The qnu and qnuK calculi, name capture and control (Extended Abstract)

Sylvain Baro, François Maurel

The $q\nu$ and $q\nu K$ calculi : name capture and control

Sylvain Baro François Maurel

March 11, 2003

Abstract

We introduce a pithy calculus obtained by splitting the λ binder of the λ-calculus into two constructions : a pure binder ν, *à la* π-calculus, and a combinator for β-reduction. This splitting allows a powerful – yet controlled – rebinding mechanism. Using the same splitting with continuations allows the expression of both local and global exceptions in the same *clean* calculus. We also exhibit a typing system with *variable initialisation* and *exception handling* analysis.

Introduction

The splitting of a binder into a pure binder and a combinator has been proposed by Fu Yuxi in ICALP'97 for the χ-calculus and by Joachim Parrow and Björn Victor in LICS'98 for their *fusion calculus*. This led to expressive and simple process calculi. We hereby present a similar construction in the functional setting.

We first present the syntax of the $q\nu$-calculus, its reduction rules and some properties. We then define typing rules and a variable initialisation analysis. Finally, we extend the language with continuations which allows an encoding for *real exceptions*.

1 The $q\nu$-calculus

1.1 Syntax

The $q\nu$-calculus is obtained by *splitting* the usual λ construction into two constructions: the binder ν which handles the renaming, and the combinator q for β-reduction.

Given an infinite countable set of variables $\mathcal{X}$, we define the terms of the $q\nu$-calculus as :

$$t \ ::= \ x \mid \nu x.t \mid qx.t \mid (t\ t)$$

The set of *free variables* of a term t is defined as

$$\begin{array}{lcllcl} FV(x) & \equiv & \{x\} & FV(qx.t) & \equiv & FV(t) \cup \{x\} \\ FV(\nu x.t) & \equiv & FV(t) \backslash \{x\} & FV((t_1\ t_2)) & \equiv & FV(t_1) \cup FV(t_2) \end{array}$$

For example, in the term $qx.(\nu y.qy.y\ z)$, the variables x and z are free while y is bound.

As opposed to the behaviour of the ν, the combinator q is not a binder, hence, variables in its subterm are *not protected* against capture. Renaming and closed terms are defined according to this definition of free variables. The renaming in a term only takes into account the ν binder : for example $\nu x.x$ and $\nu y.y$ are equivalent up to renaming, while $qx.x$ and $qy.y$ are not.

A term is *closed* if all its variables x are under a νx. Similarly, a term is *safe* if all its variables x are under a qx. The *unsafe variables* – those which are not under a q – are the variables which may not be initialised. The typical example is x in $\nu x.x$ which is bound but unsafe. The intuition behind these definitions is that a term is closed when all its variables are declared whereas a term is safe when all its variables are initialised.

The substitution is also defined according to the behaviour we expect from ν and q, up to renaming.

$$\begin{array}{lcllcl} x[u/x] & \equiv & u & y[u/x] & \equiv & y \\ (\nu x.t)[u/x] & \equiv & \nu x.t & (\nu y.t)[u/x] & \equiv & \nu y.t[u/x] \text{ if } y \notin FV(u) \\ (qx.t)[u/x] & \equiv & qx.t[u/x] & (qy.t)[u/x] & \equiv & qy.t[u/x] \\ (t_1\ t_2)[u/x] & \equiv & (t_1[u/x]\ t_2[u/x]) & & & \end{array}$$

1.2 Reduction rules

We consider terms up to a structural congruence $\nu x.\nu y.t \sim \nu y.\nu x.t$. The reduction rules are:

Scope extrusion	$(\nu x.t\ u)$	$\rightsquigarrow_\nu$	$\nu x.(t\ u)$	*if* $x \notin FV(u)$
β-reduction	$(qx.t\ u)$	$\rightsquigarrow_\beta$	$t[u/x]$	
Garbage collection	$\nu x.t$	$\rightsquigarrow_\gamma$	t	*if* $x \notin FV(t)$

The reduction $\rightsquigarrow$ is the union of these three reductions.

