standard definitions and properties on Term Rewriting Systems (TRSs) and Combinatory Reduction Systems (CRSs). These definitions and properties, along with the proof technique, are mainly taken from Ancona and Zucca (2002). For more technical details, see Klop (1987) and Klop *et al.* (1993).

[†] Recall that the technical development in this paper is given for the *pure R*-calculus, as defined in Figure 1. If we had chosen to present the *R*-calculus as a parametric calculus built on top of an underlying *core* calculus, as is done, for instance, for *CMS* in Ancona and Zucca (2002), confluence would, of course, hold only under suitable assumptions for the core level, and, analogously, for other properties (see Fagorzi (2005) for this approach).

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Note that the standard CR property holds since the calculus does not include *local substitution rules*, which allows us to replace variables in local definitions. When we add these rules, only a relaxed form of confluence holds, which is called *up to information content*, as we will describe and prove in Section 4.5.

Definition 2.1. An *Abstract Reduction System* (ARS) is a pair $(A, (\rightarrow_\alpha)_{\alpha \in I})$ consisting of a set A and a sequence of binary relations $\rightarrow_\alpha$ on A, which are also called reduction or rewrite relations.

If for $a, b \in A$ we have $(a, b) \in \rightarrow_\alpha$, we write $a \rightarrow_\alpha b$. The reflexive, transitive reflexive and transitive reflexive symmetric closures of $\rightarrow_\alpha$ are written $\overset{=}{\rightarrow}_\alpha$, $\overset{*}{\rightarrow}_\alpha$ and $\leftrightarrow_\alpha$, respectively. The composition $\rightarrow_\alpha \circ \rightarrow_\beta$ is defined by $a \rightarrow_\alpha \circ \rightarrow_\beta b$ if $a \rightarrow_\alpha c \rightarrow_\beta b$ for some $c \in A$. The converse relation of $\rightarrow_\alpha$ is written $\leftarrow_\alpha$.

Definition 2.2. If $\rightarrow_\alpha$ and $\rightarrow_\beta$ are reduction relations on A, then we say that they *commute* if $\overset{*}{\leftarrow}_\beta \circ \overset{*}{\rightarrow}_\alpha = \overset{*}{\rightarrow}_\alpha \circ \overset{*}{\leftarrow}_\beta$.
We say that they *strongly commute* if $\leftarrow_\beta \circ \rightarrow_\alpha = \overset{=}{\rightarrow}_\alpha \circ \overset{=}{\leftarrow}_\beta$.

Fact 2.3. If $\rightarrow_\alpha$ and $\rightarrow_\beta$ strongly commute, then they commute.

Definition 2.4. A reduction relation is said to be Church–Rosser (or confluent) if $\overset{*}{\rightarrow}$ is self-commuting.

From Hindley (1964) we know that the following two theorems hold.

Theorem 2.5. Let $\left(A, \rightarrow_\alpha, \rightarrow_\beta\right)$ be an ARS. If for all $a, b, c \in A$, with $a \rightarrow_\alpha b$ and $a \rightarrow_\beta c$, there exists $d \in A$ such that $b \overset{*}{\rightarrow}_\beta d$ and $c \overset{=}{\rightarrow}_\alpha d$, then $\rightarrow_\alpha$ and $\rightarrow_\beta$ commute.

Theorem 2.6 (Hindley). Let $(A, (\rightarrow_\alpha)_{\alpha \in I})$ be an ARS such that for all $\alpha, \beta \in I$, $\rightarrow_\alpha$ and $\rightarrow_\beta$ commutes. Then the union $\bigcup_{\alpha \in I} \rightarrow_\alpha$ is CR.

See Klop (1987) and Klop *et al.* (1993) for more on the notions of a Term Rewriting System and a Combinatory Reduction System, respectively.

The following definitions and theorem for TRSs also apply to CRSs (by replacing the word 'variable' by 'meta-variable').

Definition 2.7. A term is *linear* if it contains no multiple occurrences of the same variable. A TRS is *left-linear* if the left-hand side of each of its reduction rules is a linear term.

Definition 2.8. Let $\alpha \rightarrow \beta$ and $\gamma \rightarrow \delta$ be two rewrite rules such that α is unifiable with a non-variable subterm of γ. That is, there is a context $\mathbb{E}$, a non-variable term t and a substitution σ such that $\gamma = \mathbb{E}[t]$ and $\alpha\{\sigma\} = t\{\sigma\}$; as a consequence, the term $\gamma\{\sigma\} (= \mathbb{E}[t]\{\sigma\})$ can be reduced in two possible ways: $\mathbb{E}[t]\{\sigma\} \rightarrow \mathbb{E}[\beta]\{\sigma\}$ and $\gamma\{\sigma\} \rightarrow \delta\{\sigma\}$.

The pair $\left(\mathbb{E}[\beta]\{\sigma\}, \delta\{\sigma\}\right)$ is called a *critical pair*. If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \delta$ are the same rewrite rule, we also require that α is unifiable with a *proper* non-variable subterm of $\gamma (= \alpha)$.

Definition 2.9. A TRS is *orthogonal* if it is left-linear and there are no critical pairs.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Theorem 2.10. Every orthogonal TRS is CR.

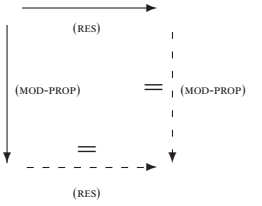
Theorem 2.11. Let C_1 and C_2 be two left-linear CRSs defined on two disjoint alphabets. Then $C_1 \cup C_2$ is CR if and only if C_1 and C_2 are CR.

See van Oostrom (1994, Theorem 3.4.29, page 126) for a proof of Theorem 2.11.

We now state some lemmas that will be useful for proving that the R-calculus is CR.

Lemma 2.12. Rule (MOD-PROP) strongly commutes with (RES).

Proof. We prove that the diagram

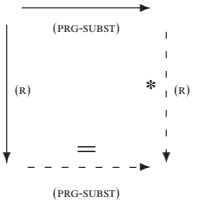


commutes.

If the (RES) redex has the form $\mathscr{R}\left[\iota; o; \rho \mid e\right] \uparrow$ and either the (MOD-PROP) redex is $\mathscr{R}\left[\iota; o; \rho \mid e\right]$ (which is a critical pair) or it is inside either $\mathscr{R}$, or o or ρ, the diagram strongly commutes by applying (RES) as horizontal step and no vertical step; in all other cases the diagram strongly commutes by applying (RES) as a horizontal step and (MOD-PROP) as a vertical step. □

Lemma 2.13. Rule (PRG-SUBST) commutes with respect to itself and the rules (MOD-PROP), (RES), (SUM-LIFT), (R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (RUN-LIFT) and (RES-LIFT).

Proof. From Theorem 2.5, we have to prove that the diagram



commutes, where (R) =(PRG-SUBST), (MOD-PROP), (RES), (SUM-LIFT), (R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (RUN-LIFT) and (RES-LIFT). If the (PRG-SUBST) step duplicates the (R) redex (that is, this redex is inside the substituted term), then the diagram commutes by applying (R) twice as a vertical step and (PRG-SUBST) as a horizontal step. In all other cases, including the critical ones (for instance: (R) = (MOD-PROP) and the (R) redex has the form $\mathscr{R}\left[\iota; o; \rho \mid e\right]$, with $\left[\iota; o; \rho \mid e\right]$ the (PRG-SUBST) redex; (R) = (SUM-LIFT) and the (R) redex has the form $\left[\iota_1; o_1; \rho_1 \mid e\right] \uparrow + [\iota_2; o_2; \rho_2]$, with $\left[\iota_1; o_1; \rho_1 \mid e\right]$ the (PRG-SUBST) redex), the diagram commutes by applying (R) as a vertical step and (PRG-SUBST) as a horizontal step. □

We can now state the Church–Rosser result.

Theorem 2.14 (Church–Rosser). The reduction relation $\longrightarrow$ is confluent.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$
\begin{aligned}
\ll x \gg &= x \\
\ll \left[\iota;\, X_j \overset{j\in J}{\mapsto} e_j;\, x_k \overset{k\in K}{\mapsto} e_k\right] \gg &= \left[\iota;\, X_j \overset{j\in J}{\mapsto} \ll e_j \gg;\, x_k \overset{k\in K}{\mapsto} \ll e_k \gg\right] \\
\ll e_1 + e_2 \gg &= \ll e_1 \gg + \ll e_2 \gg \\
\ll {}_{\sigma^\iota}|e|_{\sigma^o} \gg &= {}_{\sigma^\iota}|\ll e \gg|_{\sigma^o} \\
\ll \mathsf{freeze}_\sigma e \gg &= \mathsf{freeze}_\sigma \ll e \gg \\
\ll e.X \gg &= (\ll e \gg\downarrow_X)\uparrow
\end{aligned}
$$

Fig. 7. Translation of *CMS* into the *R*-calculus

Proof. Let R_C denote the reduction relation consisting only of the rules (MOD-PROP), (PRG-SUBST), (RES), (SUM-LIFT), (R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (RUN-LIFT) and (RES-LIFT). From Theorem 2.6, Fact 2.3 and Lemmas 2.12 and 2.13 (all other pairs of rules in R_C commute trivially since they are non-overlapping), we have that R_C is CR.

Furthermore, let R_M denote the reduction relation consisting of the remaining rules, that is, (M-SUM), (M-FREEZE), (M-REDUCT) and (RUN). Trivially, R_M is an orthogonal CRS, so, by Theorem 2.10, it is CR.

Now, clearly, the *R*-calculus can be obtained as the disjoint union of the two left-linear CRSs R_C and R_M, so we can conclude by Theorem 2.11. □

2.7. *Subsuming module calculi*

We now show that the *R*-calculus is an extension of traditional module calculi, such as *CMS*.

Since the *CMS* syntax is a subset of the syntax of the *R*-calculus, except for the selection operator, we just have to encode selection, and this can be done by using the run and result operators, as shown in Figure 7. In this way, the semantics by unfolding of *CMS* selection is simulated in a way that is similar to what happens in the m-calculus (Wells and Vestergaard 2000), where the selected output component is evaluated in the context of all local module definitions. This is illustrated by the following example.

Consider the *CMS* expression

$$(\mathsf{freeze}_{Y\mapsto Y}([y \mapsto Y\,;\, X \mapsto x;\, x \mapsto 2 + y] + [;\, Y \mapsto 1;\,])).X\,,$$

which in *CMS* reduces as follows:

$$
\begin{aligned}
&\xrightarrow{\text{(SUM)}} && (\mathsf{freeze}_{Y\mapsto Y}[y \mapsto Y\,;\, X \mapsto x, Y \mapsto 1;\, x \mapsto 2 + y]).X \\
&\xrightarrow{\text{(FREEZE)}} && [;\, X \mapsto x, Y \mapsto 1;\, x \mapsto 2 + y, y \mapsto 1].X \\
&\xrightarrow{\text{(SEL)}} && [;\, Y_1 \mapsto 2 + y;\, x \mapsto 2 + y, y \mapsto 1].Y_1 \\
&\xrightarrow{\text{(SEL)}} && 2 + [;\, Y_2 \mapsto 1;\, x \mapsto 2 + y, y \mapsto 1].Y_2 \\
&\xrightarrow{\text{(SEL)}} && 2 + 1 \longrightarrow 3\,.
\end{aligned}
$$

This is translated into

$$(\mathsf{freeze}_{Y\mapsto Y}([y \mapsto Y\,;\, X \mapsto x;\, x \mapsto 2 + y] + [;\, Y \mapsto 1;\,]))\downarrow_X\uparrow\,,$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

which in the R-calculus reduces as follows:

$$
\begin{aligned}
&\xrightarrow{\text{(SUM)}} && (\mathsf{freeze}_{Y\mapsto Y}\,[y \mapsto Y\,;\, X \mapsto x, Y \mapsto 1;\, x \mapsto 2 + y])\downarrow_X \uparrow \\
&\xrightarrow{\text{(FREEZE)}} && [;\, X \mapsto x, Y \mapsto 1;\, x \mapsto 2 + y, y \mapsto 1]\downarrow_X \uparrow \\
&\xrightarrow{\text{(RUN)}} && \big[;\, X \mapsto x;\, x \mapsto 2 + y, y \mapsto 1 \mid x\big]\uparrow \\
&\xrightarrow{\text{(PRG-SUBST)}} && \big[;\, X \mapsto x;\, x \mapsto 2 + y, y \mapsto 1 \mid 2 + y\big]\uparrow \\
&\xrightarrow{\text{(PRG-SUBST)}} && \big[;\, X \mapsto x;\, x \mapsto 2 + y, y \mapsto 1 \mid 2 + 1\big]\uparrow \\
&\longrightarrow && \big[;\, X \mapsto x;\, x \mapsto 2 + y, y \mapsto 1 \mid 3\big]\uparrow \\
&\xrightarrow{\text{(RES)}} && 3\,.
\end{aligned}
$$

See Fagorzi (2005) for a formal statement and a proof outline of the correctness of the simulation (that is, the fact that the translation preserves the *CMS* semantics).

3. Type systems

In this section, we illustrate three different type systems for the calculus. All of them prevent errors caused by an ill-formed application of module operators, analogously to what happens in standard module calculi such as *CMS* (Ancona and Zucca 2002)[†]. However, they differ in the way they handle 'missing component' errors, that is, situations in which program execution requires a module component that is neither currently available nor can be provided by reconfiguration steps.

First we show a *CMS*-like type system, that is, a type system analogous to that given in Ancona and Zucca (2002) for the *CMS* calculus. More precisely, the rules for module operators are pretty much the same[‡], while the rules for the operators run and result are driven by the rule for *CMS* selection. In this type system, programs can only be extracted from configurations with no input components.

Then we present a more sophisticated type system that also allows, in some cases, program extraction from configurations with input components. Indeed, a dependency analysis ensures that the program will never need these components.

Finally, we consider an alternative to both the refined but expensive dependency analysis and the cheaper but more restrictive *CMS*-like analysis. That is, we present a type system that allows program extraction from any configuration, delaying at run-time checks about the availability of required components. To this end, we extend the semantics of the R-calculus to raise an error if a required component cannot become available by performing reconfiguration steps.

3.1. *A CMS-like type system*

In this section we define $\mathscr{T}^{\mathsf{CMS}}$, which is a type system for the R-calculus that forms a natural extension to the one given in Ancona and Zucca (2002) for *CMS*. Indeed, as

[†] In more detail, the statically performed checks are made on the compatibility of input components and the disjointness of output components in sum, and the existence of an output name in renaming, freeze and run.

[‡] The only differences are that the typing rules for freeze and reduct operators are relaxed according to their more liberal semantics, and that to simplify the presentation we do not consider type annotations in terms.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

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

Fig. 8. $\mathscr{T}^{\mathsf{CMS}}$ types

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

Fig. 9. $\mathscr{T}^{\mathsf{CMS}}$ well-formed types

happens in *CMS* for selection, here the extraction of the running program is allowed only on configurations with no input components.

Types are defined in Figure 8, and include *module types* and *configuration types*.

A module type consists of an *input signature* π^{ι} and an *output signature* π^{o}. A signature is a sequence $X_i : \tau_i{}^{i \in I}$ of pairs consisting of a component name and a type. In the following, we will identify all signatures that represent the same set of pairs (that is, the ordering and repetitions are immaterial).

In a configuration type, the first two components π^{ι} and π^{o} have the same meaning as for module types, while τ is the type of the program running in the configuration.

Both module and configuration types are well formed if the signatures π^{ι} and π^{o} turn out to be maps from component names into well-formed types. This is formalised by the rules in Figure 9.

In the rest of the paper $_ , _$ denotes sequence concatenation.

Figure 10 gives the typing rules for deriving judgments of the form $\Gamma \vdash e : \tau$, with the meaning 'e is a well-formed expression with type τ in the context Γ'.

A context Γ is a finite (possibly empty) sequence of assignments of well-formed types to variables where variable repetition is allowed.

The predicate $\Gamma(x) = \tau$ is defined inductively as follows:

— $\varnothing(x) = \tau$ is false for any variable x and type τ;

— $(\Gamma, x : \tau)(x') = \begin{cases} \tau & \text{if } x = x' \\ \tau' & \text{if } x \neq x' \text{ and } \Gamma(x') = \tau' . \end{cases}$

We will use the following auxiliary functions and predicates:

— $\mathsf{dom}(\varnothing) = \varnothing$ and $\mathsf{dom}(\Gamma, x : \tau) = \mathsf{dom}(\Gamma) \cup \{x\}$;

— $\Gamma \subseteq \Gamma'$ if and only if for all variables x, if $\Gamma(x) = \tau$, then $\Gamma'(x) = \tau$.

In the judgments we identify all contexts Γ and Γ' such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \subseteq \Gamma$.

Rule (VAR) is standard.

Rule (M-BASIC) says that a basic module is well formed in a context Γ if there exists a context $x_p : \tau_p{}^{p \in I \cup K}$ assigning types to both deferred and local variables such that each defining expression of either an output or local component has a type in Γ extended with this context, and, moreover, the resulting module type is well formed.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

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

$$\text{(M-BASIC)}\ \frac{\left\{\Gamma, x_p : \tau_p{}^{p\in I\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\} \vdash \left[X_i : \tau_i{}^{i\in I}; X_j : \tau_j{}^{j\in J}\right]}{\Gamma \vdash \left[x_i \overset{i\in I}{\mapsto} X_i; X_j \overset{j\in J}{\mapsto} e_j; x_k \overset{k\in K}{\mapsto} e_k\right] : \left[X_i : \tau_i{}^{i\in I}; X_j : \tau_j{}^{j\in J}\right]}$$

$$\text{(M-SUM)}\ \frac{\Gamma \vdash e_1 : \left[\pi_1^I; \pi_1^O\right] \quad \Gamma \vdash e_2 : \left[\pi_2^I; \pi_2^O\right]}{\Gamma \vdash e_1 + e_2 : \left[\pi_1^I, \pi_2^I; \pi_1^O, \pi_2^O\right]} \quad \begin{array}{l}\pi_1^I \text{ and } \pi_2^I \text{ compatible}\\ \mathsf{dom}(\pi_1^O) \cap \mathsf{dom}(\pi_2^O) = \varnothing\end{array}$$

$$\text{(M-REDUCT)}\ \frac{\Gamma \vdash e : \left[\pi_1^I, \pi_2^I; \pi^O\right]}{\Gamma \vdash {}_{\sigma^I|}e_{|\sigma^O} : \left[\tilde{\pi}^I, \pi_2^I; \tilde{\pi}^O\right]} \quad \begin{array}{l}\mathsf{dom}(\sigma^I) \cap \mathsf{dom}(\pi_2^I) = \varnothing\\ \tilde{\pi}^I \text{ and } \pi_2^I \text{ compatible}\\ \sigma^I|_{\mathsf{dom}(\pi_1^I)} : \pi_1^I \to \tilde{\pi}^I\\ \sigma^O : \tilde{\pi}^O \to \pi^O\end{array}$$

$$\text{(M-FREEZE)}\ \frac{\Gamma \vdash e : \left[\pi_1^I, \pi_2^I; \pi^O\right]}{\Gamma \vdash \mathsf{freeze}_\sigma e : \left[\pi_2^I; \pi^O\right]} \quad \begin{array}{l}\mathsf{dom}(\sigma) \cap \mathsf{dom}(\pi_2^I) = \varnothing\\ \sigma|_{\mathsf{dom}(\pi_1^I)} : \pi_1^I \to \pi^O\end{array}$$

$$\text{(BASIC)}\ \frac{\begin{array}{l}\Gamma, x_p : \tau_p{}^{p\in I\cup K} \vdash e : \tau\\ \left\{\Gamma, x_p : \tau_p{}^{p\in I\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\} \vdash \left[X_i : \tau_i{}^{i\in I}; X_j : \tau_j{}^{j\in J}; \tau\right]\end{array}}{\Gamma \vdash \left[x_i \overset{i\in I}{\mapsto} X_i; X_j \overset{j\in J}{\mapsto} e_j; x_k \overset{k\in K}{\mapsto} e_k \mid e\right] : \left[X_i : \tau_i{}^{i\in I}; X_j : \tau_j{}^{j\in J}; \tau\right]}$$

$$\text{(SUM)}\ \frac{\Gamma \vdash e_1 : \left[\pi_1^I; \pi_1^O; \tau\right] \quad \Gamma \vdash e_2 : \left[\pi_2^I; \pi_2^O\right]}{\Gamma \vdash e_1 + e_2 : \left[\pi_1^I, \pi_2^I; \pi_1^O, \pi_2^O; \tau\right]} \quad \begin{array}{l}\pi_1^I \text{ and } \pi_2^I \text{ compatible}\\ \mathsf{dom}(\pi_1^O) \cap \mathsf{dom}(\pi_2^O) = \varnothing\end{array}$$

$$\text{(REDUCT)}\ \frac{\Gamma \vdash e : \left[\pi_1^I, \pi_2^I; \pi^O; \tau\right]}{\Gamma \vdash {}_{\sigma^I|}e_{|\sigma^O} : \left[\tilde{\pi}^I, \pi_2^I; \tilde{\pi}^O; \tau\right]} \quad \begin{array}{l}\mathsf{dom}(\sigma^I) \cap \mathsf{dom}(\pi_2^I) = \varnothing\\ \tilde{\pi}^I \text{ and } \pi_2^I \text{ compatible}\\ \sigma^I|_{\mathsf{dom}(\pi_1^I)} : \pi_1^I \to \tilde{\pi}^I\\ \sigma^O : \tilde{\pi}^O \to \pi^O\end{array}$$

$$\text{(FREEZE)}\ \frac{\Gamma \vdash e : \left[\pi_1^I, \pi_2^I; \pi^O; \tau\right]}{\Gamma \vdash \mathsf{freeze}_\sigma e : \left[\pi_2^I; \pi^O; \tau\right]} \quad \begin{array}{l}\mathsf{dom}(\sigma) \cap \mathsf{dom}(\pi_2^I) = \varnothing\\ \sigma|_{\mathsf{dom}(\pi_1^I)} : \pi_1^I \to \pi^O\end{array}$$

$$\text{(RUN)}\ \frac{\Gamma \vdash e : [\pi^I; \pi^O]}{\Gamma \vdash e{\downarrow_X} : [\pi^I; \pi^O; \pi^O(X)]}$$

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

Fig. 10. $\mathscr{T}^{\mathsf{CMS}}$ typing rules

In rule (M-SUM), the two resulting signatures are obtained by concatenating the signatures of the modules in the arguments, and the two side-conditions allow sharing of input components having the same name and type and prevent output components from being shared.

In rules (M-REDUCT) and (M-FREEZE), the fact that the side-conditions have the form $\sigma : \pi_1 \to \pi_2$ ensures that the renaming σ preserves types; formally, this means that $\sigma : \mathsf{dom}(\pi_1) \to \mathsf{dom}(\pi_2)$ and $\sigma(X) = Y \Rightarrow \pi_1(X) = \pi_2(Y)$.

Rule (M-REDUCT) allows the independent renaming (through renamings σ^I and σ^O) of input and output names in the module, provided the types are preserved (the last two

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

side-conditions). The input renaming is a map from old to new names, while the output renaming is a map from new to old names.

Unlike the original formulation in Ancona and Zucca (2002), we do not require that the domain of the input renaming coincides with the set of names in the input signature of the module. Hence, the renaming only affects a subset of input names ($\mathsf{dom}(\pi_1^I)$, as specified by the first side-condition), and can also be defined on names not present in the module. The definition of σ^I on these names is simply ignored (by taking the restriction of σ^I to the domain of π_1^I).

The last two side-conditions, as well as guaranteeing type preservation, determine the resulting signature of the module. Note that the resulting input signature is obtained by concatenating the unaffected and the new input signatures. These two signatures must be compatible (as required by the second side-condition), since the domains of $\tilde{\pi}^I$ and π_2^I may not be disjoint.

Rule (M-FREEZE) allows the linking (through the renaming σ) of some input names to output names of the module, provided the types are preserved (second side-condition). The first side-condition specifies that π_1^I is the set of input names that must be frozen. These names will be removed in the resulting (input) signature.

Moreover, as happens in rule (M-REDUCT) for input renamings, and unlike what happens in Ancona and Zucca (2002), we allow the renaming σ to be defined also on names not present in the input signature of the module. Again, the definition of σ on these names is simply ignored (by taking the restriction of σ to the domain of π_1^I).

Rule (BASIC) computes the first two components of the configuration type as rule (M-BASIC) for a basic module, while τ is the type of the running program in the context extended with types for all (deferred and local) variables. Moreover, the resulting configuration type must be well formed.

Rules (SUM), (REDUCT) and (FREEZE) are the obvious extensions of the equivalent rules for modules, where the type of the running program is just propagated.

Rule (RUN) states that the configuration obtained by running the output component X of a module has the same input and output signatures, and, as its program type, the type of X (this implicitly implies $X \in \mathsf{dom}(\pi^O)$).

Rule (RES) allows us to extract the program running in a configuration with no input components. We will be able to refine this requirement when we introduce a dependency relation in the next section. Obviously, the resulting type is that of the program running in the configuration.

We have that types inferred by the given type system are always well formed, as stated by the following proposition.

Proposition 3.1. If $\Gamma \vdash e : \tau$, then $\vdash \tau$.

Proof. The proof is an easy induction on the typing rules. □

We now show soundness of the $\mathscr{T}^{\mathsf{CMS}}$ type system by proving the usual progress and subject reduction properties. We first give some lemmas that are needed for the proof of the progress property.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$\frac{\vdash \tau \diamond}{\vdash [\varnothing; \pi^o; \tau] \diamond} \qquad \frac{}{\vdash [\pi^\iota; \pi^o] \diamond}$$

Fig. 11. $\mathscr{T}^{\mathsf{CMS}}$ type system: closed types

Lemma 3.2. Given $e \in \mathsf{Exp}$ and $x \in \mathsf{FV}(e)$:

1 There exists $\mathscr{E}$ such that $e = \mathscr{E}[x]$ with $x \notin \mathsf{HB}(\mathscr{E})$.
2 If $\Gamma \vdash e : \tau$, then $x \in \mathsf{dom}(\Gamma)$.

Proof. The proof is an easy induction on the structure of e. □

Recall that the set Val of the values, ranged over by v, was defined in Figure 6.

Lemma 3.3 (Canonical forms). Given $v \in \mathsf{Val}$:

— If $\Gamma \vdash v : [\pi^\iota; \pi^o]$, then v has either the form $[\iota; o; \rho]$ or the form $\left[\iota; o; \rho \mid v'\right]\uparrow$.
— If $\Gamma \vdash v : [\pi^\iota; \pi^o; \tau]$, then v has either the form $\left[\iota; o; \rho \mid v'\right]$ or the form $\left[\iota; o; \rho \mid v'\right]\uparrow$.

Proof. The proof is an easy check. □

In order to prove the progress property (Corollary 3.7) inductively, we need a more general property (Theorem 3.6), which holds for all well-typed expressions. Before formally stating this property, we will give a brief illustration of the intuition lying behind it.

The soundness property, stating that well-typed terms do not get stuck, usually concerns ground terms, that is, terms without free variables. More precisely, subject reduction (the preservation of types by reduction) holds for all well-typed terms, whereas progress (well-typed terms are either values or can be reduced) only holds on ground terms. Indeed, a term containing a free variable can get stuck, but still be well typed since, intuitively, it can be inserted in a context where the variable is provided and leads to a well-typed ground term.

In the $\mathscr{T}^{\mathsf{CMS}}$ type system, the progress property holds on terms that are not only ground, but also have *closed type*, that is, a type where each configuration level has no input components. In particular, the top configuration level has no input components. The judgment $\vdash \tau \diamond$, with the meaning 'τ is a closed type', is defined in Figure 11.

Indeed, a term of a non-closed type can get stuck, but still be well typed since, intuitively, it can be inserted in a context where the missing component can be provided by reconfiguration steps, as illustrated by the following examples.

Example 3.4. The configuration

$$e_1 = \left[z \mapsto Z, x \mapsto X; X \mapsto 3; y \mapsto 1, w \mapsto x + 1 \mid y + z\right]$$

has type $[X : \mathsf{int}, Z : \mathsf{int}; X : \mathsf{int}; \mathsf{int}]$ and gets stuck, since program execution needs the input component Z. However, we can obtain a term of closed type by inserting e_1 into a context where both Z and X are provided, such as $\mathsf{freeze}_{X \mapsto X, Z \mapsto Z}(e_1 + [; Z \mapsto 2;])$ of type $[; X : \mathsf{int}, Z : \mathsf{int}; \mathsf{int}]$. Also, the configuration $e_2 = \left[z \mapsto Z; Z \mapsto 2; \mid \left[x \mapsto X; ; \mid x + z\right]\right]$ has a non-closed type $[Z : \mathsf{int}; Z : \mathsf{int}; [X : \mathsf{int}; ; \mathsf{int}]]$, and program execution needs input components at each of the two levels. Even in this case, we can obtain a term

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$\mathscr{B} ::= \Box \mid \left[\iota; o; \rho \mid \mathscr{B}\right].$$

Fig. 12. Contexts of basic configurations

of closed type by applying suitable module operators to the configuration: for example, $\mathsf{freeze}_{X\mapsto X}((\mathsf{freeze}_{Z\mapsto Z} e_2)\uparrow + [; X \mapsto 3;])$, which has type $[; X : \mathsf{int}; \mathsf{int}]$.

Well-typed terms that have a non-closed type can get stuck. However, in this case they either contain a free variable, or they are basic configurations where the running program depends on a deferred variable, which may be running inside a context of basic configurations, which are defined in Figure 12. This is formalised below.

Fact 3.5. If $\Gamma \vdash \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right] : \tau$ with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$, then τ is a non-closed configuration type.

Proof. The proof is an easy check. □

Theorem 3.6. If $\Gamma \vdash e : \tau$, then (at least) one of the following cases holds:

1 $e \in \mathsf{Val}$.
2 $e \longrightarrow e'$, for some $e' \in \mathsf{Exp}$.
3 $e = \mathscr{E}[x]$, $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$.
4 $e = \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$, $x \in \mathsf{dom}(\iota)$.

Proof. The proof is by induction on the typing rules.

(VAR) We derive $\Gamma \vdash x : \Gamma(x)$, so case 3 holds.

(M-BASIC) We derive $\Gamma \vdash \left[x_i \overset{i\in I}{\mapsto} X_i; X_j \overset{j\in J}{\mapsto} e_j; x_k \overset{k\in K}{\mapsto} e_k\right] : \left[X_i : {\tau_i}^{i\in I}; X_j : {\tau_j}^{j\in J}\right]$, so case 1 holds.

(M-SUM) We derive $\Gamma \vdash e_1 + e_2 : \left[\pi_1^\iota,\pi_2^\iota; \pi_1^o,\pi_2^o\right]$. By the induction hypothesis on the premise $\vdash e_1 : \left[\pi_1^\iota; \pi_1^o\right]$, one of cases 1–3 holds (case 4 is impossible by Fact 3.5).

