A calculus of lambda calculus contexts

Mirna Bognar and Roel de Vrijer (mirna@cs.vu.nl,rdv@cs.vu.nl)
Vrije Universiteit, Amsterdam

Abstract. The calculus λc serves as a general framework for representing contexts. Essential features are control over variable capturing and the freedom to manipulate contexts before or after hole filling, by a mechanism of delayed substitution. The context calculus λc is given in the form of an extension of the lambda calculus. Many notions of context can be represented within the framework; a particular variation can be obtained by the choice of a *pretyping*, which we illustrate by three examples.

1. Introduction

The central notion in this paper is that of *context*, i.e. an expression with special places, called *holes*, where other expressions can be placed. For example, in the lambda calculus, $(\lambda x.\Box)\,z$, where $\Box$ denotes a hole, is a context. In formal systems with bound variables, such as lambda calculus, a distinctive feature of placing an arbitrary expression into a hole of a context is *variable capturing*: some free variables of the expression may become bound by the binders of the context. For example, placing the expression $x\,z$ into the hole of the context above results in the expression $(\lambda x.x\,z)\,z$, where the free variable x of the expression has become bound by the binder λx of the context.

In many formal systems, the standard transformations, which are defined on expressions, are not defined on contexts. This implies that contexts are treated merely as a notation, which hinders any formal reasoning about contexts and the interaction with expressions put into their holes. Our objective is to add contexts as first-class objects, and to gain control over variable capturing and, more generally, 'communication' between a context and expressions to be put into its holes.

1.1. Motivation, applications and related work

The starting point of our research has been De Bruijn's calculus of segments, which was proposed in the context of the family of proof checkers Automath. From a broader perspective, the increasing interest in contexts has its motivation from many directions, as diverse as modeling programs and program environments, operational semantics and dealing with anaphora in natural language representation. In all these cases there is a need for manipulating contexts on the same level as expressions. The study and formalization of contexts has been the

subject of various papers; we list a few from the problem areas just mentioned.

In [6], N.G. de Bruijn introduced a lambda calculus extended with incomplete terms of a special form, called *segments*. The purpose of segments was facilitating definitions and manipulation of abbreviations in Automath (see [16]). Technically, segments can be characterized as contexts with precisely one hole at a special position. The segment calculus included means for representing segments, variables over segments and abstraction over segments. In [2, 3] H. Balsters gave a simply typed version of the segment calculus and proved confluence and subject reduction.

With the goal of optimalization of interactive proof checking, L. Magnusson (see [13]) presented an algorithm for incomplete proofs. The algorithm is designed for Martin-Löf's type theory with explicit substitutions and it is used in the proof editor ALF. The unfinished parts of a proof are denoted by placeholders, which are assigned a type and a local context. When filling in a new part of a proof into a placeholder, it is sufficient to check the new part. With the goal of representing incomplete proofs and supporting incremental proof development and higher-order unification, C. Muñoz presented in [15] a name-free explicit substitutions calculus with dependent types and with meta-variables over the missing parts of an expression. In both formalizations, filling of the missing parts of an expression is an operation, which 'commutes' with (a refined version of) β-reduction.

With the development of programming languages in mind, M. Hashimoto and A. Ohori (see [8]) proposed a typed context calculus, which is an extension of the simply typed lambda calculus. The type system specifies the variable-capturing nature of contexts with one hole using α-sensitive interface variables. The relations of β-reduction and hole-filling reduction are combined, under the restriction that no β-steps are allowed within a context. With the aim of building a theory of separate compilation and incremental programming, S.-D. Lee and D. Friedman (see [12]) designed a schema for enriching lambda calculus with contexts. They employ contexts for modeling program modules and their calculus for modeling module linking. In their calculus the binders in contexts are treated as identifiers whose binding scope is by compilation extended to objects filled into the holes. Computation is performed by β-reduction and additional compilation rules. M. Sato, T. Sakurai and R. Burstall (see [22]) defined a simply typed lambda calculus with first-class environments. The calculus is provided with operations for evaluating expressions within an environment and includes environments as function arguments.

For dealing with contexts in operational semantics, D. Sands (see [21]), contributing the idea to A. Pitts, proposed a representation of contexts with function variables for holes, meta-abstractions over variables to be captured for terms to be filled into the holes, and substitution of hole variables for hole filling. Using this representation, the operation of hole filling is freely combined with β-reduction. With the same motivation, I.A. Mason (see [14]) extended the syntax of lambda calculus with denotations for holes labeled by a substitution. He introduced two notions of variable replacement, weak and strong substitution, which differ in the behavior at hole labels. Weak substitution is used for α- and β-reduction, and for filling holes with terms. Strong substitution is used for filling holes with contexts. Hole filling is defined as an operation, which 'commutes' with β-reduction.

With the purpose of modeling binding mechanisms in natural language, M. Kohlhase, S. Kuschert and M. Müller (see [11]) introduced dynamic lambda calculus as an extension of the simply typed lambda calculus with declarations. In their approach the scope of binders sometimes extends the textual scope of a sentence. Declarations are α-sensitive and β-reduction is not defined on declarations. In addition to types, expressions are provided with modality, which describes their variable binding power.

Other contributions to the formalization of contexts have been made by C.L. Talcott [23] and S. Kahrs [9].

1.2. Our approach

Although emerging from different fields of research, with different motivations, the problem of formalizing contexts and communication can be tackled uniformly. The context calculus λc can serve as a uniform framework for representing different kinds of contexts. It is an extension of the lambda calculus with facilities for representing contexts and context-related operations such as filling the holes of a context by expressions (called 'hole filling') or by contexts (called 'composition'), and establishing the (explicit) 'communication'.

Communication is meant here in the broad sense of interaction between a context and the expressions that are put into its holes. In particular, we present a technique that allows us to control the passing of variable bindings. This regards not only potential capture of a variable by a binder in the context, but also passing on imminent substitutions that emerge from earlier β-reduction within the context. It is accomplished by giving both the context and the holes, as well as the expressions that are candidates to be filled in, a functional representation. There is an analogy to techniques used in higher-order rewriting

(see for example J.W. Klop [10], T. Nipkow [17], V. van Oostrom & F. van Raamsdonk [19]), and in the field of higher-order abstract syntax (see for example F. Pfenning & C. Elliott [20], J. Despeyroux, F. Pfenning & C. Schürmann [7]), where variable capturing is accomplished by a substitution calculus. Similar techniques are applied in the work of D. Sands which we already mentioned.

An important characteristic of our approach is that we also give the contexts a functional representation. This is accomplished by λ-abstracting the hole variables; a context is seen as a function of its holes. It is in this way that a *calculus* arises, where contexts can be freely manipulated on the object level and where hole filling is computed *within* the calculus, as opposed to hole filling as a meta-operation. Moreover, having contexts as objects is essential for having functions over contexts. Such a calculus is necessary for applications in proof checking (segment calculus) and in linguistics, as can be seen in the previously mentioned work of N.G. de Bruijn, H. Balsters and M. Kohlhase et al. A functional representation of contexts is also present in the work of M. Hashimoto and A. Ohori.

The power of our calculus is its expressivity, which is achieved by on the one hand a flexible syntax, and on the other hand the possibility of term-formation restrictions within the framework. The syntax allows for a first-class treatment of contexts by having explicit abstraction over context variables and free context manipulation. Term-formation restrictions are implemented by 'pretypings'. Via the choice of an adequate pretyping different notions of context can be represented within λc. Last but not least, the calculus can directly be translated to lambda calculus. Bearing this in mind, we perceive λc as a comfortable level of abstraction for dealing with contexts as first-class objects.

1.3. Outline

The paper is organized as follows. Section 2 introduces and analyzes first informal notions of context in the lambda calculus, and then an example of the use of contexts in proof checking. Section 3 explains the working of our context calculus. Section 4 defines the context calculus λc, the untyped version of the framework. Section 5 contains the proof that λc is confluent. Section 6 contains three examples of pretyping for different notions of context, including De Bruijn's segments. Section 7 summarizes this all and suggests some directions for future research.

2. Examples of contexts

In this section we first describe contexts as they are encountered in the lambda calculus literature, usually as an informal notational device. Then we give a simple example of the use of contexts in proof checking.

2.1. Informal notions of context in the λ-calculus

The standard theory of the untyped and simply typed lambda calculus with constants is presupposed; the interested reader is referred to [4].

A context over λ-terms, or a λ-context for short, is basically a λ-term with some holes in it. Contrary to λ-terms, λ-contexts are not considered modulo α-conversion (in the name-carrying representation), nor are λ-contexts subject to β-reduction. That means, for example, that $\lambda x.\Box \neq_\alpha \lambda y.\Box$ and $(\lambda x.x\,\Box)\,y \not\to_\beta y\,\Box$.

We distinguish two basic context-related operations, both concerned with filling holes. The first operation, called hole filling, deals with placing terms into the holes of a context. The second operation, called composition, deals with putting contexts into the holes of a context. For example, the composition of λ-contexts $\lambda x.\Box$ and $x(\lambda y.\Box)$ results in $\lambda x.x(\lambda y.\Box)$. In both operations, when a term or a context is placed into a hole of a context, variable capturing may occur. The difference between these operations is, in addition to the difference in the objects that are placed into the holes (terms vs. contexts), in the resulting object: a λ-term, in the case of hole filling; and a λ-context, in the case of composition. The resulting objects are denoted as $C[M]$ and $C[D]$ where C and D are contexts, M is a term and where $[\,]$ denotes the textual replacement of the hole(s) in C by M or D.

As a matter of fact, several variants of this simple view on contexts exist. The first possibility for variation is in the number of holes allowed in a context: precisely one (as for example in [10]), or many, including zero. The second possibility for variation is, in the case where many holes are allowed, in the way these holes are treated: as copies of the same hole, which therefore must be filled with the same term (as for example in [4]); as copies of different holes, which therefore may be filled with possibly different terms; or as combination of both treatments by distinguishing between holes and hole occurrences.

A formalization of λ-contexts should ideally provide means for representing contexts and context-related operations, as well as rules for computing these operations, and, moreover, should extend the standard rewrite relations to (the representations of) contexts. The major problem in a naïve formalization is that the standard rewrite relations do

not commute with the new context reductions. Confluence is lost and, consequently, the corresponding equational theory is inconsistent. The non-commutation of β-reduction and hole filling is demonstrated by the next example, where a representation for *hole filling*, *hf* (here seen as an operator), has been introduced, where hole filling is computed by *fill*-reduction and where β-reduction has naïvely been extended to contexts; a similar example of non-commutation can be given for α-conversion and hole filling.

Example 1. Let $C \equiv (\lambda x.x\Box)\,y$ and $M \equiv x$. Then

$$\begin{array}{lll} & hf(C,\ M) & \\ \equiv & hf((\lambda x.x\,\Box)\,y,\ x) & \qquad\qquad\qquad\quad hf\,(C,\ M) \\ \to_{fill} & (\lambda x.x\,x)\,y & \text{but} \qquad\quad \to_{\beta} \quad hf\,(y\,\Box,\ x) \\ \to_{\beta} & y\,y & \to_{fill} \quad y\,x \\ \equiv & N & \not\equiv \quad N. \end{array}$$

In the example, the reductions end in different terms because in the reduction on the left the substitution $[\![x := y]\!]$, which emerged from the rewrite step in context C is applied to the term M, while in the reduction on the right the substitution is not applied to the term M, but only to the hole, which 'forgets' it. Note that the result of the left reduction is the intended one.

What is needed is a way of denoting the intended bindings, which keeps track of α-conversion or β-reduction in the outer context and passes the effects of these reductions on to the terms (or contexts) replacing the holes. We call this interaction between holes and objects to be put into the holes *communication*. It is common to both hole filling and composition and it can be tackled separately. Then hole filling and composition reduce to replacing holes, without any communication.

Accordingly, the reduction on the right can be repaired by explicitly keeping track of these α, β-changes and applying the resulting substitution to the term after hole filling:

$$hf((\lambda x.x\Box)\,y,\ x) \to_{\beta}\ hf(y\Box^{[\![x:=y]\!]},\ x)\ \to_{fill}\ y\,(x[\![x := y]\!])\ =\ yy.$$

The problem of establishing communication is reduced to the encoding of this substitution.

2.2. An example from proof checking

What is still missing in the above account of λ-contexts is the use of variables for contexts, which then also can be abstracted. We illustrate this with an example from type theory, when used for the representation of mathematical concepts, in the realm of proof checking.

