The context cube $\lambda[\,]$: Barendregt's lambda cube with contexts

Mirna Bognar *　　Roel de Vrijer *

Abstract

We present the context cube $\lambda[]$, a system of eight context calculi corresponding to Barendregt's lambda cube with contexts. In the context cube, the expressions and meta-contexts of the lambda cube can be represented. Moreover, filling of the holes of a context is a rewrite relation.

The novelty of the context cube is that it also deals with contexts with dependent types. The simplicity of the context cube lies in the fact that it can be translated into the lambda cube. Rewriting in the context cube is confluent and strongly normalising.

Furthermore, we illustrate how a mathematical structure can be represented as a context within the context cube, in such a way that it can be used, for example, in proofs. Our representations are basically de Bruijn's segments, but with holes in types to obtain a uniform representation.

1 Introduction

Contexts are expressions with special places, called holes, where other expressions may be placed. In the lambda calculus, $(\lambda x.\,[])z$ where $[]$ denotes a hole, is an example of a context. In formal systems with binders, 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 xz into the hole of the context above results in the expression $(\lambda x.\,xz)z$, 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. That implies that contexts are treated merely as a notation, which hinders any formal reasoning about contexts. 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.

Formalisation of contexts is motivated in different realms of research as diverse as proof checking, programming languages operational semantics and natural languages. Contexts are formalised with the purpose of facilitating definitions and manipulation of abbreviations in Automath (see [Bru78, Bal87, Bal86, NGdV94]), optimalisation of interactive proof checking (see [Mag96]), representing incomplete proofs and supporting incremental proof development and higher-order unification (see [Muñ97]), for the development of programming languages (see [HO98, SSB99]), for dealing with contexts in operational semantics (see [San98, Mas99]), modeling binding mechanisms in natural language (see [KKM99]). Other contributions to the formalisation of contexts can be found in [Tal91, Kah93, LF96].

We present here the context cube, or the $\lambda[\,]$-cube for short, which is a collection of eight context calculi related to the eight systems of Barendregt's lambda cube with contexts. Because

* *Vrije Universiteit, Division of Mathematics and Computer Science, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*, `mirna@cs.vu.nl, rdv@cs.vu.nl`

meta-contexts for the lambda cube are difficult to describe by using informal notation as in the lambda calculus, indirectly, the $\lambda[\,]$-cube defines a notion of contexts for the lambda cube. The novelty of the $\lambda[\,]$-cube is that it deals with context *calculi* with dependent types, where hole filling is defined as a rewrite relation. Furthermore, the context cube includes context calculi dealing with dependent types. The simplicity of the $\lambda[\,]$-cube lies in the fact that it can be translated into the lambda cube. Rewriting in the $\lambda[]$-cube is confluent and strongly normalising. The context cube builds upon the context calculus λc, which was presented in [BdV01]; a more extensive exposition with proofs can be found in the forthcoming thesis of the first author [Bog02].

This paper is structured as follows. Section 2 discusses the problems when defining and formalising meta-contexts for the systems of the lambda cube. Section 3 is a gentle introduction to the context cube. It describes what the context cube is and the approach to formalisation of meta-contexts in the context cube. Section 4 contains the definition of the expressions, rewrite relations and typing in the systems of the context cube. Section 5 looks into the properties of the context cube and establishes the correspondence between the systems of the context cube and the systems of the lambda cube. Section 6 illustrates how the context cube can be employed for representing de Bruijn's segments with 'polymorphic types'. Segments are a special kind of contexts that can be used for representing mathematical structures in proof checking. In general, in this paper we concentrate on the expressivity related to contexts, and do not consider the expressive power of different systems of the $\lambda[\,]$-cube, because the latter are basically the same as in the λ-cube. Section 7 concludes this paper with a couple of suggestions for future work.

2 The lambda cube and meta-contexts

The lambda cube, or the λ-cube for short, which was introduced by H.P. Barendregt, comprises eight systems of typed lambda calculi, which are presented in a uniform way. For the details of the lambda cube, we refer to [Bar92].

We only remark here that the legal expressions in the systems of the λ-cube can be seen as stratified into five levels: the level of sorts, the level of types, the level of kinds, the level of elements of types and the level of elements of kinds. Moreover, on *legal* expressions, β-reduction can be seen as being restricted to the level of elements (of types or kinds), because β-redexes $(\lambda x : A.\, b)a$ are elements: by using the Generation and Substitution Lemmas for the λ-cube, there exist Γ, B and s with $\Gamma \vdash (\lambda x : A.\, b) \; : \; (\Pi x : A.\, B) \; : \; s$, $\Gamma \vdash a \; : \; A$ and $\Gamma \vdash (\lambda x : A.\, b)a \; : \; B[\![x := a]\!] \; : \; s$. With types and kinds possibly depending on elements, redexes may *occur* in types and kinds too. Again, the positions of redexes in a type or kind (and, in general, in any expression) are restricted by the fact that redexes are elements. This restriction will be used later in our approach to formalisation of contexts.

For the definition of the λ-cube and investigation of the properties of the λ-cube, meta-contexts are not needed. Accordingly, in [Bar92] there is no definition of meta-contexts. However, an attempt to give a definition of meta-contexts using informal notation as in the lambda calculus, fails. We claim here that, in order to define meta-contexts in the λ-cube in the way analogous to the definition of meta-contexts in the untyped lambda calculus, one needs a better representation of holes together with communication. By communication we mean here the communication between (reducts of) context C and expression M, which comprises establishing intended variable capturing and passing on imminent substitutions to M that emerge from earlier α-conversion and β-reduction within the context C or from substitutions applied on C.

We set out the assumptions regarding meta-contexts that seem natural, and describe the problem. The assumptions are the following.

i) In the λ-cube the attention is focused on legal expressions. Then it seems natural to define (legal) meta-contexts, informally, as legal expressions with holes. In the sequel, in the informal notation holes will be denoted by $[]^A$ where A is the type or kind of the hole.

ii) The notion of legal expressions is defined by the typing rules. Then, with meta-contexts defined as legal expressions with holes, it seems natural to define the notion of meta-contexts by using the same typing rules. In the typing derivations, it seems natural to treat holes as constants, that is, as free variables which are not allowed to be abstracted.

iii) In general, hole filling is an integral part of the definition of meta-contexts. As in the case of the untyped lambda calculus, hole filling is, informally, a literal replacement of a hole by a legal expression of the same type or kind as the hole.

With these assumptions, meta-contexts and hole filling are not well-defined. The problem lies in the fact that the typing rules employ substitution and β-conversion, while communication of a hole is not preserved under these transformations. This is illustrated by the example given below.

Example 2.1 Consider the typing statement $B : * \vdash (\lambda A : *. \lambda x : []^*. x)B : (\Pi x : []^*. []^*)$ and a fragment of its derivation:

$$\frac{B : * \vdash (\lambda A : *. \lambda x : []^*. x) : (\Pi A : *. \Pi x : []^*. []^*) \qquad B : * \vdash B : *}{B : * \vdash (\lambda A : *. \lambda x : []^*. x)B : (\Pi x : []^*. []^*)}$$

The hole of the element and the hole of the type originate from the same hole: the type of x. Therefore, they should be considered as occurrences of the same hole, and accordingly, filled by the same expression. However, while filling the hole of the element and the hole of the type by A where $A : *$ is 'legal' in the assumption, it results in an incorrect typing statement in the conclusion: $(\lambda A : *. \lambda x : A. x) : (\Pi A : *. \Pi x : A. A)$ but not $(\lambda A : *. \lambda x : A. x)B : (\Pi x : A. A)$. The problem arises because in this derivation step, the information $[\![A := B]\!]$ is not recorded by the hole in the type.

Hence, in order to define meta-contexts, a notation for holes with explicitly denoted communication is necessary.

Furthermore, a formalisation of contexts includes also (at least) extending the definition of α-conversion and β-reduction to contexts and defining hole filling. In the case of lambda calculus, it is well-known that α-conversion and β-reduction do not 'commute' with filling of holes (see for example [BdV01, HO98, San98]).