— If $e_1 \in \mathsf{Val}$, then, by Lemma 3.3, there are the following two cases:

(A) If $e_1 \equiv [\iota_1; o_1; \rho_1]$, we apply the induction hypothesis on the premise $\vdash e_2 : \left[\pi_2^\iota; \pi_2^o\right]$, one of 1–3 holds (case 4 is impossible by Fact 3.5).

(a) If $e_2 \in \mathsf{Val}$, then, by Lemma 3.3, there are two cases to be considered:

(i) If $e_2 \equiv [\iota_2; o_2; \rho_2]$,

$$[\iota_1; o_1; \rho_1] + [\iota_2; o_2; \rho_2] \xrightarrow{\text{(M-SUM)}} [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2] .$$

Note that all (implicit and explicit) side-conditions of the rule are satisfied: indeed, the only condition that cannot be guaranteed by α-conversion is $\mathsf{dom}(o_1) \cap \mathsf{dom}(o_2) = \varnothing$, which follows from the second side-condition of typing rule (M-SUM);

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(ii) If $e_2 \equiv \left[\iota_2; o_2; \rho_2 \mid v_2\right]\uparrow$,

$$[\iota_1; o_1; \rho_1] + \left[\iota_2; o_2; \rho_2 \mid v_2\right]\uparrow \xrightarrow{\text{(R-SUM-LIFT)}} \left[\iota_2; o_2; \rho_2 \mid [\iota_1; o_1; \rho_1] + v_2\right]\uparrow .$$

(b) If $e_2 \longrightarrow e_2'$, the thesis follows by applying the reduction rule $(\mathscr{E})$.

(c) If $e_2 = \mathscr{E}[x]$ with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then, we also have $[\iota_1; o_1; \rho_1] + \mathscr{E}$ is a one-hole context with $x \notin \mathsf{HB}([\iota_1; o_1; \rho_1] + \mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$.

(B) If $e_1 \equiv \left[\iota_1; o_1; \rho_1 \mid v_1\right]\uparrow$,

$$\left[\iota_1; o_1; \rho_1 \mid v_1\right]\uparrow + e_2 \xrightarrow{\text{(SUM-LIFT)}} \left[\iota_1; o_1; \rho_1 \mid v_1 + e_2\right]\uparrow .$$

Note that the side-conditions of this rule can always be satisfied by α-conversion.

— If $e_1 \longrightarrow e_1'$, the thesis follows by application of the reduction rule $(\mathscr{E})$.

— If $e_1 = \mathscr{E}[x]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then, also, $\mathscr{E} + e_2$ is a one-hole context with $x \notin \mathsf{HB}(\mathscr{E} + e_2)$ and $x \in \mathsf{dom}(\Gamma)$.

(M-REDUCT) We derive $\Gamma \vdash {}_{\sigma^\iota|}e_{|\sigma^o} : \left[\tilde{\pi}^\iota, \pi_2^\iota; \tilde{\pi}^o\right]$. By the induction hypothesis on the premise $\Gamma \vdash e : \left[\pi_1^\iota, \pi_2^\iota; \pi^o\right]$, one of 1–3 holds (case 4 is impossible by Fact 3.5).

— If $e \in \mathsf{Val}$, then, by Lemma 3.3, there are the following two cases:

(A) If $e \equiv [\iota; o; \rho]$, from the third side-condition we have $\sigma^\iota|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \tilde{\pi}^\iota$, hence, for a partition of ι into ι_1, ι_2 such that $\mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma^\iota) = \varnothing$ and $\mathsf{cod}(\iota_1) \subseteq \mathsf{dom}(\sigma^\iota)$, we have

$${}_{\sigma^\iota|}[\iota_1, \iota_2; o; \rho]_{|\sigma^o} \xrightarrow{\text{(M-REDUCT)}} [\sigma^\iota \circ \iota_1, \iota_2; o \circ \sigma^o; \rho] .$$

Note that all (implicit and explicit) side-conditions of the rule are satisfied. Indeed, $\mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma^\iota) = \varnothing$ and $\mathsf{cod}(\iota_1) \subseteq \mathsf{dom}(\sigma^\iota)$ are satisfied by the chosen partition, and $\mathsf{cod}(\sigma^o) \subseteq \mathsf{dom}(o)$ holds by the fourth side-condition of the typing rule (M-REDUCT).

(B) If $e \equiv \left[\iota; o; \rho \mid v\right]\uparrow$,

$${}_{\sigma^\iota|}\left[\iota; o; \rho \mid v\right]\uparrow_{|\sigma^o} \xrightarrow{\text{(REDUCT-LIFT)}} \left[\iota; o; \rho \mid {}_{\sigma^\iota|}v_{|\sigma^o}\right]\uparrow .$$

— If $e \longrightarrow e'$, the thesis follows by application of the reduction rule $(\mathscr{E})$.

— If $e = \mathscr{E}[x]$ with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then, also, ${}_{\sigma^\iota|}\mathscr{E}_{|\sigma^o}$ is a one-hole context with $x \notin \mathsf{HB}\left({}_{\sigma^\iota|}\mathscr{E}_{|\sigma^o}\right)$ and $x \in \mathsf{dom}(\Gamma)$.

(M-FREEZE) We derive $\Gamma \vdash \mathsf{freeze}_\sigma e : \left[\pi_2^\iota; \pi^o\right]$. By the induction hypothesis on the premise $\Gamma \vdash e : \left[\pi_1^\iota, \pi_2^\iota; \pi^o\right]$, one of 1–3 holds (case 4 is impossible by Fact 3.5).

— If $e \in \mathsf{Val}$, then, by Lemma 3.3, there are the following two cases:

(A) If $e \equiv [\iota; o; \rho]$, from the second side-condition of the typing rule (M-FREEZE), we have $\sigma|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \pi^o$, and hence for a partition of ι into ι_1, ι_2 such that $\mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma) = \varnothing$ and $\mathsf{cod}(\iota_1) \subseteq \mathsf{dom}(\sigma)$,

$$\mathsf{freeze}_\sigma\left[x_i \overset{i\in I}{\mapsto} X_i, \iota_2; o; \rho\right] \xrightarrow{\text{(M-FREEZE)}} \left[\iota_2; o; \rho, x_i \overset{i\in I}{\mapsto} o(\sigma(X_i))\right]$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

with $\iota_1 = x_i \overset{i\in I}{\mapsto} X_i$. Note that all (implicit and explicit) side-conditions of the rule are satisfied. Indeed, $\mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma) = \varnothing$ and $\{X_i \mid i \in I\} = \mathsf{cod}(\iota_1) \subseteq \mathsf{dom}(\sigma)$ are satisfied by the chosen partition, and $\{\sigma(X_i) \mid i \in I\} \subseteq \mathsf{dom}(o)$ holds by the second side-condition of the typing rule (M-FREEZE).

(B) If $e \equiv [\iota; o; \rho \mid v]\uparrow$,

$$\mathsf{freeze}_\sigma [\iota; o; \rho \mid v]\uparrow \xrightarrow{\text{(REDUCT-LIFT)}} [\iota; o; \rho \mid \mathsf{freeze}_\sigma v]\uparrow .$$

— If $e \longrightarrow e'$, the thesis follows by application of the reduction rule ($\mathscr{E}$).

— If $e = \mathscr{E}[x]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then, also, $\mathsf{freeze}_\sigma \mathscr{E}$ is a one-hole context with $x \notin \mathsf{HB}(\mathsf{freeze}_\sigma \mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$.

(BASIC) We derive

$$\Gamma \vdash \left[x_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} e_j;\ x_k \overset{k\in K}{\mapsto} e_k \mid e\right] : \left[X_i : \tau_i{}^{i\in I};\ X_j : \tau_j{}^{j\in J};\ \tau\right] .$$

Set

$$[\iota; o; \rho \mid e] = \left[x_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} e_j;\ x_k \overset{k\in K}{\mapsto} e_k \mid e\right] .$$

By the induction hypothesis on the premise $\Gamma, x_p : \tau_p{}^{p\in I\cup K}, \vdash e : \tau$, one of cases 1–4 holds.

— If $e \in \mathsf{Val}$, case 1 holds.

— If $e \longrightarrow e'$, the thesis follows by application of the reduction rule ($\mathscr{E}$).

— If $e \equiv \mathscr{E}[x]$ with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma, x_p : \tau_p{}^{p\in I\cup K})$, there are the following three cases to be considered:

(A) If $x \neq x_p$ for all $p \in I \cup K$, then, also, $[\iota; o; \rho \mid \mathscr{E}[x]]$ is a one-hole context with $x \notin \mathsf{HB}([\iota; o; \rho \mid \mathscr{E}])$ and $x \in \mathsf{dom}(\Gamma)$.

(B) If $x = x_k$, $k \in K$, that is, $x \in \mathsf{dom}(\rho)$, then

$$[\iota; o; \rho \mid \mathscr{E}[x]] \xrightarrow{\text{(PRG-SUBST)}} [\iota; o; \rho \mid \mathscr{E}\{\rho(x)\}] .$$

(C) If $x = x_i$, $i \in I$, that is, $x \in \mathsf{dom}(\iota)$, then $[\iota; o; \rho \mid e] = \mathscr{B}[\iota; o; \rho \mid \mathscr{E}[x]]$, with $\mathscr{B} = \Box$.

— If $e = \mathscr{B}[\iota'; o'; \rho' \mid \mathscr{E}[x]]$ with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota')$, then $[\iota; o; \rho \mid e] = \mathscr{B}'[\iota'; o'; \rho' \mid \mathscr{E}[x]]$, with $\mathscr{B}' = [\iota; o; \rho \mid \mathscr{B}]$.

(SUM) We derive $\Gamma \vdash e_1 + e_2 : [\pi_1^\iota, \pi_2^\iota; \pi_1^o, \pi_2^o; \tau]$. By the induction hypothesis on the premise $\Gamma \vdash e_1 : [\pi_1^\iota; \pi_2^o; \tau]$, one of cases 1–4 holds.

— The first three cases are analogous to those for (M-SUM) (when $e_1, e_2 \in \mathsf{Val}$, with $e_1 \equiv [\iota_1; o_1; \rho_1 \mid v]$ and $e_2 \equiv [\iota_2; o_2; \rho_2]$, we apply (MOD-PROP)+(M-SUM)).

— If $e_1 = \mathscr{B}[\iota_1; o_1; \rho_1 \mid \mathscr{E}[x]]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota_1)$ (that is, e_1 has either the form $[\iota_1; o_1; \rho_1 \mid \mathscr{E}[x]]$ with $\mathscr{B} = \Box$, or the form

$$[\iota_1'; o_1'; \rho_1' \mid \mathscr{B}'[\iota_1; o_1; \rho_1 \mid \mathscr{E}[x]]]$$

with $\mathscr{B} = [\iota_1'; o_1'; \rho_1' \mid \mathscr{B}']$). In this case the thesis follows in a similar way to the case for (M-SUM) with $e_1 \in \mathsf{Val}$.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(REDUCT) We derive $\Gamma \vdash {}_{\sigma^\iota|}e_{|\sigma^o} : \left[\tilde{\pi}^\iota, \pi_2^\iota; \tilde{\pi}^o; \tau\right]$. By the induction hypothesis on the premise $\Gamma \vdash e : \left[\pi_1^\iota, \pi_2^\iota; \pi^o; \tau\right]$, one of cases 1–4 holds.

— The first three cases are analogous to those for (M-REDUCT) (when $e \equiv \left[\iota; o; \rho \mid v\right] \in \mathsf{Val}$, we apply (MOD-PROP)+(M-REDUCT)).

— If $e = \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$ (that is, e either has the form $\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$ with $\mathscr{B} = \Box$, or the form $\left[\iota'; o'; \rho' \mid \mathscr{B}'\left[\iota; o; \rho \mid \mathscr{E}[x]\right]\right]$ with $\mathscr{B} = \left[\iota'; o'; \rho' \mid \mathscr{B}'\right]$). In this case the thesis follows in a similar way to the case for (M-REDUCT) with $e \in \mathsf{Val}$.

(FREEZE) We derive $\Gamma \vdash \mathsf{freeze}_\sigma e : \left[\pi_2^\iota; \pi^o; \tau\right]$. By the induction hypothesis on the premise $\Gamma \vdash e : \left[\pi_1^\iota, \pi_2^\iota; \pi^o; \tau\right]$, one of 1–4 holds.

— The first three cases are analogous to those for (M-FREEZE) (when $e \equiv \left[\iota; o; \rho \mid v\right] \in \mathsf{Val}$, we apply (MOD-PROP)+(M-FREEZE)).

— If $e = \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$ (that is, e either has the form $\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$ with $\mathscr{B} = \Box$, or the form $\left[\iota'; o'; \rho' \mid \mathscr{B}'\left[\iota; o; \rho \mid \mathscr{E}[x]\right]\right]$ with $\mathscr{B} = \left[\iota'; o'; \rho' \mid \mathscr{B}'\right]$). In this case the thesis follows in a similar way to the case for (M-FREEZE) with $e \in \mathsf{Val}$.

(RUN) We derive $\Gamma \vdash e\downarrow_X : [\pi^\iota; \pi^o; \pi^o(X)]$. By the induction hypothesis on the premise $\Gamma \vdash e : [\pi^\iota; \pi^o]$, one of 1–3 holds (case 4 is impossible by Fact 3.5).

— If $e \in \mathsf{Val}$, by Lemma 3.3, there are the following two cases:

(A) If $e \equiv [\iota; o; \rho]$,

$$[\iota; o; \rho]\downarrow_X \xrightarrow{\text{(RUN)}} \left[\iota; o; \rho \mid o(X)\right].$$

Note that the implicit side-condition $X \in \mathsf{dom}(o)$ is satisfied since the expression $\pi^o(X)$ must be defined.

(B) If $e \equiv \left[\iota; o; \rho \mid v\right]\uparrow$,

$$\left[\iota; o; \rho \mid v\right]\uparrow\downarrow_X \xrightarrow{\text{(RUN-LIFT)}} \left[\iota; o; \rho \mid v\downarrow_X\right]\uparrow.$$

— If $e \longrightarrow e'$, the thesis follows by application of the reduction rule $(\mathscr{E})$.

— If $e = \mathscr{E}[x]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then, also, $\mathscr{E}\downarrow_X$ is a one-hole context with $x \notin \mathsf{HB}(\mathscr{E}\downarrow_X)$ and $x \in \mathsf{dom}(\Gamma)$.

(RES) We derive $\Gamma \vdash e\uparrow : \tau$. By the induction hypothesis on the premise $\Gamma \vdash e : [\varnothing; \pi^o; \tau]$, one of 1–4 holds.

— If $e \in \mathsf{Val}$, by Lemma 3.3, there are the following two cases:

(A) If $e \equiv \left[\varnothing; o; \rho \mid v\right]$, for some $v \in \mathsf{Val}$ ($\iota = \varnothing$ since $\pi^\iota = \varnothing$):

(a) If $\mathsf{FV}(v) \cap \mathsf{dom}(\rho) = \varnothing$, we have

$$\left[\varnothing; o; \rho \mid v\right]\uparrow \xrightarrow{\text{(RES)}} v.$$

Note that the side-condition $\mathsf{FV}(e) \cap \mathsf{dom}(\iota, \rho) = \varnothing$ is satisfied.

(b) Otherwise, $v = \mathscr{E}[x]$ with $x \in \mathsf{dom}(\rho)$ and we have

$$\left[\varnothing; o; \rho \mid \mathscr{E}[x]\right]\uparrow \xrightarrow{\text{(PRG-SUBST)}} \left[\varnothing; o; \rho \mid \mathscr{E}\{\rho(x)\}\right]\uparrow.$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(B) If $e \equiv \left[\iota; o; \rho \mid v\right]\uparrow$,

$$\left[\iota; o; \rho \mid v\right]\uparrow\uparrow \xrightarrow{\text{(RES-LIFT)}} \left[\iota; o; \rho \mid v\uparrow\right]\uparrow .$$

— If $e \longrightarrow e'$, the thesis follows by application of the reduction rule $(\mathscr{E})$.

— If $e = \mathscr{E}[x]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then, also, $\mathscr{E}\uparrow$ is a one-hole context with $x \notin \mathsf{HB}(\mathscr{E}\uparrow)$ and $x \in \mathsf{dom}(\Gamma)$.

— If $e = \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$, then, since $\pi^\iota = \varnothing$, we know e must have the form $\left[\varnothing; o'; \rho' \mid \mathscr{B}'\left[\iota; o; \rho \mid \mathscr{E}[x]\right]\right]$ (the case $e = \left[\iota; o; \rho \mid \mathscr{E}[x]\right]$ with $\mathscr{B} = \Box$ is impossible). Set $e' = \mathscr{B}'\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$. There are the following two cases:

(A) If $\mathsf{FV}(e') \cap \mathsf{dom}(\rho') = \varnothing$, then

$$\left[\varnothing; o'; \rho' \mid e'\right]\uparrow \xrightarrow{\text{(RES)}} e'.$$

Note that the side-condition $\mathsf{FV}(e') \cap \mathsf{dom}(\varnothing, \rho') = \varnothing$ is satisfied.

(B) If $\mathsf{FV}(e') \cap \mathsf{dom}(\rho') \neq \varnothing$, then, by Lemma 3.2, there exists $\mathscr{E}'$ such that $e' = \mathscr{E}'[y]$ with $y \notin \mathsf{HB}\left(\mathscr{E}'\right)$, $y \in \mathsf{dom}(\rho')$. Hence

$$\left[\varnothing; o'; \rho' \mid \mathscr{E}'[y]\right]\uparrow \xrightarrow{\text{(PRG-SUBST)}} \left[\varnothing; o'; \rho' \mid \mathscr{E}'\{\rho(y)\}\right]\uparrow . \qquad \Box$$

We can now prove the progress property as a special case of the previous theorem.

Corollary 3.7 (Progress). If $\varnothing \vdash e : \tau$ and $\vdash \tau \diamond$, then either $e \in \mathsf{Val}$ or $e \longrightarrow e'$ for some $e' \in \mathsf{Exp}$.

Proof. The thesis follows from Theorem 3.6. Indeed, the third case cannot occur since $\Gamma = \varnothing$, and the fourth case is impossible by Fact 3.5. $\Box$

The following lemmas are required to prove the Subject Reduction Theorem.

Lemma 3.8 (Weakening). If $\Gamma \vdash e : \tau$, then $\Gamma' \vdash e : \tau$ for any context Γ' such that $\Gamma \subseteq \Gamma'$.

Proof. The proof is an easy induction on the typing rules. $\Box$

Lemma 3.9 (Strengthening). If $\Gamma \vdash e : \tau$, then $\Gamma' \vdash e : \tau$ for any context Γ' such that $\Gamma' \subseteq \Gamma$ and $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma')$.

Proof. The proof is an easy induction on the typing rules. $\Box$

Lemma 3.10 (Substitution). If $\Gamma \vdash \mathscr{E}[x] : \tau$, $\Gamma(x) = \tau_x$ and $\Gamma \vdash e : \tau_x$, then $\Gamma \vdash \mathscr{E}\{e\} : \tau$.

Proof. The proof is by induction on the derivation of the judgment $\Gamma \vdash \mathscr{E}[x] : \tau$. $\Box$

Lemma 3.11. If $\Gamma \vdash \mathscr{R}\left[\iota; o; \rho \mid e\right] : [\pi^\iota; \pi^o; \tau]$, then $\Gamma, \Gamma^\iota, \Gamma^\rho \vdash e : \tau$, for some Γ^ι, Γ^ρ such that $\mathsf{dom}(\Gamma^\iota) = \mathsf{dom}(\iota)$ and $\mathsf{dom}(\Gamma^\rho) = \mathsf{dom}(\rho)$.

Proof. The proof is by induction on the structure of $\mathscr{R}$.

— $\Box$: in this case judgment $\Gamma \vdash \left[\iota; o; \rho \mid e\right] : [\pi^\iota; \pi^o; \tau]$ can be derived only by (BASIC), hence $\Gamma, \Gamma^\iota, \Gamma^\rho \vdash e : \tau$, with $\Gamma^\iota = \pi^\iota \circ \iota$ and Γ^ρ such that

$$\left\{\Gamma, \Gamma^\iota, \Gamma^\rho \vdash \rho(x) : \Gamma^\rho(x) \mid x \in \mathsf{dom}(\rho)\right\},$$

hence $\mathsf{dom}(\Gamma^\iota) = \mathsf{dom}(\iota)$ and $\mathsf{dom}(\Gamma^\rho) = \mathsf{dom}(\rho)$.

— All other cases can be proved easily by application of the induction hypothesis. $\Box$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Lemma 3.12. If $\Gamma \vdash \mathscr{R}\left[\iota; o; \rho \mid e\right] : \tau$ and $\mathscr{R}[\iota; o; \rho] \xrightarrow[\alpha]{} \mathscr{R}'\left[\iota'; o'; \rho'\right]$, then $\Gamma \vdash \mathscr{R}'\left[\iota'; o'; \rho' \mid \alpha(e)\right] : \tau$.

Proof. The proof is by induction on the structure of $\mathscr{R}$. □

Theorem 3.13 (Subject Reduction). If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.

Proof. We use induction on the reduction rules:

($\mathscr{E}$) It is easy to prove all cases, for any possible context $\mathscr{E}$, by exploiting the induction hypothesis.

(M-SUM) We derive

$$[\iota_1; o_1; \rho_1] + [\iota_2; o_2; \rho_2] \longrightarrow [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2]$$

and by hypothesis we have

$$\Gamma \vdash [\iota_1; o_1; \rho_1] + [\iota_2; o_2; \rho_2] : \tau \tag{1}$$

This last judgment can only be derived by using (M-SUM), hence $\tau = \left[\pi_1^\iota, \pi_2^\iota; \pi_1^o, \pi_2^o\right]$ and

$$\Gamma \vdash [\iota_h; o_h; \rho_h] : \left[\pi_h^\iota; \pi_h^o\right], \ h \in \{1,2\}. \tag{2}$$

We first observe that, for Proposition 3.1, τ must be well formed. Judgments in (2) can only be derived by using rule (M-BASIC), hence we must have

$$\begin{aligned} [\iota_h; o_h; \rho_h] &= \left[x_i \overset{i \in I_h}{\mapsto} X_i; X_j \overset{j \in J_h}{\mapsto} e_j; x_k \overset{k \in K_h}{\mapsto} e_k\right] \\ \left[\pi_h^\iota; \pi_h^o\right] &= \left[X_i : \tau_i{}^{i \in I_h}; X_j : \tau_j{}^{j \in J_h}\right] \end{aligned}$$

and

$$\left\{\Gamma, x_p : \tau_p{}^{p \in I_h \cup K_h} \vdash e_q : \tau_q \mid q \in J_h \cup K_h\right\} \tag{3}$$

with $h \in \{1,2\}$. We now apply to all the judgments in (3) the Strengthening Lemma (Lemma 3.9) with

$$\left(\Gamma \backslash \{x_q \mid q \in I_k \cup K_k\}\right), x_p : \tau_p{}^{p \in I_h \cup K_h} \subseteq \Gamma, x_p : \tau_p{}^{p \in I_h \cup K_h},$$

with $h,k \in \{1,2\}$ and $h \neq k$ (indeed, to perform the (M-SUM) reduction step, we must have $\mathsf{dom}(\iota_h, \rho_h) \cap \mathsf{dom}(\iota_k, \rho_k) = \varnothing$ and $\mathsf{dom}(\iota_k, \rho_k) \cap \mathsf{FV}([\iota_h; o_h; \rho_h]) = \varnothing$). Then, by applying to all resulting judgments the Weakening Lemma 3.8 with

$$\left(\Gamma \backslash \{x_q \mid q \in I_k \cup K_k\}\right), x_p : \tau_p{}^{p \in I_h \cup K_h} \subseteq x_p : \tau_p{}^{p \in I_1 \cup K_1 \cup I_2 \cup K_2},,$$

with $h,k \in \{1,2\}$ and $h \neq k$ (indeed, to perform the (M-SUM) reduction step we must have $\mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$), we get

$$\left\{x_p : \tau_p{}^{p \in I_1 \cup K_1 \cup I_2 \cup K_2},, \vdash e_q : \tau_q \mid q \in J_h \cup K_h\right\}, \ h \in \{1,2\}.$$

We can now give the following derivation:

$$\frac{\left\{x_p : \tau_p{}^{p \in I_1 \cup I_2 \cup K_1 \cup K_2}, \vdash e_q : \tau_q \mid q \in J_1 \cup J_2 \cup K_1 \cup K_2\right\} \vdash \tau}{\vdash [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2] : \left[\pi_1^\iota, \pi_2^\iota; \pi_1^o, \pi_2^o\right]} \text{(M-BASIC)}$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Indeed

$$[\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2] = \left[x_i \overset{i\in(I_1\cup I_2)}{\mapsto} X_i; X_j \overset{j\in(J_1\cup J_2)}{\mapsto} e_j; x_k \overset{k\in(K_1\cup K_2)}{\mapsto} e_k\right]$$
$$\left[\pi_1^\iota,\pi_2^\iota; \pi_1^o,\pi_2^o\right] = \left[X_i : \tau_i{}^{i\in(I_1\cup I_2)}; X_j : \tau_j{}^{j\in(J_1\cup J_2)}\right].$$

(M-REDUCT) We derive

$$_{\sigma^\iota|}[\iota_1, \iota_2; o; \rho]_{|\sigma^o} \longrightarrow [\sigma^\iota \circ \iota_1, \iota_2; o \circ \sigma^o; \rho]$$

with

$$\mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma^\iota) = \varnothing, \tag{1}$$

and by hypothesis we have

$$\Gamma \vdash {}_{\sigma^\iota|}[\iota_1, \iota_2; o; \rho]_{|\sigma^o} : \tau. \tag{2}$$

This last judgment can only be derived by using (M-REDUCT), hence $\tau = \left[\tilde{\pi}^\iota,\pi_2^\iota; \tilde{\pi}^o\right]$ and

$$\Gamma \vdash [\iota_1, \iota_2; o; \rho] : \left[\pi_1^\iota, \pi_2^\iota; \pi^o\right] \tag{3}$$
$$\sigma^\iota|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \tilde{\pi}^\iota \tag{4}$$
$$\sigma^o : \tilde{\pi}^o \to \pi^o \tag{5}$$
$$\mathsf{dom}(\sigma^\iota) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing. \tag{6}$$

We first observe that, for Proposition 3.1, τ must be well formed.
Judgment (3) can only be derived by using rule (M-BASIC), hence we must have

$$[\iota_1, \iota_2; o; \rho] = \left[x_i \overset{i\in I_1\cup I_2}{\mapsto} X_i; X_j \overset{j\in J}{\mapsto} e_j; x_k \overset{k\in K}{\mapsto} e_k\right],$$

with $x_i \overset{i\in I_h}{\mapsto} X_i = \iota_h, \in \{1, 2\}$,

$$\left[\pi_1^\iota, \pi_2^\iota; \pi^o\right] = \left[X_i : \tau_i{}^{i\in I_1\cup I_2}; X_j : \tau_j{}^{j\in J}\right]$$

and

$$\left\{\Gamma, x_p : \tau_p{}^{p\in I_1\cup I_2\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\}. \tag{7}$$

We can now consider the following subset of judgments in (7):

$$\left\{\Gamma, x_p : \tau_p{}^{p\in I_1\cup I_2\cup K} \vdash e_r : \tau_r \mid r \in R\right\} \tag{8}$$

with $X_r \overset{r\in R}{\mapsto} e_r = o \circ \sigma^o$ and $X_r : \tau_r{}^{r\in R} = \tilde{\pi}^o$, such that for all $r \in R$, $\sigma^o(X_r) = X_j$, $e_r = e_j$ and $\tau_r = \tau_j$, $j \in J$. Note that judgments in (8) are a (possibly equal) subset of those in (7) since, form (5), $\mathsf{cod}(\sigma^o) \subseteq \mathsf{dom}(o)$ and σ^o may be non-surjective.
Furthermore, if σ^ι is non-surjective, we have that

$$\sigma^\iota \circ \iota_1 = x_i \overset{i\in I_1}{\mapsto} \sigma^\iota(X_i), x_f \overset{f\in F}{\mapsto} X_f$$

such that for all $f \in F$, x_f is a fresh new dummy variable (recall that in $\sigma^\iota \circ \iota_1$ we have a fresh new dummy variable for each name in $\mathsf{dom}(\tilde{\pi}^\iota) \setminus \{\sigma^\iota(X) \mid X \in$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$\mathsf{cod}(\iota_1)\}$). We can consider ${\tau_f}^{f\in F}$ such that $\sigma^\iota(X_i) : {\tau_i}^{i\in I_1}, X_f : {\tau_f}^{f\in F} = \tilde{\pi}^\iota$ and apply to all judgments in (8) and to a subset of those in (7) the Weakening Lemma 3.8 with $\Gamma, x_p : {\tau_p}^{p\in I_1\cup I_2\cup K} \subseteq \Gamma, x_p : {\tau_p}^{p\in I_1\cup I_2\cup K}, x_f : {\tau_f}^{f\in F}$, giving:

$$\left\{\Gamma, x_p : {\tau_p}^{p\in I_1\cup I_2\cup K}, x_f : {\tau_f}^{f\in F} \vdash e_q : \tau_q \mid q \in K\right\} \tag{7'}$$

$$\left\{\Gamma, x_p : {\tau_p}^{p\in I_1\cup I_2\cup K}, x_f : {\tau_f}^{f\in F} \vdash e_r : \tau_r \mid r \in R\right\}. \tag{8'}$$

We can now give the following derivation:

$$\frac{\vdash \tau \quad (7') \quad (8')}{\vdash [\sigma^\iota \circ \iota_1, \iota_2; o \circ \sigma^o; \rho] : \tau}\ {\scriptstyle (\text{M-BASIC})}$$

indeed,

$$[\sigma^\iota \circ \iota_1, \iota_2; o \circ \sigma^o; \rho] = \left[x_i \overset{i\in I_1}{\mapsto} \sigma^\iota(X_i), x_f \overset{f\in F}{\mapsto} X_f, x_i \overset{i\in I_2}{\mapsto} X_i; X_r \overset{r\in R}{\mapsto} e_r; x_k \overset{k\in K}{\mapsto} e_k\right]$$

$$\tau = \left[\sigma^\iota(X_i) : {\tau_i}^{i\in I_1}, X_f : {\tau_f}^{f\in F}, X_i : {\tau_i}^{i\in I_2}; X_r : {\tau_r}^{r\in R}\right].$$

(M-FREEZE) We derive

$$\mathsf{freeze}_\sigma\left[x_i \overset{i\in I_1}{\mapsto} X_i, \iota; o; \rho\right] \longrightarrow \left[\iota; o; \rho, x_i \overset{i\in I_1}{\mapsto} o(\sigma(X_i))\right]$$

with

$$\mathsf{cod}(\iota) \cap \mathsf{dom}(\sigma) = \varnothing, \tag{1}$$

and by hypothesis we have

$$\Gamma \vdash \mathsf{freeze}_\sigma\left[x_i \overset{i\in I_1}{\mapsto} X_i, \iota; o; \rho\right] : \tau. \tag{2}$$

This last judgment can only be derived using (M-FREEZE), so $\tau = \left[\pi_2^\iota; \pi^o\right]$, with

$$\Gamma \vdash \left[x_i \overset{i\in I_1}{\mapsto} X_i, \iota; o; \rho\right] : \left[\pi_1^\iota, \pi_2^\iota; \pi^o\right] \tag{3}$$

$$\sigma|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \pi^o \tag{4}$$

$$\mathsf{dom}(\sigma) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing. \tag{5}$$

Judgment (3) can only be derived by using (M-BASIC), hence we must have

$$\left[x_i \overset{i\in I_1}{\mapsto} X_i, \iota; o; \rho\right] = \left[x_i \overset{i\in I_1\cup I_2}{\mapsto} X_i; X_j \overset{j\in J}{\mapsto} e_j; x_k \overset{k\in K}{\mapsto} e_k\right]$$

$$\left[\pi_1^\iota, \pi_2^\iota; \pi^o\right] = \left[X_i : {\tau_i}^{i\in I_1\cup I_2}; X_j : {\tau_j}^{j\in J}\right]$$

with $X_i : {\tau_i}^{i\in I_h} = \pi_h^\iota$, $h \in \{1, 2\}$, and

$$\left\{\Gamma, x_p : {\tau_p}^{p\in I_1\cup I_2\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\}. \tag{6}$$

Note that we have $\mathsf{dom}(\pi_1^\iota) = \{X_i \mid i \in I_1\}$ and

$$\mathsf{dom}(\pi_2^\iota) = \{X_i \mid i \in I_2\}. \tag{7}$$

Indeed, we have that $\mathsf{dom}(\pi_1^\iota, \pi_2^\iota) = \{X_i \mid i \in I_1 \cup I_2\}$ (from (3)) and $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\pi_1^\iota, \pi_2^\iota) = \pi_1^\iota$ ($\mathsf{dom}(\pi_1^\iota) \subseteq \mathsf{dom}(\sigma^\iota)$ from (4) and $\mathsf{dom}(\sigma^\iota) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

from (5)) and $\mathsf{dom}(\sigma) \cap \{X_i \mid i \in I_1 \cup I_2\} = \{X_i \mid i \in I_1\}$ ($\{X_i \mid i \in I_1\} \subseteq \mathsf{dom}(\sigma')$ from well-formedness of $o(\sigma(X_i))$ for all $i \in I$ and $\mathsf{cod}(\iota) \cap \mathsf{dom}(\sigma') = \varnothing$ from (1)).