A context that is suitable for reasoning about reflexive relations, and hence in a way representing the notion of reflexive relation, could be the following:

$$\lambda A : Set.\lambda R : (A \to A \to Set).\lambda rfl : (\forall x : A.Rxx).\Box.$$

Let us call this context $\underline{refl}$. Then, an argument on reflexive relations, say a piece of mathematical text *text*, can be performed within this context, via hole filling:

$$hf\,(\underline{refl}, text).$$

In *text* the identifiers A, R, rfl can then be used. In a larger piece of text this can happen more than once. Say in a proof term

$$P(hf\,(\underline{refl}, text_1), hf\,(\underline{refl}, text_2), hf\,(\underline{refl}, text_3)).$$

An efficient representation, without the need to repeat $\underline{refl}$, could then be given as:

$$(\lambda c.P(hf\,(c, text_1), hf\,(c, text_2), hf\,(c, text_3)))\underline{refl}.$$

Once expressions of this kind are allowed, also explicit hole abstraction becomes very natural, if not unavoidable.

The use of the context $\underline{refl}$ that we indicated is also a typical example of a *segment*, according to N.G. de Bruijn [6]. A technical treatment of segments using our calculus λc will be given in Section 6.3.

3. An introduction to λc

The main aspects of the context calculus are sketched. A formal description of the calculus is given in Section 4.

Contexts. A context will be considered as a function over the possible contents of its holes. For this reason, in the context calculus hole variables $h, g, k, \ldots$ are introduced and contexts are represented as functions over (one or many) hole variables. The abstractor* for hole variables is denoted by δ_n, where $n \in I\!N$ is the number of variables which δ_n binds.

Communication. At first sight, it seems natural to use explicit substitutions (see for example [1]) for communication, by for example labeling holes with a substitution, viz. $\Box^\sigma$. This idea can be found e.g. in the work on contexts of L. Magnusson [13], C. Muñoz [15], I.A. Mason [14]

* The symbol δ is used also by M. Hashimoto and A. Ohori for abstracting hole variables, but only as a 'unary' abstractor.

and M. Sato et al. [22]). In the present paper it is our objective to reduce the whole matter of context manipulation to the very basic and well-understood notions of λ-abstraction and β-reduction. An explicit substitution calculus could then be used to eliminate β-reduction again, for example with the purpose of giving an efficient implementation. At this point we think it is profitable to separate the two issues.

So we take a basic, lambda-calculus-like approach, and solve the problem of encoding communication by using the fact that in lambda calculus substitution emerges as the result of a β-step: $M\,[\![x := N]\!] \leftarrow_\beta (\lambda x.M)N$. Since it is convenient to use multiple substitutions, we will introduce new constructors $\Lambda_n \_ . \_$ for multiple abstraction of n variables and $\_\langle \_, \ldots, \_ \rangle_n$ for multiple ($n+1$-ary) application, together with a multiple version ($\textcircled{m}\beta$) of the β-rule. This is illustrated by the following example.

Example 2. The reduction on the right in Example 1 of Section 2 now becomes, in a reverse order of the steps (the hole-filling constructor *hf* is still auxiliary, indices are implicit and h is a hole variable):

$$\begin{array}{lll} yy & = & y(x[\![x := y]\!]) \\ & \leftarrow_{\textcircled{m}\beta} & y((\Lambda x.x)\langle y\rangle) \\ & \leftarrow_{\mathit{fill}} & \mathit{hf}(y(h\langle y\rangle),\ \Lambda x.x) \\ & \leftarrow_{\beta} & \mathit{hf}((\lambda x.x(h\langle x\rangle))y,\ \Lambda x.x) \end{array}$$

where the last term shows the new representations of the hole[†] $(h\langle x\rangle)$ and of the *communicating term* $(\Lambda x.x)$.

In general, holes are represented as multiple applications of hole variables to a sequence of terms that keeps track of the relevant $\alpha\beta$-changes. Such a representation of holes resembles the representation of meta-variables in higher-order rewriting, where a similar binding effect is encoded. Communicating terms and, in the case of composition, communicating contexts are represented as multiple abstractions over variables that will become bound by the binders of the context where they will eventually be placed. When a communicating term is placed into the hole, communication can be computed by applying a generalized form of the β-rule

$$(\Lambda x_1, \ldots, x_n.U)\langle V_1, \ldots, V_n\rangle \ \rightarrow_{\textcircled{m}\beta}\ U[\![x_1 := V_1, \ldots, x_n := V_n]\!],$$

recovering the binding intention and passing the changes.

Hole filling and composition. Since we represent contexts as functions over holes (i.e. as abstractions over hole variables), hole filling simply boils down to (multiple) β-reduction. Thus, in our representation

[†] The notation $\langle\ \rangle$ in the representation of holes is also used by D. Sands.

of Example 1 we get $(\delta h.\lambda x.x(h\langle x\rangle)y)\lceil\Lambda x.x\rceil \to_{fill} \lambda x.x((\Lambda x.x)\langle x\rangle)y$, where $\_\lceil\_\rceil$ denotes a hole-filling constructor. In general, a context representation may be a function over many holes and consequently, hole filling may involve filling many holes simultaneously, viz.

$$(\delta_n h_1,\ldots,h_n.U)\lceil V_1,\ldots,V_n\rceil_n \to_{fill} U[\![h_1 := V_1,\ldots,h_n := V_n]\!]\,.$$

Here, the number of holes (i.e. the index of δ_n) equals the number of arguments between the brackets (i.e. the index of $\_\lceil\_,\ldots,\_\rceil_n$). Note that, in general, the arity of $\_\lceil\_,\ldots,\_\rceil_n$ is $n+1$.

Also composition may involve filling many holes simultaneously. This explains the need for composition operators $\circ_n$ for arbitrary n. However, the rewrite relation of composition is more complicated than in the case of hole filling: it includes some shifting of abstractions. This is explained by the following example of a binary composition.

Example 3. Let $C \equiv \lambda x.\Box$ and $D \equiv x(\lambda y.\Box)$ be two λ-contexts. Then the composition of the two results in the λ-context $\lambda x.x(\lambda y.\Box)$. Note that the hole of the result of the composition is the 'lifted' hole of the second λ-context, which potentially binds variables x as well as y.

In the context calculus, these λ-contexts are represented as $C_c \equiv \delta g.\lambda x.g\langle x\rangle$ and $D_c \equiv \delta h.x(\lambda y.h\langle y\rangle)$. Because the second context is going to be put into the hole of C_c, it is provided with means of communication: the preamble Λx and 'lifted' hole $h\langle x,y\rangle$ adapted for this purpose, viz. $D'_c \equiv \Lambda x.\delta h.x(\lambda y.h\langle x,y\rangle)$. The composition puts the second context into the hole, and moves the abstraction δh to the beginning of C_c, so that the whole becomes an abstraction over the 'lifted' hole h of D_c. The composition rewrite step should result in $C_c \circ D'_c \to_\circ \delta h.\lambda x.(\Lambda x.x(\lambda y.h\langle x,y\rangle))\langle x\rangle$, where $\circ$ is the composition constructor in λc. Note that by performing the ensuing communication step this term reduces to $\delta h.\lambda x.x(\lambda y.h\langle x,y\rangle)$, which is a representation of the resulting composition in lambda calculus.

The $\circ$-step of the example is an instance of the binary-composition rewrite rule ($\circ$):

$$(\delta g.U)\circ(\Lambda u_1,\ldots,u_n.\delta h.V) \to_\circ \delta h.U[\![g := \Lambda u_1,\ldots,u_n.V]\!]$$

where δh is shifted to the beginning of the reduct (after the variable h has been renamed if it occurs free in U). In the example a binary composition was used. In general, if a context representation is a function over n holes, the composition $\circ_n$ will involve $n+1$ contexts: one outer context and n contexts that are filled into the holes of the outer context. The resulting context is a context over the holes of the n

contexts; hence, the composition will shift the hole abstractions of all n contexts to the beginning of the reduct.

Framework. In the context calculus the building blocks can freely be combined to form λc-terms: variables, abstractions, applications and compositions. If a context contains many occurrences of a hole, these may be given the same name, like in for example the λc-term $\delta h.\, \lambda x.(h\langle x\rangle)(h\langle x\rangle)$. If a context contains many holes, these can be represented by different hole variables, as in $\delta h, g.\, \lambda x.(h\langle x\rangle)(\lambda y.\, g\langle x, y\rangle)$. An alternative representation is $\delta h.\, \delta g.\, \lambda x.(h\langle x\rangle)(\lambda y.\, g\langle x, y\rangle)$, where the holes are filled sequentially. Last but not least, the calculus may include variables over contexts and functions over contexts, witnessing the true first-class treatment of contexts.

In λc different notions of context can be represented. However, considering a calculus with contexts of a specific form, a criterion for well-definedness of such a calculus is of course that that specific form of contexts is preserved under transformations such as substitution, α- and β-rewriting, hole filling and composition.

Pretyping. The flexibility of the framework can be controlled by 'pretyping', that is, by restricting the λc-term formation. The aim of these restrictions is to gain more control over the form of λc-terms. Pretyping works in λc like typing does in lambda calculus. In a typed lambda calculus, each variable has a type and term formation is led by a set of typing rules. Analogously, a set of pretyping rules controls the λc-term formation. By means of pretyping, λc-terms can be restricted to representations of contexts with only one hole, variables in abstractions can be ensured to match their arguments, or the whole context calculus λc can be restricted to a subset of term constructors and rewrite rules, for example.

4. Definition of λc

This section conveys the definitions of the basic notions of the (untyped) framework.

As we have explained in the previous section, in addition to the lambda-calculus constructors, in λc there are two more pairs of abstractors and applicators, namely $(\Lambda, \langle\,\rangle)$ and $(\delta, \lceil\,\rceil)$, and, moreover, a composition constructor $\circ$. The pair $(\Lambda, \langle\,\rangle)$ and the rule $(@\beta)$ will be used for representing and computing communication. The pair $(\delta, \lceil\,\rceil)$ and the rule (*fill*) will be used for representing contexts and hole filling. The constructor $\circ$ and the rule $(\circ)$ will be used for composition. Hence, these added constructors with the rules together form a part of

the calculus that will be concerned with representing and computing context-related operations.

We now give the definition of λc. Let $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ as usual, and let $\mathcal{V}$ be a countably infinite set of variables.

Definition 1. (Λc) The set of un(pre)typed λc-terms Λc is defined inductively by

$$\begin{array}{rrrr} U ::= u \mid & (\lambda u.U) \mid & (UU) \mid & \\ & (\Lambda_n u_1, \ldots, u_n.U) \mid & (U\langle U, \ldots, U\rangle_n) \mid & \\ & (\delta_n u_1, \ldots, u_n.U) \mid & (U\lceil U, \ldots, U\rceil_n) \mid & (\circ_n(U, U, \ldots, U)) \end{array}$$

where $u, u_1, \ldots, u_n \in \mathcal{V}$ and $U, \ldots, U$ abbreviate n U's .

Notation. In general, a sequence of objects $a_1, \ldots, a_n$ will be abbreviated by the vector $\vec{a}$, where the vector is empty if $n = 0$, and the length of the vector will be denoted by $|\vec{a}|$. Thus, variables $u_1, \ldots, u_n$ in $(\Lambda_n u_1, \ldots, u_n.U)$ and $(\delta_n u_1, \ldots, u_n.U)$ will be abbreviated by $\vec{u}$, and terms $U_1, \ldots, U_n$ in the expressions $(U\langle U_1, \ldots, U_n\rangle_n)$, $(U\lceil U_1, \ldots, U_n\rceil_n)$ and $(\circ_n(U, U_1, \ldots, U_n))$ will be abbreviated by $\vec{U}$, where if $n = 0$ the preceding comma is dropped in the case of $\lceil\ \rceil$ and $\circ$. We will even omit the index n and assume that the arities of Λ, $\langle\ \rangle$, δ, $\lceil\ \rceil$ and $\circ$ and the number of their arguments match. Furthermore, if $\circ$ is binary, it is used in infix notation. As usual, standard abbreviations regarding brackets apply. In the remainder, the following convention considering typical elements will be adopted (if not explicitly stated otherwise): $i, j, l, m, n \in \mathbb{N}$, $u, u', u_i, v, w, \ldots \in \mathcal{V}$ and $U, U', U_i, V, W \ldots \in \Lambda c$.

Although the context calculus is designed to work with contexts as first-class objects, we could still make use of a notion of (meta-)contexts over λc-terms. A meta-context in λc is a λc-term with some holes, all of which are considered different. If C is a meta-context with n holes and $\vec{U}$ are n λc-terms, then the hole filling results in the λc-term $C[\vec{U}]$ where the i^{th} hole in C is replaced by U_i, for $1 \leq i \leq n$. Composition is defined analogously. In both operations, variable capturing may occur.