1.3 Properties

Following the initial motivations, the λ-calculus can be encoded inside the $q\nu$-calculus by the translation $[\![\lambda x.t]\!] \equiv \nu x.qx.[\![t]\!]$ which has the expected behaviour : $t \rightsquigarrow_\lambda u \Rightarrow [\![t]\!] \rightsquigarrow_\nu \rightsquigarrow_\beta \rightsquigarrow_\gamma^* [\![u]\!]$. We therefore use in the sequel $\lambda x.t$ as a shorthand for $\nu x.qx.t$.

Let $choice = \lambda t.\lambda u.\nu x.(\lambda y.(qx.y\ u)\ (qx.x\ t))$. For all t and u, $((choice\ t)\ u)$ reduces to t or u. Hence the $q\nu$-calculus is not confluent. Therefore, we use reduction strategies to enforce determinism. In the sequel, we explore specifically both call-by-value and call-by-name weak strategies (no reductions under a q).

1.4 Examples

The following terms – which do not belong to the λ-calculus – should help to understand the $q\nu$-calculus :

- A *rebinder* is a term capturing a variable during reduction. A typical rebinder is $r_x \equiv \lambda y.qx.y$ which can be used to write the identity function $id \equiv \nu x.(\lambda f.(f\ x)\ r_x)$ where x is rebound by r_x at runtime.
- The term $\nu x.x$ is closed, unsafe and doesn't reduce.

2 Typing

We mimic the λ-calculus simple typing rules in the $q\nu$-calculus and afterwards enrich this system with *unsafe variables evaluation* analysis.

2.1 Simple types

Given a countable set of atoms (ranged over by X), simple types (ranged over by $A, B \ldots$) are built using the following grammar : $A := X \mid A \to A$.

Judgements are of the form $\Gamma \vdash t : A$ where the context Γ is a set of distinct variables with a simple type, t is a term and A is a type. Judgements are considered up to context permutation.

The typing rules for the $q\nu$-calculus are as follows

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : A \vdash qx.t : A \to B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \nu x.t : B}$$

These typing rules extend the ones of the λ-calculus with the translation $\lambda x.t = \nu x.qx.t$. The combinator q is not a binder, hence the variable x is not discarded from the context in the rule for q. Moreover, each variable has to be declared by a ν.

Proposition 1 (Subject reduction)
Let $\Gamma \vdash t : A$ be a derivable judgement. If t reduces to u then $\Gamma \vdash u : A$ is derivable.

Proposition 2 (Strong normalization)
Any typable term is strongly normalizing.

2.2 Analysis

Consider $id = \nu x.(\lambda f.(f\ x)\ \lambda y.qx.y)$. This term reduces to $\lambda x.x$. The terms id and $\nu x.x$ are both well typed and unsafe but their reductions differ. In weak reduction, the unsafety of id disappears during the reduction process whereas $\nu x.x$ leads to the evaluation of an uninitialized variable.

By annotating types with variables and extending simple typing rules, we are able to detect this kind of "errors" and distinguish the previous terms.

3 The K

The λ_κ-calculus is an extension of the λ-calculus with continuation such that $\kappa x.t = \text{call/cc}\ \lambda x.t$. In the same way λ is split, κ can be split into two parts : the already defined ν and a combinator K such that $\kappa x.t = \nu x.Kx.t$. The syntax of the $q\nu K$-calculus is $t\ ::=\ x \mid \nu x.t \mid qx.t \mid (t\ t) \mid Kx.t \mid \langle x, \mathcal{C}[\ ]\rangle$.

The $Kx.t$ construction evaluates t with the current continuation ($\langle x, \mathcal{C}[\ ]\rangle$) bound to x, where $\mathcal{C}[\ ]$ is the current evaluation context. The variable x in $\langle x, \mathcal{C}[\ ]\rangle$ is needed here for the nesting of Kx which doesn't occur in the λ_κ-calculus.

