J. Functional Programming **9** (6): 625–647, November 1999. Printed in the United Kingdom 625

A confluent λ-calculus with a catch/throw mechanism

TRISTAN CROLARD

Laboratoire Preuves, Programmes et Systèmes Université Paris 7, France
(*e-mail:* crolard@ufr-info-p7.jussieu.fr)

Abstract

We derive a confluent λ-calculus with a catch/throw mechanism (called $\lambda_{\mathbf{ct}}$-calculus) from Parigot's $\lambda\mu$-calculus. We also present several translations from one calculus into the other which are morphisms for the reduction. We use them to show that the $\lambda_{\mathbf{ct}}$-calculus is a retract of $\lambda\mu$-calculus (these calculi are isomorphic if we consider only convertibility). As a by-product, we obtain the subject reduction property for the $\lambda_{\mathbf{ct}}$-calculus, as well as the strong normalization for $\lambda_{\mathbf{ct}}$-terms typable in the second order classical natural deduction.

Capsule Review

Parigot's $\lambda\mu$-calulus provides a very neat extension of λ–calculus by 'control features' making clear the connection to classical logic by using the latter as a typing system for $\lambda\mu$–terms.

However, from the point of view of programming it appears as more natural to consider a control extension $\lambda_{\mathbf{ct}}$ of λ–calculus based on the operators **catch** and **throw** which allow for writing and reading of continuations *aka* evaluation contexts. The author gives a translation Λ^{ι}_{μ} of $\lambda_{\mathbf{ct}}$ to $\lambda\mu$-calculus by expanding **catch** and **throw** into appropriate macros. He defines a reduction system on $\lambda_{\mathbf{ct}}$ generating the theory induced by the translation Λ^{ι}_{μ} to $\lambda\mu$–calculus. This translation turns out to establish a 1-1-correspondence between the two calculi modulo the considered conversion theories. More important, however, is that the translation Λ^{ι}_{μ} is a morphism of reduction systems and there is a backward translation $\Lambda^{s}_{\mathbf{ct}}$ with $\Lambda^{s}_{\mathbf{ct}} \circ \Lambda^{\iota}_{\mu} = \mathrm{Id}_{\mathbf{ct}}$ such that $\Lambda^{s}_{\mathbf{ct}}$ is a reduction morphism as well. Thus, $\lambda_{\mathbf{ct}}$ appears as a retract of $\lambda\mu$-calculus via morphisms preserving reduction. However, as shown in the paper $\lambda\mu$–calculus cannot appear as a retract of $\lambda_{\mathbf{ct}}$ via reduction morphisms.

Based on these results one may reduce confluence of $\lambda_{\mathbf{ct}}$ to confluence of $\lambda\mu$-calculus (already proved by Parigot) and normalisation of a typed version of $\lambda_{\mathbf{ct}}$ to normalisation of typed $\lambda\mu$-calculus (already proved by Parigot). Moreover, the typed version of $\lambda_{\mathbf{ct}}$ gives sort of 'logical meaning' to the operators **catch** and **throw** which is interesting in its own.

1 Introduction

In the last four years, several extensions of the λ-calculus with some catch/throw mechanism have been proposed by Nakano (1994*a*; 1994*b*; 1995) and by Sato (1997) and Kameyama (1997; 1998). In these papers, the authors consider the catch/throw mechanism as 'intrinsically non-deterministic', and thus investigate non-confluent calculi or confine themselves to some specific evaluation strategy. For instance in

Nakano (1994*b*), the non-deterministic feature of the catch/throw mechanism is introduced by the following rule:

$$C[\mathbf{throw}\ \alpha\ t] \mapsto \mathbf{throw}\ \alpha\ t$$

where the context $C[\bullet]$ does not capture α or any individual/tag variable occurring freely in t. Let us now look at the following example from Nakano (1994*b*):

$$M \equiv \mathbf{catch}\ \alpha\ ((\lambda x.\lambda y.1\ (\mathbf{throw}\ \alpha\ 2)\ (\mathbf{throw}\ \alpha\ 3))$$

If we assume that we also have the two following rules: $\mathbf{catch}\ \alpha\ \mathbf{throw}\ \alpha\ t \rightarrow \mathbf{catch}\ \alpha\ t$ and $\mathbf{catch}\ \alpha\ t \rightarrow t$ when α does not occur free in t, we have three possible normal forms, depending on the evaluation strategy:

$$\begin{array}{l} M \rightarrow_\beta \mathbf{catch}\ \alpha\ ((\lambda y.1\ (\mathbf{throw}\ \alpha\ 2)) \rightarrow_\beta \mathbf{catch}\ \alpha\ 1 \rightarrow 1 \\ M \mapsto \mathbf{catch}\ \alpha\ \mathbf{throw}\ \alpha\ 2 \rightarrow \mathbf{catch}\ \alpha\ 2 \rightarrow 2 \\ M \mapsto \mathbf{catch}\ \alpha\ \mathbf{throw}\ \alpha\ 3 \rightarrow \mathbf{catch}\ \alpha\ 3 \rightarrow 3 \end{array}$$

In this paper, however, we will see that if we weaken the rule $\mapsto$, it is possible to define a confluent λ-calculus with a catch/throw mechanism. This calculus, called $\lambda_{\mathbf{ct}}$-calculus, is in fact derived from Parigot's $\lambda\mu$-calculus. We will present several 'canonical' morphisms for the reduction from one calculus to the other (this notion of canonical translation will be formalized).

Then we show that the $\lambda_{\mathbf{ct}}$-calculus is a canonical retract of the $\lambda\mu$-calculus. This will enable us to derive the confluence of the $\lambda_{\mathbf{ct}}$-calculus from the confluence of the $\lambda\mu$-calculus. We also prove that the converse is not true (the $\lambda\mu$-calculus is not a canonical retract of the $\lambda_{\mathbf{ct}}$-calculus), since there is no surjective canonical morphism from the $\lambda_{\mathbf{ct}}$-calculus to the $\lambda\mu$-calculus. Both calculi are, however, isomorphic if we consider terms up to renaming/simplification.

As a by-product of these translations, we also obtain the subject reduction property, as well as the strong normalization for terms typable in the second order classical natural deduction.

As usual with control operators, the catch/throw mechanism is easier to introduce in the framework of abstract stack machines. Indeed, control operators are aimed to handle the continuation (i.e. the rest of the computation to be performed, see Felleisen *et al.* (1986; 1987) or Reynolds (1993) for a survey), and precisely, in abstract stack machines, the continuation is represented by the stack. In the remainder of this introduction, we thus consider two simple extensions of Krivine's abstract machine: the $\lambda\mu$-machine and the $\lambda_{\mathbf{ct}}$-machine. The former is designed for evaluating $\lambda\mu$-terms, as suggested by Parigot (1993) and developed by Beus and Streicher (1998) and De Groote (1999), while the latter is provided with a catch/throw mechanism.

In section 2, we derive the reduction rules of the $\lambda_{\mathbf{ct}}$-calculus from the rules of the $\lambda\mu$-calculus. In section 3, we prove that the $\lambda_{\mathbf{ct}}$-calculus is a canonical retract of the $\lambda\mu$-calculus, and as a corollary we obtain the confluence of the $\lambda_{\mathbf{ct}}$-calculus. In section 4, we take advantage of the fact that the classical natural deduction may be seen as a type system for the $\lambda_{\mathbf{ct}}$-calculus, as well as for the $\lambda\mu$-calculus.

1.1 An abstract machine for the λμ-calculus

We first recall the syntax of $\lambda\mu$-terms (as usual, we use $x, y, z \ldots$ as λ-variables and $\alpha, \beta, \gamma \ldots$ as μ-variables). The set of $\lambda\mu$-terms is inductively defined as follows.

Definition 1.1.1
If t, u are $\lambda\mu$-terms then the following terms are also $\lambda\mu$-terms (where α and β range over μ-variables and x ranges over λ-variables):

$$x, \quad (t\ u), \quad \lambda x.t, \quad \mu\alpha[\beta]t$$

Remark. The μ-operator is a binder (the μ-variable α is bound in $\mu\alpha[\beta]t$).

We now define the $\lambda\mu$-machine as a rewrite system. For that purpose we recall some common definitions (see Beus and Streicher (1998), for instance): a closure is inductively defined as a triple ⟨$\lambda\mu$-term, closure-environment, stack-environment⟩, where a closure-environment is a list of pairs (λ-variable, closure), a stack-environment is a list of pairs (μ-variable, stack) and a stack is a list of closures.