3 Introduction to the context cube

We describe here the context cube informally; the definition is given in the next section.

A collection of context calculi related to the λ-cube. The $\lambda[\,]$-cube defines a collection of eight context calculi. These context calculi extend the eight systems of the λ-cube with contexts. The $\lambda[\,]$-cube provides an adequate notation for meta-contexts of the λ-cube, where β-reduction on contexts is gained for free.

Context calculi. On (representation of) contexts, also the standard relations and operations are defined: typing, substitution, α-conversion, and β-reduction. Furthermore, the systems of the $\lambda[\,]$-cube include variables over (representations of) contexts and functions over (representation of) contexts.

We stress that context-related rewriting in the $\lambda[\,]$-cube is defined on the same level where β-rewriting is allowed in the λ-cube, namely on the level of elements. This is the level where the problem of commutation between β-reduction and hole filling occurred in the informal notation. Incidentally, the consequence of this choice is that we stay within the framework of the λ-cube.

The notion of contexts. Indirectly, the $\lambda[\,]$-cube defines a notion of contexts for the λ-cube. It formalises contexts with many holes, which in turn may occur an arbitrary number of times. A hole may occur at any position where otherwise a variable may occur. The holes of a context are filled sequentially.

Typing. The typing relation in each system of the $\lambda[\,]$-cube makes the assumptions listed above. Moreover, the typing relation will satisfy the following: if C is a context of type or kind T, then

$C[\vec{a}]$ is of type or kind $T[\vec{a}]$ for suitable expressions $\vec{a}$. This restriction filters out typing statements like $(\lambda x : []^{*}. x) : (\Pi x : A. A)$, where the typing relation assumes more information about filling of holes than available in the expression and its type or kind.

The typing relation in each system of the $\lambda[\,]$-cube is defined in such a way that the levels are preserved: if a meta-context of the λ-cube is on the level of elements, types or sorts, then its representation within the $\lambda[\,]$-cube is at the same level.

Properties. Technically, we arrive at a very simple extension of the λ-cube. The syntax of the expressions is extended with new abstractions and applications related to representing contexts. These new expression constructors can be understood as macros over the syntax of the λ-cube. In particular, each system of the $\lambda[\,]$-cube can be translated into the corresponding system of the λ-cube. That means that the whole $\lambda[\,]$-cube can be encoded into the λ-cube.

The $\lambda[\,]$-cube is rather expressive. For example, as we will show later, 'polymorphic' de Bruijn's segments with dependent types can be represented within the $\lambda[\,]$-cube.

Representing the lambda cube with contexts. We give the intuition behind the representation method; see also Example 4.5 and [BdV01].

- Legal expressions of the λ-cube: These expressions can be represented directly in the $\lambda[\,]$-cube, because each system $\lambda[S]$ of the $\lambda[\,]$-cube is an extension of the corresponding system λS of the λ-cube.

- Holes: Holes are represented by hole variables $h, g, H, \ldots$ applied to a sequence of expressions $\vec{N}$ that keep track of the relevant α, β-changes in the context. So, holes are represented as $h\langle\vec{N}\rangle$, where $\_\langle\_\rangle$ denotes multiple application.

- Communication: In addition to multiple application, which is used in the representation of communication around the holes, we introduce the multiple abstractions Λ and $\underline{\Lambda}$ for representing the expressions that are to be put into holes and their types or kinds. In general, if M is an expression that is intended to be put into a hole and where $\vec{x}$ are intended to become bound upon filling, and if T is the type of M, then M and T are represented in the $\lambda[\,]$-cube as $\Lambda\vec{x}{:}\vec{A}.\,M$ and $\underline{\Lambda}\vec{x}{:}\vec{A}.\,T$ respectively, and the intention is that $\Lambda\vec{x}{:}\vec{A}.\,M$ is of type $\underline{\Lambda}\vec{x}{:}\vec{A}.\,T$. Upon filling the expression into a hole $h\langle\vec{N}\rangle$, that is upon replacing the hole variable by the expression, commutation can be computed by a multiple β-rewrite step:

$$(\Lambda\vec{x}{:}\vec{A}.\,M)\langle\vec{N}\rangle \to M\,[\![\vec{x} := \vec{N}]\!]\,.$$

  There is an analogy to techniques used in higher-order rewriting (see for example [Klo80, Nip93, OR93]), and in the field of higher-order abstract syntax (see for example [PE88, DPS97]), where a similar binding effect is encoded by a substitution calculus.

  It may be noted that an alternative approach to the problem of communication could make use of explicit substitutions. Pure type systems with explicit substitutions have been studied in for example [Blo99].

- Contexts, hole filling: Contexts are considered as functions over the possible contents of its holes; that is, contexts are functions over hole variables. The abstractor for the hole variables is denoted by δ. Hole variables of a context will be abstracted sequentially. In general, if C is a context of type T_C with n holes (holes are represented as above), then C is represented as $\delta h_1{:}T_1.\,\ldots\delta h_n{:}T_n.\,C$ and its type is represented as $\underline{\delta} h_1{:}T_1.\,\ldots\underline{\delta} h_n{:}T_n.\,T_C$. In general, in (the representation of) types and kinds, hole variables are abstracted by $\underline{\delta}$. The intention is again that $\delta h_1{:}T_1.\,\ldots\delta h_n{:}T_n.\,C$ is of type $\underline{\delta} h_1{:}T_1.\,\ldots\underline{\delta} h_n{:}T_n.\,T_C$.

  Hole filling is then represented sequentially as follows. If $\delta h{:}T.\,U$ is a context and if $\Lambda\vec{x}{:}\vec{A}.\,M$ is a communicating expression, then hole filling is represented as $(\delta h{:}T.\,U)\lceil\Lambda\vec{x}{:}\vec{A}.\,M\rceil$ and computed by a version of the β-rewrite rule:

$$(\delta h{:}T.\,U)\lceil\Lambda\vec{x}{:}\vec{A}.\,M\rceil \to U\,[\![h := \Lambda\vec{x}{:}\vec{A}.\,M]\!]\,.$$

If $\underline{\delta}h{:}T.T_U$ is the type of the context $\delta h{:}T.U$, then the type of $(\delta h{:}T.U)\lceil \Lambda\vec{x}{:}\vec{A}.M\rceil$ is $T_U[\![h := \Lambda\vec{x}{:}\vec{A}.M]\!]$.

Composition can be encoded within the $\lambda[\,]$-cube.

- Sorts: Using the constructors of the $\lambda[\,]$-cube, we will represent terms and types, as in the λ-cube, and moreover, contexts and communicating terms. Contexts will not be considered as a new sort of expressions, because contexts are terms or types with holes. Hence, the dependencies between contexts, terms and types can be expressed by $*$ and $\Box$ in a way that is analogous to controlling[1] the dependencies in the λ-cube.

  However, communicating expressions will be considered as new sorts of expressions, because such expressions are combined with contexts, terms and types in a different way. For example, a context may depend on a communicating expression, but a communicating expression may only depend on terms and types. For denoting the sorts of communicating expressions two sorts are introduced: $\triangle_*$, which denotes communicating types, and $\triangle_\Box$, which denotes communicating kinds.

In sum, the elements are treated as in the context calculus λc (see [BdV01]); the abstractors $\underline{\Lambda}$ and $\underline{\delta}$ are introduced for representing the types and kinds (with holes). Holes in types and kinds are filled by substitution, filling of holes in elements can be denoted and its computation can be postponed. This explanation will also be visible in the typing rules.

4 Definition of the context cube

This section presents the definition of expressions, rewriting and typing of the eight systems of the context cube.

Let $\mathcal{S} = \{*, \Box, \triangle_*, \triangle_\Box\}$ be the set of sorts. Let $\mathbb{V}$ be a set of variables.