We first observe that, for Proposition 3.1, τ must be well formed.

We can consider the following subset of judgments in (6):

$$\left\{\Gamma, x_p : {\tau_p}^{p \in I_1 \cup I_2 \cup K} \vdash o(\sigma(X_i)) : \tau_i \mid i \in I_1\right\}. \tag{8}$$

Note that judgments in (8) are a (possibly equal) subset of those in (6) since, from (4), for all $i \in I_1$, $\sigma(X_i) = X_j$ and $\tau_i = \tau_j$, with $j \in J$; moreover, $\sigma|_{\mathsf{dom}(\pi'_1)}$ may be non-surjective.

From (6) and (8) we can now give the following derivation:

$$\frac{\begin{array}{l}\vdash \tau \\ \left\{\Gamma, x_p : {\tau_p}^{p \in I_2 \cup K \cup I_1} \vdash e_q : \tau_q \mid q \in J \cup K\right\} \\ \left\{\Gamma, x_p : {\tau_p}^{p \in I_2 \cup K \cup I_1} \vdash o(\sigma(X_i)) : \tau_i \mid i \in I_1\right\}\end{array}}{\vdash \left[\iota;\, o;\, \rho, x_i \overset{i \in I_1}{\mapsto} o(\sigma(X_i))\right] : \tau}\ \textsc{(m-basic)}$$

Indeed:

$$\begin{aligned}&\left[\iota;\, o;\, \rho, x_i \overset{i \in I_1}{\mapsto} o(\sigma(X_i))\right] \\ &\quad = \left[x_i \overset{i \in I_2}{\mapsto} \sigma'(X_i);\, X_j \overset{j \in J}{\mapsto} e_j;\, x_k \overset{k \in K}{\mapsto} e_k, x_i \overset{i \in I_1}{\mapsto} o(\sigma(X_i))\right] \\ &\tau \;=\; \left[\sigma'(X_i) : {\tau_i}^{i \in I_2};\, X_j : {\tau_j}^{j \in J}\right].\end{aligned}$$

(mod-prop) We derive $\mathscr{R}\left[\iota;\, o;\, \rho \mid e\right] \longrightarrow \mathscr{R}'\left[\iota';\, o';\, \rho' \mid \alpha(e)\right]$, with $\Gamma \vdash \mathscr{R}\left[\iota;\, o;\, \rho \mid e\right] : \tau$. In this case the thesis follows by applying Lemma 3.12 to this last judgment and to the premise the reduction rule (mod-prop).

(prg-subst) We derive $\left[\iota;\, o;\, \rho \mid \mathscr{E}[x]\right] \longrightarrow \left[\iota;\, o;\, \rho \mid \mathscr{E}\{\rho(x)\}\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\rho)$, and, by hypothesis, we have $\Gamma \vdash \left[\iota;\, o;\, \rho \mid \mathscr{E}[x]\right] : \tau$. This last judgment can only be derived by using (basic), hence we must have

$$\begin{aligned}\left[\iota;\, o;\, \rho \mid \mathscr{E}[x]\right] &= \left[x_i \overset{i \in I_1 \cup I_2}{\mapsto} X_i;\, X_j \overset{j \in J}{\mapsto} e_j;\, x_k \overset{k \in K}{\mapsto} e_k \mid \mathscr{E}[x]\right] \\ \tau &= \left[X_i : {\tau_i}^{i \in I};\, X_j : {\tau_j}^{j \in J};\, \tau'\right].\end{aligned}$$

In particular, from the premises of (basic), we have $\Gamma, x_p : {\tau_p}^{p \in I \cup K} \vdash e : \tau'$ and, since $x \in \mathsf{dom}(\rho)$, we also have $\Gamma, x_p : {\tau_p}^{p \in I \cup K} \vdash \rho(x) : \Gamma^{\rho}(x)$. Therefore, using the Substitution Lemma (Lemma 3.10), we obtain $\Gamma, x_p : {\tau_p}^{p \in I \cup K} \vdash \mathscr{E}\{\rho(x)\} : \tau'$, and the thesis follows by application of the typing rule (basic), observing that τ must be well formed for Proposition 3.1.

(run) We derive $[\iota;\, o;\, \rho]{\downarrow_X} \longrightarrow \left[\iota;\, o;\, \rho \mid o(X)\right]$ and, by hypothesis, $\Gamma \vdash [\iota;\, o;\, \rho]{\downarrow_X} : \tau$. This judgment can only be derived by using (run), hence $\tau = [\pi^{\iota};\, \pi^{o};\, \pi^{o}(X)]$ and $\Gamma \vdash [\iota;\, o;\, \rho] : [\pi^{\iota};\, \pi^{o}]$. This last judgment can only be derived by applying

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(M-BASIC), hence we must have

$$[\iota; o; \rho] = \left[x_i \overset{i\in I}{\mapsto} X_i; X_j \overset{j\in J}{\mapsto} e_j; x_k \overset{k\in K}{\mapsto} e_k\right]$$
$$[\pi^\iota; \pi^o] = \left[X_i : {\tau_i}^{i\in I}; X_j : {\tau_j}^{j\in J}\right]$$

and

$$\left\{\Gamma, x_p : {\tau_p}^{p\in I\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\}. \tag{1}$$

We first observe that, for Proposition 3.1, τ must be well formed. We can now give the following derivation:

$$\frac{(1)\quad (2)\quad \Gamma, x_p : {\tau_p}^{p\in I\cup K} \vdash o(X) : \pi^o(X) \quad \vdash \tau}{\vdash \left[\iota; o; \rho \mid o(X)\right] : [\pi^\iota; \pi^o; \pi^o(X)]}\ (\text{BASIC})$$

(RES) We derive $\left(\mathscr{R}\left[\iota; o; \rho \mid e\right]\right)\uparrow \longrightarrow e$, with

$$\mathsf{FV}(e) \cap \mathsf{dom}(\iota, \rho) = \varnothing, \tag{1}$$

and, by hypothesis, we have $\Gamma \vdash \left(\mathscr{R}\left[\iota; o; \rho \mid e\right]\right)\uparrow : \tau$. This last judgment can only be derived using the typing rule (RES), hence we must have $\Gamma \vdash \mathscr{R}\left[\iota; o; \rho \mid e\right] : [\varnothing; \pi^o; \tau]$. From this last judgment, using Lemma 3.11, we get $\Gamma, \Gamma^\iota, \Gamma^\rho \vdash e : \tau$. Finally, we can use the Strengthening Lemma (Lemma 3.9) with $\Gamma \setminus \mathsf{dom}(\Gamma^\iota, \Gamma^\rho) \subseteq \Gamma, \Gamma^\iota, \Gamma^\rho$ (from (1)) and the Weakening Lemma (Lemma 3.8) with $\Gamma \setminus \mathsf{dom}(\Gamma^\iota, \Gamma^\rho) \subseteq \Gamma$, to conclude $\Gamma \vdash e : \tau$.

(SUM-LIFT) We derive $\left[\iota_1; o_1; \rho_1 \mid e\right]\uparrow + [\iota_2; o_2; \rho_2] \longrightarrow \left[\iota_1; o_1; \rho_1 \mid e + [\iota_2; o_2; \rho_2]\right]\uparrow$ and by hypothesis we have $\Gamma \vdash \left[\iota_1; o_1; \rho_1 \mid e\right]\uparrow + [\iota_2; o_2; \rho_2] : \tau$. We first observe that, for Proposition 3.1, τ must be well formed. Moreover, there are two cases to be considered: τ is either a module type or a configuration type. We consider the latter case (the former can be handled in a similar way).

The unique way to derive the given typing judgment is by using (SUM), (RES) and (BASIC). In particular, for (SUM) we must have $\tau = \left[\pi_1^\iota, \pi_2^\iota; \pi_1^o, \pi_2^o; \tau'\right]$ and

$$\Gamma \vdash \left[\iota_1; o_1; \rho_1 \mid e\right]\uparrow : \left[\pi_1^\iota; \pi_1^o; \tau'\right] \tag{1}$$
$$\Gamma \vdash [\iota_2; o_2; \rho_2] : \left[\pi_2^\iota; \pi_2^o\right]. \tag{2}$$

From (1), for (RES), we must have $\iota_1 = \varnothing$ and

$$\Gamma \vdash \left[\varnothing; o_1; \rho_1 \mid e\right] : \left[\varnothing; \pi^o; \left[\pi_1^\iota; \pi_1^o; \tau'\right]\right]$$

for some π^o, moreover, for rule (BASIC),

$$\left[\varnothing; o_1; \rho_1 \mid e\right] = \left[\varnothing; X_j \overset{j\in J_1}{\mapsto} e_j; x_k \overset{k\in K_1}{\mapsto} e_k \mid e\right]$$
$$\left[\varnothing; \pi^o; \left[\pi_1^\iota; \pi_1^o; \tau'\right]\right] = \left[\varnothing; X_j : {\tau_j}^{j_1\in J_1}; \left[\pi_1^\iota; \pi_1^o; \tau'\right]\right]$$

and

$$\left\{\Gamma, x_p : {\tau_p}^{p\in K_1} \vdash e_q : \tau_q \mid q \in J_1 \cup K_1\right\} \tag{4}$$
$$\Gamma, x_p : {\tau_p}^{p\in K_1} \vdash e : \left[\pi_1^\iota; \pi_1^o; \tau'\right]. \tag{5}$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$\tau \in \mathsf{Type}$	$::=$		**type**
	$\mid$	$[\pi^i; \pi^o; \mathscr{D}]$	module type
	$\mid$	$[\pi^i; \pi^o; \mathscr{D}; \chi \Rightarrow \tau]$	configuration type
π	$=$	$X_i : \tau_i^{\,i \in I}$	signature
$\mathscr{D}$	$\subseteq$	$\mathsf{Name} \times \mathsf{Name}$	dependency relation
χ	$\subseteq$	Name	required names

Fig. 13. $\mathscr{TS}^{\mathsf{dep}}$ types

For the side-condition $\mathsf{FV}([\iota_2; o_2; \rho_2]) \cap \mathsf{dom}(\iota_1, \rho_1) = \varnothing$ of (SUM-LIFT), we can apply the Weakening Lemma (Lemma 3.8) to (2), to give

$$\Gamma, x_p : \tau_p{}^{p \in K_1} \vdash [\iota_2; o_2; \rho_2] : \left[\pi_2^i; \pi_2^o\right] . \tag{2'}$$

Hence, we have the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x_p : \tau_p{}^{p \in K_1} \vdash e + [\iota_2; o_2; \rho_2] : \left[\pi^i, \pi_2^i; \pi^o, \pi_2^o; \tau'\right]}\,(\text{SUM}) \qquad \vdash \tau}{\Gamma \vdash \left[\varnothing; o_1; \rho_1 \mid e + [\iota_2; o_2; \rho_2]\right] : \left[\varnothing; \pi_1^o; \left[\pi^i, \pi_2^i; \pi^o, \pi_2^o; \tau'\right]\right]}\,(\text{BASIC})}{\Gamma \vdash \left[\varnothing; o_1; \rho_1 \mid e + [\iota_2; o_2; \rho_2]\right] \uparrow : \left[\pi^i, \pi_2^i; \pi^o, \pi_2^o; \tau'\right]}\,(\text{RES})$$

(4) (5) (2′)

The remaining cases ((R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (RUN-LIFT) and (RES-LIFT)) are handled in a similar way to rule (SUM-LIFT). □

3.2. *A type system with dependency analysis*

In this section we define a more sophisticated type system $\mathscr{TS}^{\mathsf{dep}}$ for the R-calculus, which allows programs also to be extracted from configurations with input components. This type system relies on a dependency analysis that ensures that the program will never try to access module components that cannot become available by performing reconfiguration steps.

Types are defined in Figure 13. Module types are extended (with respect to those used in the $\mathscr{T}^{\mathsf{CMS}}$ type system) by the additiuon of a binary relation $\mathscr{D}$ on Name called the *dependency relation*. Intuitively, $\mathscr{D}$ keeps track of the input names an output name depends on and is used to give a more liberal rule for the result operator†. We write $Y \overset{\mathscr{D}}{\rightarrow} X$ for $(Y, X) \in \mathscr{D}$. Intuitively, this means that Y depends on X, that is, Y is an output component defined by an expression that (transitively) refers to a deferred variable x mapped to the input component X.

Configuration types are also extended by the addition of both the dependency relation $\mathscr{D}$ and a component $\chi \Rightarrow \tau$ where χ is a subset of Name, which is called the set of *required names*. Intuitively, $\chi \Rightarrow \tau$ means that the program in the configuration has type τ if all (input) names in χ can be provided eventually (with the proper type).

† The definition of $\mathscr{D}$ and the related operators were inspired by the CMS_v calculus (Hirschowitz and Leroy 2002). However, we have preferred here to consider the inverse relation and do not need to deal with labelled multi-graphs.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

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

$$\text{(M-BASIC)}\ \frac{\begin{array}{c}\vdash \left[X_i : \tau_i{}^{i\in I};\ X_j : \tau_j{}^{j\in J}\right] \\ \left\{\Gamma, x_p : \tau_p{}^{p\in I\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\}\end{array}}{\Gamma \vdash \left[x_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} e_j;\ x_k \overset{k\in K}{\mapsto} e_k\right] : \left[X_i : \tau_i{}^{i\in I};\ X_j : \tau_j{}^{j\in J};\ \mathscr{D}^{[\iota;o;\rho]}\right]}$$

$$\text{(M-SUM)}\ \frac{\Gamma \vdash e_1 : \left[\pi_1^\iota;\ \pi_1^o;\ \mathscr{D}_1\right] \quad \Gamma \vdash e_2 : \left[\pi_2^\iota;\ \pi_2^o;\ \mathscr{D}_2\right]}{\Gamma \vdash e_1 + e_2 : \left[\pi_1^\iota,\pi_2^\iota;\ \pi_1^o,\pi_2^o;\ \mathscr{D}_1 \cup \mathscr{D}_2\right]}\ \begin{array}{l}\pi_1^\iota \text{ and } \pi_2^\iota \text{ compatible} \\ \mathsf{dom}(\pi_1^o) \cap \mathsf{dom}(\pi_2^o) = \varnothing\end{array}$$

$$\text{(M-REDUCT)}\ \frac{\Gamma \vdash e : \left[\pi_1^\iota,\pi_2^\iota;\ \pi^o;\ \mathscr{D}\right]}{\Gamma \vdash {}_{\sigma^\iota|}e_{|\sigma^o} : \left[\tilde{\pi}^\iota,\pi_2^\iota;\ \tilde{\pi}^o;\ {}_{\sigma^\iota|}\mathscr{D}_{|\sigma^o}\right]}\ \begin{array}{l}\mathsf{dom}(\sigma^\iota) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing \\ \tilde{\pi}^\iota \text{ and } \pi_2^\iota \text{ compatible} \\ \sigma^\iota|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \tilde{\pi}^\iota \\ \sigma^o : \tilde{\pi}^o \to \pi^o\end{array}$$

$$\text{(M-FREEZE)}\ \frac{\Gamma \vdash e : \left[\pi_1^\iota,\pi_2^\iota;\ \pi^o;\ \mathscr{D}\right]}{\Gamma \vdash \mathsf{freeze}_\sigma e : \left[\pi_2^\iota;\ \pi^o;\ \mathsf{freeze}_\sigma \mathscr{D}\right]}\ \begin{array}{l}\mathsf{dom}(\sigma) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing \\ \sigma|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \pi^o\end{array}$$

$$\text{(BASIC)}\ \frac{\begin{array}{c}\vdash \left[X_i : \tau_i{}^{i\in I};\ X_j : \tau_j{}^{j\in J}\right] \\ \left\{\Gamma, x_p : \tau_p{}^{p\in I\cup K} \vdash e_q : \tau_q \mid q \in J \cup K\right\} \qquad \Gamma, x_p : \tau_p{}^{p\in I\cup K} \vdash e : \tau\end{array}}{\Gamma \vdash \left[x_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} e_j;\ x_k \overset{k\in K}{\mapsto} e_k \mid e\right] : \left[\pi^\iota;\ \pi^o;\ \mathscr{D}^{[\iota;o;\rho]};\ \chi^{[\iota;o;\rho|e]} \Rightarrow \tau\right]}$$

$$\text{(SUM)}\ \frac{\Gamma \vdash e_1 : \left[\pi_1^\iota;\ \pi_1^o;\ \mathscr{D}_1;\ \chi \Rightarrow \tau\right] \quad \Gamma \vdash e_2 : \left[\pi_2^\iota;\ \pi_2^o;\ \mathscr{D}_2\right]}{\Gamma \vdash e_1 + e_2 : \left[\pi_1^\iota,\pi_2^\iota;\ \pi_1^o,\pi_2^o;\ \mathscr{D}_1 \cup \mathscr{D}_2;\ \chi \Rightarrow \tau\right]}\ \begin{array}{l}\pi_1^\iota \text{ and } \pi_2^\iota \text{ compatible} \\ \mathsf{dom}(\pi_1^o) \cap \mathsf{dom}(\pi_2^o) = \varnothing\end{array}$$

$$\text{(REDUCT)}\ \frac{\Gamma \vdash e : \left[\pi_1^\iota,\pi_2^\iota;\ \pi^o;\ \mathscr{D};\ \chi \Rightarrow \tau\right]}{\Gamma \vdash {}_{\sigma^\iota|}e_{|\sigma^o} : \left[\tilde{\pi}^\iota,\pi_2^\iota;\ \tilde{\pi}^o;\ {}_{\sigma^\iota|}\mathscr{D}_{|\sigma^o};\ \sigma^\iota(\chi) \Rightarrow \tau\right]}\ \begin{array}{l}\mathsf{dom}(\sigma^\iota) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing \\ \tilde{\pi}^\iota \text{ and } \pi_2^\iota \text{ compatible} \\ \sigma^\iota|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \tilde{\pi}^\iota \\ \sigma^o : \tilde{\pi}^o \to \pi^o\end{array}$$

$$\text{(FREEZE)}\ \frac{\Gamma \vdash e : \left[\pi_1^\iota,\pi_2^\iota;\ \pi^o;\ \mathscr{D};\ \chi \Rightarrow \tau\right]}{\Gamma \vdash \mathsf{freeze}_\sigma e : \left[\pi_2^\iota;\ \pi^o;\ \mathsf{freeze}_\sigma \mathscr{D};\ \mathsf{freeze}_\sigma^{\mathscr{D}} \chi \Rightarrow \tau\right]}\ \begin{array}{l}\mathsf{dom}(\sigma) \cap \mathsf{dom}(\pi_2^\iota) = \varnothing \\ \sigma|_{\mathsf{dom}(\pi_1^\iota)} : \pi_1^\iota \to \pi^o\end{array}$$

$$\text{(RUN)}\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o;\ \mathscr{D}]}{\Gamma \vdash e{\downarrow_X} : [\pi^\iota;\ \pi^o;\ \mathscr{D};\ \chi \Rightarrow \pi^o(X)]}\ \chi = \{Y \mid X \overset{\mathscr{D}}{\to} Y\}$$

$$\text{(RES)}\ \frac{\Gamma \vdash e : [\pi^\iota;\ \pi^o;\ \mathscr{D};\ \varnothing \Rightarrow \tau]}{\Gamma \vdash e{\uparrow} : \tau}$$

Fig. 14. $\mathscr{TS}^{\mathsf{dep}}$ typing rules

Figure 14 gives the typing rules for deriving judgments of the form $\Gamma \vdash e : \tau$, with the meaning 'e is a well-formed expression with type τ in the context Γ', where contexts are defined as in the $\mathscr{T}^{\mathsf{CMS}}$ type system.

Rules for the module subset, that is, (VAR), (M-BASIC), (M-SUM), (M-REDUCT) and (M-FREEZE), correspond to those in the $\mathscr{T}^{\mathsf{CMS}}$ type system except for the calculation of dependencies.

In rule (M-BASIC), $\mathscr{D}^{[\iota;o;\rho]}$ denotes the dependency relation induced by a basic module $[\iota; o; \rho]$, which is defined in Figure 15. For each output name Y, this relation keeps track of all input names that Y depends on through ρ.

In rule (M-SUM), the dependency relation is the union of the dependency relations of the arguments.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Auxiliary relations

— $y \to_\rho x$ if $y \in \mathsf{dom}(\rho)$ and $x \in \mathsf{FV}(\rho(y))$ $(x \in \mathsf{dom}(\iota) \cup \mathsf{dom}(\rho))$
— $\to_\rho{}^*$ is the transitive and reflexive closure of $\to_\rho$

Operators on dependency relations

Basic: $\mathscr{D}^{[\iota;o;\rho]} \subseteq \mathsf{dom}(o) \times \mathsf{cod}(\iota)$ is defined by

$$Y \overset{\mathscr{D}^{[\iota;o;\rho]}}{\to} X \text{ if } y \to_\rho{}^* x \text{ with } y \in \mathsf{FV}(o(Y)) \text{ and } \iota(x) = X$$

Reduct: ${}_{\sigma^\iota}|\mathscr{D}|_{\sigma^o} \triangleq \{(Y, \sigma^\iota(X)) \mid \sigma^o(Y) \overset{\mathscr{D}}{\to} X\}$
Freeze: $\mathsf{freeze}_\sigma \mathscr{D}$ is defined by

— $\mathsf{freeze}_{\sigma, X \mapsto X'}(\mathscr{D}) \triangleq \mathsf{freeze}_\sigma\left((\mathscr{D} \cup \mathscr{D}^{\mathsf{add}}_{X \mapsto X'}) \setminus \mathscr{D}^{\mathsf{remove}}_X\right)$, where
$\mathscr{D}^{\mathsf{add}}_{X \mapsto X'} \triangleq \left\{(Y, Z) \mid Y \overset{\mathscr{D}}{\to} X \wedge X' \overset{\mathscr{D}}{\to} Z\right\}$
$\mathscr{D}^{\mathsf{remove}}_X \triangleq \{(Y, X) \mid Y \overset{\mathscr{D}}{\to} X\}$
— $\mathsf{freeze}_\varnothing(\mathscr{D}) = \mathscr{D}$

Operators on sets of required names

Basic: $\chi^{[\iota;o;\rho|e]} \triangleq \left\{X \mid y \to_\rho^* x,\ y \in \mathsf{FV}(e),\ \iota(x) = X\right\}$
Reduct: $\sigma^\iota(\chi) = \{\sigma^\iota(X) \mid X \in \chi\}$
Freeze: $\mathsf{freeze}^{\mathscr{D}}_\sigma \chi \triangleq (\chi \cup \chi^{\mathscr{D}}_{\mathsf{add}}) \setminus \mathsf{dom}(\sigma)$, where

$$\chi^{\mathscr{D}}_{\mathsf{add}} = \{Z \mid X \in \chi \wedge \sigma(X) \overset{\mathsf{freeze}_\sigma \mathscr{D}}{\to} Z\}$$

Fig. 15. Dependency relation and required names

In rule (M-REDUCT), the dependency relation is obtained from that in the premise by renaming all input and output names using σ^ι and σ^o, respectively.

In rule (M-FREEZE), the dependency relation $\mathsf{freeze}_\sigma \mathscr{D}$ is defined in Figure 15. The intuition behind this definition is as follows.

First, a freeze operator with renaming σ is equivalent (Ancona 1998) to a sequence of freeze operators, one for each association $X \mapsto X'$ in σ. Hence, we can define $\mathsf{freeze}_\sigma \mathscr{D}$ inductively on the elements of σ.

For each association $X \mapsto X'$ in σ, the dependency relation is changed to take into account the fact that the name X will no longer be an input name and will take as its defining expression that of the output component X'. This defining expression can in turn depend on some input name Z. As a result, any dependency $Y \overset{\mathscr{D}}{\to} X$ must be removed and a new dependency $Y \overset{\mathscr{D}}{\to} Z$ must be added for each output name Y that previously depended on X and input name Z such that X' previously depended on Z.

Example 3.14. Consider the module expression

$$\mathsf{freeze}_{X \mapsto X, Y \mapsto Y}\left[x \mapsto X, y \mapsto Y, z \mapsto Z\,;\, X \mapsto z, Y \mapsto y', W \mapsto y + z\,;\, y' \mapsto x + 1\right].$$

Basic module dependencies are $\{(X, Z), (Y, X), (W, Y), (W, Z)\}$. The dependencies of the term at the top level are obtained from this set by performing the following steps:

— Since we freeze the input name X, we replace the dependency (Y, X) by (Y, Z) (indeed, X is bound to an expression depending on Z).

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

— Since we freeze the input name Y, we replace the dependency (W, Y) by (W, Z) (indeed, Y is bound to an expression depending on Z).

Note that dependencies on the name Z are unaffected.

In the typing rules (BASIC), (SUM), (REDUCT) and (FREEZE), the dependency relation for configurations is computed in exactly the same way as for modules.

In rule (BASIC), $\chi^{[\iota;o;\rho|e]}$ denotes the set of names that the expression e depends on (either directly or indirectly through ρ).

Since the sum operator just glues a module to a configuration without connecting any input with output names, in rule (SUM) the set of required names is unaffected.

In rule (REDUCT), the set of required names changes accordingly with σ^{ι}.

In rule (FREEZE), the set of required names $\mathsf{freeze}^{\mathscr{D}}_{\sigma}\chi$ is obtained from χ by replacing each name X that is to be frozen (that is, $X \in \chi \cap \mathsf{dom}(\sigma)$) with all names required by its new defining expression (that is, the defining expression of $\sigma(X)$). These new names are exactly those on which $\sigma(X)$ depends in the relation $\mathsf{freeze}_{\sigma}\mathscr{D}$.

In rule (RUN), the dependency relation does not change, and the set of required names χ contains all the input names that the output component X depends on.

Finally, rule (RES) allows us to extract the program running in a configuration provided that no input name is required, that is, that all components that may be needed for program execution can be provided by reconfiguration steps. Note the difference here compared with the $\mathscr{T}^{\mathsf{CMS}}$ type system.

The following result relates the dependency relation and the set of required names to the other components of module and configuration types.

Proposition 3.15.

— If $\Gamma \vdash e : [\pi^{\iota}; \pi^{o}; \mathscr{D}]$, then $\mathscr{D} \subseteq \mathsf{dom}(\pi^{o}) \times \mathsf{dom}(\pi^{\iota})$.
— If $\Gamma \vdash e : [\pi^{\iota}; \pi^{o}; \mathscr{D}; \chi \Rightarrow \tau]$, then $\mathscr{D} \subseteq \mathsf{dom}(\pi^{o}) \times \mathsf{dom}(\pi^{\iota})$, $\chi \subseteq \mathsf{dom}(\pi^{\iota})$.

Proof. The proof is an easy induction on the typing rules. □

We now show the soundness of the $\mathscr{TS}^{\mathsf{dep}}$ type system, following the same schema as for the $\mathscr{T}^{\mathsf{CMS}}$ type system. In particular, Weakening, Strengthening and Substitution Lemmas analogous to those stated for the $\mathscr{T}^{\mathsf{CMS}}$ type system (see Lemmas 3.8, 3.9 and 3.10, respectively) hold; moreover, Lemma 3.2 still holds for the $\mathscr{TS}^{\mathsf{dep}}$ type system.

Lemma 3.16 (Canonical Forms). Given $v \in \mathsf{Val}$:

— If $\Gamma \vdash v : [\pi^{\iota}; \pi^{o}; \mathscr{D}]$, then v either has the form $[\iota; o; \rho]$ or the form $\left[\iota; o; \rho \mid v'\right]\uparrow$.
— If $\Gamma \vdash v : [\pi^{\iota}; \pi^{o}; \mathscr{D}; \chi \Rightarrow \tau]$, then v either has the form $\left[\iota; o; \rho \mid v'\right]$ or the form $\left[\iota; o; \rho \mid v'\right]\uparrow$.