The rules of the $\lambda\mu$-machine are given below. The variables CE and SE range over closure-environments and stack-environments, respectively. As usual, the notation $E(v)$, where E is a closure-environment (resp. stack-environment) and v a λ-variable (resp. μ-variable), stands for the closure (resp. stack) assigned to v in E.

- $(\langle x, CE, SE\rangle, S) \to (CE(x), S)$
- $(\langle (u\ v), CE, SE\rangle, S) \to (\langle u, CE, SE\rangle, \langle v, CE, SE\rangle :: S)$
- $(\langle \lambda x.t, CE, SE\rangle, c :: S) \to (\langle t, (x, c) :: CE, SE\rangle, S)$
- $(\langle \mu\alpha[\beta]t, CE, SE\rangle, S) \to (\langle t, CE, (\alpha, S) :: SE\rangle, ((\alpha, S) :: SE)(\beta))$

Remark. We wrote $((\alpha, S) :: SE)(\beta)$ in the last rule (and not just $SE(\beta)$) in order to deal with the case $\alpha = \beta$.

An instruction of the form $\mu\alpha[\beta]t$ is carried out as expected: the machine first binds the current continuation to name α, then restores the continuation whose name is β in the current environment (discarding the current continuation) and eventually evaluates t.

Remark. Notice that this abstract machine actually evaluates only the weak head normal form of a $\lambda\mu$-term. For further details, see Beus and Streicher (1998) and De Groote (1999).

1.2 An abstract machine with a **catch/throw** *mechanism*

We still consider two separate name-spaces for λ-variables and μ-variables (or 'tag-variables'). The set of $\lambda_{\mathbf{ct}}$-terms is inductively defined as follows.

Definition 1.2.1
If t, u are $\lambda_{\mathbf{ct}}$-terms then the following terms are also $\lambda_{\mathbf{ct}}$-terms (where α ranges over μ-variables and x ranges over λ-variables):

$$x, \quad (t\ u), \quad \lambda x.t, \quad \textbf{catch}\ \alpha\ t, \quad \textbf{throw}\ \alpha\ t$$

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

Remark. The **catch** operator is a binder (the μ-variable α is bound in **catch** α t).

The intended behaviour of the **catch** and **throw** operators is intuitively clear. To evaluate **catch** α t, the machine should bind the current continuation to name α and then evaluate t. To evaluate **throw** α t, the machine should discard the current continuation, restore the continuation whose name is α in the current environment, and eventually evaluate t. Formally, to define the $\lambda_{\mathbf{ct}}$-machine, just replace the last rule of the $\lambda\mu$-machine by the following rules:

- $(\langle \textbf{catch}\ \alpha\ t, CE, SE\rangle, S) \to (\langle t, CE, (\alpha, S) :: SE\rangle, S)$
- $(\langle \textbf{throw}\ \alpha\ t, CE, SE\rangle, S) \to (\langle t, CE, SE\rangle, SE(\alpha))$

1.3 Simulating one machine by the other

It is easy to simulate the behaviour of $\mu\alpha[\beta]t$ in the $\lambda_{\mathbf{ct}}$-machine. Indeed, let us consider a term of the form **catch** α **throw** β t. To evaluate such a term, the $\lambda_{\mathbf{ct}}$-machine binds the current stack to name α, restores the stack whose name is β in the current environment and then evaluates t: this is exactly what does the $\lambda\mu$-machine when it evaluates $\mu\alpha[\beta]t$.

Conversely, the behaviour of the **catch** and **throw** operators can also be simulated in the $\lambda\mu$-machine.

- Let us first consider a term of the form $\mu\alpha[\alpha]t$. When the $\lambda\mu$-machine evaluates such a term, it first binds the current stack to name α, then restores *this very stack*, before it evaluates t. This is exactly what the $\lambda_{\mathbf{ct}}$-machine does when it evaluates **catch** α t.
- Let us now consider the instruction $\mu\alpha[\beta]t$, where α does not occur in t. To carry out this term, the $\lambda\mu$-machine binds the current stack to name α, then restores the stack whose name is β in the current environment before it evaluates t. Since α does not occur in t, the current stack should have been discarded, and this is exactly what does the $\lambda_{\mathbf{ct}}$-machine whenever it evaluates **throw** β t.

2 The catch and throw operators and the $\lambda\mu$-calculus

In the previous section, we discussed how abstract machines can simulate one another. However, restricting ourselves to some abstract machine amounts exactly to considering a specified evaluation strategy (weak head reduction for Krivine's abstract machines). In this section, we show that the simulation of the **catch** and **throw** operators defined in the framework of abstract machines work as well when we consider the $\lambda\mu$-calculus as a confluent rewriting system: we will take advantage of this in the next section to derive a confluent λ-calculus with some catch/throw mechanism. Let us first recall the reduction rules of the $\lambda\mu$-calculus (note that the original proof of confluence of the $\lambda\mu$-calculus given in Parigot (1992) is broken by the renaming rule, however the fix is easy and presented by Py (1998)).

Reduction rules of the λμ-calculus

- The β-reduction:

$$(\lambda x.t\ u) \to t\{u/x\}$$

- The *structural* rule:

$$(\mu\alpha.t\ u) \to \mu\alpha.t\big\{[\alpha = 20](w\ u)/[\alpha]w\big\}$$

- The *renaming* rule:

$$[\beta]\mu\alpha.t \to t\{\beta/\alpha\}$$

- The *simplification* rule:

$$\mu\alpha[\alpha]t \to t \text{ if } \alpha \text{ does not occur free in } t$$

The notation $t\{u/x\}$ stands for the usual capture-avoiding substitution of the λ-variable x by u in t. The *structural substitution* $t\big\{[\alpha](w\ u)/[\alpha]w\big\}$ is defined inductively by:

- $x\{[\alpha](w\ v)/[\alpha]w\} = x$
- $(\lambda x.t)\{[\alpha](w\ v)/[\alpha]w\} = \lambda x.t\{[\alpha](w\ v)/[\alpha]w\}$
- $(t\ u)\{[\alpha](w\ v)/[\alpha]w\} = (t\{[\alpha](w\ v)/[\alpha]w\}\ u\{[\alpha](w\ v)/[\alpha]w\})$
- $(\mu\beta.t)\{[\alpha](w\ v)/[\alpha]w\} = \mu\beta.t\{[\alpha](w\ v)/[\alpha]w\}$
- $([\alpha]t)\{[\alpha](w\ v)/[\alpha]w\} = [\alpha](t\{[\alpha](w\ v)/[\alpha]w\}\ v)$
- $([\beta]t)\{[\alpha](w\ v)/[\alpha]w\} = [\beta]t\{[\alpha](w\ v)/[\alpha]w\}$ if $\alpha \neq \beta$.

Remark. We sometimes use the more explicit notation $t\{\mu\beta[\alpha](w\ u)/\mu\beta[\alpha]w\}$ for structural substitution above, since any occurrence of $[\alpha]w$ in t has actually the shape $\mu\beta[\alpha]w$.

Definition 2.0.1
We call *simple λμ-term* a term which contains no renaming/simplification redex.

Remark. The simple form of a $\lambda\mu$-term t (which is obtained from t by applying only renaming/simplification rules) is unique (modulo α-conversion), and is reached in a linear number of reduction steps. Indeed, it is easy to check that renaming and simplification rules commute with any other rule. Moreover, the application of renaming/simplification rules strictly decreases the size of the term.

Notation. We denote by $\bar{t}$ the simple form of a $\lambda\mu$-term t.

2.1 Deriving the rules

We saw in the previous section that the **catch** and **throw** operators can be simulated respectively by *catch* $\alpha\ t \equiv\ \mu\alpha[\alpha]t$ and *throw* $\alpha\ t \equiv \mu\beta[\alpha]t$, where β is a μ-variable different from α and which does not occur free in t.

Notation. We use the abbreviation $\mu\_[\alpha]t$ in the latter case, where _ stands for any

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

μ-variable different from α which does not occur free in t. Moreover, to avoid any confusion, we will use italic font for macros (while we use bold face font for built-in operators).

We call $\lambda\mu ct$-calculus the sublanguage of the $\lambda\mu$-calculus containing only the 'macros' *catch* and *throw* (in other words, where any occurrence of a subterm $\mu\alpha[\beta]t$ is either of the form $\mu\alpha[\alpha]t$, or of the form $\mu\_[\beta]t$). Unfortunately, the subset consisting of all $\lambda\mu ct$-terms is not closed under reduction because of the following rule:

$$catch\ \alpha\ throw\ \beta\ t = \mu\alpha[\alpha]\mu\_[\beta]t \to \mu\alpha[\beta]t\{\alpha/\_\} = \mu\alpha[\beta]t$$

We therefore restrict ourselves to instances of rules for which the contractum is still a $\lambda\mu ct$-term. We obtain these rules by enumerating all the redexes that may occur in a $\lambda\mu ct$-term.