Definition 4.1 (Pseudo-expressions) Let $s \in \mathcal{S}$ and $x, \vec{x} \in \mathbb{V}$. The set of pseudo-expressions A of the $\lambda[\,]$-cube is defined inductively by

$$\begin{array}{rllllll} A & ::= & s \mid & x \mid & (\lambda x : A.\,A) \mid & (A\,A) \mid & (\Pi x : A.\,A) \mid \\ & & (\Lambda_n \vec{x} : \vec{A}.\,A) \mid & (A\,\langle\vec{A}\rangle_n) \mid & (\underline{\Lambda}_n \vec{x} : \vec{A}.\,A) \mid & (\delta x : A.\,A) \mid & (A\,\lceil A\rceil) \mid \quad (\underline{\delta} x : A.\,A) \end{array}$$

where $|\vec{x}| = |\vec{A}| = n$.

Notation. As usual, standard abbreviations regarding brackets apply, including the association to the left. Also, standard abbreviations for sequences of n pseudo-expressions as $\vec{A}$ and sequences of n variables as $\vec{x}$ apply. When convenient, consecutive abstractions $(\lambda x_1 : A_1.\ \ldots (\lambda x_n : A_n.\,M))$ and $(\Pi x_1 : A_n.\ \ldots (\Pi x_n : A_n.\,M))$ will be abbreviated by $\lambda\vec{x} : \vec{A}.\,M$ and $\Pi\vec{x} : \vec{A}.\,M$, respectively, and moreover, if $A_1 \equiv \ldots \equiv A_n$, then we write $\lambda\vec{x} : A.\,M$ and $\Pi\vec{x} : A.\,M$, respectively. In the multiple abstractions Λ_n and $\underline{\Lambda}_n$, if $n = 0$ then we write $\Lambda\epsilon.\,M$ and $\underline{\Lambda}\epsilon.\,A$. Furthermore, we will omit the index n of Λ, $\underline{\Lambda}$, and $\langle\rangle$, and assume that the arities of these constructors and the number of their arguments match. Arbitrary expressions will be denoted by A,B,C,.... We will distinguish between the variables for holes and other variables: the hole variables will be denoted by $h, g, k, h_1, H, G\ldots$; other variables will be denoted by $x, y, x', x_1, \alpha, \gamma, \ldots$.

Substitution over pseudo-expressions is defined in a standard way; we skip the definition here.

On pseudo-expressions, rewrite relations are defined via rewrite rules. The rewrite rules are split into the standard β-reduction and the context-related rewriting. The context-related rewriting comprises communication and hole filling.

[1] Recall that in the λ-cube the sort $*$ contains types, i.e. $\tau : *$ amounts to saying that τ is a type, and that the sort $\Box$ contains *kinds*, i.e. $\kappa : \Box$ amounts to saying that κ is a kind. In the λ-cube the dependencies that are allowed in a particular system are expressed by $\mathcal{R} \subseteq \{*, \Box\} \times \{*, \Box\}$. The pair (s_1, s_2) denotes the dependency of the elements of sort s_2 on the elements of sort s_1. For example, the pair $(*, \Box)$ denotes that elements of kinds may depend on elements of types.

Definition 4.2 (Rewrite rules) The rewrite relations of the $\lambda[\,]$-cube are induced on pseudo-expressions by two collections of rewrite rules, the lambda calculus rewrite rule and the context rewrite rules. The two collections of rewrite rules are given below.

i) The lambda calculus rewrite rule is:

$$(\lambda x : A.\, B)C \;\to_\beta\; B[\![x := C]\!] \qquad (\beta)$$

ii) The context rewrite rules are:

$$(\Lambda\vec{x} : \vec{A}.\, B)\langle\vec{C}\rangle \;\to_{ⓜ\beta}\; B[\![\vec{x} := \vec{C}]\!] \qquad (ⓜ\beta)$$

$$(\delta h : A.\, B)\lceil C\rceil \;\to_{\mathit{fill}}\; B[\![h := C]\!] \qquad (\mathit{fill})$$

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

The context rewrite rules can be understood as instances of the β-rewrite rule, because they too are application–abstraction elimination rules. The rule (ⓜβ) is a multiple version of the β-rule and the rule (*fill*) is a version of the β-rule dealing with an abstraction over a hole variable.

The systems of the context cube are presented in a uniform way, analogously to the definition of λ-cube (see [Bar92]). The eight systems of the $\lambda[\,]$-cube are $\lambda[\to]$, $\lambda[2]$, $\lambda[P]$, $\lambda[P2]$, $\lambda[\underline{\omega}]$, $\lambda[\omega]$, $\lambda[P\underline{\omega}]$, and $\lambda[C]$. The names for these systems are derived from the names of the systems of the λ-cube. As we will show later, system $\lambda[S]$ of the $\lambda[\,]$-cube corresponds to the system λS of the λ-cube with contexts.

Each of the eight systems is defined on legal expressions with respect to a particular collection of the typing rules. The typing rules are divided into the general typing rules and the specific typing rules. The general typing rules are the rules that are present in each of the eight systems; these rules guide the formation of elements, allow for conversion in their types and kinds, and guide the formation of legal bases. The specific typing rules control the formation of types and kinds according to the allowed dependencies between terms and types[2] in a particular system of the $\lambda[\,]$-cube. The dependencies that are allowed in a particular system are expressed by $\mathcal{R} \subseteq \{*, \Box\}\times\{*, \Box\}$ and each system includes a different subset of the specific typing rules. Hence, the specific rules are parametrised by $\mathcal{R}$.

The eight systems of the $\lambda[\,]$-cube and the dependencies $\mathcal{R}$ allowed in a particular system are displayed in Table 1. The dependencies $\mathcal{R}$ of a system $\lambda[S]$ are the same as the dependencies of the system λS of the λ-cube.

system $\lambda[S]$	parametrised by $\mathcal{R}$			
$\lambda[\to]$	$(*,*)$			
$\lambda[2]$	$(*,*)$	$(\Box,*)$		
$\lambda[P]$	$(*,*)$		$(*,\Box)$	
$\lambda[P2]$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda[\underline{\omega}]$	$(*,*)$			$(\Box,\Box)$
$\lambda[\omega]$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda[P\underline{\omega}]$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
$\lambda[C]$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

Table 1: The systems of the $\lambda[\,]$-cube

Statements, declarations and pseudo-bases are defined analogously to the corresponding notion in the λ-cube. A *statement* is of the form $A : B$ with $A, B \in \mathcal{T}$. A *declaration* is a statement of the form $x : A$ where $x \in \mathcal{V}$. A *pseudo-basis* is a finite, ordered sequence of declarations with distinct variables. The ordering is a consequence of the fact that if $x_1 : A_1, \ldots, x_n : A_n$ is a pseudo-basis, then A_j may depend on x_i for i and j with $1 \leq i < j \leq n$. The empty basis is denoted by $\emptyset$.

[2]Here we mean, terms and types in the terminology of, for example, the simply typed lambda calculus.

Definition 4.3 (Typing) Let $\lambda[S]$ be a system of the $\lambda[\,]$-cube, parametrised by $\mathcal{R}$. Let A and B be pseudo-expressions and let Γ be a pseudo-basis. Then $A : B$ can be derived from the pseudo-basis Γ, denoted by $\Gamma \vdash_{\lambda[S]} A : B$, if $\Gamma \vdash A : B$ can be derived using $\mathcal{R}$ and the typing rules displayed in Figure 1.

Definition 4.4 (The $\lambda[\,]$-cube) Each of the eight systems of the $\lambda[\,]$-cube are defined as follows.
Let $\lambda[S]$ be one of the systems of the $\lambda[\,]$-cube listed in Table 1. Let $\lambda[S]$ be parametrised by $\mathcal{R}$. Then, the system $\lambda[S]$ consists of the legal expressions as defined by the typing rules of Figure 1 with respect to $\mathcal{R}$, and of the rewrite relations of Definition 4.2.

Notation. If the system is irrelevant, the index $\lambda[S]$ in $\Gamma \vdash_{\lambda[S]} A : B$ is omitted. Furthermore, $\Gamma \vdash A : B : C$ is a shorthand for the two derivations $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$.