The constructors $\Lambda_n \_.\_$ and $\delta_n \_.\_$ are multiple abstractors, which bind n variables simultaneously, and the constructors $\_\langle \_, \ldots, \_\rangle_n$, $\_\lceil \_, \ldots, \_\rceil_n$ and $\circ_n(\_, \_, \ldots, \_)$ are $n+1$-ary function symbols. The free and bound variables in a λc-term are defined as in lambda calculus. As in the case of lambda calculus, λc-terms are considered equal up to α-conversion. Moreover, we assume that bound variables are renamed whenever necessary. In λc, we need multiple substitutions, which are a straightforward pointwise extension of (single) substitutions. For

$U, \vec{V} \in \Lambda c$ and m distinct variables $\vec{v}$, where m is also the number of terms in $\vec{V}$, the result $U[\![\vec{v} := \vec{V}]\!]$ of substituting V_i for free occurrences of v_i in U $(1 \leq i \leq m)$ is defined as:

$$\begin{array}{lcl} u[\![\vec{v} := \vec{V}]\!] & = & \begin{cases} V_i & : \text{ if } u = v_i \text{ for some } 1 \leq i \leq m \\ u & : \text{ otherwise,} \end{cases} \\ (\mathtt{B}\vec{u}.U')[\![\vec{v} := \vec{V}]\!] & = & \mathtt{B}\vec{u}.(U'[\![\vec{v} := \vec{V}]\!]), \\ \mathtt{F}\ (U',\ U_1, \ldots, U_n)[\![\vec{v} := \vec{V}]\!] & & \\ & = & \mathtt{F}\ (U'[\![\vec{v} := \vec{V}]\!],\ U_1[\![\vec{v} := \vec{V}]\!], \ldots, U_n[\![\vec{v} := \vec{V}]\!]). \end{array}$$

Here $\mathtt{B}$ denotes λ, Λ or δ, and $\mathtt{F}$ denotes $\cdot$ (implicit application), $\langle\ \rangle$, $\lceil\ \rceil$ or $\circ$. It is assumed that the bound variables in the second clause are renamed to avoid clashes.

Definition 2. (Context calculus λc) The context calculus λc is defined on terms of Λc with rewrite relations induced by two collections of rewrite rule schemas, the lambda calculus rewrite rule schema and the context rewrite rule schemas. The two collections of rewrite rule schemas (rewrite rules, for short) are given below.

i) The lambda calculus rewrite rule is:

$$(\lambda u.U)V \ \rightarrow_\beta \ U[\![u := V]\!]. \qquad (\beta)$$

ii) The context rewrite rules are:

$$\begin{array}{ll} (\Lambda\vec{u}.U)\langle\vec{V}\rangle \ \rightarrow_{@\beta} \ U[\![\vec{u} := \vec{V}]\!] & (@\beta) \\ (\delta\vec{u}.U)\lceil\vec{V}\rceil \ \rightarrow_{fill} \ U[\![\vec{u} := \vec{V}]\!] & (fill) \\ \circ_n((\delta_n\vec{u}.U), (\Lambda\vec{v}.\delta\vec{v}'.V_1), \ldots, (\Lambda\vec{w}.\delta\vec{w}'.V_n)) & \\ \quad \rightarrow_\circ \delta\vec{v}', \ldots, \vec{w}'.U[\![u_1 := \Lambda\vec{v}.V_1, \ldots, u_n := \Lambda\vec{w}.V_n]\!]. & (\circ) \end{array}$$

As usual, it is assumed that the bound variables in the rewrite rules are renamed to avoid clashes.

The rewrite rules $(@\beta)$, and $(fill)$ denote actually one rewrite rule for each index $n \in I\!N$ of the abstraction and application. The rewrite rule $(\circ)$ denotes a rewrite rule for each combination of the indices involved, with only one condition: the indices of $\circ$ and of the first δ have to match. Since these rewrite rules implement context-related operations in λc, we call them the context rewrite rules and denote the rewrite relation generated by these rewrite rules by $\rightarrow_c$.

Example 4. Instances of the rewrite rules $(@\beta)$, $(fill)$ and $(\circ)$ are (let x, x', y be term variables, and other variables be hole variables):

1. $(\Lambda x, x'.U)\langle V, V'\rangle \rightarrow_{@\beta} U[\![x := V, x' := V']\!]$
2. $(\delta.U)\lceil\rceil \rightarrow_{fill} U$
3. $(\delta g.U) \circ (\Lambda x, x'.\delta h.V) \rightarrow_{\circ} \delta h.U[\![g := \Lambda x, x'.V]\!]$
4. $(\delta g.U) \circ (\Lambda y.\delta h, k.V) \rightarrow_{\circ} \delta h, k.U[\![g := \Lambda y.V]\!]$
5. $\circ_2((\delta g, g'.U), (\Lambda x.\delta h.V), (\Lambda y.\delta k, k'.W))$
$\rightarrow_{\circ} \delta h, k, k'.U[\![g := \Lambda x.V, g' := \Lambda y.W]\!]$.

The last instance illustrates the composition of a two-hole context with two contexts, where the hole abstractions of the latter contexts are shifted to the beginning of the resulting context.

Remark. The context calculus can be defined more efficiently and, ultimately, translated to lambda calculus. The pairs of constructors $(\Lambda, \langle\,\rangle)$ and $(\delta, \lceil\,\rceil)$ with the corresponding rewrite rules have the same behavior, so the latter can be defined in terms of the former. Also, since compositions are functions on contexts, they can be defined as λc-terms, provided a powerful enough pretyping system is employed. In the case of Example 3 in Section 3, let $\underline{comp} \equiv \lambda c.\delta d'.\delta g'.c\lceil\Lambda x'.(d'\langle x'\rangle)\lceil g'\rceil\rceil$. Furthermore, by encoding the single abstraction and single application as special cases of the corresponding multiple one, the only constructors that are really needed are Λ and $\langle\,\rangle$ and the rewrite rule $(@\beta)$. These constructors and the rewrite rule (hence, all constructors and rewrite rules) can, in turn, be translated to the lambda calculus (using currying). This implies that the context calculus can be simulated within the lambda calculus where the computation of hole filling, composition and communication is performed in an algorithmic way, by rewriting groups of redexes simultaneously. However, we believe λc puts us at a suitable level of abstraction for working with contexts.

5. Confluence of λc

The next theorem states the main result on λc, saying that the order of computations is irrelevant.

Theorem 1. λc is confluent.

The proof is done indirectly, via higher-order rewriting systems (HRSs), a framework for term rewriting systems with binders. Since

the context calculus is such a rewriting system, it can very naturally be written as a HRS. First, the higher-order system $\mathcal{H}$ is defined, by translating the constructors and rewrite rules of the context calculus into the HRS-format. The higher-order system $\mathcal{H}$ turns out to be orthogonal: there are no critical pairs and all rewrite rules are left-linear (i.e. free variables occur at most once on the left-hand side of a rewrite rule). Since any orthogonal HRS is confluent (cf. [10], [19], [17]), it follows that $\mathcal{H}$ is confluent. Next, $\mathcal{H}$ is restricted to a subsystem, called $\mathcal{H}_{\lambda c}$, which is closed under reduction and which corresponds to the context calculus. From the properties of $\mathcal{H}$ and the theory of HRSs, it can be shown that $\mathcal{H}_{\lambda c}$ is confluent, hence the context calculus too.

The remainder of this section conveys some details of the proof. The proof considers the complete system λc, but it holds also for any subsystem of λc (which is closed under reduction). For more background on HRSs, we refer to [17].

In general, in the meta-language of HRSs, i.e. the simply typed lambda calculus, the set of types $\mathcal{T}$ is defined over a set of base types using the function constructor $\rightarrow$. Here, we restrict the set of base types to the singleton $\{0\}$. Furthermore, we assume there is a countably infinite set of typed variables at our disposal, $\mathcal{V}^{\rightarrow} = \bigcup_{\tau \in \mathcal{T}} \mathcal{V}^{\tau}$. Without loss of generality, we assume that the variables $\mathcal{V}^0$ of type 0 range over the same names as variables $\mathcal{V}$ of λc. Then, the terms of $\mathcal{H}$ are well-typed terms of the simply typed lambda calculus with constants corresponding to the constructors of the context calculus, and the rewrite rules of $\mathcal{H}$ are, loosely speaking, translations of the rewrite rules of λc.

Definition 3. (HRS $\mathcal{H}$) 1. The set of constants $C_{\mathcal{H}}$ contains the following constants ($n \in \mathbb{N}$):

$$
\begin{array}{ll}
\mathsf{abs} & : (0 \rightarrow 0) \rightarrow 0 \\
\mathsf{app} & : 0 \rightarrow 0 \rightarrow 0 \\
\mathsf{mabs}_n & : (\vec{0}_n \rightarrow 0) \rightarrow 0 \\
\mathsf{mapp}_n & : \vec{0}_{n+1} \rightarrow 0 \\
\mathsf{habs}_n & : (\vec{0}_n \rightarrow 0) \rightarrow 0 \\
\mathsf{hf}_n & : \vec{0}_{n+1} \rightarrow 0 \\
\mathsf{comp}_n & : \vec{0}_{n+1} \rightarrow 0.
\end{array}
$$

2. Terms of the HRS $\mathcal{H}$ are simply typed λ-calculus terms generated from the set of typed variables $\mathcal{V}^{\rightarrow}$ and the set of typed constants $C_{\mathcal{H}}$. The set of terms of HRS $\mathcal{H}$ is denoted by $\textsc{Term}(\mathcal{H})$.

3. The set of rewrite rules $R_{\mathcal{H}}$ of $\mathcal{H}$ contains the following rewrite rules:

$$\mathsf{app}\ (\mathsf{abs}(\lambda u.\, z\ u))\ z' \ \to_{\mathsf{b}}\ z\ z' \tag{b}$$

$$\mathsf{mapp}\ (\mathsf{mabs}(\lambda \vec{u}.\, z\ \vec{u}))\ \vec{z}' \ \to_{\mathsf{mb}}\ z\ \vec{z}' \tag{mb}$$

$$\mathsf{hf}\ (\mathsf{habs}(\lambda \vec{u}.\, z\ \vec{u}))\ \vec{z}' \ \to_{\mathsf{fill}}\ z\ \vec{z}' \tag{fill}$$

$$\begin{aligned}&\mathsf{comp}\ (\mathsf{habs}(\lambda \vec{u}.\, z\ \vec{u}))\ (\mathsf{mabs}(\lambda \vec{v}.\, \mathsf{habs}(\lambda \vec{v}'.\, z_1\ \vec{v}\ \vec{v}')))\ldots \\ &\qquad\qquad (\mathsf{mabs}(\lambda \vec{w}.\, \mathsf{habs}(\lambda \vec{w}'.\, z_n\ \vec{w}\ \vec{w}'))) \\ &\to_{\mathsf{cmp}}\ \mathsf{habs}(\lambda \vec{v}', \ldots, \vec{w}'.\, z\ (\mathsf{mabs}(\lambda \vec{v}.\, z_1\ \vec{v}\ \vec{v}'))\ldots \\ &\qquad\qquad (\mathsf{mabs}(\lambda \vec{w}.\, z_n\ \vec{w}\ \vec{w}'))).\end{aligned} \tag{cmp}$$

Proposition 1. $\mathcal{H}$ is confluent.

Proof. $\mathcal{H}$ is orthogonal. Hence, $\mathcal{H}$ is confluent. □

However, from the context calculus point of view, $\mathcal{H}$ contains too many elements: $\mathcal{H}$ contains also terms like $\mathsf{abs}(\lambda u.\, z\ u)$, $\lambda u.\, u$ and $\mathsf{app}\ x$, which are intuitively meaningless in the context calculus. The meaningful terms, in the context calculus' viewpoint, are the terms which mimic the term formation of λc: starting from variables, terms are built using abstractors and functors (in $\mathcal{H}$, constants) provided with the right number of arguments. The following definition describes such terms inductively.