- The subset of $\lambda\mu ct$-terms is clearly closed under substitution. We thus obtain the β-reduction as a derived rule (since the contractum is always a $\lambda\mu ct$-term):

$$(\lambda x.t\ u) \to t\{u/x\}$$

- Any redex of the form $\mu\alpha[\alpha]t$ that occurs in a $\lambda\mu ct$-term has the form *catch* $\alpha\ t$. The rule $\mu\alpha[\alpha]t \to t$, if α does not occur free in t, and thus yields a unique derived rule (since the contractum is still a $\lambda\mu ct$-term):

$$catch\ \alpha\ t = \mu\alpha[\alpha]t \to t$$

- Any redex of the form $(\mu\alpha.w\ v)$ (to be more specific $(\mu\alpha[\beta]u\ v)$) that occurs in a $\lambda\mu ct$-term is either of the form $(catch\ \alpha\ u)\ v$, or of the form $(throw\ \alpha\ u)\ v$. The rule $(\mu\alpha.w\ v) \to \mu\alpha.w\ \big\{[\alpha](t\ v)/[\alpha]t\big\}$ yields two derived rules (since in both cases the contractum is still a $\lambda\mu ct$-term):

$$\begin{aligned}((catch\ \alpha\ u)\ v) = (\mu\alpha[\alpha]u\ v) &\to \mu\alpha([\alpha]u)\big\{[\alpha](t\ v)/[\alpha]t\ \big\}\\ &= \mu\alpha[\alpha](u(\big\{\mu\_[\alpha](t\ v)/\mu\_[\alpha]t\ \big\}\ v)\\ &= catch\ \alpha\ (u\big\{throw\ \alpha\ (t\ v)/throw\ \alpha\ t\big\}\ v)\\ ((throw\ \alpha\ u)\ v) = (\mu\delta[\alpha]u\ v) &\to \mu\delta([\alpha]u)\big\{[\delta](t\ v)/[\delta]t\ \big\}\\ &= \mu\delta[\alpha]u = throw\ \alpha\ u\end{aligned}$$

  since, by definition of *throw*, the variable δ is different from α and does not occur free in u.
- Any redex of the form $[\alpha]\mu\beta.w$ (to be more specific, $\mu\gamma[\alpha]\mu\beta[\delta]t$) that occurs in a $\lambda\mu ct$-term has one of the four following forms: *catch* α *catch* $\beta\ t$, *throw* α *throw* $\beta\ t$, *throw* α *catch* $\beta\ t$, *catch* α *throw* $\beta\ t$. The rule $[\alpha]\mu\beta.w \to w\big\{\ \alpha/\beta\big\}$ yields four cases:

$$\begin{aligned}catch\ \alpha\ catch\ \beta\ t &= \mu\alpha[\alpha]\mu\beta[\beta]t \to \mu\alpha[\alpha]t\{\alpha/\beta\} = catch\ \alpha\ t\{\alpha/\beta\}\\ throw\ \alpha\ throw\ \beta\ t &= \mu\_[\alpha]\mu\_[\beta]t \to \mu\_[\beta]t\{\alpha/\_\} = throw\ \beta\ t\\ throw\ \alpha\ catch\ \beta\ t &= \mu\_[\alpha]\mu\beta[\beta]t \to \mu\_[\alpha]t\{\alpha/\beta\} = throw\ \alpha\ t\{\alpha/\beta\}\\ catch\ \alpha\ throw\ \beta\ t &= \mu\alpha[\alpha]\mu\_[\beta]t \to \mu\alpha[\beta]t\{\alpha/\_\} = \mu\alpha[\beta]t\end{aligned}$$

The first three cases yield three derived rules, but in the last case (as we already

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

saw) the contractum is no more a $\lambda\mu ct$-term. Nevertheless, in the special case $\alpha = \beta$ we obtain the following derived rule:

$$catch\ \alpha\ throw\ \alpha\ t = \mu\alpha[\alpha]\mu\_[\alpha]t \rightarrow \mu\alpha[\alpha]t = catch\ \alpha\ t$$

Remark. The rule $catch\ \alpha\ throw\ \beta\ t \rightarrow \mu\alpha[\beta]t$ shows that the two "macros" *catch* and *throw* are enough to express all the $\lambda\mu$-terms up to renaming.

Let us summarize the derived rules we obtained above in the following definition (where **catch** and **throw** are now native operators):

Definition 2.1.1
We call $\lambda_{\mathbf{ct}}$-calculus the λ-calculus together with the operators **catch** and **throw** defined by the 8 following rules:

1. $(\lambda x.t\ u) \rightarrow t\{u/x\}$
2. $((\mathbf{catch}\ \alpha\ t)\ u) \rightarrow \mathbf{catch}\ \alpha\ (t\ \left\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\ \right\}\ u)$
3. $((\mathbf{throw}\ \alpha\ t)\ u) \rightarrow \mathbf{throw}\ \alpha\ t$
4. $\mathbf{catch}\ \alpha\ \mathbf{catch}\ \beta\ t \rightarrow \mathbf{catch}\ \alpha\ t\{\alpha/\beta\}$
5. $\mathbf{throw}\ \alpha\ \mathbf{throw}\ \beta\ t \rightarrow\ \mathbf{throw}\ \beta\ t$
6. $\mathbf{throw}\ \alpha\ \mathbf{catch}\ \beta\ t \rightarrow \mathbf{throw}\ \alpha\ t\{\alpha/\beta\}$
7. $\mathbf{catch}\ \alpha\ \mathbf{throw}\ \alpha\ t \rightarrow\ \mathbf{catch}\ \alpha\ t$
8. $\mathbf{catch}\ \alpha\ t \rightarrow t$ if α does not occur free in t.

Notation. The *structural substitution* $t\left\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\right\}$ is defined inductively by:

- $x\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\} = x$
- $(\lambda x.t)\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\} =$
  $\lambda x.t\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\}$
- $(s\ t)\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\} =$
  $(s\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\}\ t\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\})$
- $(\mathbf{catch}\ \beta\ t)\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\} =$
  $\mathbf{catch}\ \beta\ t\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\}$
- $(\mathbf{throw}\ \alpha\ t)\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\} =$
  $\mathbf{throw}\ \alpha\ (t\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\}\ u)$
- $(\mathbf{throw}\ \beta\ t)\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\} =$
  $\mathbf{throw}\ \beta\ t\{\mathbf{throw}\ \alpha\ (w\ u)/\mathbf{throw}\ \alpha\ w\}$ if $\alpha \neq \beta$.

Definition 2.1.2
Rules 4–7 are called *renaming rules*. Rule 8 is called a *simplification rule*. A *simple $\lambda_{\mathbf{ct}}$-term* is a term which contains no renaming/simplification redex.

Remark. As for $\lambda\mu$-terms, the simple form of a $\lambda_{\mathbf{ct}}$-term t (which is obtained from t by applying only renaming and simplification rules) is unique (modulo α-conversion), and is reached in a linear number of reduction steps. Two $\lambda_{\mathbf{ct}}$-terms are said to be *equal up to renaming/simplification* if they have the same simple form.

Notation. We denote by $\bar{t}$ the simple form of a $\lambda_{\mathbf{ct}}$-term t.

3 Canonical morphisms

The construction of the $\lambda_{\mathbf{ct}}$-calculus lets us expect the existence of some canonical translations that embed each calculus into the other and which are morphisms for the reduction. We first formalize this notion of canonical translation. Then we show that the $\lambda_{\mathbf{ct}}$-calculus is a canonical retract of the $\lambda\mu$-calculus. This will enable us to derive the confluence of the $\lambda_{\mathbf{ct}}$-calculus from the confluence of the $\lambda\mu$-calculus. We also show that the converse is not true (the $\lambda\mu$-calculus is not a canonical retract of the $\lambda_{\mathbf{ct}}$-calculus) since there is no surjective canonical morphism from the $\lambda_{\mathbf{ct}}$-calculus to the $\lambda\mu$-calculus. Both calculi are however isomorphic if we consider terms up to renaming/simplification (and consequently up to convertibility).