The terminology regarding typing is similar to the same terminology in the λ-cube. A *typing statement* is of the form $\Gamma \vdash A : B$. A pseudo-basis Γ is called a *(legal) basis* if $\exists P, Q \in \mathcal{T}\ (\Gamma \vdash P : Q)$. An expression A is called a *(legal) expression* if $\exists \Gamma, B\ (\Gamma \vdash A : B$ or $\Gamma \vdash B : A)$. An expression A is called a *kind* if there is a basis Γ such that $\Gamma \vdash A : \Box$. An expression A is called a *communicating kind* if there is a basis Γ such that $\Gamma \vdash A : \triangle_\Box$. An expression A is called a *type* if there is a basis Γ such that $\Gamma \vdash A : *$. An expression A is called a *communicating type* if there is a basis Γ such that $\Gamma \vdash A : \triangle_*$. An expression A is called an *element* if $\exists B \in \mathcal{T} \exists s \in \mathcal{S}\ (\Gamma \vdash A : B : s)$.

We comment on the typing system. The typing rules of the $\lambda[\,]$-cube include the typing rules of the λ-cube. These are the general rules (*axiom*), (*var*), (*weak*), (*conv*), (*abs*) and (*app*), and the specific rule (Π). Here, in the $\lambda[\,]$-cube, these rules are of course adapted for dealing also with new sorts of expressions: the sort s in (*var*), (*weak*) and (*conv*) may also be $\triangle_*$ or $\triangle_\Box$; by the rules (*var*) and (*weak*) both term variables and hole variables may be introduced; and in the conversion $B = B'$ of the rule (*conv*) also the context-related rewriting may be employed.

In addition to the typing rules of the λ-cube, the $\lambda[\,]$-cube contains new typing rules which deal with formation of new sorts of expressions. The additional rules are the general rules (*mabs*), (*mapp*), (*habs*), and (*fill*), and the specific rules ($\underline{\Lambda}$) and ($\underline{\delta}$). The rules (*mabs*) and (*mapp*) deal with communication: they guard the typing of communication around holes and the typing of elements to be put into holes. The rules (*habs*) and (*fill*) deal with formation of contexts and filling of holes of a context.

By the rules (*mabs*) and (*habs*), the abstraction of variables is reflected in both the element and its type or kind: in the case of (*habs*), if $\Gamma, h : A \vdash b : B$ then $\Gamma \vdash (\delta h : A.\, b) : (\underline{\delta} h : A.\, B)$. This is comparable to typing of λ-abstractions by the rule (*abs*). By the rules (*mapp*) and (*fill*), the 'application' is postponed in the element but it is computed immediately in its type or kind: in the case of (*fill*), for example, if $\Gamma \vdash a : A$ and $\Gamma \vdash (\delta h : A.\, b) : (\underline{\delta} h : A.\, B)$ then $\Gamma \vdash (\delta h : A.\, b)\lceil a \rceil : B[\![x := a]\!]$. This is comparable to typing of applications by the rule (*app*). By these rules a (β-, *m*β- or *fill*-)redex is formed only in the element. Hence, in general, although a redex may also occur in a type of a kind, the redex itself is an element.

The specific rules ($\underline{\Lambda}$) and ($\underline{\delta}$) govern the formation of communicating types and communicating kinds, and the formation of the types and kinds of contexts, respectively. They are parametrised by the allowed dependencies in a particular system, which are expressed by $\mathcal{R}$. The requirement in the rule ($\underline{\Lambda}$) in case $|\vec{x}| = 0$, that there is an s' with $(s', s) \in \mathcal{R}$ ensures that no abstractions are allowed over expressions over which otherwise no abstractions are allowed. This condition excludes expressions like $\Lambda\epsilon.\, A$ and $\underline{\Lambda}\epsilon.\, *$ with $A : *$, if otherwise also $\Lambda x : B.\, A$ and $\underline{\Lambda} x : B.\, *$ are not allowed for any B (i.e. if there is no $(s, \Box)$ in $\mathcal{R}$).

By (thoroughly) inspecting the typing rules of the $\lambda[\,]$-cube, one may see that, as in the case of the λ-cube, legal expressions are stratified into levels. Because of the new sorts in the $\lambda[\,]$-cube, its level structure is more involved than in the λ-cube:

element : type : $*$	element : communicating type : $\triangle_*$
element : kind : $\Box$	element : communicating kind : $\triangle_\Box$.

Here, as in the case of the λ-cube, the levels of elements on one hand, and the levels of types and kinds on the other hand overlap.

We give an example which illustrates the expressive power of the $\lambda[\,]$-cube. It treats meta-contexts of the λ-cube: the examples considered can be understood as translations of the examples of meta-contexts. Furthermore, this example shows that the $\lambda[\,]$-cube has a greater expressivity than just representing meta-contexts. In the $\lambda[\,]$-cube also functions over (representations of) contexts can be represented, as well as intermediate results of context-related computation. At the end of this paper, we will work out a more involved example dealing with de Bruijn's segments with dependent types and their representation within the $\lambda[\,]$-cube.

Example 4.5 We consider some legal expressions of the $\lambda[\,]$-cube, which can be understood as representations of (legal) meta-contexts of the λ-cube. Throughout this example the expressions of the $\lambda[\,]$-cube will be 'translated' into informal, inadequate but more intuitive quasi-notation of the λ-cube, which uses $[]^D$ for denoting holes of type or kind D.

i) All legal expressions of the λ-cube can directly be represented within the $\lambda[\,]$-cube. See Lemma 5.7.

ii) The identity function over type A is a legal expression:

$$A : * \;\vdash\; (\lambda x{:}A.\, x) \;:\; (A{\to}A) \;:\; *.$$

iii) Consider the following examples:

$$\begin{array}{rl} A : *, B : * \vdash & (\Lambda x{:}A, y{:}B.\, x) \;:\; (\underline{\Lambda} x{:}A, y{:}B.\, A) \;:\; \triangle_* \\ A : *, B : * \vdash & (\Lambda x{:}A, y{:}B.\, A) \;:\; (\underline{\Lambda} x{:}A, y{:}B.\, *) \;:\; \triangle_\Box \\ \emptyset \vdash & (\Lambda A{:}*, x{:}A.\, x) \;:\; (\underline{\Lambda} A{:}*, x{:}A.\, A) \;:\; \triangle_*. \end{array}$$

These examples involve elements that represent terms and types to be placed into a hole of a context. The element $\Lambda x{:}A, y{:}B.x$ can be understood as the term x which is to be placed into a hole in the scope of an x-abstractor and a y-abstractor; hence the communication preamble $\Lambda x{:}A, y{:}B$. Due to possible dependencies in the types as in the third example, the type of $\Lambda x{:}A, y{:}B.x$ is the $\underline{\Lambda}$-abstraction $\underline{\Lambda} x{:}A, y{:}B.\,A$. Analogously, the statements $(\Lambda x{:}A, y{:}B.\,A) : (\underline{\Lambda} x{:}A, y{:}B.*)$ and $(\Lambda A{:}*, x{:}A.\,x) : (\underline{\Lambda} A{:}*, x{:}A.\,A)$ can be explained. We mention only that we now deal with also communicating types (for example, $\Lambda x{:}A, y{:}B.\,A$), and with types in communication (for example, A in $\Lambda A{:}*, x{:}A.\,x$).

iv) Hole variables are of a communicating type or of a communicating kind; for example:

$$\emptyset \;\vdash\; h \;:\; (\underline{\Lambda} A{:}*\,.\,\Pi x{:}A.\,*) \;:\; \triangle_\Box.$$

v) The example

$$\emptyset \;\vdash\; (\underline{\delta} h{:}(\underline{\Lambda} A'{:}*\,.\,*).\,\Pi A{:}*\,.\,\Pi x{:}\,h\langle A\rangle\,.\,*) \;:\; \Box$$

can be understood as a representation of the meta-context $\Pi A{:}*\,.\,\Pi x{:}[]^*.\,*$. The hole variable in $\Pi A{:}*\,.\,\Pi x{:}\,h\langle A\rangle\,.\,*$ is abstracted by a $\underline{\delta}$-abstraction because $\Pi A{:}*\,.\,\Pi x{:}[]^*.\,*$ is a kind with a hole.

vi) The example

$$\emptyset \;\vdash\; (\underline{\delta} h{:}(\underline{\Lambda}\epsilon.\,*).\,h\langle\rangle) \;:\; *$$

can be understood as a representation of the meta-context $[]^*$. Because the meta-context $[]^*$ is considered here as a type with a hole, its representation is a $\underline{\delta}$-abstraction. Note that the communication of the hole $[]^*$ is void: hence, the 'empty' multiple abstraction $\underline{\Lambda}\epsilon.*$ as the type of h and the 'empty' multiple application $h\langle\rangle$ as the representation of the hole.