Definition 4. (TERM($\mathcal{H}_{\lambda c}$)) Let TERM($\mathcal{H}_{\lambda c}$) be the smallest subset of the terms of $\mathcal{H}$ defined inductively as follows

1. $\mathcal{V}^0 \subseteq$ TERM($\mathcal{H}_{\lambda c}$); and
2. if $u, \vec{u} \in \mathcal{V}^0$ and $s, t, \vec{s} \in$ TERM($\mathcal{H}_{\lambda c}$), then $\mathsf{abs}(\lambda u.\, s)$, $\mathsf{app}\ s\ t$, $\mathsf{mabs}(\lambda \vec{u}.\, s)$, $\mathsf{mapp}\ s\ \vec{s}$, $\mathsf{habs}(\lambda \vec{u}.\, s)$, $\mathsf{hf}\ s\ \vec{s}$, $\mathsf{comp}\ s\ \vec{s}$ are all elements of TERM($\mathcal{H}_{\lambda c}$).

Equivalently, one could say that TERM($\mathcal{H}_{\lambda c}$) contains the elements s of TERM($\mathcal{H}$) such that s is of type 0, s is in long $\beta\eta$-normal form, and s contains only variables of type 0.

By the next proposition, TERM($\mathcal{H}_{\lambda c}$) is closed under the rewrites of $\mathcal{H}$. Accordingly, $\mathcal{H}_{\lambda c} = \langle$TERM($\mathcal{H}_{\lambda c}$), $\mathcal{R}_{\mathcal{H}}\rangle$ may safely be called a sub-HRS of $\mathcal{H}$.

Proposition 2. Let $s \in$ TERM($\mathcal{H}_{\lambda c}$). If $s \to t$ then $t \in$ TERM($\mathcal{H}_{\lambda c}$).

Proof. First, one proves that if $l^\sigma{\downarrow_\beta} \in$ TERM($\mathcal{H}_{\lambda c}$) then $r^\sigma{\downarrow_\beta} \in$ TERM($\mathcal{H}_{\lambda c}$), for every rewrite rule $l \to r$ of $\mathcal{H}$ and HRS-substitution σ. This is based on the fact that, by definition, HRS-substitutions

assign long $\beta\eta$-normal forms, and if $l^\sigma{\downarrow_\beta} \in \textsc{Term}(\mathcal{H}_{\lambda c})$, then $\forall z \in \mathrm{dom}(\sigma).\ \sigma(z) = \lambda\vec{v}.s$ with $\vec{v}, s \in \textsc{Term}(\mathcal{H}_{\lambda c})$. Then the proposition follows by induction to s. □

Corollary 1. The sub-HRS $\mathcal{H}_{\lambda c}$ is confluent.

Proof. Confluence of $\mathcal{H}_{\lambda c}$ is a corollary of Proposition 1 using the fact that subsystems of the confluent ones, are confluent themselves. □

The definition of lifting spells out the intuition used when defining $\mathcal{H}_{\lambda c}$. Lifting specifies the one-to-one correspondence between the terms of λc and $\textsc{Term}(\mathcal{H}_{\lambda c})$. We will call the inverse function projection and denote it by $\llbracket\ \text{-}\ \rrbracket$.

Definition 5. (Lifting Λc to $\textsc{Term}(\mathcal{H}_{\lambda c})$) Let $U \in \Lambda c$. The translation (*lifting*) of U to sub-HRS $\mathcal{H}_{\lambda c}$, $\ulcorner U \urcorner$ is defined by the structural induction on U:

$$
\begin{array}{lcl}
\ulcorner u \urcorner & = & u \\
\ulcorner\ \lambda u.U\ \urcorner & = & \mathsf{abs}(\lambda u.\ulcorner U \urcorner) \\
\ulcorner\ UV\ \urcorner & = & \mathsf{app}\ \ulcorner U \urcorner\ \ulcorner V \urcorner \\
\ulcorner\ \Lambda\vec{u}.U\ \urcorner & = & \mathsf{mabs}(\lambda\vec{u}.\ulcorner U \urcorner) \\
\ulcorner\ U\langle\vec{U}\rangle\ \urcorner & = & \mathsf{mapp}\ \ulcorner U \urcorner\ \ulcorner \vec{U} \urcorner \\
\ulcorner\ \delta\vec{u}.U\ \urcorner & = & \mathsf{habs}(\lambda\vec{u}.\ulcorner U \urcorner) \\
\ulcorner\ U\lceil\vec{U}\rceil\ \urcorner & = & \mathsf{hf}\ \ulcorner U \urcorner\ \ulcorner \vec{U} \urcorner \\
\ulcorner\ \circ(U,\vec{U})\ \urcorner & = & \mathsf{comp}\ \ulcorner U \urcorner\ \ulcorner \vec{U} \urcorner
\end{array}
$$

where the variables u, $\vec{u}$ on the right-hand sides are all of type 0.

Proposition 3. 1. Let $U, \vec{U} \in \Lambda c$. If $U \to U_1 \to \ldots \to U_n$ then $\ulcorner U \urcorner \to \ulcorner U_1 \urcorner \to \ldots \to \ulcorner U_n \urcorner$.

2. Let $s, \vec{s} \in \textsc{Term}(\mathcal{H}_{\lambda c})$. If $s \to s_1 \to \ldots \to s_n$ then $\llbracket s \rrbracket \to \llbracket s_1 \rrbracket \to \ldots \to \llbracket s_n \rrbracket$.

Proof. The proofs of the two parts resemble each other and are conducted by induction to n. The crux of the proofs is that a lifted contraction of a rewrite rule (ρ) in λc is a contraction of the corresponding rewrite rule (r) in $\mathcal{H}_{\lambda c}$, and that a projected contraction of a rewrite rule (r) in $\mathcal{H}_{\lambda c}$ is a contraction of the corresponding rewrite rule (ρ) in λc, which is proved by case analysis on the rewrite rules of λc and $\mathcal{H}_{\lambda c}$, respectively. □

Then, λc is confluent: Each pair of diverging reductions $U \twoheadrightarrow V_1$ and $U \twoheadrightarrow V_2$ of λc can be lifted to a pair of diverging reductions

$\llbracket U \rrbracket \twoheadrightarrow \llbracket V_1 \rrbracket$ and $\llbracket U \rrbracket \twoheadrightarrow \llbracket V_2 \rrbracket$, respectively, in $\mathcal{H}_{\lambda c}$, by Proposition 3 (i). Since $\mathcal{H}_{\lambda c}$ is confluent (Proposition 1), there is a pair of converging reductions $\llbracket V_1 \rrbracket \twoheadrightarrow t$ and $\llbracket V_2 \rrbracket \twoheadrightarrow t$ in $\mathcal{H}_{\lambda c}$. The converging reductions can be projected back to converging reductions from V_1 and V_2, i.e. $V_1 = \lfloor\!\lfloor \llbracket V_1 \rrbracket \rfloor\!\rfloor \twoheadrightarrow \lfloor\!\lfloor t \rfloor\!\rfloor$ and $V_2 = \lfloor\!\lfloor \llbracket V_2 \rrbracket \rfloor\!\rfloor \twoheadrightarrow \lfloor\!\lfloor t \rfloor\!\rfloor$ respectively, in λc, by Proposition 3(ii) and by the fact that lifting and projecting are each others inverse.

Remark. Actually, even a stronger result than confluence holds: in $\mathcal{H}_{\lambda c}$ any pair of (unions of) rewrite rules commutes. This commutation property rests on the Prism Theorem for higher-order rewriting (cf. [18]), which is employed in confluence proofs via developments. The commutation property of rewriting in $\mathcal{H}_{\lambda c}$ implies the commutation property of rewriting in λc.

6. Three case studies of pretyping

The three examples of this section illustrate the flexibility of the framework λc that can be obtained by fine-tuning the pretyping rules. The first example is a calculus for representing the untyped lambda calculus with λ-contexts. The representations can be manipulated in λc, but there are no variables or functions over (representations of) λ-contexts. The second example describes a calculus over simply typed lambda calculus with λ-contexts. By ignoring the types this example becomes an extension of the first example with first-class contexts. The third example is a calculus for simply typed lambda calculus with a special kind of contexts, namely De Bruijn's segments.

6.1. Context calculus for untyped λ-contexts

The pretyping presented in this section deals with the terms and contexts of the untyped lambda calculus. We will first stipulate what kind of λ-contexts we are interested in, and give the definitions of hole filling and composition accordingly. Next, we will give a translation and pretyping system λc^{λ} for these contexts and state some of its properties. Finally, we will address the adequacy of the context representation within λc^{λ}.

The λ-contexts of this section are λ-terms with $n \in \mathbb{N}$ holes, i.e.

$$\begin{array}{lll} M & ::= & x \mid \lambda x.M \mid M\,N \\ C & ::= & x \mid \Box \mid \lambda x.C \mid C\,D. \end{array}$$

We consider each occurrence of $\Box$ as a different hole and assume the holes in a λ-context to be ordered from left to right.

Two operations on λ-contexts are considered, namely hole filling and composition. For the purpose of preciseness, in this section we will make a distinction between the notations of these operations (by the function names *hf* and *comp*) and notations for their result (using [ ]). Hole filling, denoted by *hf*, is the function from λ-contexts and λ-terms to λ-terms which to a λ-context C with n holes and n λ-terms $\vec{M}$ assigns the λ-term $C[\vec{M}]$, which is the result of replacing the i^{th} hole in C by M_i (for $1 \leq i \leq n$). Composition, denoted by *comp*, is a function from λ-contexts to λ-contexts defined analogously. In sum, $hf(C,\ \vec{M}) = C[\vec{M}]$ and $comp(C,\ \vec{D}) = C[\vec{D}]$. Being defined as functions, hole filling and composition may be composed in a more complex expression. However, mixing these functions with term or context formation is not allowed. In this connection we clarify some (ambiguous) conventional notations: for example, $\lambda x.C[M]$ is either the result of $hf((\lambda x.\Box),\ hf(C,\ M))$, or of $hf(comp((\lambda x.\Box),\ C),\ M)$.

In the sequel $P, \vec{Q}, \vec{R}$ range over expressions possibly containing the meta-operations *hf* and *comp*. Here P and $\vec{Q}$ will be used for expressions resulting in λ-contexts, and $\vec{R}$ for expressions resulting in λ-terms. The λ-context that is the result of evaluating P is denoted by P^*; likewise for Q's and R's.

The representation of λ-terms and λ-contexts within λc is implemented by the translation. Translation of the λ-contexts and the context-related functions to λc requires some preprocessing, which involves meta-level (α-sensitive) observations being made explicit. For this purpose, two functions are assumed: $\textsc{Nrh}(P)$, which returns the number of holes in the result of P (i.e. in P^*); and $\textsc{Bnd}(P, i)$, which returns the list of all variables such that the i^{th} hole of P^* lies in their scope ($1 \leq i \leq \textsc{Nrh}(P)$). For instance, $\textsc{Nrh}(comp(\lambda x.\Box,\ \Box\Box)) = 2$ and $\textsc{Bnd}(comp(\lambda x.\Box,\ \Box\Box), 1) = x$. Regarding Bnd, we assume that there are no overshadowed binders in a λ-context, as there are in $\lambda x.\lambda x.\Box$, but Bnd (and later, the λc-terms $\textsc{Lft}(Q, \vec{x})$) could have be defined otherwise to deal with overshadowed variables.

The λ-terms and λ-contexts are translated to λc by the translation function $[\![\_]\!]$. The translation function behaves like the identity function on λ-terms while on λ-contexts it replaces holes by 'labeled' hole variables and adds a preamble: if $\textsc{Nrh}(C) = n$, $\textsc{Bnd}(C, 1) = \vec{x}$, ...,$\textsc{Bnd}(C, n) = \vec{y}$, then

$$\begin{aligned} [\![M]\!] &= M \\ [\![C]\!] &= \delta\vec{h}.C[h_1\langle\vec{x}\rangle, \ldots, h_n\langle\vec{y}\rangle]. \end{aligned}$$

The translation function extends to the composed objects as: if $\textsc{Nrh}(P) = n$, $\textsc{Bnd}(P,1) = \vec{x}, \ldots, \textsc{Bnd}(P,n) = \vec{y}$, then

$$\begin{array}{lcl} [\![\ hf(P,\ \vec{R})\ ]\!] & = & [\![P]\!]\,\lceil \Lambda\vec{x}.[\![R_1]\!], \ldots, \Lambda\vec{y}.[\![R_n]\!]\rceil \\ [\![\ comp(P,\ \vec{Q})\ ]\!] & = & \circ([\![P]\!], \Lambda\vec{x}.\textsc{Lft}([\![Q_1]\!], \vec{x}), \ldots, \Lambda\vec{y}.\textsc{Lft}([\![Q_n]\!], \vec{y})) \end{array}$$

where $\textsc{Lft}$ lifts the holes of Q_i. In general $\textsc{Lft}$ is defined by: if $\textsc{Nrh}(Q_i) = m$, $\textsc{Bnd}(Q_i,1) = \vec{y}, \ldots, \textsc{Bnd}(Q_i,m) = \vec{z}$, then

$$\textsc{Lft}([\![Q_i]\!], \vec{x}) = \delta\vec{g}.[\![Q_i]\!]\,\lceil \Lambda\vec{y}.\, g_1\langle\vec{x}\vec{y}\rangle, \ldots, \Lambda\vec{z}.\, g_m\langle\vec{x}\vec{z}\rangle\rceil.$$

In sum, translation of an expression P explicitly involves hole filling, composition, communication and lifting of holes.

