The context calculus λc

Extended Abstract

Mirna Bognar * Roel de Vrijer *

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 so-called pretyping. By way of an example we treat the contexts of Hashimoto & Ohori.

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 λ-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 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. 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.

The starting point of our research has been De Bruijn's calculus of segments, which was proposed in the context of Automath. More in general, the increasing interest in contexts has its motivation from many directions, as diverse as modelling programs and program environments, 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 formalisation of contexts has been the subject of various papers; we list a few from the problem areas just mentioned.

* *Vrije Universiteit, Department of Theoretical Computer Science, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands,* mirna@cs.vu.nl, rdv@cs.vu.nl

In [Bru78], De Bruijn introduced a λ-calculus extended with incomplete terms of a special form, called *segments*. The purpose of segments was facilitating definitions and manipulation of abbreviations in the proof-checkers family Automath (see [NGV94]). 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 [Bal87, Bal94] Balsters gave a simply typed version of the segment calculus and proved confluence and subject reduction. A characterisation of segments by means of the context calculus is given in Bognar & de Vrijer [BV99].

With the development of programming languages in mind, Hashimoto and Ohori (see [HO98]) 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. Sato, Sakurai and Burstall (see [SSB99]) 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.

With the purpose of modelling binding mechanisms in natural language, Kohlhase, Kuschert and Müller (see [KKM99]) 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.

Although emerging from different fields of research, with different motivations, the problem of formalising 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, where variable capturing is accomplished by a substitution calculus. (Refer e.g. to Nipkow [Nip93], van Oostrom & van Raamsdonk [OR93].) Similar techniques are used in the field of higher-order abstract syntax, see e.g. Pfenning & Elliott [PE88], Despeyroux, Pfenning & Schürmann [DPS97]. There have been earlier proposals to use these techniques for the formalization of contexts by, among others, Talcott [Tal91] and Sands [San98].

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. The mechanism of pretyping is demonstrated in Section 5 on a specific form of lambda calculus contexts.

2 Informal notions of context in the λ-calculus

In this section we describe contexts as they are encountered in the λ-calculus literature, usually as an informal notational device. Unfortunately, λ-calculus contexts are not compatible with α- and β-reduction. The standard theory of the untyped and simply typed lambda calculus with constants is presupposed; the interested reader is referred to [Bar84].

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 $(\lambda x.x\Box)y \not\to_\beta y\Box$.

There are two basic context-related operations, both concerned with filling holes. The first operation, hole filling, denoted by $[\ ]$, deals with placing terms into the holes of a context. The second operation, 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.

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, 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 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 formalisation 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 formalisation 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 *hole filling* constructor $\oplus$ has been introduced and where hole filling is computed by $fill$-reduction; a similar example of non-commutation can be given for α-conversion and hole filling.

Example. Let $C \equiv (\lambda x.x\Box)y$ and $t \equiv x$. Then

$$\begin{array}{lll} C \oplus t & \equiv & ((\lambda x.x\Box)y) \oplus x \\ & \to_{fill} & (\lambda x.xx)y \\ & \to_\beta & yy \\ & \equiv & s' \end{array} \qquad \text{but} \qquad \begin{array}{lll} C \oplus t & \to_\beta & (y\Box) \oplus x \\ & \to_{fill} & yx \\ & \not\equiv & s' \end{array}$$

In the example, the reductions end in different terms because in the reduction on the left the substitution $[\![x := y]\!]$ that emerged from the rewrite step in context C is applied to the term t, whilst in the reductions on the right the substitution is not applied to term t, 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, that 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. This problem of *communication* between holes and objects to be put into the holes 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.

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

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

3 A gentle introduction to λc

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

Contexts. A context can be considered as a function over the possible contents of its holes. For this reason, in the context calculus hole variables are introduced and contexts are represented as functions over hole variables.