To be more specific, we will define two canonical translations $\Lambda^{\iota}_{\mathbf{ct}}$ and $\Lambda^{s}_{\mathbf{ct}}$ of $\lambda\mu$-terms into $\lambda_{\mathbf{ct}}$-terms and two canonical translations Λ^{ι}_{μ} and Λ^{s}_{μ} of $\lambda_{\mathbf{ct}}$-terms into $\lambda\mu$-terms such that:

- $\Lambda^{s}_{\mathbf{ct}} \circ \Lambda^{\iota}_{\mu} = \mathrm{Id}_{\mathbf{ct}}$, and thus $\Lambda^{s}_{\mathbf{ct}}$ is surjective and Λ^{ι}_{μ} is injective.
- $\Lambda^{s}_{\mu} \circ \Lambda^{\iota}_{\mathbf{ct}} = \mathrm{Id}_{\mu}$, and thus Λ^{s}_{μ} is surjective and $\Lambda^{\iota}_{\mathbf{ct}}$ is injective.
- Λ^{ι}_{μ} and $\Lambda^{s}_{\mathbf{ct}}$ are morphisms, and thus $\langle \Lambda^{\iota}_{\mu}, \Lambda^{s}_{\mathbf{ct}} \rangle$ is a retraction pair.
- $\Lambda^{\iota}_{\mathbf{ct}}$ is a morphism, but Λ^{s}_{μ} is not a morphism (since there is no surjective canonical morphism from the $\lambda_{\mathbf{ct}}$-calculus to the $\lambda\mu$-calculus).

3.1 Morphisms

We give here the formal definition of a morphism. As usual, for any relation $\rightarrow$ we denote by $\rightarrow^{\star}$ the reflexive, transitive closure of $\rightarrow$.

Definition 3.1.1
Given two calculus $\mathbf{c}_1$ and $\mathbf{c}_2$ and a mapping Φ from $\mathbf{c}_1$ to $\mathbf{c}_2$, we say that Φ is a morphism for $\rightarrow_{\mathbf{c}_1}$ iff for any terms t, u of $\mathbf{c}_1$:

$$t \rightarrow_{\mathbf{c}_1} u \quad \text{implies} \quad \Phi(t) \rightarrow^{\star}_{\mathbf{c}_2} \Phi(u)$$

Remarks

- A morphism for the reduction also preserves convertibility. In other words, if Φ is a morphism for $\rightarrow_{\mathbf{c}_1}$ and if $=_{\mathbf{c}_i}$ denotes the reflexive, symmetric, transitive closure of $\rightarrow_{\mathbf{c}_i}$:

$$t =_{\mathbf{c}_1} u \quad \text{implies} \quad \Phi(t) =_{\mathbf{c}_2} \Phi(u)$$

- A mapping Φ which preserves one-step reduction:

$$t \rightarrow_{\mathbf{c}_1} u \quad \text{implies} \quad \Phi(t) \rightarrow_{\mathbf{c}_2} \Phi(u)$$

  is of course a morphism according to the previous definition.
- If Φ is an injective morphism and the relation $\rightarrow_{\mathbf{c}_1}$ is irreflexive (i.e. $t \not\rightarrow_{\mathbf{c}_1} t$ for any t, which is usually the case for one-step reduction) then if $\rightarrow^{+}_{\mathbf{c}_i}$ denotes the transitive closure of $\rightarrow_{\mathbf{c}_i}$:

$$t \rightarrow_{\mathbf{c}_1} u \quad \text{implies} \quad \Phi(t) \rightarrow^{+}_{\mathbf{c}_2} \Phi(u)$$

3.2 Canonical translations

Let us notice that there is a very natural bijection between simple terms of both calculi (see Proposition 3.2.3). A translation from one calculus into the other is thus said to be canonical if it extends this natural bijection. Conversely, we recover this natural bijection from any canonical translation when we consider terms equal up to renaming/simplification (in both calculi).

Definition 3.2.1
We define the mapping Λ^s_μ from $\lambda_{\mathbf{ct}}$-terms to λ_μ-terms by induction:

- $\Lambda^s_\mu(x) = x$, if x is a λ-variable,
- $\Lambda^s_\mu((u\ v)) = (\Lambda^s_\mu(u)\ \Lambda^s_\mu(v))$
- $\Lambda^s_\mu(\lambda x.t) = \lambda x.\Lambda^s_\mu(t)$
- $\Lambda^s_\mu(\mathbf{catch}\ \alpha\ t) = \begin{cases} \mu\alpha[\beta]\Lambda^s_\mu(u) & \text{if } t \text{ has the form } \mathbf{throw}\ \beta\ u \\ \mu\alpha[\alpha]\Lambda^s_\mu(t) & \text{otherwise} \end{cases}$
- $\Lambda^s_\mu(\mathbf{throw}\ \alpha\ t) = \mu\delta[\alpha]\Lambda^s_\mu(t)$, where δ is a fresh μ-variable.

Definition 3.2.2
We define the mapping $\Lambda^s_{\mathbf{ct}}$ from $\lambda\mu$-terms to $\lambda_{\mathbf{ct}}$-terms by induction:

- $\Lambda^s_{\mathbf{ct}}(x) = x$, if x is a λ-variable,
- $\Lambda^s_{\mathbf{ct}}((u\ v)) = (\Lambda^s_{\mathbf{ct}}(u)\ \Lambda^s_{\mathbf{ct}}(v))$
- $\Lambda^s_{\mathbf{ct}}(\lambda x.t) = \lambda x.\Lambda^s_{\mathbf{ct}}(t)$
- $\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]t) = \begin{cases} \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(t) & \text{if } \alpha = \beta \\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t) & \text{if } \alpha \neq \beta \text{ and } \alpha \text{ is not free in } t \\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t) & \text{otherwise} \end{cases}$

Remark. For any $\lambda_{\mathbf{ct}}$-term (resp λ_μ-term) t, the free λ-variables and μ-variables are the same in t and $\Lambda^s_{\mathbf{ct}}(t)$ (resp. $\Lambda^s_\mu(t)$).

Proposition 3.2.3
The mapping $\Lambda^s_{\mathbf{ct}}$ (resp. Λ^s_μ) is a bijection between simple forms of both calculi.

Proof
Check that if t is a simple $\lambda_{\mathbf{ct}}$-term then $\Lambda^s_{\mathbf{ct}}(\Lambda^s_\mu(t)) = t$, and conversely if t is a simple λ_μ-term then $\Lambda^s_\mu(\Lambda^s_{\mathbf{ct}}(t)) = t$. □

Remark. Notice that neither $\Lambda^s_{\mathbf{ct}}$ nor Λ^s_μ is a bijection (if we do not restrict the domain to simple terms). Indeed, $\Lambda^s_{\mathbf{ct}}$ is not injective since if t is a $\lambda\mu$-term such that $\alpha \neq \beta$ and α occurs free in t:

$$\Lambda^s_{\mathbf{ct}}(\mu\alpha[\alpha]\mu\_[\beta]t) = \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t) = \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]t)$$

Besides, Λ^s_μ is not injective since if t is a $\lambda_{\mathbf{ct}}$-term which has not the form $\mathbf{throw}\ \beta\ u$:

$$\Lambda^s_\mu(\mathbf{catch}\ \alpha\ \mathbf{throw}\ \alpha\ t) = (\mu\alpha[\alpha]\Lambda^s_\mu(t)) = \Lambda^s_\mu(\mathbf{catch}\ \alpha\ t)$$

Definition 3.2.4
A translation Ψ from the $\lambda_{\mathbf{ct}}$-calculus into the λ_μ-calculus is said to be *canonical* if for any $\lambda_{\mathbf{ct}}$-term t, $\overline{\Psi(t)} = \Lambda^s_\mu(\bar{t})$. Conversely, a translation Φ from the

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

λ_μ-calculus into the $\lambda_{\mathbf{ct}}$-calculus is said to be *canonical* if for any $\lambda\mu$-term t, $\overline{\Phi(t)} = \Lambda^s_{\mathbf{ct}}(\bar{t})$.

Remark. A canonical translation is thus a translation which maps sequences of control operators of one calculus onto sequences of control operators of the other calculus and leaves the rest unchanged. In particular, a term without control operator is translated into itself.

Proposition 3.2.5
The translations Λ^s_μ and $\Lambda^s_{\mathbf{ct}}$ are canonical.

Proof
Check by induction on the $\lambda_{\mathbf{ct}}$-term (resp. $\lambda\mu$-term) t, $\overline{\Lambda^s_\mu(t)} = \Lambda^s_\mu(\bar{t})$ (resp. $\overline{\Lambda^s_{\mathbf{ct}}(t)} = \Lambda^s_{\mathbf{ct}}(\bar{t})$). □

3.3 Injective canonical morphisms

In this section we define two injective canonical morphisms $\Lambda^\iota_{\mathbf{ct}}$ and Λ^ι_μ from one calculus into the other. Since these translations are injective, they will allow us to derive the strong normalization of one calculus from the strong normalization of the other calculus (see section 4).