Next, we give a pretyping that matches the translations. Let the set of base pretypes be $\mathcal{BT} = \{\mathtt{t}\}$. Then the pretypes $\rho \in \mathcal{P}$ are defined as

$$\rho ::= \mathtt{t} \mid [\vec{\mathtt{t}}]\mathtt{t} \mid [\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t} \mid [\vec{\mathtt{t}}]([\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t}),$$

where $[\,]$ binds stronger than $\times$, which in turn binds stronger than $\Rightarrow$.

The pretyping uses two bases, the basis Γ containing declarations of the form $x : \mathtt{t}$, and the basis Δ containing declarations of the form $h : [\vec{\mathtt{t}}]\mathtt{t}$. The bases are split because the elements of Γ are used as true variables whereas the elements of Δ serve as markers, in the sense that they are used for marking the beginning (abstraction) and endings (i.e. holes) of a context. The new type constructors $[\,]$ and $\_ \times \ldots \times \_ \Rightarrow$ are introduced for distinguishing between different constructors of λc (namely, $[]$ for Λ and $\langle\,\rangle$, and $\_\times\ldots\times\_\Rightarrow$ for δ and $\lceil\,\rceil$), as will become clear in the pretyping rules. In the pretyping rules, some vectors are suggestively indexed by their length, and $\vec{U} : \vec{\mathtt{t}}$ denotes the pointwise pretyping $U_i : \mathtt{t}$ for $1 \leq i \leq |\vec{U}| = |\vec{\mathtt{t}}|$. Furthermore, both Γ and Δ are, without loss of generality, assumed to contain distinct variables.

Definition 6. (Pretyping rules for λc^{λ}) A term $U \in \Lambda c$ is pretypable by ρ from the bases Γ, Δ, if $\Gamma, \Delta \vdash U : \rho$ can be derived using the pretyping rules displayed in Figure 1.

The pretyping rules can be explained as follows. The rules (*var*), (*abs*) and (*app*) are the rules that guard the well-formedness of the untyped λ-terms. The rules (*hvar*) and (*habs*) are used in forming representations of λ-contexts. The rules (*mabs*), ($mabs_c$) and (*mapp*) are used for pretyping communication around holes and representations of λ-terms and λ-contexts to be put into holes. The rules (*fill*) and (*comp*) are used in pretyping when filling holes.

$$(var)\quad \frac{(x:\mathtt{t})\in\Gamma}{\Gamma,\Delta\ \vdash\ x:\mathtt{t}}$$

$$(abs)\quad \frac{\Gamma,x:\mathtt{t},\Delta\ \vdash\ U:\mathtt{t}}{\Gamma,\Delta\ \vdash\ \lambda x.U:\mathtt{t}}$$

$$(app)\quad \frac{\Gamma,\Delta\ \vdash\ U:\mathtt{t}\quad \Gamma,\Delta\ \vdash\ V:\mathtt{t}}{\Gamma,\Delta\ \vdash\ UV:\mathtt{t}}$$

$$(hvar)\quad \frac{(h:[\vec{\mathtt{t}}]\mathtt{t})\in\Delta}{\Gamma,\Delta\ \vdash\ h:[\vec{\mathtt{t}}]\mathtt{t}}$$

$$(habs)\quad \frac{\Gamma,\Delta,h_1:[\vec{\mathtt{t}}_{m_1}]\mathtt{t},\ldots,h_n:[\vec{\mathtt{t}}_{m_n}]\mathtt{t}\ \vdash\ U:\mathtt{t}}{\Gamma,\Delta\ \vdash\ \delta\vec{h}.U:[\vec{\mathtt{t}}_{m_1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{m_n}]\mathtt{t}\Rightarrow\mathtt{t}}$$

$$(mabs)\quad \frac{\Gamma,\vec{x}:\vec{\mathtt{t}},\Delta\ \vdash\ U:\mathtt{t}}{\Gamma,\Delta\ \vdash\ \Lambda\vec{x}.U:[\vec{\mathtt{t}}]\mathtt{t}}$$

$$(mapp)\quad \frac{\Gamma,\Delta\ \vdash\ U:[\vec{\mathtt{t}}]\mathtt{t}\quad \Gamma,\Delta\ \vdash\ \vec{V}:\vec{\mathtt{t}}}{\Gamma,\Delta\ \vdash\ U\langle\vec{V}\rangle:\mathtt{t}}$$

$$(mabs_c)\quad \frac{\Gamma,\vec{x}:\vec{\mathtt{t}},\Delta\ \vdash\ U:[\vec{\mathtt{t}}_{m_1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{m_n}]\mathtt{t}\Rightarrow\mathtt{t}}{\Gamma,\Delta\ \vdash\ \Lambda\vec{x}.U:[\vec{\mathtt{t}}]([\vec{\mathtt{t}}_{m_1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{m_n}]\mathtt{t}\Rightarrow\mathtt{t})}$$

$$(fill)\quad \frac{\Gamma,\Delta\ \vdash\ U:[\vec{\mathtt{t}}_{m_1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{m_n}]\mathtt{t}\Rightarrow\mathtt{t}\quad \Gamma,\Delta\ \vdash\ V_i:[\vec{\mathtt{t}}_{m_i}]\mathtt{t}}{\Gamma,\Delta\ \vdash\ U\lceil\vec{V}\rceil:\mathtt{t}}$$

$$(comp)\quad \frac{\begin{array}{c}\Gamma,\Delta\ \vdash\ U:[\vec{\mathtt{t}}_{m_1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{m_n}]\mathtt{t}\Rightarrow\mathtt{t}\\ \Gamma,\Delta\ \vdash\ V_i:[\vec{\mathtt{t}}_{m_i}]([\vec{\mathtt{t}}_{j_1^i}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{j_{l_i}^i}]\mathtt{t}\Rightarrow\mathtt{t})\end{array}}{\begin{array}{l}\Gamma,\Delta\ \vdash\ \circ(U,\vec{V}):\\ [\vec{\mathtt{t}}_{j_1^1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{j_{l_1}^1}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{j_1^n}]\mathtt{t}\times\ldots\times[\vec{\mathtt{t}}_{j_{l_n}^n}]\mathtt{t}\Rightarrow\mathtt{t}\end{array}}$$

Figure 1. Pretyping rules for λc^{λ}

Definition 7. (λc^{λ}) The terms of λc^{λ} are the well-pretyped terms of λc according to Definition 6. The rewrite rules are the rules (β), $(@\beta)$, (fill) and a version of $(\circ)$ of λc, now over well-pretyped terms:

$$(\lambda x.U)\,V \to_{\beta} U[\![x := V]\!]$$

$$(\Lambda \vec{u}.U)\,\langle \vec{V} \rangle \to_{@\beta} U[\![\vec{u} := \vec{V}]\!]$$

$$(\delta \vec{u}.U)\,\lceil \vec{V} \rceil \to_{\mathit{fill}} U[\![\vec{u} := \vec{V}]\!]$$

$$\circ((\delta \vec{g}.U), (\Lambda \vec{x}.\delta \vec{h}.V_1), \ldots, (\Lambda \vec{y}.\delta \vec{k}.V_n)) \to_{\circ} \delta \vec{h}, \ldots, \vec{k}.U[\![g_1 := \Lambda \vec{x}.V_1, \ldots, g_n := \Lambda \vec{y}.V_n]\!].$$

We investigate some properties of the pretyping.

Proposition 4. (Subject reduction) If $\Gamma, \Delta \vdash U : \rho$ and $U \twoheadrightarrow V$, then $\Gamma, \Delta \vdash V : \rho$.

Proof. The proof is a standard one and relies on the generation lemma, which gives a correspondence between the structure of a λc^{λ}-term and its pretype. □

With the pretyping being a Curry-style typing, not all pretypable λc-terms do have a unique pretype. For example, the λc-term $\delta h.x$ is pretypable by both $[\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}$ and $[\,]\mathtt{t} \Rightarrow \mathtt{t}$. Moreover, the calculus λc^{λ} is not strongly normalizing. For example, $\vdash (\lambda x.xx)(\lambda x.xx) : \mathtt{t}$ and as we know, this term can be endlessly rewritten. However, the context-related rewriting is weakly normalizing.

Proposition 5. (Weak normalization) The reduction generated by $\to_c$ in λc^{λ} is weakly normalizing.

Proof. By the leftmost-innermost strategy each λc^{λ}-term reduces to the normal form w.r.t. $\to_c$: let

$$(\sharp(\circ\text{-symbols in } U), \sharp(\mathit{fill}\text{-redexes in } U), \sharp(@\beta\text{-redexes in } U))$$

be the measure $m(U)$ of an arbitrary λc^{λ}-term U and check that $m(U)$ decreases by each context rewrite step in which the leftmost-innermost redex is contracted. □

Next, we investigate some properties of representations of λ-terms and λ-contexts. Translations of λ-terms and λ-contexts are indeed pretypable in λc^{λ}, as is stated in the next proposition.

Proposition 6. Let R and P be expressions resulting in a λ-term and a λ-context, respectively. Then,

1. $\Gamma \vdash \llbracket R \rrbracket : \mathtt{t}$.
2. $\Gamma \vdash \llbracket P \rrbracket : [\vec{\mathtt{t}}]\,\mathtt{t} \times \ldots \times [\vec{\mathtt{t}}]\mathtt{t} \Rightarrow \mathtt{t}$.

Proof. The proof is conducted by simultaneous induction on the structure of R and P. □

The adequacy of the context representation within λc^{λ} is the subject of the following two propositions. The first proposition claims that the two ways of computing $\llbracket\, hf(P,\, \vec{R})\, \rrbracket$ and $\llbracket\, comp(P,\, \vec{Q})\, \rrbracket$ result in the same λc-terms. Let $U{\downarrow_c}$ denote the normal form of U w.r.t. the context rewrite rules.

Proposition 7. Let P be an expression resulting in a λ-term or a λ-context. Then $\llbracket P^* \rrbracket = \llbracket P \rrbracket{\downarrow_c}$.

Proof. The proof is conducted by induction to P. □

The second proposition states that in λc^{λ} hole filling may be postponed. More precisely, it states that β-steps can be performed in the representation of a λ-context within λc^{λ} before hole filling. Recall that in lambda calculus, β-steps may be performed in a λ-context only after filling the holes of the λ-context.

Proposition 8. Let P be an expression resulting in a λ-context with n holes and $\vec{R}$ be n expressions resulting in λ-terms. Let $\textsc{Bnd}(P,1) = \vec{x}, \ldots, \textsc{Bnd}(P,n) = \vec{y}$. Suppose $\llbracket P \rrbracket \twoheadrightarrow_{\beta} V$. Then there is a λ-term N such that $P^*[\vec{R}^*] \twoheadrightarrow_{\beta} N$ and $V\,\lceil \Lambda\vec{x}.\llbracket R_1 \rrbracket, \ldots, \Lambda\vec{y}.\llbracket R_n \rrbracket \rceil \twoheadrightarrow_c N$.