Communication. We take a basic, lambda-calculus-like approach, and solve this problem by using the fact that in lambda calculus substitution emerges as the result of a β-step: $t\,[\![x := t']\!] \leftarrow_\beta (\lambda x.t)\,t'$. Since it is convenient to use multiple substitution, we will introduce new constructors Λ for multiple abstraction and $@\!\!\!\!\bigcirc\!\!\!\!m$ for multiple application, together with a multiple version of the β-rule. The second reduction in the example above now becomes (the hole filling constructor $\oplus$ is still auxiliary and h is a hole variable):

$$\begin{array}{lll} ((\lambda x.x\,(@\!\!\!\!\bigcirc\!\!\!\!m\ h\ x))y) \oplus (\Lambda x.x) & \rightarrow_\beta & (y\,(@\!\!\!\!\bigcirc\!\!\!\!m\ h\ y)) \oplus (\Lambda x.x) \\ & \rightarrow_{fill} & y\,(@\!\!\!\!\bigcirc\!\!\!\!m\ (\Lambda x.x)\ y) \\ & \rightarrow_{@\!\!\!\!\bigcirc\!\!\!\!m\beta} & yy \end{array}$$

where the first term reveals the new representations of the hole $(@\!\!\!\!\bigcirc\!\!\!\!m\ h\ x)$ and the term $(\Lambda x.x)$.

In general, 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. The holes are represented as multiple applications of hole variables to a sequence of terms that keeps track of the relevant $\alpha\beta$-changes. When a communicating term is placed into the hole, communication can be computed by applying a generalized form of the β-rule

$$@\!\!\!\!\bigcirc\!\!\!\!m\ (\Lambda x_1, ...x_n.U)\ V_1...V_n \quad \rightarrow \quad U[\![x_1 := V_1, ..., x_n := V_n]\!]$$

recovering the binding intention and passing the changes.

Hole filling and composition. The rewrite relations hole filling and composition depend on the structure of a particular notion of context. However, since we represent contexts as

abstractions over hole variables, hole filling simply boils down to (multiple) β-reduction. So, in our running example we get $(\lambda h.\lambda x.x(@\, h\ x)\, y)(\Lambda x.x) \to_\beta \lambda x.x(@\, (\Lambda x.x)\ x)\, y$.

In the case of composition, the rewrite relation involves some shifting of abstractions. For example, let $C \equiv \lambda x.\Box$ and $D \equiv x(\lambda y.\Box)$ be two λ-contexts. Then the composition of the two, here denoted by $C \circ D$, results in the context $\lambda x.x(\lambda y.\Box)$. Note that the hole of the result of the composition is the hole of the second context, which potentially binds variables x as well as y. In the context calculus, these contexts are represented as $C_c \equiv \lambda h.\lambda x.@\, h\ x$ and $D_c \equiv \lambda g.x(\lambda y.@\, g\ y)$. Because the second context is going to be put into the hole of C_c, it is provided with means of communication, adapted for this purpose: $D'_c \equiv \Lambda x.\lambda g.x(\lambda y.@\, g\ xy)$. The composition puts the second context into the hole, and moves the λg abstraction at the beginning of C, so that the whole is an abstraction over the second hole g, the hole of D. The composition rewrite step should result in $C_c \circ D'_c \to_\circ \lambda g.\lambda x.@\, (\Lambda x.x(\lambda y.@\, g\ xy))\ x$. This is an instance of the composition rewrite rule:

$$(\lambda h.U) \circ (\Lambda u_1 ... u_n.\lambda g.V) \quad \to_\circ \quad \lambda g.U[\![h := \Lambda u_1 ... u_n.V]\!]$$

where λg is shifted to the beginning of the reduct. Note that by performing the ensuing communication step this term reduces to $\lambda g.\lambda x.x(\lambda y.@\, g\ xy)$, which is a representation of the resulting composition in lambda calculus.

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 $\lambda h.\lambda x.(@\, h\ x)(@\, h\ x)$. All occurrences of the same hole should have the same number of arguments, though. If a context contains many holes, these can be represented by different hole variables, as in $\lambda h.\lambda g.\lambda x.(@_1\ h\ x)\ (\lambda y.@_2\ g\ xy)$. Note that here the holes are filled sequentially using the β-rule. An alternative representation is $\Lambda h, g.\lambda x.(@_1\ h\ x)\ (\lambda y.@_2\ g\ xy)$. Last but not least, the calculus may include variables over contexts and functions over contexts, witnessing the true first-class treatment of contexts.

Pretyping. The flexibility of the framework can be controlled by 'pretyping', 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, or variables in abstractions can be ensured to match their arguments, for example.

4 Definition of λc

This section contains the definition of the (untyped) framework. In the absence of pretyping, which naturally controls term-formation, a couple of assumptions are made about composition by using labelled composition constructors and composition rewrite rules of special form. With pretyping defined, such assumptions can in general be enforced by the pretyping rules.