3.3.1 From the $\lambda_{\mathbf{ct}}$-calculus towards the $\lambda\mu$-calculus

Definition 3.3.1
We define the mapping Λ^ι_μ from $\lambda_{\mathbf{ct}}$-terms to $\lambda\mu$-terms by induction:

- $\Lambda^\iota_\mu(x) = x$, if x is a λ-variable,
- $\Lambda^\iota_\mu((u\ v)) = (\Lambda^\iota_\mu(u)\ \Lambda^\iota_\mu(v))$
- $\Lambda^\iota_\mu(\lambda x.t) = \lambda x.\Lambda^\iota_\mu(t)$
- $\Lambda^\iota_\mu(\mathbf{catch}\ \alpha\ t) = \mu\alpha[\alpha]\Lambda^\iota_\mu(t)$
- $\Lambda^\iota_\mu(\mathbf{throw}\ \alpha\ t) = \mu\delta[\alpha]\Lambda^\iota_\mu(t)$ where δ is a fresh μ-variable.

Remark. For any $\lambda_{\mathbf{ct}}$-terms t, $\Lambda^\iota_\mu(t)$ is a $\lambda\mu ct$-term.

Proposition 3.3.2
The mapping Λ^ι_μ is canonical.

Proof
Check by induction on the $\lambda_{\mathbf{ct}}$-term t that $\overline{\Lambda^\iota_\mu(t)} = \Lambda^s_\mu(\bar{t})$. □

We easily show that Λ^ι_μ is a morphism for the reduction.

Proposition 3.3.3
For any $\lambda_{\mathbf{ct}}$-terms t, t', if $t \to_{\mathbf{ct}} t'$ then $\Lambda^\iota_\mu(t) \to_{\lambda\mu} \Lambda^\iota_\mu(t')$.

Proof
By construction, since for any $\lambda_{\mathbf{ct}}$-term t, $\Lambda^\iota_\mu(t)$ is a $\lambda\mu ct$-term and any reduction rule of the $\lambda_{\mathbf{ct}}$-calculus comes from some reduction rule of the $\lambda\mu$-calculus. □

3.3.2 From the $\lambda\mu$-calculus towards the $\lambda_{\mathbf{ct}}$-calculus

Definition 3.3.4
We define the mapping $\Lambda^{\iota}_{\mathbf{ct}}$ from $\lambda\mu$-terms to $\lambda_{\mathbf{ct}}$-terms by induction:

- $\Lambda^{\iota}_{\mathbf{ct}}(x) = x$, if x is a λ-variable,
- $\Lambda^{\iota}_{\mathbf{ct}}((u\ v)) = (\Lambda^{\iota}_{\mathbf{ct}}(u)\ \Lambda^{\iota}_{\mathbf{ct}}(v))$
- $\Lambda^{\iota}_{\mathbf{ct}}(\lambda x.t) = \lambda x.\Lambda^{\iota}_{\mathbf{ct}}(t)$
- $\Lambda^{\iota}_{\mathbf{ct}}(\mu\alpha[\beta]t) = \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^{\iota}_{\mathbf{ct}}(t)$

Proposition 3.3.5
The mapping $\Lambda^{\iota}_{\mathbf{ct}}$ is canonical.

Proof
Check by induction on the $\lambda\mu$-term t that $\overline{\Lambda^{\iota}_{\mathbf{ct}}(t)} = \Lambda^{s}_{\mathbf{ct}}(\bar{t})$. □

We now show that $\Lambda^{\iota}_{\mathbf{ct}}$ is a morphism for the reduction.

Lemma 3.3.6
For any $\lambda\mu$-terms u, v and any variable x free in u:

$$\Lambda^{\iota}_{\mathbf{ct}}(u\{v/x\}) = \Lambda^{\iota}_{\mathbf{ct}}(u)\{\Lambda^{\iota}_{\mathbf{ct}}(v)/x\}$$

Proof
By induction on the term u. □

Lemma 3.3.7
For any instance $u \to v$ of a rule of the $\lambda\mu$-calculus we have $\Lambda^{\iota}_{\mathbf{ct}}(u) \to^{\star}_{\mathbf{ct}} \Lambda^{\iota}_{\mathbf{ct}}(v)$.

Proof
We consider each rule of the $\lambda\mu$-calculus:

- Case of the β-reduction $(\lambda x.u\ v) \to_{\beta} u\{v/x\}$:

$$\Lambda^{\iota}_{\mathbf{ct}}((\lambda x.u\ v)) = (\lambda x.\Lambda^{\iota}_{\mathbf{ct}}(u)\ \Lambda^{\iota}_{\mathbf{ct}}(v)) \to_{\beta} \Lambda^{\iota}_{\mathbf{ct}}(u)\{\Lambda^{\iota}_{\mathbf{ct}}(v)/x\} = \Lambda^{\iota}_{\mathbf{ct}}(u\{v/x\})$$

  by the substitution lemma.

- Case of the rule $(\mu\alpha.u\ v) \to \mu\alpha.u\ \big\{[\alpha](t\ v)/[\alpha]t\big\}$

$$\begin{aligned} &\Lambda^{\iota}_{\mathbf{ct}}(\mu\alpha.u\ v) \\ &\quad = \quad (\mathbf{catch}\ \alpha\ \Lambda^{\iota}_{\mathbf{ct}}(u)\ \Lambda^{\iota}_{\mathbf{ct}}(v)) \\ &\quad \to_2 \quad \mathbf{catch}\ \alpha\ (\Lambda^{\iota}_{\mathbf{ct}}(u)\big\{\mathbf{throw}\ \alpha\ (t\ \Lambda^{\iota}_{\mathbf{ct}}(v))/\mathbf{throw}\ \alpha\ t\big\}\ \Lambda^{\iota}_{\mathbf{ct}}(v)) \\ &\quad = \quad \Lambda^{\iota}_{\mathbf{ct}}(\mu\alpha.u\big\{[\alpha](t\ v)/[\alpha]t\ \big\}) \end{aligned}$$

- Case of the rule $[\beta]\mu\alpha t \to t\{\beta/\alpha\}$, i.e. $\mu\gamma[\beta]\mu\alpha[\delta]t \to \mu\gamma([\delta]t)\{\beta/\alpha\}$

$$\begin{aligned} \Lambda^{\iota}_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\delta]t) &= \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \delta\ \Lambda^{\iota}_{\mathbf{ct}}(t) \\ &\to_6 \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ (\mathbf{throw}\ \delta\ \Lambda^{\iota}_{\mathbf{ct}}(t))\{\beta/\alpha\} \\ &\to_5 \mathbf{catch}\ \gamma\ (\mathbf{throw}\ \delta\ \Lambda^{\iota}_{\mathbf{ct}}(t))\{\beta/\alpha\} \\ &= \Lambda^{\iota}_{\mathbf{ct}}(\mu\gamma([\delta]t)\{\beta/\alpha\}) \end{aligned}$$

- Case of the rule $\mu\alpha[\alpha]t \to t$ if α does not occur free in t.

$$\begin{aligned} \Lambda^{\iota}_{\mathbf{ct}}(\mu\alpha[\alpha]t) &= \mathbf{catch}\ \alpha\ \mathbf{throw}\ \alpha\ \Lambda^{\iota}_{\mathbf{ct}}(t) \\ &\to_7 \mathbf{catch}\ \alpha\ \Lambda^{\iota}_{\mathbf{ct}}(t) \\ &\to_8 \Lambda^{\iota}_{\mathbf{ct}}(t) \end{aligned}$$

□

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

In the sequel, we will need the usual concept of context (i.e. a term with a *hole*). Given a context, denoted by $C[\bullet]$, the notation $C[t]$ stands for the term obtained by replacing in $C[\bullet]$ the symbol $\bullet$ (the hole) by the term t. Let us now prove the following lemma:

Lemma 3.3.8
For any context $C[\bullet]$ and any $\lambda\mu$-terms u, v, if $\Lambda^{\iota}_{\mathbf{ct}}(u) \rightarrow_{\mathbf{ct}} \Lambda^{\iota}_{\mathbf{ct}}(v)$ then:

$$\Lambda^{\iota}_{\mathbf{ct}}(C[u]) \rightarrow_{\mathbf{ct}} \Lambda^{\iota}_{\mathbf{ct}}(C[v])$$

Proof
If we extend $\Lambda^{\iota}_{\mathbf{ct}}$ to $\lambda\mu$-contexts by taking $\Lambda^{\iota}_{\mathbf{ct}}(\bullet) = \bullet$, we easily prove that for any $\lambda\mu$-context C and any $\lambda\mu$-term t, we have $\Lambda^{\iota}_{\mathbf{ct}}(C[t]) = \Lambda^{\iota}_{\mathbf{ct}}(C)\{\Lambda^{\iota}_{\mathbf{ct}}(t)\}$. □

Proposition 3.3.9
For any $\lambda\mu$-terms u, v, if $u \rightarrow_{\lambda\mu} v$ then $\Lambda^{\iota}_{\mathbf{ct}}(u) \rightarrow^{\star}_{\mathbf{ct}} \Lambda^{\iota}_{\mathbf{ct}}(v)$.