Proof. The proof is illustrated in Figure 2. Technically, for the labeling of the converging reductions the commutation property of $\rightarrow_{\beta}$ and $\rightarrow_c$ (see the concluding remark of Section 5) is used. □

6.2. Context calculus for simply typed λ-calculus

The pretyped calculus $\lambda c^{\rightarrow}$ given in this section describes the simply typed lambda calculus with λ-contexts (*i*) with many holes, which may occur manifold, (*ii*) where holes are filled sequentially, (*iii*) including composition and (*iv*) including context variables and functions over

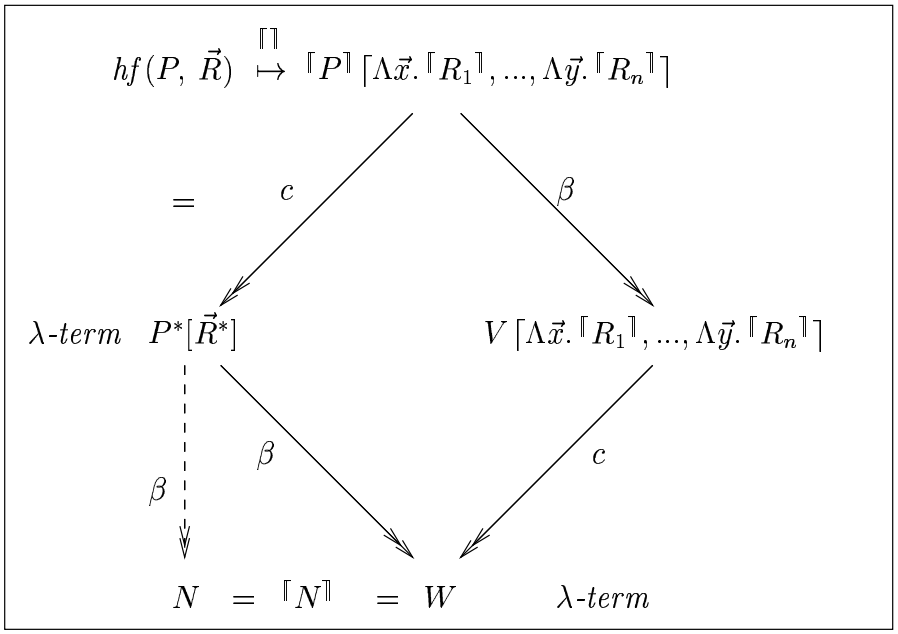

Figure 2. Hole filling can be postponed

(representations of) λ-contexts. The representation of λ-contexts follows the description given in the introduction. The pretyping rules of $\lambda c^{\rightarrow}$ for the most part follow the typing rules of the calculus of M. Hashimoto and A. Ohori (cf. [8]). In this section, we will first define the pretypes, the pretyping rules and the calculus $\lambda c^{\rightarrow}$. Then, we will summarize some properties of $\lambda c^{\rightarrow}$ and briefly compare $\lambda c^{\rightarrow}$ to the calculus of M. Hashimoto and A. Ohori [8]. Finally, we will name a variation of this pretyping.

Let $\mathcal{BT}$ denote the set of base types with $\mathtt{a} \in \mathcal{BT}$. The τ-pretypes ($\tau \in \mathcal{T}$) and the ρ-pretypes ($\rho \in \mathcal{P}$) are defined as

$$\tau ::= \mathtt{a} \mid \tau \rightarrow \tau \mid [\vec{\tau}]\tau \Rightarrow \tau \quad \text{and} \quad \rho ::= \tau \mid [\vec{\tau}]\tau.$$

Here, $\rightarrow$ associates to the right, $\rightarrow$ binds stronger than $[\,]$ and $[\,]$ binds stronger than $\Rightarrow$. The τ-pretypes are used for pretyping representations of λ-terms and λ-contexts, and the ρ-pretypes are also used for pretyping communicating objects and holes. The pretyping uses two bases, the basis Γ containing declarations of the form $x : \tau$, and the basis Δ containing declarations of the form $h : [\vec{\tau}]\tau$. Similarly to the case in λc^{λ}, the new type constructors $[\,]$ and $\Rightarrow$ are introduced for better correspondence with the constructors of λc (namely, $[\,]$ for Λ and $\langle\,\rangle$, and $\Rightarrow$ for δ and $\lceil\,\rceil$).

In the remainder, the following notation will be used: $\tau, \sigma, \tau', \vec{\tau} \ldots \in \mathcal{T}$, $\rho, \rho' \in \mathcal{P}$, and $\vec{U} : \vec{\tau}$ denotes the pointwise pretyping $U_i : \tau_i$ for $1 \leq i \leq |\vec{\tau}|$.

Definition 8. (Pretyping rules for $\lambda c^{\rightarrow}$) A term $U \in \Lambda c$ is pretypable by ρ from the bases Γ, Δ, if $\Gamma, \Delta \vdash U : \rho$ can be derived using the pretyping rules displayed in Figure 3. The set of well-pretyped λc-terms will be denoted by $\Lambda c^{\rightarrow}$.

We briefly comment on the pretyping rules. The rules (*var*), (*abs*) and (*app*) are the familiar Church-style typing rules for $\lambda^{\rightarrow}$, now ranging also over (representations of) λ-contexts. The rules (*hvar*), (*habs*) and (*fill*) are their respective counterparts dealing with hole variables. The rules (*mabs*) and (*mapp*) are used for pretyping communication, where the components (variables in the abstraction and arguments in the application) are all of τ-pretypes. Note that only the 'unary' hole abstractor δ and the binary hole filler $\lceil\ \rceil$ are employed.

Figure 4 is an example of pretyping in $\lambda c^{\rightarrow}$.

Definition 9. ($\lambda c^{\rightarrow}$) The terms of $\lambda c^{\rightarrow}$ are the well-pretyped terms of λc according to Definition 8. The rewrite rules are the rules (β), ($@\beta$) and (*fill*) of λc, now over well-pretyped terms:

$$(\lambda x : \tau . U) V \rightarrow_{\beta} U[\![x := V]\!]$$

$$(\Lambda \vec{x} : \vec{\tau} . U) \langle \vec{V} \rangle \rightarrow_{@\beta} U[\![\vec{x} := \vec{V}]\!]$$

$$(\delta h : [\vec{\tau}]\tau . U) \lceil V \rceil \rightarrow_{fill} U[\![h := V]\!].$$

Note that there is no composition in $\lambda c^{\rightarrow}$. This is because composition is definable: for every (context) U of pretype $[\vec{\tau}]\tau \Rightarrow \tau'$ and every (communicating context) V of pretype $[\vec{\tau}]([\vec{\sigma}]\sigma \Rightarrow \tau)$ the following closed pretypable λc-term can act as a composition constructor in $\underline{comp}\ U\ V$,

$$\begin{array}{rcl} \underline{comp} & \equiv & \lambda c\colon [\vec{\tau}]\tau \Rightarrow \tau' . \delta d\colon [\vec{\tau}]([\vec{\sigma}]\sigma \Rightarrow \tau) . \delta g\colon [\vec{\sigma}]\sigma . c \lceil \Lambda \vec{x}\colon \vec{\tau} . (d \langle \vec{x} \rangle) \lceil g \rceil \rceil \\ & : & ([\vec{\tau}]\tau \Rightarrow \tau') \rightarrow ([\vec{\tau}]([\vec{\sigma}]\sigma \Rightarrow \tau) \Rightarrow ([\vec{\sigma}]\sigma \Rightarrow \tau')). \end{array}$$

Consequently, the composition constructor, pretyping rule and rewrite rule are omitted.

From the lambda calculus viewpoint, the λc-terms of a τ-pretype without free hole variables are representations of λ-terms, λ-contexts, functions over these elements. The other λc-terms are intermediate representations of λ-contexts and communicating objects.

We have the following results.

$$(var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma,\Delta\ \vdash\ x:\tau}$$

$$(abs)\quad \frac{\Gamma,x:\tau,\Delta\ \vdash\ U:\tau'}{\Gamma,\Delta\ \vdash\ \lambda x:\tau.U:\tau\to\tau'}$$

$$(app)\quad \frac{\Gamma,\Delta\ \vdash\ U:\tau\to\tau'\quad \Gamma,\Delta\ \vdash\ V:\tau}{\Gamma,\Delta\ \vdash\ UV:\tau'}$$

$$(hvar)\quad \frac{(h:[\vec{\tau}]\tau)\in\Delta}{\Gamma,\Delta\ \vdash\ h:[\vec{\tau}]\tau}$$

$$(habs)\quad \frac{\Gamma,\Delta,h:[\vec{\tau}]\tau\ \vdash\ U:\tau'}{\Gamma,\Delta\ \vdash\ \delta h:[\vec{\tau}]\tau.U:[\vec{\tau}]\tau\Rightarrow\tau'}$$

$$(mabs)\quad \frac{\Gamma,\vec{x}:\vec{\tau},\Delta\ \vdash\ U:\tau}{\Gamma,\Delta\ \vdash\ \Lambda\vec{x}:\vec{\tau}.U:[\vec{\tau}]\tau}$$

$$(mapp)\quad \frac{\Gamma,\Delta\ \vdash\ U:[\vec{\tau}]\tau\quad \Gamma,\Delta\ \vdash\ \vec{V}:\vec{\tau}}{\Gamma,\Delta\ \vdash\ U\langle\vec{V}\rangle:\tau}$$

$$(fill)\quad \frac{\Gamma,\Delta\ \vdash\ U:[\vec{\tau}]\tau\Rightarrow\tau'\quad \Gamma,\Delta\ \vdash\ V:[\vec{\tau}]\tau}{\Gamma,\Delta\ \vdash\ U\lceil V\rceil:\tau'}$$

Figure 3. Pretyping rules for $\lambda c^{\to}$

Proposition 9. (Uniqueness of pretypes) If $\Gamma,\Delta\vdash U:\rho_1$ and $\Gamma,\Delta\vdash U:\rho_2$ then $\rho_1\equiv\rho_2$.

Proposition 10. (Subject reduction) If $\Gamma,\Delta\ \vdash\ U\ :\ \rho$ and $U\twoheadrightarrow V$, then $\Gamma,\Delta\ \vdash\ V:\rho$.

The proofs of these two propositions are the standard ones, as in the case of $\lambda^{\to}$ *à la* Church. The proof of the second proposition uses

$$\dfrac{\dfrac{\dfrac{(h:[\mathtt{a}]\mathtt{a})\in\{h:[\mathtt{a}]\mathtt{a}\}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h:[\mathtt{a}]\mathtt{a}} \quad \dfrac{(x:\mathtt{a})\in\{z:\mathtt{a},x:\mathtt{a}\}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash x:\mathtt{a}}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h\langle x\rangle:\mathtt{a}}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash \lambda x:\mathtt{a}.h\langle x\rangle:\mathtt{a}\to\mathtt{a}} \quad \begin{array}{c}\cdot\\ \cdot\\ \cdot\end{array} \quad \dfrac{\dfrac{\dfrac{(h:[\mathtt{a}]\mathtt{a})\in\{h:[\mathtt{a}]\mathtt{a}\}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h:[\mathtt{a}]\mathtt{a}} \quad \dfrac{(z:\mathtt{a})\in\{z:\mathtt{a}\}}{z:\mathtt{a},x:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash z:\mathtt{a}}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash h\langle z\rangle:\mathtt{a}}}{z:\mathtt{a};h:[\mathtt{a}]\mathtt{a}\vdash(\lambda x:\mathtt{a}.h\langle x\rangle)(h\langle z\rangle):\mathtt{a}}}{z:\mathtt{a}\vdash\lambda h:[\mathtt{a}]\mathtt{a}.(\lambda x:\mathtt{a}.h\langle x\rangle)(h\langle z\rangle):[\mathtt{a}]\mathtt{a}\Rightarrow\mathtt{a}}$$

Figure 4. An example of pretyping in $\lambda c^{\to}$

counterparts of the generation lemma and the substitution lemma. The substitution lemma treats a multiple substitution saying that, if $\Gamma,\vec{u}:\vec{\rho},\Delta\vdash U:\rho$ and $\Gamma,\Delta\vdash\vec{V}:\vec{\rho}$ then $\Gamma,\Delta\vdash U[\![\vec{u}:=\vec{V}]\!]:\rho$, which is proved by induction to U. Using the substitution lemma one can prove that the left-hand sides of the rewrite rules have the same type as the corresponding right-hand sides. Hence, reduction preserves pretypes in $\lambda c^{\to}$.

Furthermore, reduction in $\lambda c^{\to}$ is strongly normalizing. The proof of strong normalization can be done via the natural translation [‡] of $\lambda c^{\to}$ into $\lambda^{\to}$.