Preliminary to the definition of λc-terms, the forms of composition that are allowed need to be specified. This can be accomplished by labeling each composition constructor

with an index $\iota \in \mathcal{I}$. We do not specify the precise form of the indices, but here only assume the existence of the following functions defined on $\mathcal{I}$. First $\textsc{Ar}(\iota)$, which returns the number of holes of the outer context involved in composition. Secondly $\textsc{Ind}(\iota)$, which specifies whether contexts are represented as λ- or Λ-abstractions, or as a sequence of such abstractions. Thirdly $\textsc{Ncom}(\iota)$, which specifies (the cardinality of) the communication between the holes of the main context and corresponding communicating contexts. Note by the way that in many cases, depending on the pretyping, composition will be definable.

Let $\mathcal{V}$ be a countably infinite set of variables.

Definition 4.1 (Λc) The set of un(pre)typed λc-terms Λc is defined inductively by

$$\begin{array}{rcl} U & ::= & u \mid \quad (\lambda u.U) \mid \quad (U\ U) \mid \\ & & (\Lambda_n u_1, ..., u_n.U) \mid \quad (@_n\ U\ U ... U) \mid \\ & & (\circ_\iota(U, U, ..., U) \end{array}$$

where $u \in \mathcal{V}$, $\iota \in \mathcal{I}$, $U, ..., U$ abbreviates n U's and $\textsc{Ar}(\iota) = n + 1$.

Notation. Variables $u_1, ..., u_n$ in $(\Lambda_n u_1, ..., u_n.U)$ will be abbreviated by $\vec{u}$, and n terms $U_1 ... U_n$ in the expressions $(@_n\ U\ U_1 ... U_n)$ and $(\circ_\iota(U, U_1, ..., U_n)$ will be abbreviated by $\vec{U}$, where the vectors are empty if $n = 0$. We will even omit the indices n and ι and assume that the arities of Λ, $@$ 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 maintained (if not explicitly stated otherwise): $i, j, k, m, n \in \mathbb{N}$, $u, u', u_i, v, w, ... \in \mathcal{V}$ and $U, U', U_i, V, W ... \in \Lambda c$.

The multiple abstractors Λ_n and the multiple applicators $@_m$ are generalised forms of the familiar (simple) abstractor and applicator. 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 n distinct variables $\vec{v}$, where n 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 $(1 \leq i \leq n)$ is defined as:

$$u[\![\vec{v} := \vec{V}]\!] = \begin{cases} V_i & : \text{ if } u = v_i \text{ for some } 1 \leq i \leq n \\ u & : \text{ otherwise} \end{cases},$$

$$(\mathtt{B}\vec{u}.U')[\![\vec{v} := \vec{V}]\!] = \mathtt{B}\vec{u}.(U'[\![\vec{v} := \vec{V}]\!]),$$

$$\mathtt{F}\ (U',\ U_1 ... U_m)[\![\vec{v} := \vec{V}]\!] = \mathtt{F}\ (U'[\![\vec{v} := \vec{V}]\!],\ U_1\ [\![\vec{v} := \vec{V}]\!] ... U_m[\![\vec{v} := \vec{V}]\!]).$$

Here $\mathtt{B}.$ denotes λ or Λ, and $\mathtt{F}\ (,\ )$ denotes $\cdot$, $@$ or $\circ$.

Before giving the definition of λc, an important assumption about the collection of the composition rules is explained. We assume that the arguments of the composition are of the right shape in the following sense. Suppose contexts are represented as sequences of multiple abstractions, for example $\Lambda u_1 u_2.\Lambda u_3.U$ is a representation of some context. If it is the outer context in a composition, the composition should have another three arguments that are communicating contexts represented in the same way: as sequences of multiple abstractions, e.g., $\circ((\Lambda u_1 u_2.\Lambda u_3.U), (\Lambda \vec{u}'.\Lambda \vec{u}''.U'), (\Lambda \vec{v}.\Lambda \vec{v}'.V), (\Lambda \vec{w}.\Lambda \vec{w}'.W))$. Moreover, the way the hole variables are grouped by the multiple binders should be preserved, so the composition should reduce to $\Lambda \vec{u}''\vec{v}'.\Lambda \vec{w}'.U[\![u_1 := \Lambda \vec{u}'.U', u_2 := \Lambda \vec{v}.V, u_3 := \Lambda \vec{w}.W]\!]$,

where the holes $\vec{u}''$ and $\vec{v}'$ are now under the same multiple abstractor because u_1 and u_2 were. It is assumed that all composition rules satisfy these conditions.