Proof
By lemma 3.3.7 and lemma 3.3.8. □

To round off this section, let us prove two useful lemmas:

Lemma 3.3.10
$\Lambda^{s}_{\mathbf{ct}} \circ \Lambda^{\iota}_{\mu} = \mathrm{Id}_{\mathbf{ct}}$.

Proof
Let us prove that $\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(t)) = t$ by induction on the $\lambda_{\mathbf{ct}}$-term t:

- $\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(x)) = \Lambda^{s}_{\mathbf{ct}}(x) = x$, if x is a λ-variable,
- $\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}((u\ v))) = \Lambda^{s}_{\mathbf{ct}}((\Lambda^{\iota}_{\mu}(u)\ \Lambda^{\iota}_{\mu}(v)) = (\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(u))\ \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(v))) = (u\ v)$
- $\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(\lambda x.t)) = \Lambda^{s}_{\mathbf{ct}}(\lambda x.\Lambda^{\iota}_{\mu}(t)) = \lambda x.\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(t)) = \lambda x.t$
- $\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(\mathbf{catch}\ \alpha\ t)) = \Lambda^{s}_{\mathbf{ct}}(\mu\alpha[\alpha]\Lambda^{\iota}_{\mu}(t)) = \mathbf{catch}\ \alpha\ \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(t)) = \mathbf{catch}\ \alpha\ t$
- $\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(\mathbf{throw}\ \alpha\ t))\ = \Lambda^{s}_{\mathbf{ct}}(\mu\delta[\alpha]\Lambda^{\iota}_{\mu}(t))$ where δ is a fresh μ-variable
  $= \mathbf{throw}\ \alpha\ \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(t))$ since δ does not occur in $\Lambda^{\iota}_{\mu}(t)$
  $= \mathbf{throw}\ \alpha\ t$ □

Remark. The translation $\Lambda^{s}_{\mathbf{ct}}$ is thus surjective, and the translation Λ^{ι}_{μ} is injective. However, Λ^{ι}_{μ} is clearly not surjective since Λ^{ι}_{μ} maps any $\lambda_{\mathbf{ct}}$-term to some $\lambda\mu ct$-term. We already saw that $\Lambda^{s}_{\mathbf{ct}}$ is not injective.

Lemma 3.3.11
$\Lambda^{s}_{\mu} \circ \Lambda^{\iota}_{\mathbf{ct}} = \mathrm{Id}_{\mu}$.

Proof
Let us prove that $\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(t)) = t$ by induction on the $\lambda\mu$-term t:

- $\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(x)) = \Lambda^{s}_{\mu}(x) = x$, if x is a λ-variable,
- $\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}((u\ v))) = \Lambda^{s}_{\mu}((\Lambda^{\iota}_{\mathbf{ct}}(u)\ \Lambda^{\iota}_{\mathbf{ct}}(v)) = (\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(u))\ \Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(v))) = (u\ v)$
- $\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(\lambda x.t)) = \Lambda^{s}_{\mu}(\lambda x.\Lambda^{\iota}_{\mathbf{ct}}(t)) = \lambda x.\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(t)) = \lambda x.t$
- $\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(\mu\alpha[\beta]t)) = \Lambda^{s}_{\mu}(\mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^{\iota}_{\mathbf{ct}}(t)) = \mu\alpha[\beta]\Lambda^{s}_{\mu}(\Lambda^{\iota}_{\mathbf{ct}}(t)) = \mu\alpha[\beta]t$

□

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

Remark. The translation Λ^s_μ is thus surjective, and the translation $\Lambda^\iota_{\mathbf{ct}}$ is injective. However, $\Lambda^\iota_{\mathbf{ct}}$ is clearly not surjective since $\Lambda^\iota_{\mathbf{ct}}$ maps any $\lambda\mu$-term to some $\lambda_{\mathbf{ct}}$-term in which any **catch** is followed by a **throw**. We already saw that Λ^s_μ is not injective.

3.4 The $\lambda_{\mathbf{ct}}$-calculus is a canonical retract of the $\lambda\mu$-calculus

In this section, we show that $\Lambda^s_{\mathbf{ct}}$ is a morphism, and thus $\langle \Lambda^\iota_\mu, \Lambda^s_{\mathbf{ct}} \rangle$ is a retraction pair (the $\lambda_{\mathbf{ct}}$-calculus is a canonical retract of the $\lambda\mu$-calculus).

Lemma 3.4.1
For any $\lambda\mu$-terms u, v and any variable x free in u:

$$\Lambda^s_{\mathbf{ct}}(u\{v/x\}) = \Lambda^s_{\mathbf{ct}}(u)\{\Lambda^s_{\mathbf{ct}}(v)/x\}$$

Proof
By induction on the $\lambda\mu$-term u. $\square$

Lemma 3.4.2
For any instance $u \to v$ of any rule of the $\lambda\mu$-calculus, we have $\Lambda^s_{\mathbf{ct}}(u) \to^\star_{\mathbf{ct}} \Lambda^s_{\mathbf{ct}}(v)$.

Proof
We consider each rule of the $\lambda\mu$-calculus:

- Case of the β-reduction $(\lambda x.u\ v) \to_\beta u\{v/x\}$:

  $$\Lambda^s_{\mathbf{ct}}((\lambda x.u\ v)) = (\lambda x.\Lambda^s_{\mathbf{ct}}(u)\ \Lambda^s_{\mathbf{ct}}(v)) \to_\beta \Lambda^s_{\mathbf{ct}}(u)\{\Lambda^s_{\mathbf{ct}}(v)/x\} = \Lambda^s_{\mathbf{ct}}(u\{v/x\})$$

  by the substitution lemma.

- Case of the rule $(\mu\alpha[\beta]u\ v) \to \mu\alpha([\beta]u)\big\{[\alpha](t\ v)/[\alpha]t\big\}$
  1. If $\alpha = \beta$ then $\mu\alpha([\beta]u)\big\{[\alpha](t\ v)/[\alpha]t\big\} = \mu\alpha[\alpha](u\ \big\{[\alpha](t\ v)/[\alpha]t\big\}\ v)$,

     $$\begin{array}{ll} \multicolumn{2}{l}{\Lambda^s_{\mathbf{ct}}(\mu\alpha[\alpha]u\ v)} \\ = & (\mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(u)\ \Lambda^s_{\mathbf{ct}}(v)) \\ \to_2 & \mathbf{catch}\ \alpha\ (\Lambda^s_{\mathbf{ct}}(u)\big\{\mathbf{throw}\ \alpha\ (t\ \Lambda^s_{\mathbf{ct}}(v))/\mathbf{throw}\ \alpha\ t\big\}\ \ \Lambda^s_{\mathbf{ct}}(v)) \\ = & \Lambda^s_{\mathbf{ct}}(\mu\alpha[\alpha](u\{[\alpha](t\ v)/[\alpha]t\}\ v)) \end{array}$$

     since any occurrence of a subterm $\mu\beta[\alpha]t$ (for some $\beta \neq \alpha$) is translated, by definition of $\Lambda^s_{\mathbf{ct}}$, either into **catch** β **throw** $\alpha\ \Lambda^s_{\mathbf{ct}}(t)$ or into **throw** $\alpha\ \Lambda^s_{\mathbf{ct}}(t)$.
  2. If $\alpha \neq \beta$ and α does not occur in u, $\mu\alpha([\beta]u)\{[\alpha](t\ v)/[\alpha]t\}) = \mu\alpha[\beta]u$,

     $$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]u\ v) & = & ((\mathbf{throw}\ \alpha\ \Lambda^s_{\mathbf{ct}}(u))\ \Lambda^s_{\mathbf{ct}}(v)) \\ & \to_3 & \mathbf{throw}\ \alpha\ \Lambda^s_{\mathbf{ct}}(u) \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]u) \end{array}$$