Proof. The proof is an easy check. □

Analogously to the $\mathscr{T}^{\mathsf{CMS}}$ type system, the progress property in the $\mathscr{TS}^{\mathsf{dep}}$ type system does not just hold for ground terms, but also for closed-type terms, with the less restrictive definition of closed type given in Figure 16.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$\frac{\vdash \tau \diamond}{\vdash [\pi^\iota; \pi^o; \mathscr{D}; \varnothing \Rightarrow \tau] \diamond} \qquad \frac{}{\vdash [\pi^\iota; \pi^o] \diamond}$$

Fig. 16. $\mathscr{TS}^{\mathsf{dep}}$ closed types

Example 3.17. The configuration

$$\left[z \mapsto Z\,;\, Z \mapsto 2, W \mapsto 1 + z\,;\, \mid \left[x \mapsto X\,;\,;\, y \mapsto 1 \mid 2 + y\right]\right]$$

in $\mathscr{TS}^{\mathsf{dep}}$ has the closed type

$$[Z : \mathsf{int};\, Z : \mathsf{int};\, (W, Z)\,;\, \varnothing \Rightarrow [X : \mathsf{int};\,;\, \varnothing;\, \varnothing \Rightarrow \mathsf{int}]]\ ;$$

but in $\mathscr{T}^{\mathsf{CMS}}$ it has the non-closed type

$$[Z : \mathsf{int};\, Z : \mathsf{int};\, [X : \mathsf{int};\,;\, \mathsf{int}]]\ .$$

Analogously to what we did for the $\mathscr{T}^{\mathsf{CMS}}$ type system, we now state a generalised progress property on all well-typed terms, and then get the progress property as a corollary. Recall the definition of $\mathscr{B}$ contexts in Figure 12.

Fact 3.18. If $\Gamma \vdash \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right] : \tau$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$, then τ is a non-closed configuration type.

Proof. The proof is an easy check. □

Theorem 3.19. If $\Gamma \vdash e : \tau$, then (at least) one of the following cases holds:

1 $e \in \mathsf{Val}$.
2 $e \longrightarrow e'$, for some $e' \in \mathsf{Exp}$.
3 $e = \mathscr{E}[x]$, $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$.
4 $e = \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$.

Proof. The proof is by induction on the typing rules. The proof is analogous to that for the *CMS*-like type system (Theorem 3.6). We will only show the most interesting case:

(RES) We derive $\Gamma \vdash e\uparrow : \tau$.
By the induction hypothesis on the premise $\Gamma \vdash e : [\pi^\iota; \pi^o; \mathscr{D}; \varnothing \Rightarrow \tau]$, one of 3.19–3.19 holds.

— If $e \in \mathsf{Val}$, then, by Lemma 3.16, there are the following two cases:
(a) If $e \equiv \left[\iota; o; \rho \mid v\right]$, for some $v \in \mathsf{Val}$, then:
– If $\mathsf{FV}(v) \cap \mathsf{dom}(\rho) = \varnothing$, we have

$$\left[\iota; o; \rho \mid v\right]\uparrow \xrightarrow{(\mathrm{RES})} v\,.$$

Note that the side-condition $\mathsf{FV}(e) \cap \mathsf{dom}(\iota, \rho) = \varnothing$ is satisfied since the set of required names is $\varnothing$.
– If $\mathsf{FV}(v) \cap \mathsf{dom}(\rho) \neq \varnothing$, then $v = \mathscr{E}[x]$ with $x \in \mathsf{dom}(\rho)$ and we have

$$\left[\iota; o; \rho \mid \mathscr{E}[x]\right]\uparrow \xrightarrow{(\mathrm{PRG\text{-}SUBST})} \left[\iota; o; \rho \mid \mathscr{E}\left\{\rho(x)\right\}\right]\uparrow\ .$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(b) If $e \equiv \left[\iota; o; \rho \mid v\right]\uparrow$, then

$$\left[\iota; o; \rho \mid v\right]\uparrow\uparrow \xrightarrow{\text{(RES-LIFT)}} \left[\iota; o; \rho \mid v\uparrow\right]\uparrow .$$

— If $e \longrightarrow e'$, the thesis follows by applying the reduction rule ($\mathscr{E}$).

— If $e = \mathscr{E}[x]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$, then $\mathscr{E}\uparrow$ is a one-hole context with $x \notin \mathsf{HB}(\mathscr{E}\uparrow)$ and $x \in \mathsf{dom}(\Gamma)$.

— If $e = \mathscr{B}\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$, with $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\iota)$. We have that e has the form $\left[\iota'; o'; \rho' \mid \mathscr{B}'\left[\iota; o; \rho \mid \mathscr{E}[x]\right]\right]$ with $\mathscr{B} = \left[\iota'; o'; \rho' \mid \mathscr{B}'\right]$ (the case $e = \left[\iota; o; \rho \mid \mathscr{E}[x]\right]$ with $\mathscr{B} = \Box$ is impossible since the dependency relation is empty). Set $e' = \mathscr{B}'\left[\iota; o; \rho \mid \mathscr{E}[x]\right]$. There are the following two cases:

(a) If $\mathsf{FV}(e') \cap \mathsf{dom}(\rho') = \varnothing$, then

$$\left[\iota'; o'; \rho' \mid e'\right]\uparrow \xrightarrow{\text{(RES)}} e' .$$

Note that the side-condition $\mathsf{FV}(e') \cap \mathsf{dom}(\iota', \rho') = \varnothing$ is satisfied since the set of required names is empty.

(b) If $\mathsf{FV}(e')\cap\mathsf{dom}(\rho') \neq \varnothing$, then, by Lemma 3.2, there exists $\mathscr{E}'$ such that $e' = \mathscr{E}'[y]$ with $y \notin \mathsf{HB}\left(\mathscr{E}'\right)$. Hence,

$$\left[\iota'; o'; \rho' \mid \mathscr{E}'[y]\right]\uparrow \xrightarrow{\text{(PRG-SUBST)}} \left[\iota'; o'; \rho' \mid \mathscr{E}'\{\rho(y)\}\right]\uparrow . \qquad \Box$$

Corollary 3.20 (Progress). If $\varnothing \vdash e : \tau$ and $\vdash \tau \diamond$, then either $e \in \mathsf{Val}$ or $e \longrightarrow e'$, for some $e' \in \mathsf{Exp}$.

Proof. The thesis follows from Theorem 3.6. Indeed, the third case is impossible since $\Gamma = \varnothing$, and the fourth case is impossible by Fact 3.18. $\Box$

Lemmas analogous to Lemmas 3.11 and 3.12 hold for the $\mathscr{TS}^{\mathsf{dep}}$ type system as well.

Theorem 3.21 (Subject Reduction). If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.

Proof. We use induction on the reduction rules. The proof is analogous to the one given earlier for the *CMS*-like type system (Theorem 3.13), with some additions related to the treatment of dependency relations and sets of required names. In particular:

— For (M-SUM) and (SUM), we have

$$\mathscr{D}^{[\iota_1,\iota_2;o_1,o_2;\rho_1,\rho_2]} = \mathscr{D}^{[\iota_1;o_1;\rho_1]} \cup \mathscr{D}^{[\iota_2;o_2;\rho_2]} .$$

— For (M-REDUCT) and (REDUCT), we have

$$\mathscr{D}^{[\sigma^\iota\circ\iota_1,\iota_2;o\circ\sigma^o;\rho]} = {}_{\sigma^\iota}|\mathscr{D}_{\sigma^\iota}|^{[\iota_1,\iota_2;o;\rho]}|_{\sigma^o}|_{\sigma^o} .$$

— For (M-FREEZE) and (FREEZE), we have

$$\mathscr{D}^{\left[\iota;o;\rho,x_i\overset{i\in I}{\mapsto}o(\sigma(X_i))\right]} = \mathsf{freeze}_\sigma\left(\mathscr{D}^{\left[x_i\overset{i\in I}{\mapsto}X_i,\iota;o;\rho\right]}\right).$$

— For (PRG-SUBST), we have

$$\chi^{\left[\iota;o;\rho|\mathscr{E}\{\rho(x)\}\right]} = \chi^{\left[\iota;o;\rho|\mathscr{E}[x]\right]}$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

— For (REDUCT), we have

$$\chi^{\left[\sigma^{\iota}\circ\iota_1,\iota_2;o\circ\sigma^{o};\rho|e\right]} = \sigma^{\iota}(\chi^{\sigma^{\iota}|\left[\iota_1,\iota_2;o;\rho|e\right]_{|\sigma^{o}}})$$

— Finally, for (FREEZE), we have

$$\chi^{\left[\iota;o;\rho,x_i^{i\in I}\mapsto o(\sigma(X_i))|e\right]} = \mathsf{freeze}^{\mathscr{D}}_{\sigma}\left(\chi^{\left[x_i^{i\in I}\mapsto X_i,\iota;o;\rho|e\right]}\right).$$

□

We now show that all terms that are typable in the $\mathscr{T}^{\mathsf{CMS}}$ type system are also typable in the type system based on dependency analysis.

To do this, we introduce the following function from types, signatures and contexts in the $\mathscr{TS}^{\mathsf{dep}}$ type system to types, signature and contexts in the $\mathscr{T}^{\mathsf{CMS}}$ type system:

$$\begin{aligned}
\|[\pi^{\iota};\pi^{o}]\| &= \left[\|\pi^{\iota}\|;\|\pi^{o}\|\right]\\
\|[\pi^{\iota};\pi^{o};\mathscr{D};\chi\Rightarrow\tau]\| &= \left[\|\pi^{\iota}\|;\|\pi^{o}\|;\|\tau\|\right]\\
\|X_i:\tau_i^{\,i\in I}\| &= X_i:\|\tau_i\|^{i\in I}\\
\|\Gamma,x:\tau\| &= \|\Gamma\|,x:\|\tau\|\\
\|\varnothing\| &= \varnothing.
\end{aligned}$$

We can now state the following proposition.

Proposition 3.22. If $\Gamma\vdash e:\tau$ in $\mathscr{T}^{\mathsf{CMS}}$, then $\Gamma'\vdash e:\tau'$ in $\mathscr{TS}^{\mathsf{dep}}$, with $\|\Gamma'\|=\Gamma$ and $\|\tau'\|=\tau$.

Proof. The proof is an easy induction on the $\mathscr{T}^{\mathsf{CMS}}$ rules used to derive $\Gamma\vdash e:\tau$. □

The converse of this proposition does not hold in general, as shown by the following example.

Example 3.23. The expression $\left[z\mapsto Z\,;\,Y\mapsto y,X\mapsto x;\,y\mapsto 1,x\mapsto 2+z\mid y\right]\uparrow$ is not typable in $\mathscr{T}^{\mathsf{CMS}}$ (the basic configuration has type $[Z:\mathsf{int};\,Y:\mathsf{int},X:\mathsf{int}]$, with a non-empty input signature, hence the result operator cannot be applied). However, in $\mathscr{TS}^{\mathsf{dep}}$ the basic configuration has type $[Z:\mathsf{int};\,Y:\mathsf{int},X:\mathsf{int};\,(X,Z)\,;\,\varnothing\Rightarrow\mathsf{int}]$, hence the expression has type int.

3.3. *A type system with dynamic checks*

In the previous sections we have presented two type systems for the R-calculus that have complementary advantages and disadvantages. The $\mathscr{T}^{\mathsf{CMS}}$ type system is simple, but rather restrictive, since it rejects many well-behaved terms; indeed, program extraction is not allowed on a configuration with input components, even when program execution will never require these components to proceed. On the other hand, the $\mathscr{TS}^{\mathsf{dep}}$ type system allows us to type more terms (in particular, program extraction is allowed on a configuration with input components if the program does not depend on them), but requires a rather expensive analysis. Moreover, even with the dependency analysis, some well-behaved terms are rejected, since this analysis takes into account all variables that may *possibly* be needed by the program (see Example 3.24), as usually happens with static type systems.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

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

Fig. 17. $\mathscr{T}\mathscr{S}^{\mathsf{err}}$ typing rule for the result operator

To overcome both the drawback of rejecting too many well-behaved expressions and that of an overly expensive analysis, an alternative approach can be given using a simple type system $\mathscr{T}\mathscr{S}^{\mathsf{err}}$ that delays some checks until run time. Note that this is a rather natural alternative in calculi for dynamic reconfiguration, where static analysis can become very hard. This is not the case for the *R*-calculus as designed in this paper (where the three type systems we propose, *viz*., the two static versions and the third including dynamic checks, are all reasonable choices), but it could be the case when configurations can evolve in many ways that are hard to predict at compile time, so it becomes more and more difficult to avoid the use of dynamic checks (see, for example, Ancona *et al*. (2005a) and Fagorzi (2005) for a variant of the *R*-calculus that allows *swapping* between module operators).

Formally, this type system is obtained from the $\mathscr{T}^{\mathsf{CMS}}$ type system just by allowing program extraction on arbitrary (well-typed) configurations, and raising a dynamic error when program execution needs a module component that is neither available (that is, local) nor can become available by performing reconfiguration steps. The justification for this approach is that, for any configuration context, there exist programs such that program extraction would not get stuck. The static type system only rejects configuration expressions that will get stuck regardless of the program running inside the configuration; in other words, it does not prevent 'missing component' errors, but only errors due to an ill-formed application of module operators.

The example below illustrates the differences between the three type systems.

Example 3.24. In the expressions

$[y \mapsto Y\,;\,;\, x \mapsto 2 + y \mid x]\!\uparrow$
$[y \mapsto Y\,;\,;\, x \mapsto 2 + y \mid 1]\!\uparrow$
$[y \mapsto Y\,;\,;\, x \mapsto 2 + y \mid \texttt{if true then 1 else } x]\!\uparrow$

the configurations all have the type $[Y : \mathsf{int}; \varnothing; \mathsf{int}]$, but only in the first case does the evaluation get stuck†.

As we mentioned earlier, the only typing rule in the type system with dynamic checks that is different from the $\mathscr{T}^{\mathsf{CMS}}$ type system (which is given in Figure 10) is the rule for the result operator, which is given in Figure 17.

The semantics of the calculus is also extended to allow the possibility of raising a dynamic error, as described above. That is, we extend reduction rules in Figures 3 and 4 with those in Figure 18.

† Note that in the $\mathscr{T}^{\mathsf{CMS}}$ type system all three expressions are ill-typed, while the dependency analysis allows us to type the second but not the third one.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$\text{(BASIC-ERR)}\ \frac{}{\left[\iota;\, o;\, \rho \mid \mathscr{E}[x]\right] \longrightarrow \mathsf{err}(X)} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}),\, \iota(x) = X \\ \mathscr{E}[x] \not\rightarrow ,\, \mathscr{E}[x] \notin \mathsf{Val} \\ \mathsf{FV}(\mathscr{E}[x]) \cap \mathsf{dom}(\rho) = \varnothing \end{array}$$

$$\text{(RES-ERR)}\ \frac{}{\left[\iota;\, o;\, \rho \mid \mathscr{E}[x]\right]\uparrow \longrightarrow \mathsf{err}(X)} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}),\, \iota(x) = X \\ \mathscr{E}[x] \not\rightarrow ,\, \mathscr{E}[x] \notin \mathsf{Val} \\ \mathsf{FV}(\mathscr{E}[x]) \cap \mathsf{dom}(\rho) = \varnothing \end{array}$$

$$\text{(ERR-PROP)}\ \frac{e \longrightarrow \mathsf{err}(X)}{\left[\iota;\, o;\, \rho \mid e\right] \longrightarrow \mathsf{err}(X)} \quad \mathsf{FV}(e) \cap \mathsf{dom}(\rho) = \varnothing$$

Fig. 18. *R*-calculus: new reduction rules with run-time error

In these rules, the intuitive meaning of $\mathsf{err}(X)$ is that execution is stuck because there is a reference (through some deferred variable) to a missing component X. Note that, since we do not fix an evaluation strategy, different errors may be raised depending on the strategy used.

Example 3.25. The configuration $\left[x \mapsto X, y \mapsto Y\,;\,;\,\mid x + y\right]$ can raise either $\mathsf{err}(X)$ or $\mathsf{err}(Y)$.

In rule (BASIC-ERR), an error is raised when program execution in a basic configuration is stuck because, intuitively, some required component is missing. That is, the program is a normal form but not a value, no local variable can be substituted, and the program contains a reference to a deferred variable (hence execution could proceed if this variable were available).

Note that the missing component can be provided (hence the error can be handled) by performing surrounding reconfiguration operators. To model this, the contextual closure ($\mathscr{E}$) does not propagate errors (they are not expressions).

Example 3.26. The configuration $e = \left[y \mapsto Y\,;\,;\, x \mapsto 2 + y \mid x\right]$ evaluates to an error:

$$e \xrightarrow{\text{(PRG-SUBST)}} \left[y \mapsto Y\,;\,;\, x \mapsto 2 + y \mid 2 + y\right] \xrightarrow{\text{(BASIC-ERR)}} \mathsf{err}(Y)\,.$$

However, this error can be handled by a suitable surrounding context, such as:

$$\mathsf{freeze}_{Y \mapsto Y}(e + [;\, Y \mapsto 1;\,])\,,$$

which reduces in several steps to

$$\left[;\, Y \mapsto 1;\, x \mapsto 2 + y, y \mapsto 1 \mid 3\right]\,.$$

Rule (RES-ERR) is analogous to rule (BASIC-ERR), with the only difference being that a result operator is applied to the basic configuration. Also, in this case the error is not propagated by contextual closure since the surrounding context may delay it, as illustrated by the following example.

Example 3.27. The configuration $e = \left[y \mapsto Y\,;\,;\,\mid y + [x \mapsto X\,;\, Z \mapsto 1;\,]\right]\uparrow$ evaluates to $\mathsf{err}(Y)$. However, $e\backslash Z$ reduces to $\left[y \mapsto Y\,;\,;\,\mid (y + [x \mapsto X\,;\, Z \mapsto 1;\,]) \setminus Z\right]\uparrow$ by performing a (REDUCT-LIFT) step. Analogously, an expression of the form $\left[\iota;\, o;\, \rho \mid e\right]\uparrow$ can perform a (PRG-SUBST) step if x is local at the outer level (that is, bound in ρ), and a (RES) step if x is not bound

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

at the outer level (that is, neither in ι nor in ρ. If x is deferred at the outer level (that is, say, $\iota(x) = X$), then an error at the top level for a different input name (say, $\mathsf{err}(X)$) can be raised by rule (RES-ERR).

Since errors are not propagated by contextual closure, we need an *ad hoc* rule to handle the case in which the outer enclosing operator can neither handle nor delay (by allowing further reduction steps) the error. This happens when the enclosing operator is a basic configuration and, moreover, no local variable at this level can be substituted, as modelled by rule (ERR-PROP).

Example 3.28. The expression $e = \big[;\,;\, z \mapsto 1 \mid \big[y \mapsto Y\,;\,;\, x \mapsto y + z \mid x\big]\big]$ evaluates to an error, that is,

$$\begin{aligned} e &\xrightarrow{\text{(PRG-SUBST)}} \big[;\,;\, z \mapsto 1 \mid \big[y \mapsto Y\,;\,;\, x \mapsto y + z \mid y + z\big]\big] \\ &\xrightarrow{\text{(PRG-SUBST)}} \big[;\,;\, z \mapsto 1 \mid \big[y \mapsto Y\,;\,;\, x \mapsto y + z \mid y + 1\big]\big] \\ &\xrightarrow{\text{(BASIC-ERR)}+\text{(ERR-PROP)}} \mathsf{err}(Y)\,. \end{aligned}$$

As a final remark, note that rule (RES-ERR), possibly combined with (ERR-PROP) deals with expressions that are ill typed in both the $\mathscr{T}^{\mathsf{CMS}}$ and $\mathscr{TS}^{\mathsf{dep}}$ type systems. However, here the error is raised only when the program actually needs a component that cannot be provided, as illustrated by Example 3.24, where the third configuration, which is ill typed in both $\mathscr{T}^{\mathsf{CMS}}$ and $\mathscr{TS}^{\mathsf{dep}}$, correctly reduces to a value.

On the other hand, expressions considered by rule (BASIC-ERR), possibly combined with (ERR-PROP), correspond to terms that are well typed with respect to both type systems, but have a non-closed type, and hence are not considered in the progress theorem. Here, these expressions evaluate to an error only when they actually require a component that cannot be resolved; moreover, this error can always be handled by a suitable context.

We now illustrate the results arising from this approach.

Lemma 3.29. If $e \longrightarrow \mathsf{err}(X)$, then e is either $\mathscr{B}\big[\iota;\, o;\, \rho \mid \mathscr{E}[x]\big]$ or $\mathscr{B}\big[\iota;\, o;\, \rho \mid \mathscr{E}[x]\big]\uparrow$, with $x \notin \mathsf{HB}(\mathscr{E})$, $\mathscr{E}[x] \notin \mathsf{Val}$ and $\iota(x) = X$.

Proof. The proof is by induction on the reduction rules. □

Unlike what happens for the $\mathscr{T}^{\mathsf{CMS}}$ type system, the progress property also holds here for terms of non-closed type, in the sense that they may reduce to an error.

Example 3.30. The configurations $\big[y \mapsto Y\,;\,;\mid y\big]$ and $\big[y \mapsto Y\,;\,;\mid \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ 1\ \mathtt{else}\ y\big]$, which have non-closed type in both $\mathscr{T}^{\mathsf{CMS}}$ and $\mathscr{TS}^{\mathsf{dep}}$, here reduce to an error $\mathsf{err}(Y)$ and the value $\big[y \mapsto Y\,;\,;\mid 1\big]$, respectively.

We now state the generalised progress property, which deals with possibly non-ground terms.

Theorem 3.31. If $\Gamma \vdash e : \tau$, then (at least) one of the following cases holds:

— $e \in \mathsf{Val}$.
— $e \longrightarrow e'$, for some $e' \in \mathsf{Exp}$.
— $e \longrightarrow \mathsf{err}(X)$.
— $e = \mathscr{E}[x]$, $x \notin \mathsf{HB}(\mathscr{E})$ and $x \in \mathsf{dom}(\Gamma)$.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Proof. We use induction on the typing rules. The proof is similar to that for $\mathscr{T}^{\mathsf{CMS}}$ (Theorem 3.6). □

From this theorem, the progress property follows as a corollary.

Corollary 3.32 (Progress). If $\varnothing \vdash e : \tau$, then either $e \in \mathsf{Val}$ or $e \longrightarrow e'$, for some $e' \in \mathsf{Exp}$, or $e \longrightarrow \mathsf{err}(X)$.

Proof. The thesis follows from Theorem 3.31 by taking $\Gamma = \varnothing$. □

The following lemma will be useful for proving the Subject Reduction Theorem.

Lemma 3.33. If $\Gamma \vdash \mathscr{R}\left[\iota; o; \rho \mid e\right] : [\pi^{\iota}; \pi^{o}; \tau]$, then $\Gamma, \Gamma^{\iota}, \Gamma^{\rho} \vdash e : \tau$, with $\mathsf{dom}(\Gamma^{\iota}) = \mathsf{dom}(\iota)$ and $\mathsf{dom}(\Gamma^{\rho}) = \mathsf{dom}(\rho)$.

Proof. The proof is by induction on the structure of $\mathscr{R}$. □

Theorem 3.34 (Subject Reduction). If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.

Proof. The proof is by induction on the reduction rules. The proof is analogous to that for $\mathscr{T}^{\mathsf{CMS}}$ (Theorem 3.13). □

The approach with dynamic checks can be related to the two previous (static) type systems as follows: given an expression that is well typed with respect to a static type system, its evaluation can raise an error only when its type is non-closed. Formally, we have the following proposition.

Proposition 3.35. If $\varnothing \vdash e : \tau$ in either $\mathscr{T}^{\mathsf{CMS}}$ or $\mathscr{TS}^{\mathsf{dep}}$ and $e \longrightarrow \mathsf{err}(X)$, then $\vdash \tau \diamond$ is not derivable in the type system considered.

4. A call-by-need strategy

In this section, we present a call-by-need strategy for the R-calculus that leads to a generalisation, including reconfiguration features, of call-by-need lambda calculi as in Ariola and Felleisen (1997), Ariola and Blom (1997) and Ariola and Blom (2002), and we prove its soundness and completeness with respect to the R-calculus.

The section is organised as follows:

— In Section 4.1 we formally define and explain the call-by-need reduction rules, and then provide motivation for and an informal outline of the statements and proofs of soundness and completeness.

In the remaining sections we introduce some technical machinery required to state and prove soundness and completeness. In particular:

— In Section 4.2 we introduce a notion of *information content* for the terms of the R-calculus.

— In Section 4.3 we define an enriched reduction relation that adds some administrative rules to the R-calculus, and show that the R-calculus is complete (up to information content) with respect to the enriched reduction relation (this also implies the soundness of the call-by-need strategy which is a subset of the enriched reduction relation).

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

— In Section 4.4 we show the completeness of the call-by-need strategy.
— Finally, in Section 4.5, as a side result of our approach, we show that the enriched reduction relation is confluent up to information content.

4.1. *Call-by-need reduction rules*

The reduction rules are given in Figures 19 and 20. The basic idea is that module simplification is performed only when needed, and only once. This represents a generalisation to reconfiguration operators of ideas previously developed for let-bounded variables in lambda calculi with let-rec (Ariola and Felleisen 1997; Ariola and Blom 1997; Ariola and Blom 2002).

One-hole evaluation contexts $\mathscr{E}^{\mathsf{ev}}$ are defined at the top of Figure 19. We use the abbreviation $\mathscr{D}[x, x_n]$ to denote a set of declarations such that the evaluation of x depends transitively on evaluation of x_n, as in Ariola and Felleisen (1997), Ariola and Blom (1997) and Ariola and Blom (2002).

The values of the strategy $\mathsf{Val}^{\mathsf{need}}$ are expressions that can be directly substituted for a variable (see rule (PRG-SUBST)). A reconfiguration context applied to a basic configuration is a value if its running program is a value, even if there are still some module operators to be performed. Indeed, these reconfiguration steps are not required by the program, which has already completed its execution. *Answers* are, roughly speaking, values that also contain reconfiguration contexts that can provide definitions for deferred variables. A call-by-need strategy should avoid performing these reconfiguration steps if they are unnecessary. Hence, answers are normal forms for the strategy. Note that, compared with what happens in call-by-need lambda calculi, our answers not only keep local definitions, but also non-evaluated module operators.

Rule ($\mathscr{E}^{\mathsf{ev}}$) is analogous to rule ($\mathscr{E}$) in Figure 3, but now deals with evaluation contexts rather than arbitrary contexts, and *redexes*, ranged over by r, rather than arbitrary expressions, following the approach introduced in Felleisen and Friedman (1986). We recall that a redex is any left-hand-side of the reduction relation in the consequence of (an instantiation of) a reduction rule (other than ($\mathscr{E}^{\mathsf{ev}}$)).

Rules (M-SUM), (M-REDUCT) and (M-FREEZE) are as in Figure 3 (and hence are not listed again here). In general, the reduction of a term in the module subset is carried out as in the calculus, apart from the fact that we choose a deterministic (left-to-right) strategy for reducing sum (see evaluation contexts $\mathscr{E}^{\mathsf{ev}} + e, [\imath; o; \rho] + \mathscr{E}^{\mathsf{ev}}, {}_{\sigma^\imath|}\mathscr{E}^{\mathsf{ev}}{}_{|\sigma^o}, \mathsf{freeze}_\sigma \mathscr{E}^{\mathsf{ev}}$), and we do not allow module component simplification, so the contexts $[\imath; X \mapsto \mathscr{E}^{\mathsf{ev}}, o; \rho]$ and $[\imath; o; x \mapsto \mathscr{E}^{\mathsf{ev}}, \rho]$ are not present. Indeed, the intuition is that simplification of module components only takes place in a basic configuration when triggered by the executing program (see the comments for the (SUBST), (MERGE) and ($\mathscr{D}$-MERGE) rules below).

The reduction of a basic configuration enclosed by module operators (that is, a term of the form $\mathscr{R}[\imath; o; \rho \mid e]$) proceeds, if possible, by reduction steps of the program (evaluation contexts $[\imath; o; \rho \mid \mathscr{E}^{\mathsf{ev}}], \mathscr{E}^{\mathsf{ev}} + e, {}_{\sigma^\imath|}\mathscr{E}^{\mathsf{ev}}{}_{|\sigma^o}, \mathsf{freeze}_\sigma \mathscr{E}^{\mathsf{ev}}$).