General axiom and rules:

$$(axiom) \qquad \emptyset \vdash * : \square$$

$$(var) \qquad \frac{\Gamma \vdash A : s}{\Gamma, u : A \vdash u : A} \quad \text{with } s = *, \square, \triangle_*, \triangle_\square$$

$$(weak) \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, u : C \vdash A : B} \quad s = *, \square, \triangle_*, \triangle_\square$$

$$(conv) \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \quad B = B'}{\Gamma \vdash A : B'}$$

$$(abs) \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A. B) : s}{\Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B)}$$

$$(app) \qquad \frac{\Gamma \vdash F : (\Pi x : A. B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\, a : (B[\![x := a]\!])}$$

$$(mabs) \qquad \frac{\Gamma, \vec{x} : \vec{A} \vdash b : B \qquad \Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}. B) : \triangle_s}{\Gamma \vdash (\Lambda\vec{x} : \vec{A}. b) : (\underline{\Lambda}\vec{x} : \vec{A}. B)}$$

$$(mapp) \qquad \frac{\begin{array}{c}\Gamma \vdash F : (\underline{\Lambda}\vec{x} : \vec{A}. B) \\ \Gamma \vdash a_i : A_i[\![x_1 := a_1, ..., x_{i-1} := a_{i-1}]\!]\end{array}}{\Gamma \vdash F\langle\vec{a}\rangle : (B[\![\vec{x} := \vec{a}]\!])} \quad \text{for } 1 \leq i \leq |\vec{a}| = |\vec{A}|$$

$$(habs) \qquad \frac{\Gamma, h : A \vdash b : B \qquad \Gamma \vdash (\underline{\delta}h : A. B) : s}{\Gamma \vdash (\delta h : A. b) : (\underline{\delta}h : A. B)}$$

$$(\mathit{fill}) \qquad \frac{\Gamma \vdash F : (\underline{\delta}h : A. B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\lceil a\rceil : B[\![h := a]\!]}$$

Specific rules:

$$(\Pi) \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A. B) : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{R}$$

$$(\underline{\Lambda}) \qquad \frac{\begin{array}{c}\Gamma, x_1 : A_1, \ldots, x_i : A_i \vdash A_{i+1} : s_{i+1} \\ \Gamma, \vec{x} : \vec{A} \vdash B : s\end{array}}{\Gamma \vdash (\underline{\Lambda}\vec{x} : \vec{A}. B) : \triangle_s} \quad \begin{array}{l}\text{if } (s_i, s) \in \mathcal{R} \\ \text{for all } 1 \leq i \leq |\vec{x}|, \\ \text{and, in case } |\vec{x}| = 0, \\ \text{if } \exists s' \text{ with } (s', s) \in \mathcal{R}\end{array}$$

$$(\underline{\delta}) \qquad \frac{\Gamma \vdash A : \triangle_{s_1} \qquad \Gamma, h : A \vdash B : s_2}{\Gamma \vdash (\underline{\delta}h : A. B) : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{R}$$

Figure 1: The type system for the context cube

vii) An example involving hole abstractions is

$$\emptyset \;\vdash\; (\delta h{:}(\underline{\Lambda}\epsilon.\,*).\,\lambda x{:}\, h\langle\rangle\,.\,x) \;:\; (\underline{\delta} h{:}(\underline{\Lambda}\epsilon.\,*).\,\Pi x{:}\, h\langle\rangle\,.\,h\langle\rangle) \;:\; *.$$

This typing statement can be understood as a representation of the typing statement $\emptyset \vdash (\lambda x{:}[]^*.\,x) : (\Pi x{:}[]^*.\,[]^*) : *$.

Note that the meta-context $\lambda x{:}[]^*.\,x$ (thus also, the expression $\delta h{:}(\underline{\Lambda}\epsilon.\,*).\,\lambda x{:}\, h\langle\rangle\,.\,x$) can be treated as a 'polymorphic' version of the identity function: by introducing a hole in a type, one implicitly states that 'for all types fitting the hole' $\lambda x{:}[]^*.\,x$ is of type $\Pi x{:}[]^*.\,[]^*$.

viii) An example of a function over meta-contexts is the function that takes a 'polymorphic' identity function as an argument and instantiates it to the identity function over A:

$$A : * \;\vdash\; (\lambda c{:}(\underline{\delta} h{:}(\underline{\Lambda}\epsilon.\,*).\,\Pi x{:}\, h\langle\rangle\,.\,h\langle\rangle).\,c\,\lceil\Lambda\epsilon.\,A\rceil) \;:\; (\Pi c{:}(\underline{\delta} h{:}(\underline{\Lambda}\epsilon.\,*).\,\Pi x{:}\, h\langle\rangle\,.\,h\langle\rangle).\,\Pi x{:}A.\,A).$$

ix) The meta-context of Example 2.1 can be represented as

$$B : * \vdash (\delta h{:}(\underline{\Lambda} A{:}\,*\,.*).\,(\lambda A{:}\,*\,.\,\lambda x{:}\, h\langle A\rangle\,.\,x)B)\,\lceil\Lambda A{:}\,*\,.\,A\rceil \;:\; (\Pi x{:}B.\,B).$$

Consider the fragment of its derivation where $B : *, h : (\underline{\Lambda} A{:}\,*\,.*) \vdash B : *$:

$$\frac{B : *, h : (\underline{\Lambda} A{:}\,*\,.*) \;\vdash\; (\lambda A{:}\,*\,.\,\lambda x{:}\, h\langle A\rangle\,.\,x) \;:\; (\Pi A{:}\,*\,.\,\Pi x{:}\, h\langle A\rangle\,.\,h\langle A\rangle)}{B : *, h : (\underline{\Lambda} A{:}\,*\,.*) \;\vdash\; (\lambda A{:}\,*\,.\,\lambda x{:}\, h\langle A\rangle\,.\,x)B \;:\; (\Pi x{:}\, h\langle B\rangle\,.\,h\langle B\rangle)}$$

Note that the typing is preserved (modulo context-related rewriting) under the substitution $[\![h := \Lambda A{:}\,*\,.\,A]\!]$. The reduct of the element $(\lambda A{:}\,*\,.\,\lambda x{:}\, h\langle A\rangle\,.\,x)B$ is of the same type

$$B : *, h : (\underline{\Lambda} A{:}\,*\,.*) \;\vdash\; (\lambda B{:}\,*\,.\,\lambda x{:}\, h\langle B\rangle\,.\,x) \;:\; (\Pi x{:}\, h\langle B\rangle\,.\,h\langle B\rangle).$$

Furthermore, one may check that all rewrite sequences starting from the expression $(\delta h{:}(\underline{\Lambda} A{:}\,*\,.*).\,(\lambda A{:}\,*\,.\,\lambda x{:}\, h\langle A\rangle\,.\,x)B)\,\lceil\Lambda A{:}\,*\,.\,A\rceil$ end in $\lambda x{:}B.\,x$.

x) By introducing a hole in the type of an another hole, the latter hole becomes 'polymorphic'. Consider the previous example with a hole in the type of h:

$$A : *, a : A, H : (\underline{\Lambda} x{:}A.\,*) \vdash \quad (\delta h{:}(\underline{\Lambda} x{:}A.\,H\langle x\rangle).\,h\langle a\rangle) \;:\; (\underline{\delta} h{:}(\underline{\Lambda} x{:}A.\,H\langle x\rangle).\,H\langle a\rangle) \;:\; *.$$

See also Section 6 where such 'polymorphic' holes are used for representing segments and mathematical structures.