Definition 4.2 (Context calculus λc) The context calculus λc is defined on terms of Λc with rewrite relations induced by the following rule schemes:

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

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

In addition to these rule schemes, countably many composition rule schemes having the following form are allowed:

$$\circ((\vec{\mathtt{B}}\vec{u}.U), (\Lambda\vec{v}.\vec{\mathtt{B}}\vec{v}'.V_1), ..., (\Lambda\vec{w}.\vec{\mathtt{B}}\vec{w}'.V_n))$$
$$\to_\circ \quad \vec{\mathtt{B}}\vec{v}', ..., \vec{w}'.U[\![u_1 := \Lambda\vec{v}.V_1, ..., u_n := \Lambda\vec{w}.V_n]\!] \qquad (\circ)$$

where $\vec{\mathtt{B}}\vec{u}.U$ stands for $\lambda\vec{u}.U$ or $\Lambda u_1...u_i.\Lambda u_{i+1}....\Lambda u_j...u_n.U$ with $u_1...u_i u_{i+1}...u_j...u_n = \vec{u}$.

Remark. The context calculus can be defined more efficiently as follows. Since hole filling and compositions are functions on contexts and terms, they can be defined as λc-terms, provided a powerful enough pretyping system (in the case of last example, let $comp \equiv \Lambda cd'.\lambda g'.c(\Lambda x'.(\textcircled{m}\ d'\ x')g')$). 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 $\textcircled{m}$ and the rule $(\textcircled{m}\beta)$.

Finally, we only mention the main result of λc.

Theorem 4.3 *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.

The proof is conducted in a couple of stages. Firstly, the higher-order system $\mathcal{H}$ is defined, by translating the constructors and rewrite schemes of the context calculus. The higher-order system $\mathcal{H}$ turns out to be orthogonal (there are no critical pairs and all rules are left-linear (i.e. free variables occur at most once on the left-hand side of a rule)). Since, by a general theorem, any orthogonal HRS is confluent, 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. A full version of the proof can be found at `http://www.cs.vu.nl/~mirna/conproof.ps`.

5 An example of pretyping

The pretyped calculus $\lambda c^{\to}$ given in this section describes the simply typed lambda calculus with 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 (representations of) contexts. The pretyping rules of $\lambda c^{\to}$ generally follow the typing rules of the calculus of Hashimoto and Ohori (cf. [HO98]). In this section, we will first define the pretypes, the pretyping rules and the calculus $\lambda c^{\to}$. Then, we will

$$(var)\ \frac{(x:\tau)\in\Gamma}{\Gamma,\Delta\ \vdash\ x:\tau} \qquad (hvar)\ \frac{(h:[\vec{\tau}]\tau)\in\Delta \quad \Gamma,\Delta\ \vdash\ \vec{V}:\vec{\tau}}{\Gamma,\Delta\ \vdash\ \textcircled{m}\, h\,\vec{V}:\tau}$$

$$(abs)\ \frac{\Gamma,x:\tau,\Delta\ \vdash\ U:\tau'}{\Gamma,\Delta\ \vdash\ \lambda x:\tau.U:\tau\to\tau'} \qquad (habs)\ \frac{\Gamma,\Delta,h:[\vec{\tau}]\tau\ \vdash\ U:\tau'}{\Gamma,\Delta\ \vdash\ \lambda h:[\vec{\tau}]\tau.U:[\vec{\tau}]\tau\Rightarrow\tau'}$$

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

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

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

Figure 1: Pretyping rules for $\lambda c^{\to}$

summarize some properties of $\lambda c^{\to}$ and briefly compare $\lambda c^{\to}$ to the calculus of Hashimoto and Ohori [HO98]. Finally, we will name two variations of this pretyping.

If $\mathcal{BT}$ denotes the set of base types with $\mathtt{a}\in\mathcal{BT}$, then the τ-pretypes ($\tau\in\mathcal{T}$) and the ρ-pretypes ($\rho\in\mathcal{P}$) are defined as

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

Here, $\to$ and $\Rightarrow$ associate to the right, $\to$ binds stronger than [ ] and [ ] binds stronger than $\Rightarrow$. The τ-pretypes are used for pretyping terms and 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 $h:[\vec{\tau}]\tau$. The bases are split because the elements of Γ are used as true variables and the elements of Δ 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 $\Rightarrow$ are introduced for better correspondence with the constructors of λc (namely, [ ] for Λ and $\textcircled{m}$) and for distinguishing between two kinds of abstractions: over term or context variables, and over hole variables ($\to$ versus $\Rightarrow$), as will become clear in the pretyping rules. In the pretyping rules, $\vec{U}:\vec{\tau}$ denotes the pointwise pretyping $U_i:\tau_i$ for $1\le i\le|\vec{\tau}|$, and both Γ and Δ are, without loss of generality, assumed to contain distinct variables.