If execution of the program cannot proceed because it needs a local variable x, then substitution of this instance of x with its defining expression now only takes place after this expression has already been reduced to a value (rule (PRG-SUBST)). We recall that $\mathscr{E}\{e\}$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Evaluation contexts, values and answers

$$
\begin{array}{lll}
\mathscr{E}^{\mathsf{ev}} & ::= \Box \mid \left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}\right] \mid \left[\iota;\, o;\, \rho, \mathscr{D}[x, x_n], x_n \mapsto \mathscr{E}^{\mathsf{ev}}_n \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \\
 & \quad \mid \mathscr{E}^{\mathsf{ev}} + e \mid \left[\iota;\, o;\, \rho\right] + \mathscr{E}^{\mathsf{ev}} \mid \left[\iota, x \mapsto X;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}_1[x]\right] + \mathscr{E}^{\mathsf{ev}}_2 \\
 & \quad \mid {}_{\sigma^\iota|}\mathscr{E}^{\mathsf{ev}}_{|\sigma^o} \mid \mathsf{freeze}_\sigma \mathscr{E}^{\mathsf{ev}} \mid \mathscr{E}^{\mathsf{ev}}\downarrow_X \mid \mathscr{E}^{\mathsf{ev}}\uparrow \\
\mathscr{D}[x, x_n] & ::= x \mapsto \mathscr{E}^{\mathsf{ev}}[x_1], x_1 \mapsto \mathscr{E}^{\mathsf{ev}}_1[x_2], \ldots, x_{n-1} \mapsto \mathscr{E}^{\mathsf{ev}}_{n-1}[x_n] \\
v \in \mathsf{Val}^{\mathsf{need}} & ::= \left[\iota;\, o;\, \rho\right] \mid \mathscr{R}\left[\iota;\, o;\, \rho \mid v\right] \\
a \in \mathsf{Ans} & ::= \left[\iota;\, o;\, \rho\right] \mid \mathscr{R}\left[\iota;\, o;\, \rho \mid a\right] \mid \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a\right]\right)\uparrow
\end{array}
$$

$$
(\mathscr{E}^{\mathsf{ev}}) \quad \frac{r \longrightarrow r'}{\mathscr{E}^{\mathsf{ev}}[r] \longrightarrow \mathscr{E}^{\mathsf{ev}}[r']}
$$

Reconfiguration and substitution

$$
(\text{MOD-PROP}) \quad \frac{\mathscr{R}[\iota;\, o;\, \rho] \xrightarrow[\mathsf{need}]{\alpha} \mathscr{R}'[\iota';\, o';\, \rho']}{\mathscr{R}\left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \xrightarrow[\mathsf{need}]{} \mathscr{R}'\left[\iota';\, o';\, \rho' \mid \alpha\left(\mathscr{E}^{\mathsf{ev}}[x]\right)\right]} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}) \\ x \in \mathsf{dom}(\iota) \end{array}
$$

$$
(\text{PRG-SUBST}) \quad \frac{}{\left[\iota;\, o;\, \rho, x \mapsto v \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \xrightarrow[\mathsf{need}]{} \left[\iota;\, o;\, \rho, x \mapsto v \mid \mathscr{E}^{\mathsf{ev}}\{v\}\right]} \quad x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}})
$$

$$
(\text{SUBST}) \quad \frac{}{\begin{array}{l}\left[\iota;\, o;\, \mathscr{D}[x, x_n], x_n \mapsto v, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \xrightarrow[\mathsf{need}]{} \\ \quad \left[\iota;\, o;\, \mathscr{D}[x, x_{n-1}], x_{n-1} \mapsto \mathscr{E}^{\mathsf{ev}}_{n-1}\{v\}, x_n \mapsto v, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\end{array}} \quad x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}})
$$

Merge

$$
(\text{INPUT-MERGE}) \quad \frac{\mathscr{R}[\iota_2;\, o_2;\, \rho_2] \xrightarrow[\mathsf{need}]{\alpha} \mathscr{R}'\left[\iota'_2;\, o'_2;\, \rho'_2\right]}{\begin{array}{l}\left[\iota_1;\, o_1;\, \rho_1, x \mapsto \left(\mathscr{R}\left[\iota_2;\, o_2;\, \rho_2 \mid a\right]\right)\uparrow \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \\ \xrightarrow[\mathsf{need}]{} \left[\iota_1;\, o_1;\, \rho_1, x \mapsto \left(\mathscr{R}'\left[\iota'_2;\, o'_2;\, \rho'_2 \mid \alpha(a)\right]\right)\uparrow \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\end{array}} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}) \\ (\mathsf{FV}(a) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) \neq \varnothing \end{array}
$$

$$
(\text{MERGE}) \quad \frac{}{\begin{array}{l}\left[\iota_1;\, o_1;\, \rho_1, x \mapsto \left(\mathscr{R}\left[\iota_2;\, o_2;\, \rho_2 \mid a\right]\right)\uparrow \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \\ \xrightarrow[\mathsf{need}]{} \left[\iota_1;\, o_1;\, \rho_1, x \mapsto a, \rho_2 \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\end{array}} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}) \\ (\mathsf{FV}(a) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) = \varnothing \\ (\mathsf{FV}(o_1) \cup \mathsf{FV}(\rho_1)) \cap \mathsf{dom}(\rho_2) = \varnothing \end{array}
$$

$$
(\text{INPUT-}\mathscr{D}\text{-MERGE}) \quad \frac{\mathscr{R}[\iota_2;\, o_2;\, \rho_2] \xrightarrow[\mathsf{need}]{\alpha} \mathscr{R}'\left[\iota'_2;\, o'_2;\, \rho'_2\right]}{\begin{array}{l}\left[\iota_1;\, o_1;\, \rho_1, \mathscr{D}[x, x_n], x_n \mapsto \left(\mathscr{R}\left[\iota_2;\, o_2;\, \rho_2 \mid a\right]\right)\uparrow \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \\ \xrightarrow[\mathsf{need}]{} \left[\iota_1;\, o_1;\, \rho_1, \mathscr{D}[x, x_n], x_n \mapsto \left(\mathscr{R}\left[\iota'_2;\, o'_2;\, \rho'_2 \mid \rho(a)\right]\right)\uparrow \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\end{array}} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}) \\ (\mathsf{FV}(a) \cup \mathsf{FV}(\rho_2)) \cap \\ \quad \mathsf{dom}(\iota_2) \neq \varnothing \end{array}
$$

$$
(\mathscr{D}\text{-MERGE}) \quad \frac{}{\begin{array}{l}\left[\iota_1;\, o_1;\, \rho_1, \mathscr{D}[x, x_n], x_n \mapsto \left(\mathscr{R}\left[\iota_2;\, o_2;\, \rho_2 \mid a\right]\right)\uparrow \mid \mathscr{E}^{\mathsf{ev}}[x]\right] \\ \xrightarrow[\mathsf{need}]{} \left[\iota_1;\, o_1;\, \rho_1, \mathscr{D}[x, x_n], x_n \mapsto a, \rho_2 \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\end{array}} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}) \\ (\mathsf{FV}(a) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) = \varnothing \\ (\mathsf{FV}(o_1) \cup \mathsf{FV}(\rho_1)) \cap \mathsf{dom}(\rho_2) = \varnothing \end{array}
$$

Fig. 19. Call-by-need reduction rules

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Lifting

$$\text{(SUM-LIFT)}\quad \frac{}{\left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\uparrow + a \xrightarrow[\text{need}]{} \left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}[x] + a\right]\uparrow} \quad x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}), x \in \mathsf{dom}(\iota)$$

$$\text{(R-SUM-LIFT)}\quad \frac{}{a + \left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\uparrow \xrightarrow[\text{need}]{} \left[\iota;\, o;\, \rho \mid a + \mathscr{E}^{\mathsf{ev}}[x]\right]\uparrow} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}), x \in \mathsf{dom}(\iota) \\ \mathsf{FV}(a) \cap \mathsf{dom}(\iota, \rho) = \varnothing \end{array}$$

$$\text{(REDUCT-LIFT)}\quad \frac{}{{}_{\sigma^{\iota}|}\left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\uparrow_{|\sigma^{o}} \xrightarrow[\text{need}]{} \left[\iota;\, o;\, \rho \mid {}_{\sigma^{\iota}|}\mathscr{E}^{\mathsf{ev}}[x]_{|\sigma^{o}}\right]\uparrow} \quad x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}), x \in \mathsf{dom}(\iota)$$

$$\text{(FREEZE-LIFT)}\quad \frac{}{\mathsf{freeze}_{\sigma}\left(\left[\iota;\, o;\, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right]\uparrow\right) \xrightarrow[\text{need}]{} \left[\iota;\, o;\, \rho \mid \mathsf{freeze}_{\sigma}\mathscr{E}^{\mathsf{ev}}[x]\right]\uparrow} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}}), \\ x \in \mathsf{dom}(\iota) \end{array}$$

$$\text{(}a\text{-SUM-LIFT)}\quad \frac{}{\left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a_1\right]\right)\uparrow + a_2 \xrightarrow[\text{need}]{} \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a_1 + a_2\right]\right)\uparrow}$$

$$\text{(}a\text{-R-SUM-LIFT)}\quad \frac{}{a_1 + \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a_2\right]\right)\uparrow \xrightarrow[\text{need}]{} \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a_1 + a_2\right]\right)\uparrow} \quad \mathsf{FV}(a_1) \cap \mathsf{dom}(\iota, \rho) = \varnothing$$

$$\text{(}a\text{-REDUCT-LIFT)}\quad \frac{}{{}_{\sigma^{\iota}|}\left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a\right]\right)\uparrow_{|\sigma^{o}} \xrightarrow[\text{need}]{} \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid {}_{\sigma^{\iota}|}a_{|\sigma^{o}}\right]\right)\uparrow}$$

$$\text{(}a\text{-FREEZE-LIFT)}\quad \frac{}{\mathsf{freeze}_{\sigma}\left(\left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a\right]\right)\uparrow\right) \xrightarrow[\text{need}]{} \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid \mathsf{freeze}_{\sigma} a\right]\right)\uparrow}$$

$$\text{(}a\text{-RUN-LIFT)}\quad \frac{}{\left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a\right]\uparrow\right)\downarrow_{X} \xrightarrow[\text{need}]{} \mathscr{R}\left[\iota;\, o;\, \rho \mid a\downarrow_{X}\right]\uparrow}$$

$$\text{(}a\text{-RES-LIFT)}\quad \frac{}{\left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a\right]\uparrow\right)\uparrow \xrightarrow[\text{need}]{} \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid a\uparrow\right]\right)\uparrow}$$

Fig. 20. Call-by-need reduction rules, continued

denotes the capture-avoiding substitution of the hole in context $\mathscr{E}$ with the expression e, which was defined in Figure 2. If this is not the case, then the evaluation of the definition of x is triggered, which can trigger, recursively, the evaluation of other definitions, until a local variable x_n whose evaluation does not depend on any other is found (evaluation context $\left[\iota;\, o;\, \mathscr{D}[x, x_n], x_n \mapsto \mathscr{E}_n^{\mathsf{ev}}, \rho \mid \mathscr{E}^{\mathsf{ev}}[x]\right])^{\dagger}$. When the definition of a local variable x_n is reduced to a value, this value can be used to replace a needed reference to x_n in another definition (rule (SUBST)).

The combined effect of these rules is that the evaluation of a local component is performed only when needed and only once.

If execution of the program cannot proceed because it needs a deferred variable x, reconfiguration steps take place (rule (MOD-PROP)). Note that reconfiguration now only takes

† When no such local variable can be found, the call-by-need reduction is stuck, but in the R-calculus we would have a non-terminating reduction sequence. In both cases the information content of the term would be Ω: see Section 4.2.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

place *on demand*. Indeed, in this case reconfiguration steps might make the variable local. Moreover, if the inner enclosing operator is a sum, the evaluation of the right-hand argument is triggered in order to reduce it to a basic module and so that we can perform the sum (evaluation context $\left[\iota, x \mapsto X\,;\, o\,;\, \rho \mid \mathscr{E}_1^{\mathsf{ev}}[x]\right] + \mathscr{E}_2^{\mathsf{ev}}$).

Rules (INPUT-MERGE), (MERGE), ($\mathscr{D}$-MERGE) and (INPUT-$\mathscr{D}$-MERGE) are used when the program (either directly or transitively) requires a variable x whose definition is neither a value nor an expression that can be further reduced, but an answer. If there are no references to deferred variables (rules (MERGE) and ($\mathscr{D}$-MERGE)), we can eliminate a reconfiguration level and rearrange local bindings. In the other case (rules (INPUT-MERGE) and (INPUT-$\mathscr{D}$-MERGE)), it is necessary to first perform reconfiguration steps until there are no longer any references to deferred variables.

Rules (SUM-LIFT), (R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (a-SUM-LIFT), (a-R-SUM-LIFT), (a-REDUCT-LIFT) and (a-FREEZE-LIFT) perform the lifting of module operators only when the running program cannot be further reduced, that is, it is either an answer or needs a deferred variable x that cannot be provided at the current configuration level, but might be provided after extracting the program. Indeed, note that in this case the configuration level is a basic configuration (so, no further reconfiguration steps can be performed).

Rules (a-RUN-LIFT) and (a-RES-LIFT) allow us to lift run and result operators when the running program is an answer.

Finally, rule (RUN) is as in the calculus (and hence, is not listed again here).

Example 4.1. We will now give an example of reduction in the call-by-need semantics. Consider the configuration

$$[\,;\,;\, x \mapsto \left(\mathsf{freeze}_{Z \mapsto Z}\left[z \mapsto Z\,;\, Z \mapsto 1\,;\mid \left[\,;\, W \mapsto w\,;\, w \mapsto z+1 \mid w\right]\right]\right)\uparrow, y \mapsto x\uparrow \mid y+1\,].$$

The running program refers to the local variable y, and y in turn refers to x, whose defining expression is not yet an answer. Hence, the evaluation begins by first evaluating the defining expression of x by performing (INPUT-MERGE)+(M-FREEZE) to give

$$\left[\,;\, Z \mapsto 1\,;\, z \mapsto 1 \mid \left[\,;\, W \mapsto w\,;\, w \mapsto z+1 \mid w\right]\right]\uparrow,$$

and then performing (SUBST) and (PRG-SUBST) to give

$$\left[\,;\, Z \mapsto 1\,;\, z \mapsto 1 \mid \left[\,;\, W \mapsto w\,;\, w \mapsto 2 \mid 2\right]\right]\uparrow$$

as the (closed) answer definition for x. Now, rule (MERGE) can be applied to the top-level term to give the configuration

$$e = [\,;\,;\, x \mapsto e_x, z \mapsto 1, y \mapsto x\uparrow \mid y+1]$$

where we call e_x the configuration $\left[\,;\, W \mapsto w\,;\, w \mapsto 2 \mid 2\right]$. The evaluation then proceeds as follows:

$$\begin{aligned}
e &\xrightarrow[\text{need}]{\text{(SUBST)}} \left[\,;\,;\, x \mapsto e_x, z \mapsto 1, y \mapsto \left[\,;\, W \mapsto w\,;\, w \mapsto 2 \mid 2\right]\uparrow \mid y+1\right] \\
&\xrightarrow[\text{need}]{\text{(MERGE)}} \left[\,;\,;\, x \mapsto e_x, z \mapsto 1, y \mapsto 2, w \mapsto 2 \mid y+1\right] \\
&\xrightarrow[\text{need}]{\text{(PRG-SUBST)}} \left[\,;\,;\, x \mapsto e_x, z \mapsto 1, y \mapsto 2, w \mapsto 2 \mid 2+1\right] \\
&\xrightarrow[\text{need}]{} \left[\,;\,;\, x \mapsto e_x, z \mapsto 1, y \mapsto 2, w \mapsto 2 \mid 3\right].
\end{aligned}$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

We will now show that the call-by-need strategy is deterministic. We begin by proving a unique decomposition property.

Theorem 4.2 (Unique decomposition). Given $e \in \mathsf{Exp}$, at most one of the following cases holds:

1 $e \in \mathsf{Ans}$.
2 There exists a unique evaluation context $\mathscr{E}^{\mathsf{ev}}$, redex r and rule $_{(\mathrm{R})}$ such that $\mathscr{E}^{\mathsf{ev}}[r] = e$ and r is the left-hand side of the reduction relation in the consequence of (an instantiation of) $_{(\mathrm{R})}$.
3 There exists a unique evaluation context $\mathscr{E}^{\mathsf{ev}}$ and variable x such that $\mathscr{E}^{\mathsf{ev}}[x] = e$ and $x \notin \mathsf{HB}(\mathscr{E}^{\mathsf{ev}})$.

Proof. The proof is by induction on the structure of e. □

Corollary 4.3 (Determinacy). Given $e \in \mathsf{Exp}$, if $e \xrightarrow[\mathsf{need}]{} e_1$ and $e \xrightarrow[\mathsf{need}]{} e_2$, then $e_1 = e_2$.

In the following subsections, we will state and prove the soundness and completeness of the call-by-need strategy with respect to the R-calculus, but first we will briefly motivate and outline our approach.

As a first approximation, we would like to be able to say that the call-by-need strategy gives only and all the results we get from the R-calculus. However, this statement needs to be refined because of two main problems:

(A) In general, the call-by-need strategy terminates with an answer, which is not always a value of the R-calculus, since, intuitively, we do not want to perform further (useless) reconfiguration/program extraction steps when the program is already a value. However, we can always obtain a value of the R-calculus from an answer by performing further R-calculus steps that are not call-by-need steps.
(B) To prove completeness, we need to relate not only final results, but also intermediate steps. However, it is easy to see that arbitrary steps in the two reduction relations are not directly comparable. Notably, in the R-calculus we perform arbitrary program substitution steps in situations where in the call-by-need strategy we perform substitution and merge steps, which change the shape of a configuration by unwinding references among local definitions and by 'flattening' configuration levels. Consider the following reduction sequence in the R-calculus:

$$\begin{aligned}
&\left[z \mapsto Z\,;\,;\, x \mapsto \left[;\,;\, y \mapsto 0 \mid y + z\right]\uparrow \mid x\right] \\
&\xrightarrow{(\text{PRG-SUBST})} \left[z \mapsto Z\,;\,;\, x \mapsto \left[;\,;\, y \mapsto 0 \mid y + z\right]\uparrow \mid \left[;\,;\, y \mapsto 0 \mid y + z\right]\uparrow\right] \\
&\xrightarrow{*} \left[z \mapsto Z\,;\,;\, x \mapsto z \mid z\right].
\end{aligned}$$

In the call-by-need strategy, the same term reduces as follows:

$$\begin{aligned}
&\left[z \mapsto Z\,;\,;\, x \mapsto \left[;\,;\, y \mapsto 0 \mid y + z\right]\uparrow \mid x\right] \\
&\xrightarrow[\mathsf{need}]{(\text{MERGE})} \left[z \mapsto Z\,;\,;\, y \mapsto 0, x \mapsto y + z \mid x\right] \\
&\xrightarrow[\mathsf{need}]{\star} \left[z \mapsto Z\,;\,;\, y \mapsto 0, x \mapsto z \mid z\right].
\end{aligned}$$

As the example shows, the initial term does not reduce to the same normal form in the two reduction relations. Problem B is the harder, and needs to be solved by defining

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

a more abstract relation between terms, as is often done in completeness proofs, for example, using some form of bisimulation. Here, we take the same approach as in Ariola and Blom (1997) and Ariola and Blom (2002) and define this abstract relation in terms of the *information content* of a term. Intuitively, the information content of a term is an approximation of its final result. In this way, the two terms above are equivalent since they have the same information content (in this case, $[Z\,;\,;\,|\,\Omega]$, which means that in both cases we obtain a basic configuration with one input component Z and where we have no information on the final result of the program).

Information contents are a partially ordered set, and the partial order on information contents induces a quasi-order on terms. We now get the following statement of soundness and completeness.

Soundness: If $e \xrightarrow[\text{need}]{\star} e_1$, then $e \xrightarrow{\star} e_2$ for some e_2 with greater or equal information content.

Completeness: If $e \xrightarrow{\star} e_1$, then $e \xrightarrow[\text{need}]{\star} e_2, e_2 \xrightarrow{\star} e_3$ for some e_2, e_3 such that the reduction steps in $e_2 \xrightarrow{\star} e_3$ are not call-by-need steps (this captures the fact that there are some steps that are not performed in the call-by-need strategy – Problem A above) and e_3 has greater or equal information content compared with e_1.

In the proofs, a convenient way to show that two terms e, e' are in the quasi-order relation (that is, one has less or equal information content compared with the other) is to show that e can be reduced to e' by a reduction relation that is monotonic with respect to information content. To this end, we define an enriched reduction relation $\bullet\!\!\longrightarrow$ that adds some administrative rules (that is, rules that do not augment the information content, and hence can be used in both directions) to $\longrightarrow$, and is a superset of both $\longrightarrow$ and $\xrightarrow[\text{need}]{}$. In this way, the statement of soundness above derives from the more general fact that $\longrightarrow$ turns out to be complete (up to information content) with respect to the enriched reduction $\bullet\!\!\longrightarrow$.

As a side result of our approach, we get that the enriched reduction relation is confluent up to information content. This is an important result in itself, since it shows that it is possible to solve in a more abstract way the well-know problem (Wells and Vestergaard 2000) that confluence is broken in module calculi if we allow variable substitution in local definitions (see Section 4.5 for more comments on this issue).

4.2. *Information content*

In this subsection, we define information content for the terms of the R-calculus. We first recall a few general definitions and results from Ariola and Blom (1997) and Ariola and Blom (2002), with references to these works for more technical details and proofs.

Definition 4.4. An *ordered abstract rewriting system* (an ordered ARS for short) is a triple $(A, \bullet\!\!\longrightarrow, \preceq)$ where $(A, \bullet\!\!\longrightarrow)$ is an abstract rewriting system and $(A, \preceq)$ is a quasi order.

Given an ordered ARS $(A, \bullet\!\!\longrightarrow, \preceq)$:

(1) $\bullet\!\!\longrightarrow$ is *confluent up to* $\preceq$ if $\forall a, b, c \in A : a \bullet\!\!\longrightarrow b, a \bullet\!\!\longrightarrow c$ implies $\exists d \in A : b \bullet\!\!\longrightarrow d, c \preceq d$.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

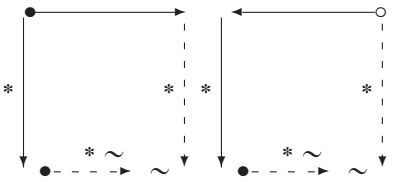
(2) Given another reduction relation $\longrightarrow \subseteq \bullet\!\!\longrightarrow$:

(i) $\longrightarrow$ is *complete for* $\bullet\!\!\longrightarrow$ *up to* $\preceq$ if $\forall a, b \in A : a \bullet\!\!\longrightarrow b$ implies $\exists c \in A : a \longrightarrow c, b \preceq c$.

(ii) $\longrightarrow$ *commutes with* $\bullet\!\!\longrightarrow$ *up to* $\preceq$ if $\forall a, b, c \in A : a \bullet\!\!\longrightarrow b, a \longrightarrow c$ implies $\exists d \in A : b \longrightarrow d, c \preceq d$.

Lemma 4.5 (Ariola and Blom (1997, Lemma 10.6)). Given an ordered ARS $(A, \bullet\!\!\longrightarrow, \preceq)$ and a relation $\longrightarrow \subseteq \bullet\!\!\longrightarrow$ that is complete for $\bullet\!\!\longrightarrow$ up to $\preceq$ and commutes with $\bullet\!\!\longrightarrow$ up to $\preceq$, we have $\bullet\!\!\longrightarrow$ is confluent up to $\preceq$.

Lemma 4.6 (Ariola and Blom (1997, Lemma 10.10). Given an ordered ARS $(A, \bullet\!\!\longrightarrow, \preceq)$, a relation $\longrightarrow \subseteq \bullet\!\!\longrightarrow$ and an equivalence $\sim \subseteq \simeq$, where $\simeq$ denotes $\preceq \cup \succeq$, and using $\circ\!\!\longrightarrow$ to denote the relation $\bullet\!\!\longrightarrow \setminus \longrightarrow$, we have for the diagrams



1 If the left diagram holds, then $\longrightarrow$ commutes with $\bullet\!\!\longrightarrow$ up to $\preceq$.
2 If the right diagram holds, then $\longrightarrow$ is complete for $\bullet\!\!\longrightarrow$ up to $\preceq$.

In the following, we will use this lemma by taking as equivalence $\sim$, the equivalence induced by the relation $\circ\!\!\longrightarrow$.

Definition 4.7. The structure $((A, \bullet\!\!\longrightarrow, \preceq), \omega, (B, \leqslant))$ is an *ARS with ordered information content* if:

(1) $(A, \bullet\!\!\longrightarrow, \preceq)$ is an ordered ARS.
(2) $\bullet\!\!\longrightarrow$ is monotonic with respect to $\preceq$.
(3) $(B, \leqslant)$ is a partially ordered set.
(4) ω is a map from A to B.
(5) $a \preceq b$ if and only if $\omega(a) \leqslant \omega(b)$.

We say that $\omega(a)$ is the *information content* of a.

We can now define information content for the terms of the R-calculus.

Definition 4.8 (Information content of *R*-calculus terms).

— Given an expression $e \in \mathsf{Exp}$, the information content of e is defined by the function ω, which returns the normal form of e with respect to the following (ω-)rules.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$
\begin{array}{lrcll}
(\omega\text{-SUM}) & e_1^\omega + e_2^\omega & \xrightarrow[\omega]{} & \Omega & e_1^\omega ::= [\iota_1; o_1; \rho_1] \mid \left[\iota_1; o_1; \rho_1 \mid e\right] \mid \Omega \\
 & & & & e_2^\omega ::= [\iota_2; o_2; \rho_2] \mid \Omega \\
(\omega\text{-REDUCT}) & {}_{\sigma^\iota}|e^\omega|_{\sigma^o} & \xrightarrow[\omega]{} & \Omega & e^\omega ::= [\iota; o; \rho] \mid \left[\iota; o; \rho \mid e\right] \mid \Omega \\
(\omega\text{-FREEZE}) & \mathsf{freeze}_\sigma e^\omega & \xrightarrow[\omega]{} & \Omega & e^\omega ::= [\iota; o; \rho] \mid \left[\iota; o; \rho \mid e\right] \mid \Omega \\
(\omega\text{-RUN}) & e^\omega \downarrow_X & \xrightarrow[\omega]{} & \Omega & e^\omega ::= [\iota; o; \rho] \mid \Omega \\
(\omega\text{-PRG-SUBST}) & \left[\iota; o; \rho \mid \mathscr{E}[x]\right] & \xrightarrow[\omega]{} & \left[\iota; o; \rho \mid \mathscr{E}[\Omega]\right] & x \in \mathsf{dom}(\iota, \rho) \\
(\omega\text{-M-SUBST}) & [\iota; o, X \mapsto \mathscr{E}[x]\,; \rho] & \xrightarrow[\omega]{} & [\iota; o, X \mapsto \mathscr{E}[\Omega]\,; \rho] & x \in \mathsf{dom}(\iota, \rho) \\
(\omega\text{-SUBST}) & \left[\iota; o, X \mapsto \mathscr{E}[x]\,; \rho \mid e\right] & \xrightarrow[\omega]{} & \left[\iota; o, X \mapsto \mathscr{E}[\Omega]\,; \rho \mid e\right] & x \in \mathsf{dom}(\iota, \rho) \\
(\omega\text{-M-GC}) & [\iota; o; \rho] & \xrightarrow[\omega]{} & [\iota; o;] & \mathsf{FV}(o) \cap \mathsf{dom}(\rho) = \varnothing \\
(\omega\text{-GC}) & \left[\iota; o; \rho \mid e\right] & \xrightarrow[\omega]{} & \left[\iota; o; \mid e\right] & (\mathsf{FV}(e) \cup \mathsf{FV}(o)) \cap \mathsf{dom}(\rho) = \varnothing \\
(\omega\text{-RES}) & \left[\iota; o; \mid e\right]\uparrow & \xrightarrow[\omega]{} & e & \mathsf{FV}(e) \cap \mathsf{dom}(\iota) = \varnothing \\
(\omega\text{-}\Omega\text{-RES}) & \Omega\uparrow & \xrightarrow[\omega]{} & \Omega\,. &
\end{array}
$$

— $\leqslant_\Omega$ is the order generated by the axiom $\Omega \leqslant_\Omega e$ for all $e \in \mathsf{Exp}$.
— $e \preceq_\omega e'$ if $\omega(e) \leqslant_\Omega \omega(e')$.
— $\simeq_\omega = \preceq_\omega \cap \succeq_\omega$.

Intuitively, the information content of a term is an approximation to its final result. Hence, terms that are redexes, such as a module operator applied to basic configurations/modules, are sent to Ω, since we do not know yet what the result will be in this case, and this Ω is propagated (first four ω-rules). When the redex is reduced, the information content can only grow (for instance, if we get either a basic module or a basic configuration). The information content of a basic module/configuration is obtained by first replacing references to variables by Ω in the program and in the definitions of output components (rules 5–7) and then forgetting local variables (rules 8 and 9). This models the fact that local definitions are not directly observable. Finally, when we extract a program, its information content is just the information content that the program has in the current basic configuration, and we can only extract Ω from Ω.

4.3. *Enriched reduction*

In this subsection, we define a reduction relation $\bullet\!\!\longrightarrow$ that adds some administrative rules to $\longrightarrow$, and show that we can give the structure of an ARS with ordered information content to the terms of the R-calculus by considering this enriched reduction relation and the definition of information content given in the previous subsection. Moreover, we show that $\longrightarrow$ is complete up to information content with respect to $\bullet\!\!\longrightarrow$.

The enriched reduction is defined by all the rules of $\longrightarrow$ (given in Figures 3 and 4) plus the new rules in Figure 21.

Merging, lifting and garbage collection rules can be performed only when some safety conditions are satisfied. In particular, in both (M-MERGE) and (PRG-MERGE), the first side-condition prevents the reduction step if either the program or local definitions depend on missing components; in lifting rules, the first side-condition requires that definitions do not refer to binding variables of the outer reconfiguration level; finally, in rules (M-GC) and (GC), the side-condition ensures that local components to be eliminated are unused inside module and

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

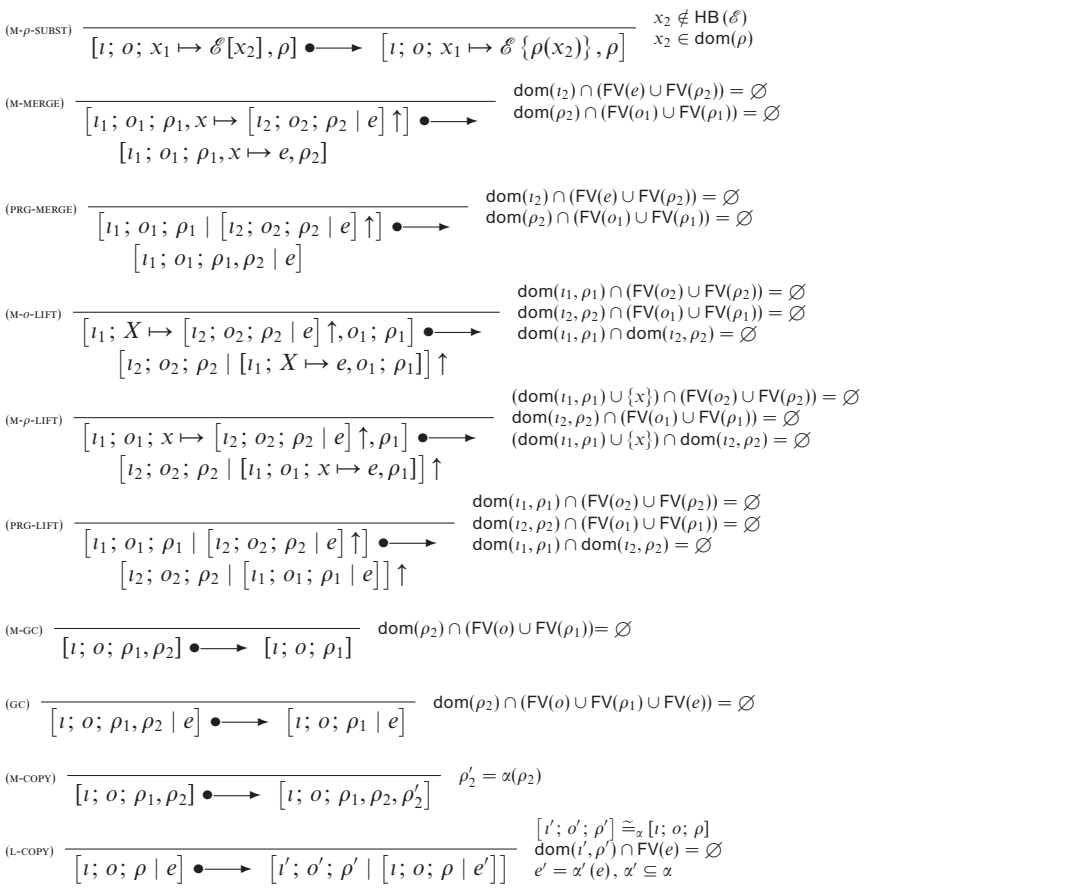


Fig. 21. Enriched reduction rules

configuration definitions, respectively. All other merging and lifting rule side-conditions can always be satisfied by an appropriate α-conversion.

Proposition 4.9. We have $\longrightarrow \subseteq \bullet\!\!\longrightarrow$ and $\xrightarrow[\mathsf{need}]{} \subseteq \bullet\!\!\longrightarrow$.