5 Properties of the context cube

In this section, we investigate the properties of the $\lambda[\,]$-cube. Technically speaking, the $\lambda[\,]$-cube is a simple extension of the λ-cube, which can easily be translated back into the λ-cube. Hence, the properties regarding typing and rewriting of the systems of the $\lambda[\,]$-cube can be stated and proved in the similar way as for the (corresponding) systems of the λ-cube.

Remark 5.1 The proofs of the statements on typing are performed in the same way as in the λ-cube. Such a piece of technical luck is a consequence of the fact that the $\lambda[\,]$-cube can be seen as a macro over the λ-cube. In particular, the $\lambda[\,]$-cube does not introduce (β-)rewriting on a level where otherwise no β-rewriting is allowed in the λ-cube. This is comparable to the situation in [SP94], and contrary to the situation in for example [KBN99] and some Automath systems (see for example [NGdV94]), where due to the presence of applicative expressions on the level of types and kinds other, more powerful techniques in proofs are used.

The underlying calculi of the $\lambda[\,]$-cube and the λ-cube. We consider the underlying calculus of the $\lambda[\,]$-cube. We define a translation from the underlying calculus of the $\lambda[\,]$-cube into the underlying calculus of the λ-cube. We also show that the translation preserves rewriting between pseudo-expressions. One can also show that the underlying calculus of the $\lambda[\,]$-cube is confluent.

Definition 5.2 (Translation to the calculus of the λ-cube)

i) Define the translation function $\ulcorner\_\urcorner$ from the pseudo-expressions of the $\lambda[\,]$-cube into the pseudo-expressions of the λ-cube as

$$
\begin{array}{ll}
\ulcorner s \urcorner & = \begin{cases} s & \text{if } s \in \{*, \square\} \\ s' & \text{if } s \equiv \triangle_{s'} \end{cases} \\
\ulcorner u \urcorner & = u \\
\ulcorner \lambda x : A.\, B \urcorner & = \lambda x : \ulcorner A \urcorner .\, \ulcorner B \urcorner \\
\ulcorner A B \urcorner & = \ulcorner A \urcorner \ulcorner B \urcorner \\
\ulcorner \Pi x : A.\, B \urcorner & = \Pi x : \ulcorner A \urcorner .\, \ulcorner B \urcorner \\
\ulcorner \Lambda \vec{x} : \vec{A}.\, B \urcorner & = \lambda x_1 : \ulcorner A_1 \urcorner . \ldots \lambda x_n : \ulcorner A_n \urcorner .\, \ulcorner B \urcorner \\
\ulcorner A \langle \vec{B} \rangle \urcorner & = \ulcorner A \urcorner \ulcorner B_1 \urcorner \ldots \ulcorner B_n \urcorner \\
\ulcorner \underline{\Lambda} \vec{x} : \vec{A}.\, B \urcorner & = \Pi x_1 : \ulcorner A_1 \urcorner . \ldots \Pi x_n : \ulcorner A_n \urcorner .\, \ulcorner B \urcorner \\
\ulcorner \delta h : A.\, B \urcorner & = \lambda h : \ulcorner A \urcorner .\, \ulcorner B \urcorner \\
\ulcorner A \lceil B \rceil \urcorner & = \ulcorner A \urcorner \ulcorner B \urcorner \\
\ulcorner \underline{\delta} h : A.\, B \urcorner & = \Pi h : \ulcorner A \urcorner .\, \ulcorner B \urcorner .
\end{array}
$$

ii) Let Γ be a pseudo-basis. Then $\ulcorner \Gamma \urcorner = \{(u : \ulcorner A \urcorner) \mid (u : A) \in \Gamma\}$.

Lemma 5.3 *Let A and B be two pseudo-expressions of the $\lambda[\,]$-cube. If $A \to B$ in the $\lambda[\,]$-cube, then $\ulcorner A \urcorner \twoheadrightarrow \ulcorner B \urcorner$ in the λ-cube.*

Proof: The proof is conducted by induction to the structure of A. More specifically, the following can be shown: (i) if $A \to_\beta B$ then $\ulcorner A \urcorner \to_\beta \ulcorner B \urcorner$; (ii) if $A \to_{\mathit{fill}} B$ then $\ulcorner A \urcorner \to_\beta \ulcorner B \urcorner$; and (iii) if $A \to_{\textcircled{m}\beta} B$ then $\ulcorner A \urcorner \twoheadrightarrow_\beta \ulcorner B \urcorner$. In the third case, the translation of a $\textcircled{m}\beta$-step is empty if the redex which is contracted is of the form $(\Lambda\epsilon.\, b)\langle\rangle$. ▯

Typing properties of the systems of the $\lambda[\,]$-cube. Each system of the $\lambda[\,]$-cube has the subject reduction property. This is proved in an analogous way as for the λ-cube; we state here only two results.

Lemma 5.4 (Substitution) *If $\Gamma, \vec{x} : \vec{A}, \Gamma' \vdash b : B$ and if $\Gamma \vdash a_i : A_i[\![x_1{:=}a_1, \ldots, x_{i-1}{:=}a_{i-1}]\!]$ for $1 \le i \le |\vec{a}|$, then $\Gamma, (\Gamma'[\![\vec{x} := \vec{a}]\!]) \vdash (b[\![\vec{x} := \vec{a}]\!]) : (B[\![\vec{x} := \vec{a}]\!])$.*

Lemma 5.5 *Let $\Gamma \vdash A : B$. Then there is a sort s such that $B \equiv s$ or $\Gamma \vdash B : s$.*

Proof: The proof is done by induction to the length of derivation $\Gamma \vdash A : B$. The proof also uses the Generation lemma and the Substitution lemma. ▯

Lemma 5.6 (Subject reduction) *If $\Gamma \vdash A : B$ and $A \to A'$ then $\Gamma \vdash A' : B$.*

Typing in $\lambda\square$ is not unique, because β-conversions is allowed in types. For example, if $\Gamma = \{A : *, x : A\}$ then both $\Gamma \vdash x : A$ and $\Gamma \vdash x : (\lambda B : *.\, B) A$.

The $\lambda[\,]$-cube and the λ-cube. The correspondence between the $\lambda[\,]$-calculus and the λ-cube can be summarised by two sentences: (i) the system λS of the λ-cube can trivially be translated into the system $\lambda[S]$ of the $\lambda[\,]$-cube; (ii) the system $\lambda[S]$ of the $\lambda[\,]$-cube can be translated into the system λS of the λ-cube. The translation of the system λS into the system $\lambda[S]$ is the identity function. The translation of the system $\lambda[S]$ into the system λS is given by Definition 5.2. Both translations preserve the notion of legal expressions and of legal bases. Moreover, both translations preserve rewrite steps on legal expressions.

Lemma 5.7 *i) If* $\Gamma \vdash_{\lambda S} A : B$*, then* $\Gamma \vdash_{\lambda[S]} A : B$*.*

ii) If A *is a legal* λS*-expression and* $A \to_\beta A'$ *in* λS*, then* $A \to_\beta A'$ *in* $\lambda[S]$*.*

Proof: The proof of the first part follows trivially from the fact that the typing rules of the λ-cube are contained in the typing rules of the $\lambda[\,]$-cube, and from the fact that the system λS of the λ-cube and the system $\lambda[S]$ are parametrised by the same set of rules $\mathcal{R}$.

For the proof of the second part, note that, by the first part, A is also a legal $\lambda[S]$-expression. Then, the statement follows from the fact that the β-rewrite rule is also a rewrite rule of the system $\lambda[S]$ and from the subject reduction property of rewriting in the system $\lambda[S]$. ▯

Lemma 5.8 *i) If* $\Gamma \vdash_{\lambda[S]} A : B$ *in the* $\lambda[\,]$*-cube, then* $\ulcorner\Gamma\urcorner \vdash_{\lambda S} \ulcorner A\urcorner : \ulcorner B\urcorner$ *in the* λ*-cube.*

ii) If A *is a legal* $\lambda[S]$*-expression and* $A \to A'$*, then* $\ulcorner A\urcorner \twoheadrightarrow \ulcorner A'\urcorner$ *in the system* λS*.*

Proof: The proof of the first part is done by translating the typing rules: it can be shown that from the translations of the premisses, the translations of the conclusions can be derived in λS. The proof uses also Lemma 5.3 to show that if $B = B'$ in the $\lambda[\,]$-cube, then $\ulcorner B\urcorner = \ulcorner B'\urcorner$ in the λ-cube.