Definition 10. (Translation of $\lambda c^{\to}$ to $\lambda^{\to}$)

i) Define $\lceil\!\rceil:\mathcal{P}\to\mathcal{T}_\lambda$ as a function that translates the pretypes to simple types:

$$\begin{array}{lcl}\lceil\,\mathtt{a}\,\rceil & = & \mathtt{a}\\ \lceil\tau\to\tau'\rceil & = & \lceil\tau\rceil\to\lceil\tau'\rceil\\ \lceil[\vec{\tau}]\tau\rceil & = & \lceil\tau_1\rceil\to\ldots\to\lceil\tau_n\rceil\to\lceil\tau\rceil\\ \lceil[\vec{\tau}]\tau\Rightarrow\tau'\rceil & = & \lceil\tau_1\rceil\to\ldots\to\lceil\tau_n\rceil\to\lceil\tau\rceil)\to\tau'.\end{array}$$

ii) Define $\lceil\!\rceil:\Lambda c^{\to}\to\Lambda^{\to}$ as

$$\begin{array}{lcl}\lceil u\rceil & = & u\\ \lceil\,\lambda x:\tau.U\,\rceil & = & \lambda x:\lceil\tau\rceil.\lceil U\rceil\\ \lceil\,U_1U_2\,\rceil & = & \lceil U_1\rceil\lceil U_2\rceil\\ \lceil\,\Lambda\vec{x}:\vec{\tau}.U\,\rceil & = & \lambda\vec{x}:\lceil\vec{\tau}\rceil.\lceil U\rceil\\ \lceil\,U\langle\vec{U}\rangle\,\rceil & = & \lceil U\rceil\lceil U_1\rceil\ldots\lceil U_n\rceil\\ \lceil\,\delta h:[\vec{\tau}]\tau.U\,\rceil & = & \lambda h:\lceil[\vec{\tau}]\tau\rceil.\lceil U\rceil\\ \lceil\,U_1\lceil U_2\rceil\,\rceil & = & \lceil U_1\rceil\lceil U_2\rceil.\end{array}$$

[‡] From the translation of $\Lambda c^{\to}$ into $\Lambda^{\to}$, the natural translation of Λc into Λ (see the concluding sentences in Section 1.2 and the concluding remark in Section 4) can be defined by dropping the (pre)types and considering δ_n and $\lceil\,\rceil_n$ for any n.

iii) Let Σ be a base. Then $\llbracket \Sigma \rrbracket = \{(u : \llbracket \rho \rrbracket) \mid (u : \rho) \in \Sigma\}$.

Proposition 11. If $\Gamma, \Delta \vdash_{\lambda c} U : \rho$ then $\llbracket \Gamma \rrbracket \cup \llbracket \Delta \rrbracket \vdash_{\lambda\to} \llbracket U \rrbracket : \llbracket \rho \rrbracket$.

Proof. By induction to the length of $\Gamma, \Delta \vdash_{\lambda c} U : \rho$. Check the pretyping rules of λc: from the translations of the preconditions, the translations of the postconditions can be derived in $\lambda^{\to}$. Then each derivation step in $\Gamma, \Delta \vdash_{\lambda c} U : \rho$ can be translated to one or more derivation steps in $\lambda^{\to}$. □

Proposition 12. If $\Gamma, \Delta \vdash_{\lambda c} U : \rho$ and $U \to V$ in the simply typed context calculus, then $\llbracket U \rrbracket \twoheadrightarrow \llbracket V \rrbracket$ in the simply typed lambda calculus.

Proof. In general, the λc-rewrite steps are translated to many β-steps, with one exception: a $\textcircled{m}\beta$-step where the multiple abstraction and the multiple application are empty, that is $(\Lambda.U)\langle\rangle \to_{\textcircled{m}\beta} U$, which in translation results in an empty β-step: $\llbracket\, (\Lambda.U)\langle\rangle \,\rrbracket = U = \llbracket U \rrbracket$. □

Proposition 13. (Strong normalization) Reduction in $\lambda c^{\to}$ is strongly normalizing.

Proof. Let $U_0 \in \Lambda c^{\to}$ and suppose r is an infinite rewrite sequence in $\lambda c^{\to}$:

$$r : \quad U_0 \to U_1 \to U_2 \to \ldots \quad \infty.$$

Note indeed that if U_0 is a $\lambda c^{\to}$-term, then so are all its reducts. Then, the translation of U's to the simply typed lambda calculus results in a rewrite sequence $\llbracket r \rrbracket$ in the simply typed lambda calculus:

$$\llbracket r \rrbracket : \quad \llbracket U_0 \rrbracket \twoheadrightarrow \llbracket U_1 \rrbracket \twoheadrightarrow \llbracket U_2 \rrbracket \twoheadrightarrow \ldots \quad \infty.$$

Because there are no infinite rewrite sequences in $\lambda^{\to}$, the tail of $\llbracket r \rrbracket$ must eventually be empty, i.e. $\llbracket U_n \rrbracket \equiv \llbracket U_{n+1} \rrbracket \equiv \ldots$. These steps can only be translations of 'empty' $\textcircled{m}\beta$-steps, i.e. $U_n \equiv C[(\Lambda.U)\langle\rangle] \to_{\textcircled{m}\beta} C[U] \equiv U_{n+1} \ldots$. However, since $\lambda c^{\to}$-terms are finite, there cannot be infinitely many such steps starting from U_n. □

The example described in this section extends the work of M. Hashimoto and A. Ohori [8] in the following sense. It includes multiple occurrences of a hole and drops their condition on the β-rule, by which β-reduction is not allowed within (representations of) λ-contexts. Moreover, $\lambda c^{\to}$ allows composition, which is not present in their system.

We conclude by mentioning a simple variation of this example of pretyping.

Untyped λ-contexts. Let $\mathcal{BT}$ contain only one pretype constant $\mathtt{t}$, consider τ-pretypes modulo $\mathtt{t} \cong \mathtt{t} \to \mathtt{t}$, and add a pretyping rule by which if $\Gamma, \Delta \vdash U : \rho$ then $\Gamma, \Delta \vdash U : \rho'$ for $\rho \cong \rho'$. Such a pretyping describes the *untyped* lambda calculus with the same kind of λ-contexts as $\lambda c^{\to}$, which again has the subject reduction property (the uniqueness of pretyping and strong normalization are lost, as expected). For example, according to this pretyping, $z : \mathtt{t} \vdash \delta h : [\mathtt{t}]\,\mathtt{t}\,.(\lambda x : \mathtt{t}.h\langle x\rangle)(h\langle z\rangle) : [\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}$ and $\vdash (\lambda x : \mathtt{t}.xx)(\lambda x : \mathtt{t}.xx) : \mathtt{t}$. This pretyping essentially has the effect of well-formedness rules on the untyped λ-terms and of typing rules on the contexts and holes, by ignoring the type constructor $\to$. Such a pretyping describes the minimal conditions which guarantee the well-pretypedness of the context machinery and do not take the types of λ-terms into consideration.

6.3. A pretyping for De Bruijn's segments

In this section a pretyping is given for a simply typed lambda calculus with segments. Segments are comparable, in our terminology, to contexts with the hole on the spine (i.e. at the leftmost position in the tree representation), like in $\lambda x.\lambda y.\Box$, $\Box\, y\, x$ and $(\lambda x.\Box)\, y$. Segments were introduced by N.G. de Bruijn in [6], in order to facilitate the use of abbreviations for segments of formulas in the family of proof checkers Automath. In this section, we will first focus on the original calculus and segments in general. Then we will describe the calculus λc^{S} and state some of its properties.

The original calculus is an extension of a name-free untyped lambda calculus. Among the added features are: a symbol for denoting a hole, segment variables (i.e. variables over contexts of a special form), functions over segments and terms, and renamings for adjusting indices (called reference transforming mappings). Each hole is labeled with variables that can bind in the term to be put into the hole. The β-reduction is defined stepwise, as it has become a tradition in explicit substitution calculi (see [1]), where the pair $(\lambda x.\_)t$ traverses through the term or segment s in the redex $(\lambda x.s)t$. In the case that s is a segment and x is in the label of its hole, the traversing pauses at the hole until eventually the hole is filled. The simply typed version of the calculus is a confluent system and has the subject reduction property (see [2] and [3]).

As a whole, the original calculus reflects the very 'Automath-ed', implementation-oriented fashion of the object processing: the objects

are represented as strings of characters and the transformations are implemented by cutting the strings, duplicating parts of the strings, inserting new characters and gluing these parts together again. Moreover, with segment variables and functions over segments, and with β-reduction defined on both terms and segments, this calculus is a true context calculus over contexts of a special form.

An important issue in a lambda calculus with segments in general, is that the specific structure of contexts is preserved under transformations. The preservation of the segment structure means in particular that if a segment is involved in a transformation, the hole on the spine cannot be multiplied nor deleted *within* (the boundaries of) the segment. The preservation of the segment structure under substitution, hole filling, composition and α-reduction, if applicable, can easily be verified. The preservation under β-reduction will now be explained in some detail. Consider a segment C and a β-redex $(\lambda x.s)t$ and distinguish the relative position of C with respect to the redex in a β-step. If the segment C is subsumed by the redex, then the segment can only be manipulated as a whole (in a structure preserving way). If the segment C subsumes the redex, then the hole is either at the spine of the redex or disjoint from the redex. In all these cases the hole is left intact under the contraction of the redex. This is due to the peculiar position of the hole in a segment and it is in general not the case with contexts with one hole at an arbitrary position: if $(\lambda x.yxx)\Box$ is a λ-context, then it reduces to the λ-context $y\Box\Box$, which has two holes.

In this section we give a pretyping for a name-carrying simply typed lambda calculus with segments λc^S. The main consequence of having name-carrying variables is that there is no need for a stepwise β-reduction, as there is in the case of name-free variables: the substitution $[\![x := t]\!]$ that arises from a β-step in $(\lambda x.s)t$ is applied immediately to s.

The representation of the lambda calculus with segments mainly follows the line described in Section 3. In particular, in λc^S segments are represented as abstractions over precisely one hole variable, which occurs at the leftmost position [§]. For instance, the segment $(\lambda x.\Box)y$ is represented as $\delta h.(\lambda x.h\langle x\rangle)y$. The pretypes and pretyping rules resemble the pretypes and pretyping rules of $\lambda c^{\rightarrow}$. The pretypes of λc^S are the τ- and ρ-pretypes of $\lambda c^{\rightarrow}$, with the same conventions and intentions. In λc^S, the τ-pretypes correspond to the frame types of the

[§] The definition of tree representation of λc-terms is a naïve extension of the tree representation of λ-terms, where δ is treated as λ, hole filling as application, Λ as a flattened version of a sequence of λ's and $\langle\,\rangle$ as a flattened version of a sequence of applications.

simply typed version of the original calculus (cf. [2], [6]) and as such they are pretypes of the first class objects of the original calculus. In the pretyping rules there are two bases involved, Γ containing variables over τ-pretypes, and Δ containing *at most one* declaration over hole variables (which is of the form $h : [\vec{\tau}]\tau$).

The pretyping system is given next.

Definition 11. (Pretyping rules for λc^S) A term $U \in \Lambda c$ is pretypable by ρ from the bases Γ, Δ, if $\Gamma, \Delta \vdash U : \rho$ can be derived using the pretyping rules displayed in Figure 5.

We comment on the pretyping rules briefly. The rules (*var*), (*abs*) and (*app*) are the rules comparable to the same rules in the simply typed lambda calculus, but in this pretyping they also range over segments. The rules (*hvar*) and (*habs*) pertain to pretyping hole variables and abstractions over hole variables. The rules (*mabs*) and (*mapp*) are used for pretyping communication. The rules (*fill*) and (*comp*) are used for pretyping hole filling and composition. Note that after the rule (*hvar*), which gives a pretype to a hole variable, the only applicable rule is (*mapp*), by which the hole variable is immediately provided with communication (i.e. the hole variables are labeled). Note also that, similarly to the previous example of pretyping, only the 'unary' δ, and the binary $\lceil\ \rceil$ and $\circ$ are used.

In all rules, the basis Γ is used like a basis in the simply typed lambda calculus. This is not the case with Δ, which strictly follows the hole variable: the basis Δ changes only by the rules (*hvar*) and (*habs*), where the hole variable is introduced or abstracted; it is empty in rules (*fill*), (*comp*), where 'completed' representations of segments are manipulated; and, it is intact in the rest of the rules, where it follows only the left branch of the term when represented as a tree.

Using this pretyping, the straightforward composition *comp* of the concluding remark in Section 4 is not pretypable: in the λc-term $\lambda c.\delta d.\delta g.c\lceil\Lambda\vec{u}.(d\langle\vec{u}\rangle)\lceil g\rceil\rceil$ the variables d and g, which should have pretypes of the form $[\vec{\tau}]\tau$ (i.e. pretypes of hole variables), are not on the spine of a context. Therefore, the composition constructor and pretyping rule (and rewrite rule) are present in the calculus.