Proof. The proof is by showing that all reduction rules in both $\longrightarrow$ and $\xrightarrow[\mathsf{need}]{}$ can be derived by reduction rules in $\bullet\!\!\longrightarrow$. □

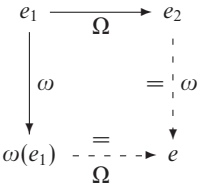
Proposition 4.10.

(1) Given $(\mathsf{Exp}, \leqslant_\Omega, \preceq_\omega)$, we have that $\leqslant_\Omega$ is monotonic with respect to $\preceq_\omega$.
(2) Given $(\mathsf{Exp}, \bullet\!\!\longrightarrow, \preceq_\omega)$, we have that $\bullet\!\!\longrightarrow$ is monotonic with respect to $\preceq_\omega$.
(3) $((\mathsf{Exp}, \bullet\!\!\longrightarrow, \preceq_\omega), \omega, (\omega(\mathsf{Exp}), \leqslant_\Omega))$ is an ARS with ordered information content.

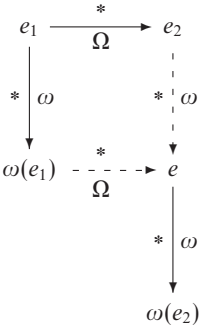
https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Proof.

(1) We first show that $\xrightarrow[\Omega]{*}$ commutes with $\xrightarrow[\omega]{*}$, where $\xrightarrow[\Omega]{}$ is just a different description of the order $\leqslant_\Omega$, which corresponds to the relation induced by the unique axiom $e \xrightarrow[\Omega]{} \Omega$. Commutativity follows from the diagram



where $\overset{=}{\dashrightarrow}$ means either zero or one step. The proof of this diagram is by case analysis on the ω-reduction step: in all cases the diagram commutes by applying either the same ω rule or performing no ω step. Now, given $e_1, e_2 \in \mathsf{Exp}$ with $e_1 \leqslant_\Omega e_2$, we get from the following diagram that $e_1 \preceq_\omega e_2$, that is, $\omega(e_1) \leqslant_\Omega \omega(e_2)$:



where

— $e_1 \xrightarrow[\Omega]{*} e_2$, since for hypothesis $e_1 \leqslant_\Omega e_2$;

— $e_1 \xrightarrow[\omega]{*} \omega(e_1)$ and $e \xrightarrow[\omega]{*} \omega(e_2)$, from the definition of ω;

— $e_2 \overset{*}{\underset{\Omega}{\dashrightarrow}} e$ and $\omega(e_1) \overset{*}{\underset{\Omega}{\dashrightarrow}} e$, since $\xrightarrow[\Omega]{}$ and $\xrightarrow[\omega]{}$ strongly commute.

(2) Given $e_1, e_2 \in \mathsf{Exp}$ with $e_1 \bullet\!\!\longrightarrow e_2$, we show that $e_1 \preceq_\omega e_2$:

— If the step is not in the calculus, the result is obvious.

— If it is due to a lift or to a (RES), then $e_1 \simeq_\omega e_2$.

— If it is due to a rule (RN) $=$ (M-SUM), (M-REDUCT), (M-FREEZE), (MOD-PROP), (RUN), then $e_1 = \mathscr{E}[r]$, with $r \overset{(RN)}{\bullet\!\!\longrightarrow} e$, $\omega(\mathscr{E}[r]) = \omega(\mathscr{E}[\Omega])$ and $\mathscr{E}[\Omega] \leqslant_\Omega \mathscr{E}[e] \equiv e_2$, hence the thesis follows from the first item.

— Finally, if it is due to rule (PRG-SUBST), we have

$$e_1 = \mathscr{E}\left[\left[\iota;\, o;\, \rho \mid \mathscr{E}'[x]\right]\right] e_2 = \mathscr{E}\left[\left[\iota;\, o;\, \rho \mid \mathscr{E}'\{\rho(x)\}\right]\right]$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

and

$$\begin{aligned}\omega(e_1) &= \omega(\mathscr{E}\left[\left[\imath; o; \rho \mid \mathscr{E}'[x]\right]\right]) \\ &= \omega(\mathscr{E}\left[\left[\imath; o; \rho \mid \mathscr{E}'[\Omega]\right]\right]) \\ &\leqslant_\Omega \omega(\mathscr{E}\left[\left[\imath; o; \rho \mid \mathscr{E}'\{\rho(x)\}\right]\right]) \\ &= \omega(e_2),\end{aligned}$$

so the thesis follows from (1).

(3) We have to check that:

(i) $(\mathsf{Exp}, \bullet\!\!\longrightarrow, \preceq_\omega)$ is an ordered ARS.
This follows from the fact that $(\mathsf{Exp}, \bullet\!\!\longrightarrow)$ is an ARS and $(\mathsf{Exp}, \preceq_\omega)$ is a quasi order.

(ii) $\bullet\!\!\longrightarrow$ is monotonic with respect to $\preceq_\omega$.
This follows from (2).

(iii) $(\omega(\mathsf{Exp}), \leqslant_\Omega)$ is a partially ordered set.
This is obvious.

(iv) ω is a function from Exp to $\omega(\mathsf{Exp})$.
This is obvious from the definition of ω.

(v) $e \preceq_\omega e'$ if $\omega(e) \leqslant_\Omega \omega(e')$.
This is obvious from the definition of $\preceq_\omega$. □

Definition 4.11 (Reduction $\circ\!\!\longrightarrow$). We use $\circ\!\!\longrightarrow$ to denote the reduction relation $\bullet\!\!\longrightarrow \setminus \longrightarrow$ and $\overset{\circ}{\longleftrightarrow}$ to denote the $\circ\!\!\longrightarrow$-conversion.

Intuitively, the rules in Figure 21 do not perform computational steps, but just transform one term into an equivalent term by changing its shape. Therefore, the new rules never augment information content, as formalised by the following proposition.

Proposition 4.12. Given $e, e' \in \mathsf{Exp}$, if $e \circ\!\!\longrightarrow e'$, then $e \simeq_\omega e'$ (hence $\overset{\circ}{\longleftrightarrow} \subseteq \simeq_\omega$).

Proof. Rules (M-ρ-SUBST), (M-MERGE), (M-GC), (GC) and (M-COPY) only act on local components, which will eventually become garbage and are then removed by $\xrightarrow[\omega]{}$.

— For (PRG-MERGE), with $\rho_i = \{x \mapsto \Omega \mid x \in \mathsf{dom}(\rho_i)\}$, $i \in \{1,2\}$, we have

$$\begin{aligned}\left[\imath_1; o_1; \rho_1 \mid \left[\imath_2; o_2; \rho_2 \mid e\right]\uparrow\right] &\xrightarrow[\omega]{*} \left[\imath_1; o_1; \rho_1 \mid \left[\imath_2; \rho_2(o_2); \rho_2 \mid \rho_2(e)\right]\uparrow\right] \\ &\xrightarrow[\omega]{} \left[\imath_1; o_1; \rho_1 \mid \left[\imath_2; \rho_2(o_2); \mid \rho_2(e)\right]\uparrow\right] \\ &\xrightarrow[\omega]{} \left[\imath_1; o_1; \rho_1 \mid \rho_2(e)\right] \text{ (side-condition of (PRG-MERGE))} \\ &\xrightarrow[\omega]{*} \left[\imath_1; \rho_1(o_1); \rho_1 \mid \rho_1(\rho_2(e))\right] \\ &\xrightarrow[\omega]{} \left[\imath_1; \rho_1(o_1); \mid \rho_1(\rho_2(e))\right]\end{aligned}$$

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

and

$$\begin{array}{rcl}\left[\iota_1; o_1; \rho_1, \rho_2 \mid e\right] & \xrightarrow[\omega]{*} & \left[\iota_1; \rho_1(\rho_2(o_1)); \rho_1, \rho_2 \mid \rho_1(\rho_2(e))\right] \\ & \equiv & \left[\iota_1; \rho_1(o_1); \rho_1, \rho_2 \mid \rho_1(\rho_2(e))\right] \text{ (side-condition of (PRG-MERGE))} \\ & \xrightarrow[\omega]{} & \left[\iota_1; \rho_1(o_1); \mid \rho_1(\rho_2(e))\right] .\end{array}$$

— For (M-O-LIFT), with $\rho_2 = \{x \mapsto \Omega \mid x \in \mathsf{dom}(\iota_2, \rho_2)\}$, we have

$$\begin{array}{rcl}\left[\iota_1; X \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow, o_1; \rho_1\right] & \xrightarrow[\omega]{*} & \left[\iota_1; X \mapsto \left[\iota_2; \rho_2(o_2); \rho_2 \mid \rho_2(e)\right]\uparrow, o_1; \rho_1\right] \\ & \xrightarrow[\omega]{} & \left[\iota_1; X \mapsto \left[\iota_2; \rho_2(o_2); \mid \rho_2(e)\right]\uparrow, o_1; \rho_1\right] \\ & \xrightarrow[\omega]{} & \left[\iota_1; X \mapsto \rho_2(e), o_1; \rho_1\right]\end{array}$$

and

$$\begin{array}{rcl}\left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; X \mapsto e, o_1; \rho_1\right]\right]\uparrow & \xrightarrow[\omega]{*} & \left[\iota_2; \rho_2(o_2); \rho_2 \mid \rho_2(\left[\iota_1; X \mapsto e, o_1; \rho_1\right])\right]\uparrow \\ & \equiv & \left[\iota_2; \rho_2(o_2); \rho_2 \mid \left[\iota_1; X \mapsto \rho_2(e), o_1; \rho_1\right])\right]\uparrow \\ & & \text{(side-condition of (M-O-LIFT))} \\ & \xrightarrow[\omega]{} & \left[\iota_2; \rho_2(o_2); \mid \left[\iota_1; X \mapsto \rho_2(e), o_1; \rho_1\right])\right]\uparrow \\ & \xrightarrow[\omega]{} & \left[\iota_1; X \mapsto \rho_2(e), o_1; \rho_1\right] .\end{array}$$

— For (M-ρ-LIFT), we proceed similarly to the (M-O-LIFT) case.

— For (L-COPY), with $\rho_h = \{x \mapsto \Omega \mid x \in \mathsf{dom}(\iota_h, \rho_h)\}$, $h \in \{1, 2\}$, we have

$$\begin{array}{rcl}\left[\iota_1; o_1; \rho_1 \mid e\right] & \xrightarrow[\omega]{*} & \left[\iota_1; \rho_1(o_1); \rho_1 \mid \rho_1(e)\right] \\ & \xrightarrow[\omega]{} & \left[\iota_1; \rho_1(o_1); \mid \rho_1(e)\right]\end{array}$$

and

$$\begin{array}{rcl}\left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e'\right]\right]\uparrow & \xrightarrow[\omega]{*} & \left[\iota_2; \rho_2(o_2); \rho_2 \mid \rho_2(\left[\iota_1; \rho_1(o_1); \rho_1 \mid \rho_1(e')\right])\right]\uparrow \\ & \equiv & \left[\iota_2; \rho_2(o_2); \rho_2 \mid \left[\iota_1; \rho_1(o_1); \rho_1 \mid \rho_2(\rho_1(e'))\right]\right]\uparrow \\ & & \text{(side-condition of (L-COPY))} \\ & \xrightarrow[\omega]{} & \left[\iota_2; \rho_2(o_2); \mid \left[\iota_1; \rho_1(o_1); \rho_1 \mid \rho_2(\rho_1(e'))\right]\right]\uparrow \\ & \xrightarrow[\omega]{} & \left[\iota_1; \rho_1(o_1); \rho_1 \mid \rho_2(\rho_1(e'))\right] ,\end{array}$$

so, we can conclude by observing that $\rho_1(e) = \rho_1(\rho_2(\rho'(e))) = \rho_1(\rho_2(e'))$.

— For (PRG-LIFT), with $\rho_2 = \{x \mapsto \Omega \mid x \in \mathsf{dom}(\iota_2, \rho_2)\}$, we have

$$\begin{array}{rcl}\left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right] & \xrightarrow[\omega]{*} & \left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; \rho_2(o_2); \rho_2 \mid \rho_2(e)\right]\uparrow\right] \\ & \xrightarrow[\omega]{} & \left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; \rho_2(o_2); \mid \rho_2(e)\right]\uparrow\right] \\ & \xrightarrow[\omega]{} & \left[\iota_1; o_1; \rho_1 \mid \rho_2(e)\right]\end{array}$$

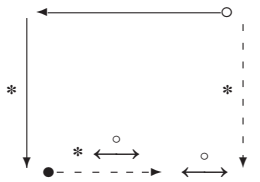
https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

and

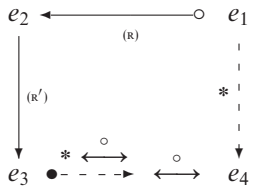
$$\begin{aligned}
\big[\iota_2;\, o_2;\, \rho_2 \mid \big[\iota_1;\, o_1;\, \rho_1 \mid e\big]\big]\uparrow \;&\xrightarrow[\omega]{*}\; \big[\iota_2;\, \rho_2(o_2);\, \rho_2 \mid \rho_2(\big[\iota_1;\, o_1;\, \rho_1 \mid e\big])\big]\uparrow \\
&\equiv\; \big[\iota_2;\, \rho_2(o_2);\, \rho_2 \mid \big[\iota_1;\, o_1;\, \rho_1 \mid \rho_2(e)\big]\big]\uparrow \\
&\qquad\qquad \text{(side-condition of } {}_{\text{(PRG-LIFT)}}\text{)} \\
&\xrightarrow[\omega]{}\; \big[\iota_2;\, \rho_2(o_2);\, \mid \big[\iota_1;\, o_1;\, \rho_1 \mid \rho_2(e)\big]\big]\uparrow \\
&\xrightarrow[\omega]{}\; \big[\iota_1;\, o_1;\, \rho_1 \mid \rho_2(e)\big]\,. \qquad \square
\end{aligned}$$

We will now show that the $\longrightarrow$ reduction relation is complete up to information content with respect to $\bullet\!\!\longrightarrow$. The proof (see Lemma 4.13) consists of showing that the diagram in Lemma 4.6 holds by taking $\overset{\circ}{\longleftrightarrow}$ as the equivalence relation to be used by the proposition above.

Lemma 4.13 (Right reduction lemma).



Proof. We prove that the diagram



commutes by case analysis on the (R) step. We will only show some interesting cases:

— If (R) = (MOD-PROP) + (M-ρ-SUBST) step, we consider all possible R-calculus steps:

(1) If the (R) step duplicates the (R′) redex (that is, if this redex is inside the substituted term), then the diagram commutes by applying (R′) as a vertical step and $\bullet\xrightarrow{(R')}$ $\xleftarrow{(R)}\!\!\circ$ as horizontal steps.

(2) Otherwise, we could have one of the following cases:

(a) If (R′) = (M-SUM), (M-REDUCT), (MOD-PROP)+(M-SUM), (MOD-PROP)+(M-REDUCT), (SUM-LIFT), (R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (RUN-LIFT) or (RES-LIFT), then the diagram commutes by applying (R′) as a vertical step and $\xleftarrow{(R)}\!\!\circ$ as a horizontal step.

(b) If (R′) = (M-FREEZE), (MOD-PROP)+(M-FREEZE), (PRG-SUBST) or (RUN), we have the following possible cases:

(i) If the (R′) step duplicates the (R) redex (that is, if this redex is inside an output component definition involved through either the freeze or run operator, or inside the local component definition substitued inside the

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

running program), the diagram commutes by applying (R') as a vertical step and $\xleftarrow{(\mathrm{R})}\circ\xleftarrow{(\mathrm{R})}\circ$ as horizontal steps.

(ii) In the case $(\mathrm{R}') = (\text{PRG-SUBST})$, with

$$\begin{aligned} e_2 &= \left[\iota; o; x_1 \mapsto \mathscr{E}_1\{\rho(x_2)\}, \rho \mid \mathscr{E}[x_1]\right] \\ e_3 &= \left[\iota; o; x_1 \mapsto \mathscr{E}_1\{\rho(x_2)\}, \rho \mid \mathscr{E}\{\mathscr{E}_1\{\rho(x_2)\}\}\right] \\ e_1 &= \left[\iota; o; x_1 \mapsto \mathscr{E}_1[x_2], \rho \mid \mathscr{E}[x_1]\right] \end{aligned}$$

the diagram commutes by applying (PRG-SUBST) twice as vertical steps, to give the configuration

$$e_4 = \left[\iota; o; x_1 \mapsto \mathscr{E}[x_2], \rho \mid \mathscr{E}\{\mathscr{E}_1[\rho(x_2)]\}\right]$$

and $\xleftarrow{(\mathrm{R})}\circ$ as a horizontal step.

(iii) In all other cases the diagram commutes by applying (R') as a vertical step and $\xleftarrow{(\mathrm{R})}\circ$ as a horizontal step.

(c) If $(\mathrm{R}') = (\text{RES})$, we could have one of the following cases:

(i) If the (R') step removes the (R) redex, that is, either this redex is inside the reconfiguration level from which the program is extracted or we have

$$\begin{aligned} e_1 &= \left(\mathscr{R}\left[\iota; o; x_1 \mapsto \mathscr{E}[x_2]\,\rho \mid e\right]\right)\uparrow \\ e_2 &= \left(\mathscr{R}\left[\iota; o; x_1 \mapsto \mathscr{E}\{\rho(x_2)\}\,\rho \mid e\right]\right)\uparrow \\ e_3 &= e, \end{aligned}$$

then in these cases the diagram commutes by applying (RES) as a vertical step and no horizontal step; hence, $e_4 = e_3$.

(ii) In all other cases the diagram commutes by applying (R') as a vertical step and $\xleftarrow{(\mathrm{R})}\circ$ as a horizontal step.

— If $(\mathrm{R}) = (\text{PRG-LIFT})$, we consider the following cases:

(1) If $(\mathrm{R}') = (\text{RES-LIFT})$ and

$$\begin{aligned} e_1 &= \left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right]\uparrow \\ e_2 &= \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e\right]\right]\uparrow\uparrow \\ e_3 &= \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e\right]\uparrow\right]\uparrow, \end{aligned}$$

we can conclude by applying no vertical step and $\bullet\xrightarrow{(\mathrm{R})}\bullet\xrightarrow{(\mathrm{R}')}$ as horizontal steps.

(2) If $(\mathrm{R}') = (\text{M-FREEZE})$, $(\text{MOD-PROP})+(\text{M-FREEZE})$, (PRG-SUBST) or (RUN), we have the following possible cases.

(a) If the (R') step duplicates the (R) redex (that is, if this redex is inside an output component definition involved through either the freeze or run operator, or inside the local component definition substitued within the running program), the diagram commutes by applying (R') as a vertical step and $\xleftarrow{(\mathrm{R})}\circ\xleftarrow{(\mathrm{R})}\circ$ as horizontal steps.

(b) In all other cases we can conclude by applying (R') as a vertical step and $\xleftarrow{(\mathrm{R})}\circ$ as a horizontal step.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

In particular:

(i) When (R') = (PRG-SUBST):

– we have

$$\begin{aligned} e_1 &= \left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right] \\ e_2 &= \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e\right]\right]\uparrow, \end{aligned}$$

where $\left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e\right]\right]$ is the (R') redex, with $\left[\iota_1; o_1; \rho_1 \mid e\right] = \mathscr{E}[x]$ and $x \in \mathsf{dom}(\rho_2)$;

– we observe that, for the side-condition of rule (R), we must have $\mathscr{E}[x] = \left[\iota_1; o_1; \rho_1 \mid \mathscr{E}_1[x]\right]$;

– hence, we apply the (R') vertical step to the redex $\left[\iota_2; o_2; \rho_2 \mid \mathscr{E}_1[x]\right]$.

(ii) But, when (R') = (PRG-SUBST):

– we have

$$\begin{aligned} e_1 &= \left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right] \\ e_2 &= \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e\right]\right]\uparrow \end{aligned}$$

where $\left[\iota_1; o_1; \rho_1 \mid e\right]$ is the (R') redex, with $e = \mathscr{E}[x]$ and $x \in \mathsf{dom}(\rho_1)$;

– for the third side-condition of (R), we can apply the (R') vertical step to $\left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid \mathscr{E}[x]\right]\uparrow\right]$.

(3) If (R') = (RES), we could have one of the following cases:

(a) If the (R') step removes the (R) redex, that is, if either this redex is inside the reconfiguration level from which the program is extracted, or we have

$$\begin{aligned} e_1 &= \left[\iota_1; o_1; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right] \\ e_2 &= \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1 \mid e\right]\right]\uparrow \\ e_3 &= \left[\iota_1; o_1; \rho_1 \mid e\right], \end{aligned}$$

then in these cases the diagram commutes by applying (R') as a vertical step and no horizontal step, so $e_4 = e_3$. Note that, in the latter case, we can safely apply (R') to $\left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow$ since for the side-condition of (R') applied to e_2 we have that $\mathsf{FV}(\left[\iota_1; o_1; \rho_1 \mid e\right]) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$ and thus $\mathsf{FV}(e) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$.

(b) In all other cases the diagram commutes by applying (R') as a vertical step and $\xleftarrow{(\mathrm{R})}\!\!\circ$ as a horizontal step. □

Theorem 4.14. Given the ARS with ordered information content

$$((\mathsf{Exp}, \bullet\!\!\longrightarrow, \preceq_\omega), \omega, (\omega(\mathsf{Exp}), \leqslant_\Omega)),$$

$\longrightarrow$ is complete for $\bullet\!\!\longrightarrow$ up to $\preceq_\omega$.

Proof. By Lemma 4.13, we get that an instance of the diagram in Lemma 4.6 holds, since $\overset{\circ}{\longleftrightarrow} \subseteq \simeq_\omega$ by Proposition 4.12. □

Theorem 4.14 implies the soundness of the call-by-need strategy, as stated in the following corollary.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Corollary 4.15. Given $e \in \mathsf{Exp}$, if $e \xrightarrow[\mathsf{need}]{} {}^{*}e_1$, then there exist e_2 such that $e \longrightarrow {}^{*}e_2$ and $e_1 \preceq_{\omega} e_2$.

Proof. The thesis follows from Theorem 4.14 since $\xrightarrow[\mathsf{need}]{}$ is a subset of $\bullet\!\!\longrightarrow$ (Proposition 4.14). □

4.4. *Completeness of the call-by-need strategy*

We will now prove the completeness of the $\xrightarrow[\mathsf{need}]{}$ reduction relation with respect to the R-calculus (Theorem 4.20). The proof is based on Lemma 4.19 later in this section, which shows that the left-hand diagram in Lemma 4.6 holds, by taking $\overset{\circ}{\longleftrightarrow}$ as the equivalence relation.

Definition 4.16 (Reduction $\xrightarrow[R\backslash\mathsf{need}]{}$). We use $\xrightarrow[R\backslash\mathsf{need}]{}$ to denote the reduction relation $\longrightarrow \setminus \xrightarrow[\mathsf{need}]{}$.

We need the following propositions to prove Lemma 4.19.

Proposition 4.17. We have

$$\left[\iota_1;\, o_1;\, \rho_1, x \mapsto \left[\iota_2;\, o_2;\, \rho_2 \mid e\right]\uparrow \mid \mathscr{E}\left[\left[\iota_2;\, o_2;\, \rho_2 \mid e\right]\uparrow\right]\right] \;\bullet \overset{*}{\overset{\circ}{\longleftrightarrow}}\; \left[\iota_1;\, o_1;\, \rho_1, x \mapsto e, \rho_2 \mid \mathscr{E}[e]\right]$$

under the assumptions $(\mathsf{FV}(e) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) = \varnothing$ and $\mathsf{HB}(\mathscr{E}) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing$.

Proof. We use structural induction on the context $\mathscr{E}$ to prove that

$$\mathscr{E}\left[\left[\iota_2;\, o_2;\, \rho_2 \mid e\right]\uparrow\right] \;\bullet \overset{*}{\longrightarrow}\; \left[\iota_2;\, o_2;\, \rho_2 \mid \mathscr{E}[e]\right]\uparrow \tag{1}$$

under the assumptions

$$\mathsf{HB}(\mathscr{E}) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing,\ \mathsf{FV}(\mathscr{E}) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing \ \text{ and } \ \mathsf{HB}(\mathscr{E}) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing. \tag{2}$$

Note that the final two side-conditions in (2) can always be satisfied by α-conversion.

(a) $\mathscr{E} \equiv \left[\iota;\, o, X \mapsto \mathscr{E}';\, \rho\right]$:

$$\begin{aligned}\left[\iota;\, o, X \mapsto \mathscr{E}'\left[\left[\iota_2;\, o_2;\, \rho_2 \mid e\right]\uparrow\right];\, \rho\right] \;&\overset{\circ}{\longleftrightarrow}\; \left[\iota;\, o, X \mapsto \left[\iota_2;\, o_2;\, \rho_2 \mid \mathscr{E}'[e]\right]\uparrow;\, \rho\right] \\ &\qquad \text{(by the induction hypothesis)} \\ &\bullet\!\!\longrightarrow\; \left[\iota_2;\, o_2;\, \rho_2 \mid \left[\iota;\, o, X \mapsto \mathscr{E}'[e];\, \rho\right]\right]\uparrow \\ &\qquad \text{(by } {}_{(\text{M-}o\text{-LIFT})}\text{)}.\end{aligned}$$

Note that the only side-condition of $_{(\text{M-}o\text{-LIFT})}$ not guaranteed by α-conversion, that is, $\mathsf{dom}(\iota, \rho) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing$, follows from (2).

(b) Cases $\mathscr{E} \equiv \left[\iota;\, o, X \mapsto \mathscr{E}';\, \rho \mid e\right]$, $\mathscr{E} \equiv \left[\iota;\, o;\, \rho, x \mapsto \mathscr{E}'\right]$ and $\mathscr{E} \equiv \left[\iota;\, o;\, \rho, x \mapsto \mathscr{E}' \mid e\right]$ are similar to (a): we use the induction hypothesis and $_{(\text{MOD-PROP})}+_{(\text{M-}o\text{-LIFT})}$, $_{(\text{M-}\rho\text{-LIFT})}$ and $_{(\text{MOD-PROP})}+_{(\text{M-}\rho\text{-LIFT})}$, respectively; and again the only side-condition of these rules that cannot be guaranteed by α-conversion, that is, $\mathsf{dom}(\iota, \rho) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing$ in the former case and $(\mathsf{dom}(\iota, \rho) \cup \{x\}) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing$ in the latter two cases, follows from (2).

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(c) $\mathscr{E} \equiv [\iota; o; \rho \mid \mathscr{E}']$:

$$\begin{aligned}[\iota; o; \rho \mid \mathscr{E}'[[\iota_2; o_2; \rho_2 \mid e]\uparrow]] \quad &\overset{\circ}{\longleftrightarrow} \quad [\iota; o; \rho \mid [\iota_2; o_2; \rho_2 \mid \mathscr{E}'[e]]\uparrow] \\ &\qquad\qquad \text{(by the induction hypothesis)} \\ &\bullet\longrightarrow \quad [\iota_2; o_2; \rho_2 \mid [\iota; o; \rho \mid \mathscr{E}'[e]]]\uparrow \text{ (by } \textsc{(prg-lift)}).\end{aligned}$$

Note that in this case also the only side-condition of (PRG-LIFT) that cannot be guaranteed by α-conversion, that is, $\mathsf{dom}(\iota, \rho) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing$, follows from (2).

(d) $\mathscr{E} \equiv \mathscr{E}' + e$:

$$\begin{aligned}\mathscr{E}'[[\iota_2; o_2; \rho_2 \mid e']\uparrow] + e \quad &\overset{\circ}{\longleftrightarrow} \quad [\iota_2; o_2; \rho_2 \mid \mathscr{E}'[e']]\uparrow + e \\ &\qquad\qquad \text{(by the induction hypothesis)} \\ &\bullet\longrightarrow \quad [\iota_2; o_2; \rho_2 \mid \mathscr{E}'[e'] + e]\uparrow \qquad \text{(by } \textsc{(sum-lift)}).\end{aligned}$$

(e) Case $\mathscr{E} \equiv e + \mathscr{E}'$ is similar to (d) using the induction hypothesis and rule (R-SUM-LIFT).

(f) $\mathscr{E} \equiv {}_{\sigma^I}|\mathscr{E}'|_{\sigma^O}$:

$$\begin{aligned}{}_{\sigma^I}|\mathscr{E}'[[\iota_2; o_2; \rho_2 \mid e]\uparrow]|_{\sigma^O} \quad &\overset{\circ}{\longleftrightarrow} \quad {}_{\sigma^I}|[\iota_2; o_2; \rho_2 \mid \mathscr{E}'[e]]\uparrow|_{\sigma^O} \\ &\qquad\qquad \text{(by the induction hypothesis)} \\ &\bullet\longrightarrow \quad [\iota_2; o_2; \rho_2 \mid {}_{\sigma^I}|\mathscr{E}'[e]|_{\sigma^O}]\uparrow \qquad \text{(by } \textsc{(reduct-lift)}).\end{aligned}$$

(g) Cases $\mathscr{E} \equiv \mathsf{freeze}_\sigma \mathscr{E}'$, $\mathscr{E} \equiv \mathscr{E}'\downarrow_X$ and $\mathscr{E} \equiv \mathscr{E}'\uparrow$ are similar to (f): we use the induction hypothesis and the (FREEZE-LIFT), (RUN-LIFT) and (RES-LIFT) rules, respectively.

Using this equivalence, we can now prove the statement in the proposition:

$$\begin{aligned}&[\iota_1; o_1; \rho_1, x \mapsto [\iota_2; o_2; \rho_2 \mid e]\uparrow \mid \mathscr{E}[[\iota_2; o_2; \rho_2 \mid e]\uparrow]] \\ &\bullet\text{-}\overset{*}{\text{-}}\!\!\rightarrow \quad [\iota_1; o_1; \rho_1, x \mapsto [\iota_2; o_2; \rho_2 \mid e]\uparrow \mid [\iota_2; o_2; \rho_2 \mid \mathscr{E}[e]]\uparrow] && \text{(by (1))} \\ &\circ\longrightarrow \quad [\iota_1; o_1; \rho_1, x \mapsto [\iota_2; o_2; \rho_2 \mid e]\uparrow, \rho_2 \mid \mathscr{E}[e]] && \text{(by } \textsc{(prg-merge)}) \\ &\circ\longrightarrow \quad [\iota_1; o_1; \rho_1, x \mapsto \rho(e), \rho_2, \rho(\rho_2) \mid \mathscr{E}[e]] && \text{(by } \textsc{(mod-prop)}+\textsc{(m-merge)}) \\ &\longleftarrow\!\circ \quad [\iota_1; o_1; \rho_1, x \mapsto e, \rho_2 \mid \mathscr{E}[e]] && \text{(by } \textsc{(m-copy)}).\end{aligned}$$

Note that, in the first step, we can correctly apply rule (1) since the only side-condition which cannot be guaranteed by α-conversion is satisfied by hypothesis; the (PRG-MERGE) step can safely be performed since by hypothesis $(\mathsf{FV}(e) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) = \varnothing$ and $\mathsf{FV}(\mathscr{E}) \cap \mathsf{FV}(\iota_2) = \varnothing$ (from (2)); moreover, in the (MOD-PROP)+(M-MERGE) step, the local environment ρ_2 and the program e are α-renamed to avoid clashes. □

Proposition 4.18. We have that

$$\begin{aligned}&[\iota_1; o_1; \rho_1, x \mapsto [\iota_2; o_2; \rho_2 \mid e]\uparrow \mid \mathscr{E}\{[\iota_2; o_2; \rho_2 \mid e]\uparrow\}] \\ &\qquad\qquad \overset{\circ}{\longleftrightarrow} [\iota_2; o_2; \rho_2 \mid [\iota_1; o_1; \rho_1, x \mapsto e \mid \mathscr{E}[x]]]\uparrow\end{aligned}$$

under the assumption $(\mathsf{dom}(\iota_1, \rho_1) \cup \{x\}) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing$.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Proof.