Definition 5.1 (Pretyping rules for $\lambda c^{\to}$) 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.

We comment on the rules shortly. The rules (var), (abs) and (app) are the familiar Church style typing rules for $\lambda^{\to}$. The rules $(hvar)$, $(habs)$ and $(fill)$ are their respective counterparts for the hole variables. By the rule $(hvar)$, hole variables are immediately provided with communication, generalizing the interface technique in the type system in [HO98]. Note that a plain hole variable is not pretypable. However, the more general rule, $\Gamma,\Delta\ \vdash\ h:[\vec{\tau}]\tau$ if $(h:[\vec{\tau}]\tau)\in\Delta$, is also posssible in our setting. The rule $(fill)$ is actually an application rule. An alternative understanding of the rules $(habs)$ and $(fill)$ is that the (simple) abstractor and applicator are duplicated in order to distinguish between abstractions over term or context variables and abstractions over hole variables,

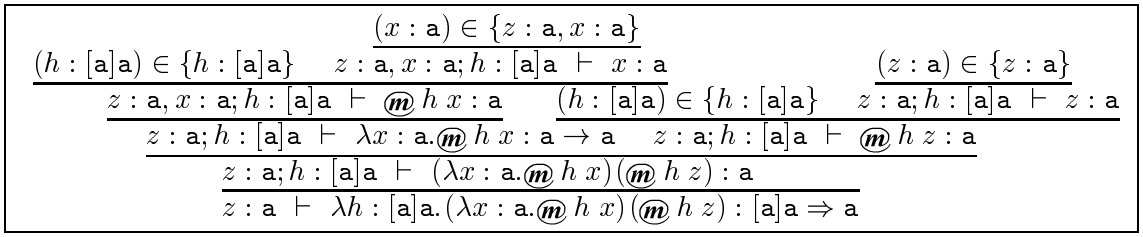

Figure 2: An example of pretyping in $\lambda c^{\to}$

and the corresponding applications. The rules (*mabs*) and (*mapp*) are added for pretyping communication. By the rule (*mabs*) no hole variables may occur in a communication.

Note that there is no composition in the pretyping rules. This is because composition is definable within $\lambda c^{\to}$: 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 $comp\, U\, V$,

$$comp \equiv \lambda c : [\vec{\tau}]\tau \Rightarrow \tau'.\lambda d : [\vec{\tau}]([\vec{\sigma}]\sigma \Rightarrow \tau).\lambda g[\vec{\sigma}]\sigma.c\ (\Lambda \vec{u} : \vec{\tau}.(\textcircled{m}\ d\ \vec{u})\ (\Lambda \vec{v} : \vec{\sigma}.\textcircled{m}\ g\ \vec{v}))$$
$$: ([\vec{\tau}]\tau \Rightarrow \tau') \to ([\vec{\tau}]([\vec{\sigma}]\sigma \Rightarrow \tau) \Rightarrow ([\vec{\sigma}]\sigma \Rightarrow \tau')).$$

Consequently, the composition constructor, rewrite rule and pretyping rule are omitted from $\lambda c^{\to}$.

Figure 2 is an example of pretyping in $\lambda c^{\to}$.

Definition 5.2 ($\lambda c^{\to}$) The terms of $\lambda c^{\to}$ are the well-pretyped terms of λc according to Definition 5.1. The rewrite rules are the rules (β) and ($\textcircled{m}\beta$) of λc, now over well-pretyped terms.

We have the following results.

Proposition 5.3 (Uniqueness of pretypes) *If* $\Gamma, \Delta \vdash U : \rho_1$ *and* $\Gamma, \Delta \vdash U : \rho_2$ *then* $\rho_1 \equiv \rho_2$.

Proposition 5.4 (Subject reduction) *If* $\Gamma, \Delta \vdash U : \rho$ *and* $U \twoheadrightarrow V$, *then* $\Gamma, \Delta \vdash V : \rho$.