Definition 12. (Segment calculus λc^S) The segment calculus is defined as a subsystem of λc on pretypable terms with the following rules:

$$(\lambda u : \tau.U)V \to_\beta U[\![u := V]\!]$$

$$(\Lambda\vec{u} : \vec{\tau}.U)\langle\vec{V}\rangle \to_{\textcircled{m}\beta} U[\![\vec{u} := \vec{V}]\!]$$

$$(var)\quad \frac{(u:\tau)\in\Gamma}{\Gamma\ \vdash\ u:\tau}$$

$$(abs)\quad \frac{\Gamma,u:\tau,\Delta\ \vdash\ U:\tau'}{\Gamma,\Delta\ \vdash\ \lambda u:\tau.U:\tau\to\tau'}$$

$$(app)\quad \frac{\Gamma,\Delta\ \vdash\ U:\tau\to\tau'\quad \Gamma\ \vdash\ V:\tau}{\Gamma,\Delta\ \vdash\ UV:\tau'}$$

$$(hvar)\quad \frac{\{(h:[\vec{\tau}]\tau)\}=\Delta}{\Gamma,\Delta\ \vdash\ h:[\vec{\tau}]\tau}$$

$$(habs)\quad \frac{\Gamma,h:[\vec{\tau}]\tau\ \vdash\ U:\tau'}{\Gamma\ \vdash\ \delta h.U:[\vec{\tau}]\tau\Rightarrow\tau'}$$

$$(fill)\quad \frac{\Gamma\ \vdash\ U:[\vec{\tau}]\tau\Rightarrow\tau'\quad \Gamma\ \vdash\ V:[\vec{\tau}]\tau}{\Gamma\ \vdash\ U\lceil V\rceil:\tau'}$$

$$(mabs)\quad \frac{\Gamma,\vec{u}:\vec{\tau},\Delta\ \vdash\ U:\tau}{\Gamma,\Delta\ \vdash\ \Lambda\vec{u}:\vec{\tau}.U:[\vec{\tau}]\tau}$$

$$(mapp)\quad \frac{\Gamma,\Delta\ \vdash\ U:[\vec{\tau}]\tau\quad \Gamma\ \vdash\ \vec{V}:\vec{\tau}}{\Gamma,\Delta\ \vdash\ U\langle\vec{V}\rangle:\tau}$$

$$(comp)\quad \frac{\Gamma\ \vdash\ U:[\vec{\tau}]\tau\Rightarrow\tau'\quad \Gamma\ \vdash\ V:[\vec{\tau}]([\vec{\sigma}]\sigma\Rightarrow\tau)}{\Gamma\ \vdash\ U\circ V:[\vec{\sigma}]\sigma\Rightarrow\tau'}$$

Figure 5. Pretyping rules for λc^S

$$(\delta u : [\vec{\tau}]\tau.U)\,\lceil V \rceil \to_{fill} U[\![u := V]\!]$$
$$(\delta h : [\vec{\tau}]\tau.U) \circ (\Lambda \vec{v} : \vec{\tau}.\, \delta g : [\vec{\sigma}]\sigma.V) \to_\circ \delta g : [\vec{\sigma}]\sigma.U[\![h := \Lambda \vec{v} : \vec{\tau}.V]\!].$$

The main results are the following.

Proposition 14. (Uniqueness of pretypes) If $\Gamma, \Delta \vdash U : \rho_1$ and $\Gamma, \Delta \vdash U : \rho_2$ then $\rho_1 \equiv \rho_2$.

Proposition 15. (Substitution Lemma)

i) If $\Gamma, \vec{x} : \vec{\tau}, \Delta \vdash U : \rho$ and $\Gamma \vdash \vec{V} : \vec{\tau}$ then $\Gamma, \Delta \vdash U[\![\vec{x} := \vec{V}]\!] : \rho$.

ii) If $\Gamma, h : [\vec{\tau}]\tau \vdash U : \rho$ and $\Gamma \vdash V : [\vec{\tau}]\tau$ then $\Gamma \vdash U[\![h := V]\!] : \rho$.

iii) If $\Gamma, h : [\vec{\tau}]\tau \vdash U : \rho$ and $\Gamma, g : [\vec{\sigma}]\sigma \vdash V : [\vec{\tau}]\tau$ then $\Gamma, g : [\vec{\sigma}]\sigma \vdash U[\![h := V]\!] : \rho$.

Proposition 16. (Subject reduction) If $\Gamma, \Delta \vdash U : \rho$ and $U \twoheadrightarrow V$, then $\Gamma, \Delta \vdash V : \rho$.

Proposition 17. (Strong normalization) Reduction in λc^S is strongly normalizing.

The proofs of the unicity of pretyping, subject reduction and strong normalization are conducted in a standard way, resembling the proofs in $\lambda c^{\to}$. Only in the case of substitution, care has to be taken when dealing with Δ, especially in the third case.

Note that the subject reduction property implies not only the pretypablity of the reduct but also the preservation of the segment structure. That is, one can prove that in the λc^S-terms of pretype $[\vec{\tau}]\tau \Rightarrow \tau'$ (i.e. representations of segments) the hole occurs at the leftmost position. Then, the preservation of pretyping under reduction implies the preservation of the structure of segment representation.

A variation of this pretyping is a pretyping of untyped lambda calculus with segments, which can be accomplished by allowing pretypes over only one constant $\mathtt{t}$ and considering the pretypes modulo equality $\mathtt{t} \cong \mathtt{t} \to \mathtt{t}$, like in the previous example of pretyping.

Digression. The distinguished position of the hole in a segment has an interesting consequence on the type of a segment in a typed version of the calculus. Because the hole is positioned at the end of the spine, these contexts have a polymorphic type in the following sense. In general, the form of a context C restricts the type of terms t that may be placed into the hole, but the type of $C[t]$ eventually depends on the type of t. Take, for example, the λ-context $C \equiv \lambda x : \tau.\Box\, x\, x$. If a λ-term t is to be placed into the hole, its type should be an arrow type $\tau \to \tau \to \alpha$,

where α denotes an arbitrary type, and the type of the result $C[t]$ is then $\tau \to \alpha$. Furthermore, this context can bind a variable of type τ in the term placed into the hole, because the hole is in the scope of $(\lambda x : \tau)$-binder. Then, the type of C is $[\tau](\tau \to \tau \to \alpha) \Rightarrow (\tau \to \alpha)$, for an arbitrary type α. However, for the sake of simplicity of the example, we treated here only a restricted version of this pretyping where holes have a fixed pretype. A polymorphic pretyping of a calculus with segments is given in [5].

7. Conclusion and future work

This paper presents a uniform framework, the context calculus λc, for representing multiform lambda calculus contexts. The context calculus can be enhanced with pretypings, which restrict the form of λc-terms. Three examples of pretyping have been given. It has been indicated that the context calculus satisfies the confluence property and that all three pretypings satisfy the subject reduction property.

Our plans for future work encompass the following.

First, we intend to design a context calculus parametrized over the object-language, that is, a context calculus parametrized over an arbitrary signature and additional rewrite rules over the signature. In the case of λc, the object-language is the lambda calculus and the meta-language is the rest of λc: the context-related constructors (Λ, $\langle\rangle$, δ, $\lceil\rceil$ and $\circ$) and the context rewrite rules (Definition 2 (ii)). We consider λc to be a good case study for designing such a general context calculus, since we believe the meta-language of λc can be reused in combination with an arbitrary object-language.

Second, we plan to add labels to our communication mechanism in order to avoid the ordering of arguments, that is to allow accessing the arguments by a label as in $(\Lambda x_1^{a_1}, \ldots, x_n^{a_n}.U)\langle V_1^{b_1}, \ldots, V_n^{b_n}\rangle$ where $b_1 \ldots b_n$ is a permutation of $a_1 \ldots a_n$.

Next, we intend to explore the possibility of using pretyped segment-like contexts in automated reasoning for representing mathematical structures, like relations, monoids, groups etc. Loosely speaking, a mathematical structure can be represented as a context where the 'body' is a sequence of abstractions. This is illustrated by the example of reflexive binary relations, which we treated in Section 2.2:

$$\lambda A : Set.\lambda R : (A \to A \to Set).\lambda rfl : (\forall x : A.Rxx).\Box.$$

Such a representation could for example be used to support internal definitions of subtyping.

Acknowledgements

We would like to thank Vincent van Oostrom for suggesting several improvements.

References

1. Abadi, M., L. Cardelli, P.-L. Curien, and J.-J. Lévy: 1991, 'Explicit Substitutions'. *Journal of Functional Programming* **1**(4), 375–416.
2. Balsters, H.: 1987, 'Lambda calculus extended with segments'. In: *Mathematical logic and theoretical computer science (College Park, Md., 1984–1985)*. New York: Dekker, pp. 15–27.
3. Balsters, H.: 1994, '*Lambda calculus extended with segments*: Chapter 1, Sections 1.1 and 1.2 (Introduction)'. In: R. P. Nederpelt, J. H. Geuvers, and R. C. d. Vrijer (eds.): *Selected papers on Automath*. Amsterdam: North-Holland, pp. 339–367.
4. Barendregt, H.: 1984, *The Lambda Calculus, its Syntax and Semantics*, Vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, revised edition. (Second printing 1985).
5. Bognar, M. and R. d. Vrijer: 1999, 'Segments in the context of contexts'. Preprint, Vrije Universiteit Amsterdam.
6. Bruijn, N. d.: 1978, 'A namefree lambda calculus with facilities for internal definition of expressions and segments'. Technical Report 78-WSK-03, Technological University Eindhoven.
7. Despeyroux, J., F. Pfenning, and C. Schürmann: 1997, 'Primitive recursion for higher-order abstract syntax'. In: *Typed lambda calculi and applications (Nancy, 1997)*. Berlin: Springer, pp. 147–163.
8. Hashimoto, M. and A. Ohori: 1998, 'A typed context calculus'. *Sūrikaisekikenkyūsho Kōkyūroku* **1023**, 76–91. Type theory and its application to computer systems (Japanese) (Kyoto, 1997).
9. Kahrs, S.: 1993, 'Context rewriting'. In: *Conditional term rewriting systems (Pont-à-Mousson, 1992)*. Berlin: Springer, pp. 21–35.
10. Klop, J.: 1980, *Combinatory Reduction Systems*, Mathematical Centre Tracts Nr. 127. Amsterdam: CWI. PhD Thesis.
11. Kohlhase, M., S. Kuschert, and M. Müller: 1999, 'Dynamic lambda calculus'. Preprint, available at `http://www.ags.uni-sb.de/~kohlhase`.
12. Lee, S.-D. and D. Friedman: 1996, 'Enriching the lambda calculus with contexts: Toward a theory of incremental program construction'. In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*. pp. 239–250.
13. Magnusson, L.: 1996, 'An algorithm for checking incomplete proof objects in type theory with localization and unification'. In: *Types for proofs and programs (Torino, 1995)*. Berlin: Springer, pp. 183–200.
14. Mason, I. A.: 1999, 'Computing with Contexts'. *Higher-Order and Symbolic Computation* **12**, 171–201.
15. Muñoz, C.: 1997, 'Dependent Types with Explicit Substitutions: A meta-theoretical development'. In: *Proceedings of the International Workshop TYPES '96*.

16. Nederpelt, R. P., J. H. Geuvers, and R. C. d. Vrijer: 1994, *Selected Papers on Automath*, Vol. 133 of *Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland.
17. Nipkow, T.: 1993, 'Orthogonal Higher-Order Rewrite Systems are Confluent'. In: *Proceedings of the International Conference on Typed Lambda Calculi and Application*. pp. 306–317.
18. Oostrom, V. v.: 1995, 'Development Closed Critical Pairs'. In: *Proceedings of the 2nd International Workshop on Higher-Order Algebra, Logic, and Term Rewriting (HOA '95)*, Vol. 1074 of *Lecture Notes in Computer Science*. Springer, pp. 185–200.
19. Oostrom, V. v. and F. v. Raamsdonk: 1993, 'Comparing Combinatory Reduction Systems and Higher-order Rewrite Systems'. Technical Report CS-R9361, CWI. Extended abstract in Proceedings of HOA'93.
20. Pfenning, F. and C. Elliott: 1988, 'Higher-order abstract syntax'. In: *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. pp. 199–208.
21. Sands, D.: 1998, 'Computing with contexts, a simple approach'. *Electronic Notes in Theoretical Computer Science* **10**.
22. Sato, M., T. Sakurai, and R. Burstall: 1999, 'Explicit environments'. In: J.-Y. Girard (ed.): *Proceedings of the 4th International Conference on Typed Lambda Calculi and Application*. pp. 340–354.
23. Talcott, C. L.: 1991, 'Binding Structures'. In: V. Lifschitz (ed.): *Artificial Intelligence and Mathematical Theory of Computation*.

Address for Offprints:
Vrije Universiteit
Department of Theoretical Computer Science
de Boelelaan 1081a
1081 HV Amsterdam
The Netherlands