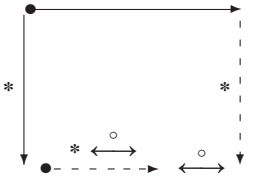
$$
\begin{array}{ll}
& \left[\iota_1; o_1; \rho_1, x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow \mid \mathscr{E}\left[\left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right]\right] \\
\overset{\circ}{\longleftrightarrow} & \left[\iota_1; o_1; \rho_1, x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow \mid \left[\iota_2; o_2; \rho_2 \mid \mathscr{E}[e]\right]\uparrow\right] \qquad \text{(by (1))} \\
\circ\!\!\longrightarrow & \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1, x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow \mid \mathscr{E}[e]\right]\right]\uparrow \qquad \text{(by (PRG-LIFT))} \\
\circ\!\!\longrightarrow & \left[\rho(\iota_2); \rho(o_2); \rho(\rho_2) \mid \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1, x \mapsto \rho(e) \mid \mathscr{E}[e]\right]\right]\right]\uparrow \\
& \qquad \text{(by (MOD-PROP)+(M-}\rho\text{-LIFT))} \\
\longleftarrow\!\!\circ & \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1, x \mapsto e \mid \mathscr{E}[e]\right]\right]\uparrow \qquad \text{(by (L-COPY))}.
\end{array}
$$

Note that the first side-condition in (2) is satisfied since in the statement of the proposition we have the capture-avoiding substitution of the hole of $\mathscr{E}$ with $\left[\iota_2; o_2; \rho_2 \mid e\right]$. The (PRG-LIFT) step can be performed safely since, by hypothesis,

$$(\mathsf{dom}(\iota_1, \rho_1) \cup \{x\}) \cap (\mathsf{FV}(o_2) \cup \mathsf{FV}(\rho_2)) = \varnothing .$$

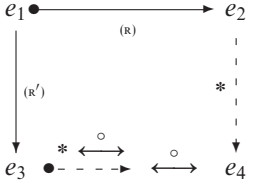
For the same reason, the (MOD-PROP)+(M-ρ-LIFT) step can also be safely performed. Finally, note that in the (MOD-PROP)+(M-ρ-LIFT) step the lifted components and the program are α-renamed to avoid clashes. □

Lemma 4.19 (Left reduction lemma). The following diagram holds:



Moreover, the number of R-calculus steps in the right reduction that are also call-by-need steps is not greater than the number of R-calculus and call-by-need steps in the left reduction. It is strictly smaller if a descendant of a call-by-need step in the top enriched reduction is contracted in the (left) calculus reduction.

Proof. We use case analysis on the enriched reduction step.



In the following we show how the previous diagram commutes by illustrating the horizontal steps.

The commutativity of the above diagram for all enriched reduction rules that are also R-calculus rules follows from the Church–Rosser property of the R-calculus (Theorem 2.14). Hence, we only have to prove the commutativity for the new rules. We will only show

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

some interesting cases:

— If (R) = (M-ρ-SUBST), (M-COPY), (L-COPY), (MOD-PROP)+(M-ρ-SUBST) or (MOD-PROP)+(M-COPY), we have the following cases:

(A) If the (R) step duplicates the (R′) redex (that is, if this redex is inside either the substituted term or the copied assignments), the diagram commutes with two vertical steps (rule (R′)) and one horizontal step (rule(R)).

(B) Otherwise, the following cases are possible.

(a) If (R′) = (M-SUM), (M-REDUCT), (MOD-PROP)+(M-SUM), (MOD-PROP)+(M-REDUCT), (SUM-LIFT), (R-SUM-LIFT), (REDUCT-LIFT), (FREEZE-LIFT), (RUN-LIFT) or (RES-LIFT), then the diagram commutes with one vertical step (rule (R′)) and one horizontal step (rule (R)).

(b) If (R′) = (M-FREEZE), (MOD-PROP)+(M-FREEZE), (PRG-SUBST), (RUN) or (RES), we have the following possible cases:

(i) If the (R′) step duplicates the (R) redex (that is, if this redex is inside an output component definition involved through either the freeze or run operator or inside the local component definition substitued in the running program), the diagram commutes with one vertical step (rule (R′)) and two horizontal steps (rule (R)).

(ii) Otherwise, we could have one of the following critical pairs:

1. If

$$\begin{aligned} e_1 &= \left[\iota;\, o;\, x_1 \mapsto \mathscr{E}_1[x_2]\,, \rho \mid \mathscr{E}[x_1]\right] \\ e_2 &= \left[\iota;\, o;\, x_1 \mapsto \mathscr{E}_1\{\rho(x_2)\}\,, \rho \mid \mathscr{E}[x_1]\right] \\ e_3 &= \left[\iota;\, o;\, x_1 \mapsto \mathscr{E}_1[x_2]\,, \rho \mid \mathscr{E}\{\mathscr{E}_1[x_2]\}\right] \end{aligned}$$

and (R) = (MOD-PROP)+(M-ρ-SUBST), (R′) = (PRG-SUBST), then the diagram commutes by applying (R′) as a vertical step and (R′) followed by (R) as a horizontal step; hence, $e_4 = \left[\iota;\, o;\, x_1 \mapsto \mathscr{E}_1\{\rho(x_2)\}\,, \rho \mid \mathscr{E}\{\mathscr{E}_1\{\rho(x_2)\}\}\right]$.

2. If

$$\begin{aligned} e_1 &= \left(\mathscr{R}\left[\iota;\, o;\, \rho \mid e\right]\right)\uparrow \\ e_2 &= \left(\mathscr{R}'\left[\iota';\, o';\, \rho' \mid \alpha\left(e'\right)\right]\right)\uparrow \\ e_3 &= e \end{aligned}$$

with (R) = (MOD-PROP) plus (M-FREEZE), (M-REDUCT), (M-SUM), (M-ρ-SUBST), (M-MERGE), (M-GC) or (M-COPY) and (R′) = (RES), then the diagram commutes by simply applying (R′) as a vertical step and no horizontal step.

3. If the critical pair is with

$$\begin{aligned} e_1 &= \left[\iota_1;\, X \mapsto \left[\iota_2;\, o_2;\, \rho_2 \mid e_2\right]\uparrow, o_1;\, \rho_1 \mid e\right]\uparrow \\ e_2 &= \left[\iota_2;\, o_2;\, \rho_2 \mid \left[\iota_1;\, X \mapsto e_2, o_1;\, \rho_1 \mid e\right]\right]\uparrow\uparrow \\ e_3 &= e \end{aligned}$$

with (R) = (MOD-PROP)+(M-o-LIFT) and (R′) = (RES), then the diagram commutes by applying (R′) as a vertical step and (RES-LIFT) once and (RES) twice as horizontal steps.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

4. The critical pair with

$$e_1 = \left[\iota_1; o_1; x \mapsto \left[\iota_2; o_2; \rho_2 \mid e_2\right]\uparrow, \rho_1 \mid e\right]\uparrow,$$

with $_{(R)} = {}_{(\text{MOD-PROP})}{+}_{(\text{M-}\rho\text{-LIFT})}$ and $_{(R')} = {}_{(\text{RES})}$ is handled similarly to 3.

5. In the case

$$e_1 = \left[\iota; o; \rho \mid e\right]\uparrow$$
$$e_2 = \left[\iota'; o'; \rho' \mid \left[\iota; o; \rho \mid \alpha'(e)\right]\right]\uparrow$$

with $\left[\iota'; o'; \rho'\right] \widetilde{=}_{\alpha} [\iota; o; \rho]$ and $\alpha' \subseteq \alpha$, $e_3 = e$, $_{(R)} = {}_{(\text{MOD-PROP})}{+}_{(\text{L-COPY})}$ and $_{(R')} = {}_{(\text{RES})}$, the diagram commutes by applying $_{(R')}$ as a vertical step and $_{(\text{RES})}$ twice as horizontal steps. Note that we certainly have $\mathsf{FV}(\left[\iota; o; \rho \mid \alpha'(e)\right]) \cap \mathsf{dom}(\iota', \rho') = \varnothing$ and $\mathsf{FV}(\alpha'(e)) \cap \mathsf{dom}(\iota, \rho) = \varnothing$ (from $_{(R')}$ applied to e_1 and the side-conditions of rule $_{(\text{L-COPY})}$).

(iii) In all other cases the diagram commutes with one vertical step (rule $_{(R')}$) and one horizontal step (rule $_{(R)}$).

— If $_{(R)} = {}_{(\text{M-MERGE})}$, $_{(\text{PRG-MERGE})}$, $_{(\text{M-GC})}$, $_{(\text{GC})}$, $_{(\text{MOD-PROP})}{+}_{(\text{M-MERGE})}$ or $_{(\text{MOD-PROP})}{+}_{(\text{PRG-MERGE})}$, we consider all possible R-calculus steps:

(A) If the $_{(R)}$ step eliminates the $_{(R')}$ redex (that is, this redex is inside one of the removed either output or local component), the diagram commutes with no vertical step and one horizontal step (rule$_{(R)}$).

(B) If $_{(R')} = {}_{(\text{M-SUM})}$, $_{(\text{M-REDUCT})}$, $_{(\text{MOD-PROP})}{+}_{(\text{M-SUM})}$, $_{(\text{MOD-PROP})}{+}_{(\text{M-REDUCT})}$, $_{(\text{SUM-LIFT})}$, $_{(\text{R-SUM-LIFT})}$, $_{(\text{REDUCT-LIFT})}$, $_{(\text{FREEZE-LIFT})}$, $_{(\text{RUN-LIFT})}$ or $_{(\text{RES-LIFT})}$, the diagram commutes with one vertical step (rule $_{(R')}$) and one horizontal step (rule $_{(R)}$).

(C) If $_{(R')} = {}_{(\text{M-FREEZE})}$, $_{(\text{MOD-PROP})}{+}_{(\text{M-FREEZE})}$, $_{(\text{PRG-SUBST})}$, $_{(\text{RUN})}$ or $_{(\text{RES})}$, we have the following possible cases:

(a) If the $_{(R')}$ step duplicates the $_{(R)}$ redex (that is, if this redex is inside an output component definition involved by either the freeze or the run operator or inside the local component definition substituted in the running program), the diagram commutes with one vertical step (rule $_{(R')}$) and two horizontal steps (rule $_{(R)}$).

(b) Otherwise, we could have one of the following critical pairs:

6. If

$$e_1 = \left[\iota_1; o_1; \rho_1, x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow \mid \mathscr{E}[x]\right]$$

with $_{(R)} = {}_{(\text{MOD-PROP})}{+}_{(\text{M-MERGE})}$ and

$$e_2 = \left[\iota_1; o_1; \rho_1, x \mapsto e, \rho_2 \mid \mathscr{E}[x]\right]$$

with $_{(R')} = {}_{(\text{PRG-SUBST})}$ and

$$e_3 = \left[\iota_1; o_1; \rho_1, x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow \mid \mathscr{E}\left\{\left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right\}\right],$$

we can conclude by performing $_{(R')}$ as a vertical step, to give

$$e_4 = \left[\iota_1; o_1; \rho_1, x \mapsto e, \rho_2 \mid \mathscr{E}\{e\}\right]$$

and by applying Proposition 4.17 to e_3 and e_4. Note that all the conditions required to apply this proposition are satisfied, since, from $_{(R)}$ applied to

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

e_1, we have that $(\mathsf{FV}(e) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) = \varnothing$ and, moreover, the second condition is guaranteed by the fact that the substitution $\mathscr{E}\{[\iota_2; o_2; \rho_2 \mid e]\uparrow\}$ is capture avoiding.

7. If
$$e_1 = [\iota_1; o_1; \rho_1 \mid [\iota_2; o_2; \rho_2 \mid e]\uparrow]$$
with $_{(R)} = {}_{(\text{PRG-MERGE})}$ and
$$e_2 = [\iota_1; o_1; \rho_1, \rho_2 \mid e]$$
with $_{(R')} = {}_{(\text{RES})}$ and
$$e_3 = [\iota_1; o_1; \rho_1 \mid e] \ ,$$
we can conclude by applying no vertical step and $_{(\text{GC})}$ as a horizontal step. Note that the side-condition of $_{(\text{GC})}$ is satisfied as, from rule $_{(R')}$ applied to e_1, we have $\mathsf{FV}(e) \cap \mathsf{dom}(\rho_2) = \varnothing$, and from rule $_{(R)}$ applied to e_1, we also have $(\mathsf{FV}(o_1) \cup \mathsf{FV}(\rho_1)) \cap \mathsf{dom}(\rho_2) = \varnothing$.

(c) In all other cases the diagram commutes with one vertical step (rule $_{(R')}$) and one horizontal step (rule $_{(R)}$).

— If $_{(R)} = {}_{(\text{M-}o\text{-LIFT})}$, $_{(\text{M-}\rho\text{-LIFT})}$, $_{(\text{MOD-PROP})}+{}_{(\text{M-}o\text{-LIFT})}$, $_{(\text{MOD-PROP})}+{}_{(\text{M-}\rho\text{-LIFT})}$ or $_{(\text{PRG-LIFT})}$, we consider all possible R-calculus steps.

(A) If $_{(R')} = {}_{(\text{M-FREEZE})}$, $_{(\text{MOD-PROP})}+{}_{(\text{M-FREEZE})}$, $_{(\text{PRG-SUBST})}$ or $_{(\text{RUN})}$, and if the $_{(R')}$ step duplicates the $_{(R)}$ redex (that is, if this redex is inside one of the output component definitions involved through either the freeze or run operator, or inside the local component definition substituted within the running program), the diagram commutes with one vertical step (rule $_{(R')}$) and two horizontal steps (rule $_{(R)}$).

(B) Otherwise, we could have one of the following critical pairs:

8. If
$$e_1 = [\iota_1; X \mapsto [\iota_2; o_2; \rho_2 \mid e]\uparrow, o_1; \rho_1] + [\iota_3; o_3; \rho_3]$$
with $_{(R)} = {}_{(\text{M-}o\text{-LIFT})}$ and
$$e_2 = [\iota_2; o_2; \rho_2 \mid [\iota_1; o_1, X \mapsto e; \rho_1]]\uparrow + [\iota_3; o_3; \rho_3]$$
with $_{(R')} = {}_{(\text{M-SUM})}$ and
$$e_3 = [\iota_1, \iota_3; o_1, X \mapsto [\iota_2; o_2; \rho_2 \mid e]\uparrow, o_3; \rho_1, \rho_3] \ ,$$
we can conclude by applying $_{(\text{SUM-LIFT})}$ and $_{(\text{M-SUM})}$ as vertical steps and $\overset{(\text{M-}o\text{-LIFT})}{\circ\!\!\longrightarrow}$ as a horizontal step. Hence,
$$e_4 = [\iota_2; o_2; \rho_2 \mid [\iota_1, \iota_3; o_1, X \mapsto e, o_3; \rho_1, \rho_3]]\uparrow \ .$$

9. Critical pairs similar to the previous one, but with either $_{(R')} = {}_{(\text{M-REDUCT})}$ or $_{(R')} = {}_{(\text{M-FREEZE})}$ are handled in a similar way. That is, by performing, in the former case, $_{(\text{REDUCT-LIFT})}$ and $_{(\text{M-REDUCT})}$ as vertical steps and $\overset{(\text{M-}o\text{-LIFT})}{\circ\!\!\longrightarrow}$ as a horizontal step; and, in the latter case, $_{(\text{FREEZE-LIFT})}$ and $_{(\text{M-FREEZE})}$ as vertical steps and $\overset{(\text{M-}o\text{-LIFT})}{\circ\!\!\longrightarrow}$ as a horizontal step.

10. Critical pairs as in previous two points, but with $_{(R)} = {}_{(\text{MOD-PROP})}+{}_{(\text{M-}o\text{-LIFT})}$ and one of

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

(i) $(\mathrm{R}') = (\text{MOD-PROP})+(\text{M-SUM})$,

(ii) $(\mathrm{R}') = (\text{MOD-PROP})+(\text{M-REDUCT})$ or

(iii) $(\mathrm{R}') = (\text{MOD-PROP})+(\text{M-FREEZE})$

are handled similarly; that is, by performing

(i) (SUM-LIFT) and (MOD-PROP)+(M-SUM) as vertical steps and $\overset{(\text{MOD-PROP})+(\text{M-}o\text{-LIFT})}{\circ\!\!\longrightarrow}$ as a horizontal step,

(ii) (REDUCT-LIFT) and (MOD-PROP)+(M-REDUCT) as vertical steps and $\overset{(\text{MOD-PROP})+(\text{M-}o\text{-LIFT})}{\circ\!\!\longrightarrow}$ as a horizontal step, or

(iii) (FREEZE-LIFT) and (MOD-PROP)+(M-FREEZE) as vertical steps and $\overset{(\text{MOD-PROP})+(\text{M-}o\text{-LIFT})}{\circ\!\!\longrightarrow}$ as a horizontal step,

respectively.

11. If

$$e_1 = \left[\iota_1; X \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow, o_1; \rho_1\right]\downarrow_X$$

with $(\mathrm{R}) = (\text{M-}o\text{-LIFT})$ and

$$e_2 = \left[\iota_2; o_2; \rho_2 \mid [\iota_1; o_1, X \mapsto e; \rho_1]\right]\uparrow\downarrow_X$$

with $(\mathrm{R}') = (\text{RUN})$ and

$$e_3 = \left[\iota_1; o_1, X \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow; \rho_1 \mid \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right],$$

we can conclude by applying (RUN-LIFT) and (RUN) as vertical steps and

$$\overset{(\text{L-COPY})}{\longleftarrow\!\!\circ}\quad\overset{(\text{MOD-PROP})+(\text{M-}o\text{-LIFT})}{\longleftarrow\!\!\circ}\quad\overset{(\text{PRG-LIFT})}{\longleftarrow\!\!\circ}$$

as a horizontal step. Hence,

$$e_4 = \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1, X \mapsto e; \rho_1 \mid e\right]\right]\uparrow .$$

12. If

$$e_1 = \left[\iota_1; X \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow, o_1; \rho_1\right]$$

with $(\mathrm{R}) = (\text{M-}o\text{-LIFT})$ and

$$e_2 = \left[\iota_2; o_2; \rho_2 \mid [\iota_1; o_1, X \mapsto e; \rho_1]\right]\uparrow$$

with $(\mathrm{R}') = (\text{RES})$ and

$$e_3 = [\iota_1; o_1, X \mapsto e; \rho_1] ,$$

we can conclude by applying (R') as a vertical step and no horizontal step. Note that the side-condition of (R') is satisfied as, from rule (R') applied to e_1, we have $\mathsf{FV}(e) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$, and from (R) applied to e_1, we have $(\mathsf{FV}(o_1) \cup \mathsf{FV}(\rho_1)) \cap \mathsf{dom}(\iota_2, \rho_2) = \varnothing$.

13. Critical pairs due to rules (M-ρ-LIFT) and (PRG-LIFT) are handled similarly to those due to rule (M-o-LIFT) (see previous points), except that now instead of the critical pair

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

due to rule (RUN), we have the following.
If

$$\begin{aligned} e_1 &= \left[\iota_1; o_1; x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow, \rho_1 \mid \mathscr{E}[x]\right] \\ e_2 &= \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1, x \mapsto e \mid \mathscr{E}[x]\right]\right]\uparrow \\ e_3 &= \left[\iota_1; o_1; x \mapsto \left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow, \rho_1 \mid \mathscr{E}\left\{\left[\iota_2; o_2; \rho_2 \mid e\right]\uparrow\right\}\right] \end{aligned}$$

with (R) = (MOD-PROP)+(M-ρ-LIFT) and (R′) = (PRG-SUBST), we can conclude by performing (R′) as a vertical step to give $e_4 = \left[\iota_2; o_2; \rho_2 \mid \left[\iota_1; o_1; \rho_1, x \mapsto e \mid \mathscr{E}\{e\}\right]\right]\uparrow$, and then applying Proposition 4.18 to e_3 and e_4.

(C) In all other cases the diagram commutes with one vertical step (rule (R′)) and one horizontal step (rule (R)).

We will now prove the 'moreover' part of the theorem. The result that the number of R-calculus and call-by-need steps in the right reduction is no greater than the number of R-calculus and call-by-need steps in the left reduction follows from the previous case analysis. While the fact that the number of R-calculus and call-by-need steps in the right reduction is strictly smaller if a descendant of a call-by-need step in the top enriched reduction is contracted in the left R-calculus reduction is proved by induction on the number of steps in the sequence. □

Theorem 4.20. Given $e \in \mathsf{Exp}$, if $e \longrightarrow^* e_1$, there exist e_2 and e_3 such that we have $e \xrightarrow[\mathsf{need}]{} {}^* e_2 \xrightarrow[R\setminus\mathsf{need}]{} {}^* e_3$ and $e_1 \preceq_\omega e_3$.

Proof. We use induction on the number of call-by-need steps in $e \longrightarrow^* e_1$.

If $e \xrightarrow{n} e_1$, that is, the reduction contains n call-by-need steps, we first apply call-by-need steps that are not R-calculus steps (if any) to e to give $e \xrightarrow[\mathsf{need}]{} {}^* e'$. By Lemma 4.19, there exists e_1' such that $e' \xrightarrow{\leqslant n} e_1'$ with $e_1 \preceq_\omega e_1'$.

We now distinguish between two possible cases for the reduction sequence $e' \xrightarrow{\leqslant n} e_1'$:

(1) If it contains no call-by-need steps, we can conclude by taking $e_2 \equiv e'$ and $e_3 \equiv e_1'$.

(2) If some call-by-need step are carried out in $e' \longrightarrow^* e_1'$, the sequence has the form $e' \xrightarrow[R\setminus\mathsf{need}]{} {}^* e_{11}' \xrightarrow[\mathsf{need}]{(R)} e_{12}' \longrightarrow^* e_1'$, where rule (R) contracts the first needed redex. In this case, this redex must be the descendant of a needed redex in e', so we have $e' \xrightarrow[\mathsf{need}]{(R)} e''$. Then, by applying Lemma 4.19, we have that there exists e_1'' such that $e'' \xrightarrow{<n} e_1''$ with $e_1' \preceq_\omega e_1''$. We can now apply the induction hypothesis to this reduction sequence, to get that there exist e_2 and e_3 such that $e'' \xrightarrow[\mathsf{need}]{} {}^* e_2'' \xrightarrow[R\setminus\mathsf{need}]{} {}^* e_3''$ such that $e_1'' \preceq_\omega e_3''$. □

4.5. *Confluence of the enriched reduction up to information content*

Theorem 4.21. Given the ARS with ordered information content

$$((\mathsf{Exp}, \bullet\!\!\longrightarrow, \preceq_\omega), \omega, (\omega(\mathsf{Exp}), \leqslant_\Omega)),$$

we have that $\longrightarrow$ commutes with $\bullet\!\!\longrightarrow$ up to $\preceq_\omega$.

Proof. By Lemma 4.19 we have that an instance of the left diagram in Lemma 4.6 holds. □

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Theorem 4.22. Given the ordered ARS $(\mathsf{Exp}, \bullet\!\!\longrightarrow, \preceq_\omega)$, we have that $\bullet\!\!\longrightarrow$ is confluent up to $\preceq_\omega$.

Proof. The thesis follows from Lemma 4.5 and Theorems 4.14 and 4.21. □

The above result shows that it is possible to solve in a more abstract way the well-known problem (see the example below) that confluence is broken in module calculi if we allow variable substitution in local definitions (rule $_{(\text{M-}\rho\text{-SUBST})}$). This problem is usually solved by imposing restrictions, such as disallowing substitution for mutually recursive local definitions (Ariola and Klop 1997; Wells and Vestergaard 2000).

Example 4.23. The basic module $[;;x \mapsto [;;w \mapsto y], y \mapsto [;;w \mapsto x]]$, by performing a $_{(\text{M-}\rho\text{-SUBST})}$ step, can reduce to either $[;;x \mapsto [;;w \mapsto [;;w \mapsto x]], y \mapsto [;;w \mapsto x]]$ or $[;;x \mapsto [;;w \mapsto y], y \mapsto [;;w \mapsto [;;w \mapsto y]]]$, which both infinitely reduce, never commuting. Nevertheless, at each step we can observe that the two terms have the same information content, which is $[;;x \mapsto \Omega, y \mapsto \Omega]$.

5. Conclusion

In this section, we first summarise the contribution of the paper, discussing the role of the *R*-calculus among other basic computational models. We then outline some work towards using (formalisms based on) the *R*-calculus to model reconfiguration features in real systems. Finally, we briefly review related work and give pointers to further research directions.

5.1. *Summary and discussion*

In this paper, we have presented the *R*-calculus, which is a calculus where standard execution (that is, execution of a single computation described by a fragment of code) can be interleaved with operations at the meta-level, which can manipulate the context in which this computation takes place in various ways. We have proposed different ways to prevent errors due to a lack of some required component, either by a purely static type system or by a combination of static and run-time checks. Moreover, we have defined a call-by-need strategy that performs module simplification only when needed and only once, and have proved its soundness and completeness. Finally, we have proved that the calculus is confluent in the usual sense if we do not include local substitution rules, and that even if we add these rules, a relaxed version of confluence holds, which is called *up to information content*. Intuitively, this captures the fact that the calculus is essentially deterministic.

We believe the key contribution of the paper is the development of a computational model that integrates in a simple way two different families of calculi: on the one hand, module calculi (Ancona and Zucca 1999; Wells and Vestergaard 2000; Machkasova and Turbak 2000; Ancona and Zucca 2002; Hirschowitz and Leroy 2002; Hirschowitz *et al.* 2004; Hirschowitz 2004; Makholm and Wells 2005), where the emphasis is on modelling operations for composing software fragments; and, on the other hand, let-rec

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

lambda calculi (Ariola and Felleisen 1997; Ariola and Klop 1997; Ariola and Blom 1997; Ariola and Blom 2002), where the emphasis is on modelling transformations of recursive definitions of functions. We will now discuss how the R-calculus is related to these two families of calculi in greater detail.

With respect to the literature on module calculi, the most important new feature of the R-calculus is that the operator for selecting a component from a module has been decomposed into two more primitive operators. More precisely, both, say, *CMS* (Ancona and Zucca 1999; Ancona and Zucca 2002) and the R-calculus rest on a notion of *open* modules (that is, incomplete, partially linked modules). Modules interact with each other through *names* X, Y, Z, while internal links are represented by *variables* x, y, z. So one may write things like

$[x \mapsto Z$; // import Z as x
$Y \mapsto x + 1$; // export $x + 1$ as Y
$z \mapsto x + 2, z' \mapsto z]$ // define (locally) z and z'.

Modules are thus made of three sections: input, output and local.

Given a module m, one may trigger the execution of one of its output components, say X, by writing $m.X$ in *CMS* and $m{\downarrow_X}$ in the R-calculus. However, while selection $m.X$ in *CMS* reduces just to the definition of X, where references to local variables, if any, are resolved by unfolding (hence selection is possible only on *closed* modules.), $m{\downarrow_X}$ in the R-calculus reduces to a compound of the whole of m and a (now active) copy of the definition of X. We write $[m \mid m.X]$ as an informal notation for such a compound. In this way, m still exists during the computation of X, and thus remains available for dynamic reconfiguration.

If one really wants to forget about m and only consider the result of the execution of X, one writes $m{\downarrow_X}{\uparrow}$. If, for instance, X is in turn a module, then this allows us to pass reconfiguration instructions to X without interfering with m.

The result is that the combination of $_{\downarrow_X}$ and $_{\uparrow}$ acts pretty much as a let-rec, since, roughly, for any reconfiguration instruction $\mathscr{R}$, one has that $\mathscr{R}([m \mid m.X]{\uparrow})$ reduces to $[m \mid \mathscr{R}[m.X]]{\uparrow}$, which is analogous to the rule $\mathscr{R}[\texttt{let}\ \mathit{defs}\ \texttt{in}\ e] \longrightarrow \texttt{let}\ \mathit{defs}\ \texttt{in}\ \mathscr{R}[e]$, which is usual in let-rec calculi.

As seen in Section 2.7, the combination of $_{\downarrow_X}$ and $_{\uparrow}$ also behaves like component selection in *CMS*, but is more general, since it can also be applied to open modules.

In summary, the R-calculus extends module calculi by adding the capability to select and evaluate components of *open modules*, leading to terms (which we called configurations) that, intuitively, model a program running in a context of software fragments, which can be modified in parallel with the program execution. Another important new technical result with respect to the literature on module calculi is the application of the approach based on information content introduced in Ariola and Blom (1997) and Ariola and Blom (2002), which allows us to solve in a more abstract way the well-known problem that confluence is broken in module calculi if we allow variable substitution in local definitions.

On the other hand, the R-calculus adds a new orthogonal feature to let-rec calculi, where basic terms are essentially programs running in ambients of mutually recursive definitions,

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

by equipping these terms with an interface to the outside using input and output names. In other words, the basic idea is to add primitives for manipulating ambients at a different (meta)level, and to investigate the interaction between the two levels. Moreover, the R-calculus differs from let-rec calculi in that it exploits a different computational paradigm in which the basic computational step is not β-reduction (function application), but module simplification. Indeed, it is well known that record-based calculi can provide an alternative computational paradigm in which λ-calculus can be encoded (Abadi and Cardelli 1996; Ancona and Zucca 2002). In this paper we have explored some important issues (typing problems, confluence, call-by-need strategy) in this alternative paradigm. The advantages offered by the record-based paradigm are a natural syntactic representation of a scope (a record, or basic module in the terminology of this paper) and a built-in mechanism for rebinding (by deleting and then adding a record component) without any need to introduce imperative features at the core level.