The reduction rules of the $q\nu K$-calculus are more complicated than the ones of the $q\nu$-calculus because of control. Due to lack of space, we only present without details the right weak call-by-value reduction strategy.

$$\begin{array}{lll}
\mathcal{C}[\{\nu x.t\}] & \rightsquigarrow \mathcal{C}[\nu x.\{t\}] & \mathcal{C}[\{(t\ u)\}] \rightsquigarrow \mathcal{C}[(t\ \{u\})] \\
\mathcal{C}[(t\ \{v\})] & \rightsquigarrow \mathcal{C}[(\{t\}\ v)] & \mathcal{C}[(\{\nu x.t\}\ v)] \rightsquigarrow \mathcal{C}[\nu x.(\{t\}\ v)] \text{ if } x \notin FV(v) \\
\mathcal{C}[(\{qx.t\}\ v)] & \rightsquigarrow \mathcal{C}[\{t[v/x]\}] & \mathcal{C}[\{Kx.t\}] \rightsquigarrow \mathcal{C}[\{t[\langle x, \mathcal{C}[\ ]\rangle/x]\}] \\
\mathcal{C}[\{(\langle x, \mathcal{C}'[\ ]\rangle\ t)\}] & \rightsquigarrow \mathcal{C}'[\{\nu\overrightarrow{z}.t\}] &
\end{array}$$

where $\langle x, \mathcal{C}[\ ]\rangle[\langle x, \mathcal{C}'[\ ]\rangle/x] \equiv \langle x, \mathcal{C}'[\ ]\rangle$. The $\overrightarrow{z}$ are the bound variables of $\mathcal{C}[\ ]$ minus those of $\mathcal{C}'[\ ]$.

The intuition is that to reduce $Kx.t$, first reduce $t \rightsquigarrow t_1 \rightsquigarrow \cdots$. If x is evaluated, then return its (scope extruded) argument.

Both typing and analysis can be extended for the $q\nu K$-calculus.

4 *Real programmers'* exceptions

We encode `CoreML` – a mini-language with *first class exceptions à la* ML – inside the $q\nu K$-calculus. We use a notion of *toplevel* in `CoreML` to handle uncaught exceptions. Exceptions of `CoreML` behaves in the usual (Caml) way. The translation of a term with an uncaught exception carrying a value t reduces to $\nu e.\nu\overrightarrow{x}.(e\ t)$. The uninitialized variables analysis yields for free an uncaught exceptions analysis.

The syntax of `CoreML` is

top	::=	`Toplevel`(t)	
t	::=	$x \mid \lambda x.t \mid (t\ t) \mid$ `let` $x\ =\ t$ `in` t	
	$\mid$	`Except` $x.t$	Defines of a new exception x in t
	$\mid$	`Exn`(x, t)	Builds an exception value containing t
	$\mid$	`try` t `with` $x\ \rightarrow u$	Catches an exception x and feeds u with its contained value
	$\mid$	(`raise` t)	Raises an exception value

For example, the following term reduces to 12, due to the *rescoping* of the exception e in the `try with`.

$$\texttt{Toplevel}(\texttt{Except}\ e.\texttt{let}\ f\ =\ \lambda x.(\texttt{raise}\ \texttt{Exn}(e, 12))\ \texttt{in try}\ (f\ 42)\ \texttt{with}\ e\ \rightarrow \lambda x.x)$$

Both `CoreML` and the $q\nu K$-calculus use a right weak call-by-value reduction strategy.

Translations of non straightforward constructions are :

$$\begin{array}{lll}
[\![\texttt{Toplevel}(t)]\!] & \equiv \nu\, Top.K\, Top.[\![t]\!] & \\
[\![\texttt{Except}\ e.t]\!] & \equiv \nu e.\nu e'.[\![\texttt{try}\ t\ \texttt{with}\ e\ \rightarrow \lambda x.(Top\ (e\ x))]\!] & \\
[\![\texttt{Exn}(e, t)]\!] & \equiv \lambda z.(e\ [\![t]\!]) & \\
[\![\texttt{try}\ t\ \texttt{with}\ e\ \rightarrow u]\!] & \equiv Ke'.([\![u]\!]\ Ke.(e'\ [\![t]\!])) & \\
[\![(\texttt{raise}\ t)]\!] & \equiv ([\![t]\!]\ \text{a}) & \text{where a is an arbitrary constant of the proper type.}
\end{array}$$

All this shows that the $q\nu K$-calculus can express both real exceptions and continuations.