  3. If $\alpha \neq \beta$ and α occurs in u, $\mu\alpha([\beta]u)\big\{[\alpha](t\ v)/[\alpha]t\ \big\} = \mu\alpha[\beta]u\big\{[\alpha](t\ v)/[\alpha]t\big\}$,

     $$\begin{array}{l} \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]u\ v) \\ =\ ((\mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(u))\ \Lambda^s_{\mathbf{ct}}(v)) \\ \to_2\ \mathbf{catch}\ \alpha\ ((\mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(u))\ \big\{\mathbf{throw}\ \alpha\ (t\ \Lambda^s_{\mathbf{ct}}(v))/\mathbf{throw}\ \alpha\ t\big\}\ \Lambda^s_{\mathbf{ct}}(v)) \\ =\ \mathbf{catch}\ \alpha\ (\mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(u)\ \big\{\mathbf{throw}\ \alpha\ (t\ \Lambda^s_{\mathbf{ct}}(v))/\mathbf{throw}\ \alpha\ t\big\}\ \Lambda^s_{\mathbf{ct}}(v)) \\ \to_3\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(u)\ \big\{\mathbf{throw}\ \alpha\ (t\ \Lambda^s_{\mathbf{ct}}(v))/\mathbf{throw}\ \alpha\big\} \\ =\ \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]u\{[\alpha](t\ v)/[\alpha]t\}) \end{array}$$

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

Again, since any occurrence of a subterm $\mu\beta[\alpha]t$ (for some $\beta \neq \alpha$) is translated, by definition of $\Lambda^s_{\mathbf{ct}}$, either into **catch** β **throw** α $\Lambda^s_{\mathbf{ct}}(t)$ or into **throw** α $\Lambda^s_{\mathbf{ct}}(t)$.

- Case of the rule $[\beta]\mu\alpha t \to t\{\beta/\alpha\}$, i.e. $\mu\gamma[\beta]\mu\alpha[\delta]t \to \mu\gamma([\delta]t)\{\beta/\alpha\}$

1. If $\gamma = \beta$ and $\alpha = \delta$ then $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\beta[\beta]t\{\beta/\alpha\}$

$$\begin{array}{rcl}\Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]\mu\alpha[\alpha]t) & = & \mathbf{catch}\ \beta\ \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(t)\\ & \to_4 & \mathbf{catch}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\}\\ & = & \Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]t\{\beta/\alpha\})\end{array}$$

2. If $\gamma = \beta$ and α thus does not occur in $[\delta]t$, $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\beta[\delta]t$

(a) If $\beta = \delta$ then $\mu\beta[\delta]t = \mu\beta[\beta]t$

$$\begin{array}{rcl}\Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]\mu\alpha[\beta]t) & = & \mathbf{catch}\ \beta\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\\ & \to_7 & \mathbf{catch}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\\ & = & \Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]t)\end{array}$$

(b) If $\beta \neq \delta$ and β does not occur in t

$$\begin{array}{rcl}\Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]\mu\alpha[\delta]t) & = & \mathbf{catch}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\\ & \to_8 & \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\\ & = & \Lambda^s_{\mathbf{ct}}(\mu\beta[\delta]t)\end{array}$$

(c) If $\beta \neq \delta$ and β occurs in t

$$\begin{array}{rcl}\Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]\mu\alpha[\delta]t) & = & \mathbf{catch}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\\ & = & \Lambda^s_{\mathbf{ct}}(\mu\beta[\delta]t)\end{array}$$

3. If $\gamma = \beta$ and α occurs then in $[\delta]t\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\beta([\delta]t)\{\beta/\alpha\}$

(a) If $\alpha = \delta$, this case has already been delt with.

(b) If $\alpha \neq \delta$ then $\mu\beta([\delta]t)\{\ \beta/\alpha\}= \mu\beta[\delta]t\{\ \beta/\alpha\}$

— If $\beta = \delta$ then $\mu\beta[\delta]t\{\beta/\alpha\}=\mu\beta[\beta]t\{\beta/\alpha\}$

$$\begin{array}{rcl}\Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]\mu\alpha[\beta]t) & = & \mathbf{catch}\ \beta\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\\ & \to_4 & \mathbf{catch}\ \beta\ (\mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t))\{\beta/\alpha\}\\ & = & \mathbf{catch}\ \beta\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\}\\ & \to_7 & \mathbf{catch}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\}\\ & = & \Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]t\{\beta/\alpha\})\end{array}$$

— If $\beta \neq \delta$ then

$$\begin{array}{rcl}\Lambda^s_{\mathbf{ct}}(\mu\beta[\beta]\mu\alpha[\delta]t) & = & \mathbf{catch}\ \beta\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\\ & \to_4 & \mathbf{catch}\ \beta\ (\mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t))\{\beta/\alpha\}\\ & = & \mathbf{catch}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\}\\ & = & \Lambda^s_{\mathbf{ct}}(\mu\beta[\delta]t\{\beta/\alpha\})\end{array}$$

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

4. If $\gamma \neq \beta$ and γ does not occur in $\mu\alpha[\delta]t$ (then in particular $\gamma \neq \delta$)

   (a) If $\alpha = \delta$ then $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\gamma[\beta]t\{\beta/\alpha\}$

$$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\alpha]t) & = & \mathbf{throw}\ \beta\ \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_6 & \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]t\{\beta/\alpha\}) \end{array}$$

   since $\gamma \neq \beta$ in the last equality.

   (b) If $\alpha \neq \delta$ and α does not occur in t (and thus $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\gamma[\delta]t$)

$$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\delta]t) & = & \mathbf{throw}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_5 & \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t) \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\delta]t) \end{array}$$

   since $\gamma \neq \delta$ in the last equality.

   (c) If $\alpha \neq \delta$ and α occurs in t (and thus $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\gamma[\delta]t\{\beta/\alpha\})$

$$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\delta]t) & = & \mathbf{throw}\ \beta\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_6 & \mathbf{throw}\ \beta\ (\mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t))\{\beta/\alpha\} \\ & = & \mathbf{throw}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ & \to_5 & \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\delta]t\{\beta/\alpha\}) \end{array}$$

5. If $\gamma \neq \beta$ and γ occurs in $\mu\alpha[\delta]t$

   (a) If $\alpha = \delta$ then $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\gamma[\beta]t\{\beta/\alpha\}$

$$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\alpha]t) & = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_6 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]t\{\beta/\alpha\}) \end{array}$$

   (b) If $\alpha \neq \delta$ and α does not occur in t (and thus $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\gamma[\delta]t$)

   — If $\gamma = \delta$ then $\mu\gamma[\delta]t = \mu\gamma[\gamma]t$

$$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\gamma]t) & = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{throw}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_5 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_7 & \mathbf{catch}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t) \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\gamma]t) \end{array}$$

   — If $\gamma \neq \delta$

$$\begin{array}{rcl} \Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\delta]t) & = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t) \\ & \to_5 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t) \\ & = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\delta]t) \end{array}$$

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

(c) If $\alpha \neq \delta$ and α occurs in t (and thus $\mu\gamma([\delta]t)\{\beta/\alpha\} = \mu\gamma[\delta]t\{\beta/\alpha\}$)

— If $\gamma = \delta$ then $\mu\gamma[\delta]t\{\beta/\alpha\} = \mu\gamma[\gamma]t\{\beta/\alpha\}$

$$\begin{array}{ll} \multicolumn{2}{l}{\Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\gamma]t)} \\ = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t) \\ \to_6 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ (\mathbf{throw}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t))\{\beta/\alpha\} \\ = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{throw}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ \to_5 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ \to_7 & \mathbf{catch}\ \gamma\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\gamma]t\{\beta/\alpha\}) \end{array}$$

— If $\gamma \neq \delta$ then $\mu\gamma[\gamma]t\{\beta/\alpha\}$

$$\begin{array}{ll} \multicolumn{2}{l}{\Lambda^s_{\mathbf{ct}}(\mu\gamma[\beta]\mu\alpha[\delta]t)} \\ = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t) \\ \to_6 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ (\mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t))\{\beta/\alpha\} \\ = & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \beta\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ \to_5 & \mathbf{catch}\ \gamma\ \mathbf{throw}\ \delta\ \Lambda^s_{\mathbf{ct}}(t)\{\beta/\alpha\} \\ = & \Lambda^s_{\mathbf{ct}}(\mu\gamma[\delta]t\{\beta/\alpha\}) \end{array}$$

- Case of the rule $\mu\alpha[\alpha]t \to t$ if α does not occur free in t.

$$\Lambda^s_{\mathbf{ct}}(\mu\alpha[\alpha]t) = \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(t) \to_8 \Lambda^s_{\mathbf{ct}}(t)$$