5.2. *Reconfiguration features in real systems*

The work presented in this paper belongs to a stream of research on foundations for systems supporting reconfiguration features (see the next subsection). The aim here was to focus on foundational aspects by designing a pure calculus and studying its metatheory; in this respect, the key contribution of the paper is, as discussed in the previous subsection, a new computational model that includes both module calculi and let-rec calculi.

Of course, much remains to be done if we are to use (a formalism based on) the R-calculus to model the features of real open systems; we will now briefly discuss some design alternatives that are significant in this respect, some of which we have partly explored in related papers.

Update of deprecated components. As shown in Ancona and Zucca (2002) for *CMS*, for example, overriding (the update of deprecated components) can be easily encoded in module calculi, and hence in the R-calculus as well. However, in the R-calculus as presented in this paper, the running program can only use hard-linked components, so the update of deprecated components is significant for subsequent module simplification but not for the execution of a module component. Indeed, the program in a configuration can use a component (by replacing a variable by its definition) only if it is *local*, and a component that is or has become local can no longer be changed by reconfiguration steps.

An alternative semantics is to enable the program running in a configuration to use *virtual* components as well, that is, components that are both input and output. For instance, the configuration $[x \mapsto X\,;\, X \mapsto 0;\, |\, x]$ could reduce to $[x \mapsto X\,;\, X \mapsto 0;\, |\, 0]$. This alternative has been studied in Ancona *et al.* (2005b) and Fagorzi (2005, Chapter 4) . Formally, this is achieved by introducing a new reduction rule (PRG-SUBST-VIRTUAL), shown in Figure 23, that is analogous to (PRG-SUBST) but allows the substitution of definitions of virtual components. Moreover, taking this approach, it becomes more convenient to adopt a syntax where virtual components are explicitly distinguished and, since the

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$e \in \mathsf{Exp}$	$::=$		**expression**
	$\mid$	x	variable
	$\mid$	$[\iota; v; f; \rho]$	basic module
	$\mid$	$\left[\iota; v; f; \rho \mid e\right]$	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	$::=$	$x_i \overset{i\in I}{\mapsto} X_i$	**input** assignment
v	$::=$	$x_i \overset{i\in I}{\mapsto} X_i \overset{i\in I}{\mapsto} e_i$	**virtual** assignment
f	$::=$	$X_i \overset{i\in I}{\mapsto} e_i$	**frozen** assignment
ρ	$::=$	$x_i \overset{i\in I}{\mapsto} e_i$	**local** assignment

Fig. 22. R^v-calculus: syntax

$$\text{(PRG-SUBST-VIRTUAL)}\ \frac{}{\left[\iota; v; f; \rho \mid \mathscr{E}[x]\right] \longrightarrow \left[\iota; v; f; \rho \mid \mathscr{E}\{e\}\right]}\ \begin{array}{l} x \notin \mathsf{HB}(\mathscr{E}) \\ x \mapsto X \mapsto e \in v \end{array}$$

$$\text{(M-DELETE-}v\text{)}\ \frac{}{[\iota; v, x \mapsto X \mapsto e; f; \rho]\backslash X \longrightarrow [\iota, x \mapsto X; v; f; \rho]}$$

$$\text{(M-DELETE-}f\text{)}\ \frac{}{[\iota; v; f, X \mapsto e; \rho]\backslash X \longrightarrow [\iota; v; f; \rho]}$$

$$\text{(M-FREEZE-}v\text{)}\ \frac{}{\mathsf{freeze}_X[\iota; v, x \mapsto X \mapsto e; f; \rho] \longrightarrow [\iota; v; f, X \mapsto e; \rho, x \mapsto e]}$$

$$\text{(M-FREEZE-}f\text{)}\ \frac{}{\mathsf{freeze}_X[\iota; v; f; \rho] \longrightarrow [\iota; v; f; \rho]}\ X \in \mathsf{dom}(f)$$

Fig. 23. R^v-calculus: reduction rules (excerpt)

association between the input and output part of a virtual component is by name[†], name-based versions of reduct and freeze operators. In Ancona *et al.* (2005b) and Fagorzi (2005, Chapter 4), for instance, we considered a name-based *delete* operator (that deletes a definition, making a virtual component deferred and removing a frozen component) and a *freeze* operator that makes a virtual component *frozen* (that is, only output).

The syntax of the resulting new calculus, called the R^v-calculus, is shown in Figure 22, and its reduction rules are shown in Figure 23. Roughly speaking, a basic module $[\iota; v; f; \rho]$ in the R^v-calculus corresponds to a basic module in the R-calculus where

[†] Note that there is no relation between an input and an output component with the same name in the R-calculus presented in this paper.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

the o-assignment has been split into two assignments v and f, which corresponds to virtual and frozen components, respectively, and analogously for basic configurations. More precisely, while f has the shape of an o-assignment (that is, a map from component names into expressions), v has the shape $x_i \overset{i\in I}{\mapsto} X_i \overset{i\in I}{\mapsto} e_i$, where x_i is the variable used for referring to the virtual component X_i inside the module; the notation is an abbreviation for $x_i \mapsto X_i, X_i \mapsto e_i$, which suggests that virtual components are both imported and exported. Basic modules and basic configurations must satisfy well-formedness conditions inherited from the R-calculus in the obvious way: that is, $\mathsf{dom}(\iota) \cap \mathsf{dom}(v) = \mathsf{dom}(v) \cap \mathsf{dom}(\rho) = \mathsf{dom}(\rho) \cap \mathsf{dom}(\iota) = \varnothing$; moreover, in a virtual assignment $x_i \overset{i\in I}{\mapsto} X_i \overset{i\in I}{\mapsto} e_i$, names X_i must all be distinct.

With virtual components, program execution can use two different definitions for a component if some reconfiguration step takes place in the meanwhile. This is no longer the case when components are frozen. The difference is illustrated by the following example.

Example 5.1. Given the configuration

$$\left(\mathsf{freeze}_X \left(\left[z \mapsto Z\,;\, x \mapsto X \mapsto 1, y \mapsto Y \mapsto 2;\, ;\, \mid x + y + z + x + y\right]\right) \setminus X \setminus Y\right) \\ + [\,;\, ;\, X \mapsto 3, Y \mapsto 4, Z \mapsto 0;\,]$$

and assuming we replace program variables from left to right, after a few steps we get

$$\left[\,;\, y \mapsto Y \mapsto 4, z \mapsto Z \mapsto 0;\, X \mapsto 3;\, x \mapsto 1 \mid 1 + 2 + 0 + 1 + 4\right].$$

Of course, having added rule (PRG-SUBST-VIRTUAL), the calculus is no longer confluent since, as shown by the example above, which values are substituted for variables depends on the interleaving between program execution and reconfiguration steps, so a reduction strategy must be chosen in order to have a deterministic semantics. In Ancona *et al.* (2005b) and Fagorzi (2005, Chapter 4) we presented a *lazy* strategy (that is, a strategy that performs reconfiguration steps only when execution would otherwise get stuck); we could also easily lift a call-by-need strategy like that presented in this paper (which is lazy and, moreover, ensures that no reconfiguration step is duplicated) to the calculus with explicit virtual components.

Moreover, analogously to what has been done in Ancona and Zucca (2002), where the semantics of *DCMS* (a calculus with explicit virtual components) was given by translation into *CMS*, it is possible encode the R^v-calculus in the R-calculus. The translation is type-directed, hence we first give, in Figure 24, types, well-formed types and typing rules for the R^v-calculus. A module type is a triple $\left[\pi^\iota\,;\, \pi^v\,;\, \pi^f\right]$ where π^v and π^f represent the virtual and frozen components, respectively. The rule for well-formedness of types changes to take into account the fact that there exists a unique name space for module components. Configuration types change analogously.

The translation of the R^v-calculus into the R-calculus is given in Figure 25. The notation $\Gamma \vdash \ll e \gg = e'$ means that the expression e in Γ (which is required to get types of variables) translates into the expression e'.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$
\begin{array}{lll}
\tau \in \mathsf{Type} & ::= & \textbf{type} \\
& | & \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right] \quad \text{module type} \\
& | & \left[\pi^{\iota};\, \pi^{v};\, \pi^{f};\, \tau\right] \quad \text{configuration type}
\end{array}
$$

$$
\frac{\vdash \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right] \quad \vdash \tau}{\vdash \left[\pi^{\iota};\, \pi^{v};\, \pi^{f};\, \tau\right]}
\qquad
\frac{\vdash \pi^{\iota} \quad \vdash \pi^{v} \quad \vdash \pi^{f}}{\vdash \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right]}
\begin{array}{l}
\mathsf{dom}(\pi^{\iota}) \cap \mathsf{dom}(\pi^{v}) = \varnothing \\
\mathsf{dom}(\pi^{v}) \cap \mathsf{dom}(\pi^{f}) = \varnothing \\
\mathsf{dom}(\pi^{f}) \cap \mathsf{dom}(\pi^{\iota}) = \varnothing
\end{array}
$$

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

$$
\text{(M-BASIC)}\ \frac{\begin{array}{l}\vdash \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right] \\ \left\{\Gamma, x_p : \tau_p{}^{p \in I \cup H \cup K} \vdash e_q : \tau_q \mid q \in H \cup J \cup K\right\}\end{array}}{\Gamma \vdash \left[x_i \overset{i \in I}{\mapsto} X_i;\, x_h \overset{h \in H}{\mapsto} X_h \overset{h \in H}{\mapsto} e_h;\, X_j \overset{j \in J}{\mapsto} e_j;\, x_k \overset{k \in K}{\mapsto} e_k\right] : \left[X_i : \tau_i{}^{i \in I};\, X_h : \tau_h{}^{h \in H};\, X_j : \tau_j{}^{j \in J}\right]}
$$

$$
\text{(M-SUM)}\ \frac{\Gamma \vdash e_1 : \left[\pi_1^{\iota};\, \pi_1^{v};\, \pi_1^{f}\right] \qquad \Gamma \vdash e_2 : \left[\pi_2^{\iota};\, \pi_2^{v};\, \pi_2^{f}\right]}{\Gamma \vdash e_1 + e_2 : \left[\pi^{\iota} \setminus (\pi_1^{v}, \pi_1^{f}, \pi_2^{v}, \pi_2^{f});\, \pi_1^{v}, \pi_2^{v}, \pi^{v};\, \pi_1^{f}, \pi_2^{f}\right]}
\begin{array}{l}
\pi_1^{\iota} \text{ and } \pi_2^{\iota} \text{ compatible} \\
\mathsf{dom}(\pi_1^{v}, \pi_1^{f}) \cap \mathsf{dom}(\pi_2^{v}, \pi_2^{f}) = \varnothing \\
\pi^{v} = (\pi_1^{\iota} \cap \pi_2^{f}), (\pi_2^{\iota} \cap \pi_1^{f})
\end{array}
$$

$$
\text{(M-DELETE-}v\text{)}\ \frac{\Gamma \vdash e : \left[\pi^{\iota};\, \pi^{v}, X : \tau;\, \pi^{f}\right]}{\Gamma \vdash e \backslash X : \left[\pi^{\iota}, X : \tau;\, \pi^{v};\, \pi^{f}\right]}
$$

$$
\text{(M-DELETE-}f\text{)}\ \frac{\Gamma \vdash e : \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}, X : \tau\right]}{\Gamma \vdash e \backslash X : \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right]}
$$

$$
\text{(M-FREEZE)}\ \frac{\Gamma \vdash e : \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right]}{\Gamma \vdash \mathsf{freeze}_X e : \left[\pi^{\iota};\, \pi^{v} \setminus X;\, \pi^{f}\right]}\ X \in \mathsf{dom}(\pi^{v}, \pi^{f})
$$

$$
\text{(RUN)}\ \frac{\Gamma \vdash e : \left[\pi^{\iota};\, \pi^{v};\, \pi^{f}\right]}{\Gamma \vdash e{\downarrow_X} : \left[\pi^{\iota};\, \pi^{v};\, \pi^{f};\, (\pi^{v}, \pi^{f})(X)\right]}
$$

$$
\text{(RES)}\ \frac{\Gamma \vdash e : \left[\varnothing;\, \pi^{v};\, \pi^{f};\, \tau\right]}{\Gamma \vdash e{\uparrow} : \tau}
$$

Fig. 24. R^{v}-calculus: types, well-formed types and typing rules (excerpt)

In order to preserve the original non-deterministic semantics of the R^{v}-calculus, the R-calculus must be extended with (an analogue of) the (PRG-SUBST-VIRTUAL) rule. Otherwise, the translation only preserves reduction sequences where this rule is never applied; if the type system restricts configuration terms to those without virtual components, the translation guarantees at least that every term of the R^{v}-calculus that has a semantics (that is, does not get stuck) has a semantics in the translation as well (intuitively, a strategy where execution of *freeze* operators has precedence over the rule (PRG-SUBST-VIRTUAL) is chosen). The more restrictive type system is obtained simply by changing rule (RES) to

$$
\text{(RES)}\ \frac{\Gamma \vdash e : \left[\varnothing;\, \varnothing;\, \pi^{f};\, \tau\right]}{\Gamma \vdash e{\uparrow} : \tau}
$$

and considering at the top level configuration types with no virtual components.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

$$\text{(VAR)}\ \frac{}{\Gamma \vdash \ll x \gg= x}$$

$$\text{(M-BASIC)}\ \frac{\begin{array}{c}\{\Gamma, x_p : \tau_p{}^{p\in I\cup H\cup K} \vdash e_q : \tau_q \mid q \in H \cup J \cup K\} \\ \{\Gamma, x_p : \tau_p{}^{p\in I\cup H\cup K} \vdash \ll e_q \gg= e'_q \mid q \in H \cup J \cup K\}\end{array}}{\begin{array}{l}\Gamma \vdash \ll \left[x_i \overset{i\in I}{\mapsto} X_i;\ x_h \overset{h\in H}{\mapsto} X_h \overset{h\in H}{\mapsto} e_h;\ X_j \overset{j\in J}{\mapsto} e_j;\ x_k \overset{k\in K}{\mapsto} e_k\right] \gg= \\ \qquad\qquad \left[\iota, x_h \overset{h\in H}{\mapsto} X_h;\ X_h \overset{h\in H}{\mapsto} e'_h, X_j \overset{j\in J}{\mapsto} e'_j;\ x_k \overset{k\in K}{\mapsto} e'_k\right]\end{array}}$$

$$\text{(M-SUM)}\ \frac{\Gamma \vdash \ll e_1 \gg= e'_1 \qquad \Gamma \vdash \ll e_2 \gg= e'_2}{\Gamma \vdash \ll e_1 + e_2 \gg= e'_1 + e'_2}$$

$$\text{(M-DELETE)}\ \frac{\Gamma \vdash \ll e \gg= e'}{\Gamma \vdash \ll e \backslash X \gg= e' \backslash X}$$

$$\text{(M-FREEZE)}\ \frac{\Gamma \vdash e : \left[\pi^\iota; \pi^v; \pi^f\right] \qquad \Gamma \vdash \ll e \gg= e'}{\Gamma \vdash \ll \mathsf{freeze}_X e \gg= \mathsf{freeze}_\sigma e'}\ \begin{array}{l}\sigma : X : \tau \to X : \tau \\ \tau = (\pi^v, \pi^f)(X)\end{array}$$

$$\text{(RUN)}\ \frac{\Gamma \vdash \ll e \gg= e'}{\Gamma \vdash \ll e\downarrow_X \gg= e'\downarrow_X}$$

$$\text{(RES)}\ \frac{\Gamma \vdash \ll e \gg= e'}{\Gamma \vdash \ll e\uparrow \gg= e'\uparrow}$$

Fig. 25. Translation of the R^v-calculus into the R-calculus (excerpt)

Control of the program over reconfiguration. In the call-by-need strategy, execution of the program always has priority over the reconfiguration operations. Moreover, in the R-calculus the order in which reconfiguration steps should be executed is statically planned. An interesting area of further work is the investigation of mechanisms for allowing the program to have more control over reconfiguration steps: for instance, the forcing of a reconfiguration step (an administrator replacing a deprecated component with a newer version should be able to decide when the replacement occurs) or determining the execution of one reconfiguration operator rather than another.

The former ability can be partly recovered by using *control variables*. For instance, in Example 5.1 variable z can be seen as a control variable that, put in a program, forces the reconfiguration steps that update the definitions of X and Y (see Ancona *et al.* (2005b) and Fagorzi (2005, Chapter 4) for more comments and examples). Of course, a more user-friendly version of this mechanism should be expressed using higher-level language constructs to be encoded by control variables.

For the latter ability, a limited form of control between the order of reconfiguration operators has been considered in Ancona *et al.* (2005a), where we introduced a *low-priority* freeze that can be swapped with other module operators (roughly speaking, this operator allows us to choose between different components available for linking

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

and selecting the one that is actually needed for program execution, as happens, for example, in dynamic loading, as in Java.

The dynamic retrieval of components from the 'outside world'. Finally, a very important issue that is not addressed in this paper is the retrieval of components that are not statically available. In the *R*-calculus, the program irreversibly fails (that is, either gets stuck or raises an error in the version with dynamic checks) if a required component is missing. However, a web browser, instead of exiting, asks the user for missing plugins when needed. More generally, in the *R*-calculus the program cannot interact with the 'outside world', since execution is modelled by a mere reduction (that is, binary) relation. In order to model reconfiguration in an open context more appropriately, a better setting would be labelled transition systems in the style of process calculi.

The key new problem to be faced in this case is the fact that, since the fragments composing an application are not all available at compile-time, we cannot perform a global static analysis; however, we would still like to guarantee that execution will never crash. We have investigated this issue in Fagorzi and Zucca (2006a), Fagorzi and Zucca (2006b) and Fagorzi and Zucca (2006c). In Fagorzi and Zucca (2006a), we model the outside world in a very simple way by extending the *R*-calculus with a *receive* primitive whose semantics is given by a labelled reduction step that models the dynamic retrieval of code from the outside. In Fagorzi and Zucca (2006b) and Fagorzi and Zucca (2006c), we present a parametric process calculus with standard send/receive primitives that abstracts from the language used for writing mobile code (that is, we no longer take the *R*-calculus but an arbitrary calculus providing some ingredients). In both papers, we want to be able to statically check the internal consistency of local code by only specifying a required type for missing code. Then we perform dynamic checks that ensure that code received from the outside, which is assumed to travel with its type, can be successfully accepted, without requiring us to type-check the whole code again. The approach we follow is advocated, for instance, in Bettini *et al.* (2005) in the context of a coordination language for mobile processes that exchange object-oriented code.

5.3. *Related and further work*

As mentioned above, the work presented in this paper originates in a stream of research (Ancona *et al.* 2003; Ancona *et al.* 2004; Ancona *et al.* 2005a; Fagorzi and Zucca 2004; Ancona *et al.* 2005b) on foundations for systems supporting reconfiguration features in which the system structure can dynamically change after the execution of an application has started. We have already discussed some of this work in the previous section; here we briefly complete the overview.

In Ancona *et al.* (2003) we presented a first proposal of a model for dynamic linking. This model was rather different from the *R*-calculus, notably by being imperative; however, we first introduced there a dependency-based type syste and an operator (called *dynamic linking*), whose behaviour roughly corresponds to that of the *freeze* operator with a lazy (in the sense explained above) strategy.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

The model presented in Ancona *et al.* (2004) corresponds roughly to a subset of the *R*-calculus (in particular, it has no run or result operators, and hence no support for higher-order configurations) with a deterministic lazy strategy. Variants allowing more unpredictable reconfiguration features have been explored, as already summarised above, in Ancona *et al.* (2005a), where we introduced a low-priority freeze, and in Ancona *et al.* (2005b), where we introduced virtual components. In Fagorzi and Zucca (2004) we showed an application of the model in Ancona *et al.* (2005a) to encode a Java-like mechanism of multiple class loaders. Finally, we have already mentioned recent (subsequent to this paper) work (Fagorzi and Zucca 2006a; Fagorzi and Zucca 2006b; Fagorzi and Zucca 2006c), where we investigate the integration of the *R*-calculus with primitives for concurrency and code mobility.

The approach in this paper differs from all the above mentioned works in that we focus on designing a new computational model and investigating its metatheory. In particular, we introduce the *R*-calculus as *a pure calculus* (that is, where no strategy is considered), include lifting rules in the calculus (which are usually not considered in module calculi, but are significant here since they can augment the computational power), and show its internal consistency by proving confluence. Other key novelties compared with previous work are the comparison of different type systems and, in particular, the investigation of dynamic checks, the development of a strategy that is not only lazy but by-need, and the exploitation for a record-based calculus of the approach based on information content, leading to a proof of the completeness of the strategy, and the confluence of an enriched calculus with local substitution rules.

In addition to the literature on module and let-rec calculi, other work that has directly inspired this paper is that on dynamic software updating in, for example, Hicks *et al.* (2001), Bierman *et al.* (2003), Stoyle *et al.* (2005) and Hicks and Nettles (2005), where the basic primitive is an *update* primitive that when run changes some part of the local code.

On the more applicative side, although the area of unanticipated software evolution continues to attract considerable interest, with its foundations studied in, for example, Mens and Kniesel (2004), there has been little work on the development of abstract models for dynamic linking and updating. Apart from the wide literature concerning concrete dynamic linking mechanisms in existing programming environments (Drossopoulou 2001; Drossopoulou *et al.* 2003; Buckley *et al.* 2005; Lagorio 2006; Drossopoulou *et al.* 2006), we would like mention Abadi *et al.* (2004), where dynamic linking is studied as the programming language counterpart of the axiom of choice, and the module system defined in Liu and Smith (2004), where static linking, dynamic linking and cross-computation communication are all defined in a uniform framework.

There are at least two important directions for further work. First, we would like to investigate the application of the calculus to the modelling of dynamic binding, rebinding and marshalling mechanisms, as, for instance, in Moreau (1998) and Bierman *et al.* (2003), which presents a call-by-value λ-calculus that delays the instantiation of identifiers in such a way that computations can use the most recent versions of rebound definitions.

Moreover, an important issue, which, as mentioned above, we are currently working on, is the integration with mobility aspects, that is, the design of calculi for reconfiguration

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

where, roughly speaking, code to be used for reconfiguring the running program can migrate from a different process. This direction is related to the former when we consider the migration of code with free variables that can be dynamically rebound in the receiver.

Acknowledgments

We would like to offer our warm thanks to Davide Ancona for our previous work together, which was the basis for the present development, and Zena Ariola for her comments on Fagorzi (2005), which directly triggered many of the new ideas in this paper. Moreover, we thank the anonymous referees for their helpful comments, which greatly improved the presentation of the paper. In particular, comments from one referee led directly to some simplification in the design of the calculus and to the writing of Section 5.2.

References

Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*, Monographs in Computer Science, Springer.

Abadi, M., Gonthier, G. and Werner, B. (2004) Choice in dynamic linking. In: FOSSACS'04 – Foundations of Software Science and Computation Structures 2004. *Springer-Verlag Lecture Notes in Computer Science* **2987** 12–26.

Ancona, D. (1998) *Modular Formal Frameworks for Module Systems*, Ph.D. thesis, Dipartimento di Informatica, Università di Pisa.

Ancona, D., Fagorzi, S. and Zucca, E. (2003) A calculus for dynamic linking. In: Blundo, C. and Laneve, C. (eds.) Italian Conference on Theoretical Computer Science 2003. *Springer-Verlag Lecture Notes in Computer Science* **2841** 284–301.

Ancona, D., Fagorzi, S. and Zucca, E. (2004) A calculus with lazy module operators. In: Lévy, J.-J., Mayr, E. W. and Mitchell, J. C. (eds.) *TCS 2004 (IFIP International Conference on Theoretical Computer Science)*, Kluwer Academic Publishers 423–436.

Ancona, D., Fagorzi, S. and Zucca, E. (2005a) A calculus for dynamic reconfiguration with low priority linking. In: WOOD'04 – Workshop on Object-Oriented Development. *Electronic Notes in Theoretical Computer Science* **138** (2) 3–35.

Ancona, D., Fagorzi, S. and Zucca, E. (2005b) Mixin modules for dynamic rebinding. In: De Nicola, R. and Sangiorgi, D. (eds.) TGC 2005 – Trustworthy Global Computing. *Springer-Verlag Lecture Notes in Computer Science* **3705** 279–298.

Ancona, D. and Zucca, E. (1998) A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science* **8** (4) 401–446.

Ancona, D. and Zucca, E. (1999) A primitive calculus for module systems. In: Nadathur, G. (ed.) PPDP'99 – Principles and Practice of Declarative Programming. *Springer-Verlag Lecture Notes in Computer Science* **1702** 62–79.

Ancona, D. and Zucca, E. (2002) A calculus of module systems. *Journal of Functional Programming* **12** (2) 91–132.

Ariola, Z. M. and Blom, S. (1997) Cyclic lambda calculi. In: Abadi, M. and Ito, T. (eds.) Theoretical Aspects of Computer Software. *Springer-Verlag Lecture Notes in Computer Science* **1281** 77–106.

Ariola, Z. M. and Blom, S. (2002) Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic* **117** (1-3) 95–168.

Ariola, Z. M. and Felleisen, M. (1997) The call-by-need lambda calculus. *Journal of Functional Programming* **7** (3) 265–301.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Ariola, Z. M. and Klop, J. W. (1997) Lambda calculus with explicit recursion. *Information and Computation* **139** (2) 154–233.

Bettini, L., Venneri, B. and Bono, V. (2005) MOMI: a calculus for mobile mixins. *Acta Informatica* **42** (2-3) 143–190.

Bierman, G., Hicks, M. W., Sewell, P., Stoyle, G. and Wansbrough, K. (2003) Dynamic rebinding for marshalling and update, with destruct-time λ. In Runciman, C. and Shivers, O. (eds.) *International Conference on Functional Programming 2003*, ACM Press 99–110.

Buckley, A., Murray, M., Eisenbach, S. and Drossopoulou, S. (2005) Flexible bytecode for linking in .NET. *Electronic Notes in Theoretical Computer Science* **141** (1) 75–92.

Drossopoulou, S. (2001) Towards an abstract model of Java dynamic linking and verification. In: Harper, R. (ed.) TIC'00 – Third Workshop on Types in Compilation (Selected Papers). *Springer-Verlag Lecture Notes in Computer Science* **2071** 53–84.

Drossopoulou, S., Lagorio, G. and Eisenbach, S. (2003) Flexible models for dynamic linking. In: Degano, P. (ed.) ESOP 2003 – European Symposium on Programming 2003. *Springer-Verlag Lecture Notes in Computer Science* **2618** 38–53.

Drossopoulou, S., Lagorio, G. and Eisenbach, S. (2006) A flexible model for dynamic linking in Java and C#. *Theoretical Computer Science* (to appear).

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

Fagorzi, S. and Zucca, E. (2004) A case-study in encoding configuration languages: Multiple class loaders. *Journal of Object Technology* **3** (11) 31–53.

Fagorzi, S. and Zucca, E. (2005) A calculus for reconfiguration. In: DCM 2005 – International Workshop on Developments in Computational Models. *Electronic Notes in Theoretical Computer Science* **135** (3).

Fagorzi, S. and Zucca, E. (2006a) A calculus of components with dynamic type-checking. In: Formal Aspects of Component Software (FACS'06). *Electronic Notes in Theoretical Computer Science* (to appear).

Fagorzi, S. and Zucca, E. (2006b) A framework for type safe exchange of mobile code. In: TGC 2006 – Trustworthy Global Computing. *Springer-Verlag Lecture Notes in Computer Science* (to appear).

Fagorzi, S. and Zucca, E. (2006c) A framework for type safe exchange of mobile code. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova. (Extended version of Fagorzi and Zucca (2006b)).

Felleisen, M. and Friedman, D. P. (1986) Control operators, the SECD-machine, and the lambda calculus. In: 3rd Working Conference on the Formal Description of Programming Concepts, Ebberup, Denmark 193–219.

Hicks, M. W., Moore, J. and Nettles, S. (2001) Dynamic software updating. In: *PLDI'01 – ACM Conference on Programming Language Design and Implementation*, ACM Press.

Hicks, M. W. and Nettles, S. (2005) Dynamic software updating. *ACM Transactions on Programming Languages and Systems* **27** (6) 1049–1096.

Hindley, J. R. (1964) *The Church–Rosser Property and a Result in Combinatory Logic*, Ph.D. thesis, University of Newcastle-upon-Tyne.

Hirschowitz, T. (2004) Rigid mixin modules. In: Kameyama, Y. and Stuckey, P. J. (eds.) FLOPS 2004 – Functional and Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **2998** 214–228.

Hirschowitz, T. and Leroy, X. (2002) Mixin modules in a call-by-value setting. In: Le Métayer, D. (ed.) ESOP 2002 – European Symposium on Programming 2002. *Springer-Verlag Lecture Notes in Computer Science* **2305** 6–20.

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press

Hirschowitz, T., Leroy, X. and Wells, J. B. (2004) Call-by-value mixin modules: Reduction semantics, side effects, types. In: Schmidt, D. A. (ed.) ESOP 2003 – European Symposium on Programming 2003. *Springer-Verlag Lecture Notes in Computer Science* **2986** 64–78.

Klop, J. (1987) Term rewriting systems: a tutorial. *Bulletin of EATCS* **32** 143–182.

Klop, J., van Oostrom, V. and van Raamsdonk, F. (1993) Combinatory reduction systems: introduction and survey. Technical Report CS-R9362, CWI.

Lagorio, G. (2006) Dynamic linking of polymorphic bytecode. In: 8th International Workshop on Formal Techniques for Java-like Programs.

Liu, Y. D. and Smith, S. F. (2004) Modules with interfaces for dynamic linking and communication. In: Odersky, M. (ed.) ECOOP'04 – Object-Oriented Programming. *Springer-Verlag Lecture Notes in Computer Science* **3086** 414–439.

Machkasova, E. and Turbak, F. (2000) A calculus for link-time compilation. In: ESOP 2000 – European Symposium on Programming 2000. *Springer-Verlag Lecture Notes in Computer Science* **1782** 260–274.

Makholm, H. and Wells, J. B. (2005) Type inference, principal typings, and let-polymorphism for first-class mixin modules. In: Danvy, O. and Pierce, B. C. (eds.) *International Conference on Functional Programming 2005*, ACM Press 156–167.

Mens, T. and Kniesel, G. (2004) Workshop on foundations of unanticipated software evolution. ETAPS 2004 (available at http://joint.org/fuse2004/).

Moreau, L. (1998) A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation* **11** (3) 233–279.

Stoyle, G., Hicks, M. W., Bierman, G., Sewell, P. and Neamtiu, I. (2005) Mutatis mutandis: safe and predictable dynamic software updating. In: *ACM Symposium on Principles of Programming Languages 2005*, ACM Press 183–194.

van Oostrom, V. (1994) *Confluence for Abstract and Higher-Order Rewriting*, Ph.D. thesis, Vrije Universiteit, Amsterdam.

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

https://doi.org/10.1017/S0960129507006238 Published online by Cambridge University Press