Proposition 5.5 (Strong normalization) *Reduction in* $\lambda c^{\to}$ *is strongly normalizing.*

The proofs of the first two propositions are the standard ones, as in the case of $\lambda^{\to}$ *à la* Church. The proof of the second proposition uses counterparts of the generation lemma and the substitution lemma. Note that, because of the choice of the rule (*hvar*), the subject reduction property does not hold for a multiple version of η-rule, e.g. $\Lambda \vec{u} : \vec{\tau}.\textcircled{m}\ h\ \vec{u} \to_\eta h$ and h is not pretypable. The proof of strong normalization can be done via the natural translation of $\lambda c^{\to}$ into $\lambda^{\to}$.

The example described in this section extends the work of Hashimoto and Ohori [HO98]. 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) contexts. Moreover, $\lambda c^{\to}$ allows composition, which is not present in their system.

We conclude by mentioning two simple variations of this example of pretyping, which can even be combined. These variations illustrate the expressive power of the framework λc, which results from the possibility of fine-tuning the pretyping rules.

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 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 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, $z : \mathtt{t} \vdash \lambda h : [\mathtt{t}]\,\mathtt{t}\,.(\lambda x : \mathtt{t}.(@\, h\ x)(@\, h\ z) : [\mathtt{t}]\mathtt{t} \Rightarrow \mathtt{t}$ and $\vdash (\lambda x : \mathtt{t}.x\,x)(\lambda x : \mathtt{t}.x\,x) : \mathtt{t}$. This pretyping essentially has the effect of rules for well-formedness of untyped λ-terms and of typing rules on the contexts and holes, by ignoring the type constructor $\to$.

λ-contexts with one hole. As contexts "with one hole" we consider those which translate to λc-terms where each subterm has at most one free hole variable. Such contexts can be captured by imposing extra conditions on Δ. The conditions involve leaving Δ out (but keeping h if present, in the rules (var), $(habs)$ and $(fill)$) or splitting Δ of the post-condition into a finite union of Δ_i's (not necessarily disjoint) that are used by the pre-conditions (in the rules (app) and $(mapp)$), and in all cases, allowing Δ('s) to contain at most one element (in the rules $(hvar)$, (abs), (app), $(mabs)$ and $(mapp)$). These conditions restrain the pretyping of terms that can act as composition constructors, which seem to have a subterm with two free variables of ρ-pretypes; for instance d and g in the subterm $(@\, d\ \vec{u})\ (\Lambda\vec{v} : \vec{\sigma}.@\, g\ \vec{v})$ of the $\lambda c^{\to}$-term *comp* given above. Therefore, a composition constructor $\circ$, and the following composition pretyping rule and composition rewrite rule are added

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

$$(\lambda h : [\vec{\tau}]\tau.U) \circ (\Lambda\vec{u} : \vec{\tau}.\lambda g : [\vec{\sigma}]\sigma.V) \ \to_\circ \ \lambda g : [\vec{\sigma}]\sigma.U[\![h := \Lambda\vec{u} : \vec{\tau}.V]\!] \qquad (\circ)$$

The composition pretyping rule resembles the composition on functions: if $V : A \to B$ and $U : B \to C$ then $U \circ V : A \to C$ with $A \equiv \sigma, B \equiv \tau$ and $C \equiv \tau'$. In this rule, communication is added, pretyped by $[\vec{\tau}]$ and $[\vec{\sigma}]$, which 'move' through the pretypes following the movement of the corresponding abstractions in the composition rewrite rule. Even with the additional composition pretyping rule and rewrite rule, in the calculus obtained by such pretyping the properties of uniqueness of pretyping, subject reduction and strong normalization are preserved.

Acknowledgements

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

References

[Bal87] Herman 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.

[Bal94] H. Balsters. *Lambda calculus extended with segments*: Chapter 1, Sections 1.1 and 1.2 (Introduction). In R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors, *Selected papers on Automath*, pages 339–367. North-Holland, Amsterdam, 1994.

[Bar84] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, revised edition, 1984. (Second printing 1985).

[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.

[BV99] Mirna Bognar and Roel de Vrijer. Segments in the context of contexts. Preprint, Vrije Universiteit Amsterdam, 1999.

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

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

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

[NGV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 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] David Sands. Computing with contexts, a simple approach. *Electronic Notes in Theoretical Computer Science*, 10, 1998.

[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.