The second part is a corollary of the first part of this lemma, Lemma 5.3 and the closure property of rewriting in the system λS (i.e. the subject reduction property of λS). ▯

Rewriting properties of the systems of the $\lambda[\,]$-cube. Rewriting in each of the systems of the $\lambda[\,]$-cube is confluent and strongly normalising. The confluence proof is a standard one; the proof of strong normalisation property is conducted via the translation into the λ-cube and the strong normalisation property of rewriting in the λ-cube.

Theorem 5.9 (Confluence) *Rewriting in each system of the* $\lambda[\,]$*-cube is confluent.*

Proof: The proof relies on the subject reduction property (Lemma 5.6) and the confluence property of the underlying calculus of the $\lambda[\,]$-cube. ▯

Theorem 5.10 (Strong normalisation) *Rewriting in any system of the* $\lambda[\,]$*-cube is strongly normalising.*

Proof: The proof is conducted via translation to the lambda cube. One shows that, if there is an infinite rewrite sequence in a system $\lambda[S]$, then this rewrite sequence can be translated into an infinite rewrite sequence of the corresponding system λS of the lambda cube. However, since there are no infinite rewrite sequences in λS, there are no such rewrite sequences in $\lambda[S]$ either. ▯

6 Applications: segments and mathematical structures

In this section we look more closely into the application of meta-contexts in proof checking and into the representation of this application within the $\lambda[\,]$-cube. It concerns the usage of meta-contexts for representing mathematical structures. The meta-contexts employed are in fact 'polymorphic' de Bruijn's segments (see [Bru78, Bru91, Bal86]) with dependent types.

Segments are contexts with the hole on the spine (i.e. at the leftmost position in the tree representation), like in the lambda terms $\lambda x.\lambda y.[]$, $[]yx$ and $(\lambda x.[])y$. The type of such a context is 'polymorphic' in the sense that the form of a segment C restricts the type of terms M that may be placed into the hole, but the type of the result of the hole filling, denoted by $C[M]$, eventually depends on the type of M. For example, if $C \equiv \lambda x:\tau.[]xx$ and if a λ-term M is to be placed into the hole, then its type should be an arrow type $\tau \to \tau \to \alpha$, where α denotes an arbitrary type, and the type of the result $C[M]$ is $\tau \to \alpha$.

When representing mathematical structures as segments, it is crucial that the types of holes are 'polymorphic', because only in this way a uniform representation can be obtained. With holes of 'polymorphic' types, such representations of mathematical structures can be used for different purposes, like for example in proofs of different lemmas involving the structure in question.

The application of meta-contexts in proof checking is illustrated by an example. From the mathematical point of view, the example treats a reflexive-euclidic relation and shows that such a relation is transitive. From the technical point of view, the example shows how mathematical structures can be represented as meta-contexts, how such representations can be combined to form representations of bigger mathematical structures, and how such representations can be used in proofs. We will conclude by considering by way of an example representation of 'polymorphic' de Bruijn's segments with dependent types in general within the $\lambda[\,]$-cube.

Sometimes we will write x^A in the abstractions instead of $x : A$.

Representing the example in the $\lambda[\,]$-cube. All the expressions considered here are of course legal. Moreover, because all of the expressions are closed, they can be derived with an empty basis. For the sake of readability, the type of the variables in abstractions will be left out if it is clear from the surrounding text what the types should be.

In general, loosely speaking, a mathematical structure can be represented as a context where the 'body' is a sequence of abstractions. In particular, a binary relation and the properties of the relation being reflexive and euclidic can be represented as the expressions $\underline{rel}$ and $\underline{re}$ in the $\lambda[\,]$-cube, where

$$\begin{array}{rcl} \underline{rel} & \equiv & \delta H.\, \delta h.\, \lambda S.\, \lambda R.\, h\langle S, R\rangle \\ & : & \underline{\delta} H^{(\underline{\Lambda} S':*,R':(S'\to S'\to *).*)}.\, \underline{\delta} h^{(\underline{\Lambda} S':*,R':(S'\to S'\to *).H\,\langle S',R'\rangle)}.\, \Pi S^*.\, \Pi R^{(S\to S\to *)}.\, H\,\langle S, R\rangle \end{array}$$

and

$$\begin{array}{rcl} \underline{re} & \equiv & \Lambda S, R.\, \delta G.\, \delta g.\, \lambda \underline{ref}.\, \lambda \underline{euc}.\, g\,\langle S, R, \underline{ref}, \underline{euc}\rangle \\ & : & \underline{\Lambda} S^*, R^{(S\to S\to *)}. \\ & & \underline{\delta} G^{(\underline{\Lambda} S':*,R':(S'\to S'\to *),r:(\Pi x:S'.\,R'xx),e:(\Pi x,y,z:S'.\,R'xy\to R'xz\to R'yz).*)}. \\ & & \underline{\delta} g^{(\underline{\Lambda} S':*,R':(S'\to S'\to *),r:(\Pi x:S'.\,R'xx),e:(\Pi x,y,z:S'.\,R'xy\to R'xz\to R'yz).\,G\,\langle S',R',r,e\rangle)}. \\ & & \Pi \underline{ref}^{(\Pi x:S.\,Rxx)}.\, \Pi \underline{euc}^{(\Pi x,y,z:S.\,Rxy\to Rxz\to Ryz)}.\; G\,\langle S, R, \underline{ref}, \underline{euc}\rangle\,. \end{array}$$

The novelty of this representation is the additional hole that captures the 'polymorphic' type of the hole of a segment. For example in $\underline{rel}$, the additional hole H enables us to say, informally, that the type of h is 'polymorphic', viz. $h : (\underline{\Lambda} S':*, R':(S\to S\to *).\,[])$. Note that this representation of segments is not a segment again, because it contains two holes.

Note also that, whenever the representation of a mathematical structure is used, first the type of this hole h has to be instantiated for the special purpose in question.

We proceed with the example. The composition of $\underline{rel}$ and $\underline{re}$ is represented by the expression $\underline{comp2}\ \underline{rel}\ \underline{re}$, where $\underline{comp2}$ is designed specifically for $\underline{rel}$ and $\underline{re}$:

$$\underline{comp2} \equiv \lambda c.\, \delta d.\, \delta K.\, \delta k.\, c\,\lceil \Lambda S, R.\, \Pi \underline{ref}.\, \Pi \underline{euc}.\, K\,\langle S, R, \underline{ref}, \underline{euc}\rangle\rceil\,\lceil \Lambda S, R.\, d\langle S, R\rangle\,\lceil K\rceil\,\lceil k\rceil\rceil$$

where $K : (\underline{\Lambda} S', R', r, e.\, *)$ and $k : (\underline{\Lambda} S', R', r, e.\, K\,\langle S', R', r, e\rangle)$. Note that the notion of compositions captured by $\underline{comp2}$ is specifically designed for this kind of 'polymorphic' segments.

The type $\underline{trans\_lemma}$ and its element $\underline{proof}$ represent the statement that a reflexive-euclidic relation is transitive and its proof and they are built using the expressions defined above:

$$\begin{array}{rl} & \underline{proof} \\ =_{def} & \underline{comp2}\ \underline{rel}\ \underline{re}\quad \lceil \Lambda S, R, \underline{ref}, \underline{euc}.\, \Pi a, b, c{:}S.\, Rab\to Rbc\to Rac\rceil \\ & \lceil \Lambda S, R, \underline{ref}, \underline{euc}.\, \lambda a, b, c{:}S.\, \lambda p{:}Rab.\, \lambda q{:}Rbc.\quad \underline{euc}\, bac(\underline{euc}\, abap(\underline{ref}\, a))q\rceil \\ : & \Pi S^*.\, \Pi R^{(S\to S\to *)}.\, \Pi \underline{ref}^{(\Pi x:S.Rxx)}.\, \Pi \underline{euc}^{(\Pi x,y,z:S.Rxy\to Rxz\to Ryz)}.\, \Pi a, b, c{:}S.\, Rab\to Rbc\to Rac \\ =_{def} & \underline{trans\_lemma}. \end{array}$$

De Bruijn's segments with polymorphic types in the $\lambda[\,]$-cube. The examples above involved in fact telescopes, that is, segments without applications, which are of the form $\lambda\vec{x} : \vec{A}.\,[]$. Here we consider an example of a segment with applications.

$$\dfrac{\dfrac{\vdots}{\Gamma_1 \vdash h\langle A,P\rangle \ : \ (\Pi x^A . H\langle A,P,x\rangle)} \qquad \dfrac{\vdots}{\Gamma_1 \vdash (fA) \ : \ A}}{\Gamma_1 =_{def} \Gamma_0, A : *, P : A \to * \vdash h\langle A,P\rangle(fA) \ : \ H\langle A,P,(fA)\rangle}\ (app)$$

$$\dfrac{\dfrac{\vdots}{\Gamma_0 \vdash (\lambda A^*. \lambda P^{A\to *}. h\langle A,P\rangle(fA)) \ : \ (\Pi A^*. \Pi P^{A\to *}. H\langle A,P,(fA)\rangle)} \qquad \dfrac{\vdots}{\Gamma_0 \vdash B : *}}{\Gamma_0 \vdash (\lambda A^*. \lambda P^{A\to *}. h\langle A,P\rangle(fA))B \ : \ (\Pi P^{B\to *}. H\langle B,P,(fB)\rangle)}\ (app)$$

with

$$\Gamma_0 \ =_{def} \ \{B : *, f : (\Pi C^*. C), H{:}(\underline{\Lambda} A^*, P^{A\to *}, x^A. *), h{:}(\underline{\Lambda} A^*, P^{A\to *}. \Pi x^A. H\langle A,P,x\rangle)\}$$

Figure 2: A fragment of a derivation

Let $S \equiv (\lambda A{:}*\,.\,\lambda P{:}(A \to *).\,[\,](f\,A))\,B$ where $B : *$ and $f : (\Pi C : *. C)$. This segment is in the $\lambda[\,]$-cube represented as

$$\begin{array}{l}
B : *, f : (\Pi c{:}*\,.\,C) \vdash \\
\delta H^{(\underline{\Lambda} A:*, P:(A\to *), x:A.\, *)}.\, \delta h^{(\underline{\Lambda} A:*, P:(A\to *).\, \Pi x:A.\, H\langle A,P,x\rangle)}.\, (\lambda A^*. \lambda P^{A\to *}. h\langle A,P\rangle(f\,A))\,B \\
: \underline{\delta} H^{(\underline{\Lambda} A:*, P:(A\to *), x:A.\, *)}.\, \underline{\delta} h^{(\underline{\Lambda} A:*, P:(A\to *).\, \Pi x:A.\, H\langle A,P,x\rangle)}.\ \Pi P^{B\to *}. H\langle B,P,(fB)\rangle\ .
\end{array}$$

Figure 2 is a fragment of its derivation which shows how the type of the segments keeps track of the applications in the segment.

7 Future work

We set out two ideas for future work. The first idea concerns adding labels in communication in order to allow for a more flexible way of communication between a context on the one hand, and terms and contexts to be put into its holes on the other hand. That is, as in

$$(\Lambda x_1^{a_1}, \ldots, x_m^{a_m}. M)\langle N_1^{b_1}, \ldots, N_n^{b_n}\rangle \to M[\![x_1 := N_{l_1}, \ldots, x_m := N_{l_m}]\!],$$

where the labels of x_i and V_{l_i} match for $1 \leq i \leq m \leq n$. By using labels the ordering of arguments $\vec{x}^{\vec{a}}$ and $\vec{N}^{\vec{b}}$ in communication can be avoided, or the arguments $\vec{N}^{\vec{b}}$ may be selected (if $a_1 \ldots a_m$ is a prefix of a permutation of $b_1 \ldots b_n$).

The second idea concerns an internal definition of subtyping. As we have seen in Section 6, contexts can be used in automated reasoning for representing mathematical structures, like relations, monoids, groups etc. Then, the aim is to define subtyping so that for example, a group can be used whenever a monoid is expected, because each group is a monoid. We believe that communication labels can effectively be employed to implement a notion of subtyping.

References

[Bal86] H. Balsters. *Lambda calculus extended with segments.* PhD thesis, Eindhoven University of Technology, 1986. Also in [NGdV94].

[Bal87] H. Balsters. Lambda calculus extended with segments. In *Mathematical logic and theoretical computer science (College Park, Md., 1984–1985)*, pages 15–27. Dekker, New York, 1987.

[Bar92] H.P. Barendregt. Lambda calculi with types. In D. Gabbay S. Abramski and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[BdV01] M. Bognar and R. de Vrijer. A calculus of lambda calculus contexts. *Journal of Automated Reasoning*, 27(1):29–59, July 2001.

[Blo99] R. Bloo. Pure type systems with explicit substitution. preprint available at `http://www.win.tue.nl/~bloo`, 1999.

[Bog02] M. Bognar. *Contexts.* PhD thesis, Vrije Universiteit, 2002. Preprint, available at `http://www.cs.vu.nl/~mirna/book.html`.

[Bru78] N.G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report 78-WSK-03, Technological University Eindhoven, 1978.

[Bru91] N.G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189–204, 1991.

[DPS97] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Typed lambda calculi and applications (Nancy, 1997)*, pages 147–163. Springer, Berlin, 1997.

[HO98] M. Hashimoto and A. Ohori. A typed context calculus. *Sūrikaisekikenkyūsho Kōkyūroku*, 1023:76–91, 1998. Type theory and its application to computer systems (Japanese) (Kyoto, 1997).

[Kah93] S. Kahrs. Context rewriting. In *Conditional term rewriting systems (Pont-à-Mousson, 1992)*, pages 21–35. Springer, Berlin, 1993.

[KBN99] F. Kamareddine, R. Bloo, and R. Nederpelt. On pi-conversion in the lambda-cube and the combination with abbreviations. *Annals of Pure and Applied Logic*, 97:27–45, 1999.

[KKM99] M. Kohlhase, S. Kuschert, and M. Müller. Dynamic lambda calculus. Preprint, available at `http://www.ags.uni-sb.de/~kohlhase`, 1999.

[Klo80] J.W. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts Nr. 127. CWI, Amsterdam, 1980. PhD Thesis.

[LF96] S.-D. Lee and D. Friedman. 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*, pages 239–250, 1996.

[Mag96] L. Magnusson. An algorithm for checking incomplete proof objects in type theory with localization and unification. In *Types for proofs and programs (Torino, 1995)*, pages 183–200. Springer, Berlin, 1996.

[Mas99] I. A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, 12:171–201, 1999.

[Muñ97] C. Muñoz. Dependent types with explicit substitutions: A meta-theoretical development. In *Proceedings of the International Workshop TYPES '96*, 1997.

[NGdV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, 1994.

[Nip93] T. Nipkow. Orthogonal Higher-Order Rewrite Systems are Confluent. In *Proceedings of the International Conference on Typed Lambda Calculi and Application*, pages 306–317, 1993.

[OR93] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. Technical Report CS-R9361, CWI, 1993. Extended abstract in Proceedings of HOA'93.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[San98] D. Sands. Computing with contexts, a simple approach. *Electronic Notes in Theoretical Computer Science*, 10, 1998.

[SP94] Paula Severi and Erik Poll. Pure Type Systems with Definitions. In A. Nerode and Yu. V. Matiyasevich, editors, *Logical Foundations of Computing Science, LFCS'94*, number 813 in LNCS, pages 316–328. Springer, 1994.

[SSB99] M. Sato, T. Sakurai, and R. Burstall. Explicit environments. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Application*, pages 340–354, 1999.

[Tal91] C. L. Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.