□

Lemma 3.4.3
For any $\lambda\mu$-context $C[\bullet]$ and any $\lambda\mu$-terms u, v, if $\Lambda^s_{\mathbf{ct}}(u) \to^\star_{\mathbf{ct}} \Lambda^s_{\mathbf{ct}}(v)$ then:

$$\Lambda^s_{\mathbf{ct}}(C[u]) \to^\star_{\mathbf{ct}} \Lambda^s_{\mathbf{ct}}(C[v])$$

Proof
By induction on the context:

- If the context is $\bullet$, $\Lambda^s_{\mathbf{ct}}(C[u]) = \Lambda^s_{\mathbf{ct}}(u) \to^\star_{\mathbf{ct}} \Lambda^s_{\mathbf{ct}}(v) = \Lambda^s_{\mathbf{ct}}(C[v])$.
- If the context is an application or an abstraction, just apply the induction hypothesis.
- If the context has the form $\mu\alpha[\beta]C[\bullet]$ then, by definition of the translation $\Lambda^s_{\mathbf{ct}}$, on the one hand,

$$\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[u]) = \begin{cases} \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(C[u]) & \textit{if } \alpha = \beta \\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[u]) & \textit{if } \alpha \neq \beta \textit{ and } \alpha \textit{ is not free in } C[u] \\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[u]) & \textit{otherwise} \end{cases}$$

and on the other hand,

$$\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[v]) = \begin{cases} \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(C[v]) & \textit{if } \alpha = \beta \\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[v]) & \textit{if } \alpha \neq \beta \textit{ and } \alpha \textit{ is not free in } C[v] \\ \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[v]) & \textit{otherwise} \end{cases}$$

By induction hypothesis $\Lambda^s_{\mathbf{ct}}(C[u]) \to^\star_{\mathbf{ct}} \Lambda^s_{\mathbf{ct}}(C[v])$, and then we consider each case:

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

1. If $\alpha = \beta$, by applying the induction assumption:

$$\begin{aligned}\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[u]) &= \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(C[u]) \\ &\to^\star_{\mathbf{ct}} \mathbf{catch}\ \alpha\ \Lambda^s_{\mathbf{ct}}(C[v]) = \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[v])\end{aligned}$$

2. If $\alpha \neq \beta$ and α does not occur free in $C[u]$, then α cannot occur free in $C[v]$ because no reduction of the $\lambda\mu$-calculus can introduce a free μ-variable, hence by applying the induction hypothesis:

$$\begin{aligned}\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[u]) &= \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[u]) \\ &\to^\star_{\mathbf{ct}} \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[v]) = \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[v])\end{aligned}$$

3. If $\alpha \neq \beta$ and α occurs free in $C[u]$, and α do not occur free then in $C[v]$ by applying the induction hypothesis and then rule (8):

$$\begin{aligned}\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[u]) &= \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[u]) \\ &\to^\star_{\mathbf{ct}} \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[v]) \\ &\to_1 \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[v]) \\ &= \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[v])\end{aligned}$$

4. If $\alpha \neq \beta$ and α occurs free in $C[u]$ and in $C[v]$ by applying the induction hypothesis:

$$\begin{aligned}\Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[u]) &= \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[u]) \\ &\to^\star_{\mathbf{ct}} \mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^s_{\mathbf{ct}}(C[v]) = \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]C[v])\end{aligned}$$

□

Proposition 3.4.4
For any $\lambda\mu$-terms u, v, if $u \to_{\lambda\mu} v$ then $\Lambda^s_{\mathbf{ct}}(u) \to^\star_{\mathbf{ct}} \Lambda^s_{\mathbf{ct}}(v)$.

Proof
By lemma 3.4.1 and lemma 3.4.3. □

Remark. The morphism $\Lambda^s_{\mathbf{ct}}$ does not map any reduction step onto at least one reduction step since $\mu\alpha[\alpha]\mu\_[\beta]t \to \mu\alpha[\beta]t$ while $\Lambda^s_{\mathbf{ct}}(\mu\alpha[\alpha]\mu\_[\beta]t) = \Lambda^s_{\mathbf{ct}}(\mu\alpha[\beta]t)$.

Corollary 3.4.5
The $\lambda_{\mathbf{ct}}$-calculus and the $\lambda\mu$-calculus are isomorphic if we consider terms up renaming/simplification.

Proof
Indeed, for any $\lambda_{\mathbf{ct}}$-term t, we have:

$$\overline{\lambda_{\mathbf{ct}}(\Lambda^\iota_\mu(t))} = \lambda_{\mathbf{ct}}(\overline{\Lambda^\iota_\mu(t)}) = \lambda_{\mathbf{ct}}(\Lambda^s_\mu(\bar{t})) = \bar{t}$$

since $\lambda_{\mathbf{ct}}$ and Λ^ι_μ are canonical. □

Corollary 3.4.6
The $\lambda_{\mathbf{ct}}$-calculus and the $\lambda\mu$-calculus are isomorphic if we consider terms up convertibility.

Proof
Since $\overline{\lambda_{\mathbf{ct}}(\Lambda^\iota_\mu(t))} = \bar{t}$ implies $\lambda_{\mathbf{ct}}(\Lambda^\iota_\mu(t)) =_{\mathbf{ct}} t$. □

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

3.5 *Confluence of the $\lambda_{\mathbf{ct}}$-calculus*

We are now able to show the confluence of the $\lambda_{\mathbf{ct}}$-calculus. Indeed, let us consider following diagram (where w exists since the $\lambda\mu$-calculus is confluent):

$$\begin{tikzcd}
 & t \arrow[dl, "\to^{\star}_{\mathbf{ct}}"'] \arrow[d, "\Lambda^{\iota}_{\mu}"'] \arrow[dr, "\to^{\star}_{\mathbf{ct}}"] & \\
u \arrow[d, "\Lambda^{\iota}_{\mu}"'] & \Lambda^{\iota}_{\mu}(t) \arrow[dl, "\to^{\star}_{\lambda\mu}"] \arrow[dr, "\to^{\star}_{\lambda\mu}"'] & v \arrow[d, "\Lambda^{\iota}_{\mu}"] \\
\Lambda^{\iota}_{\mu}(u) \arrow[d, "\Lambda^{s}_{\mathbf{ct}}"'] \arrow[dr, "\to^{\star}_{\lambda\mu}"] & & \Lambda^{\iota}_{\mu}(v) \arrow[d, "\Lambda^{s}_{\mathbf{ct}}"] \arrow[dl, "\to^{\star}_{\lambda\mu}"'] \\
\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(u)) \arrow[dr, "\to^{\star}_{\mathbf{ct}}"'] & w \arrow[d, "\Lambda^{s}_{\mathbf{ct}}"'] & \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(v)) \arrow[dl, "\to^{\star}_{\mathbf{ct}}"] \\
 & \Lambda^{s}_{\mathbf{ct}}(w) &
\end{tikzcd}$$

Theorem 3.5.1
The $\lambda_{\mathbf{ct}}$-calculus is confluent.

Proof
We have shown that for any $\lambda_{\mathbf{ct}}$-term t, $t = \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(t))$, (only the weaker property $t \to^{\star}_{\mathbf{ct}} \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(t))$ was actually needed) and the confluence of the $\lambda_{\mathbf{ct}}$-calculus results from the following diagram:

$$\begin{tikzcd}
 & t \arrow[dl, "\to^{\star}_{\mathbf{ct}}"'] \arrow[dr, "\to^{\star}_{\mathbf{ct}}"] & \\
\Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(u)) = u \arrow[dr, "\to^{\star}_{\mathbf{ct}}"'] & & v = \Lambda^{s}_{\mathbf{ct}}(\Lambda^{\iota}_{\mu}(v)) \arrow[dl, "\to^{\star}_{\mathbf{ct}}"] \\
 & \Lambda^{s}_{\mathbf{ct}}(w) &
\end{tikzcd}$$

□

3.6 *The $\lambda\mu$-calculus is not a canonical retract of the $\lambda_{\mathbf{ct}}$-calculus*

Proposition 3.6.1
There is no surjective canonical morphism from the $\lambda_{\mathbf{ct}}$ -calculus to the $\lambda\mu$-calculus.

Proof
Let us assume that Ψ is a surjective canonical morphism from the $\lambda_{\mathbf{ct}}$ -calculus to the $\lambda\mu$-calculus. Let r be a simple $\lambda\mu$-term of the form $(\mu\alpha[\beta]u\ v)$ where $\alpha \neq \beta$ and β occurs free in u. Since Ψ is surjective, there is a $\lambda_{\mathbf{ct}}$-term s such that $\Psi(s) = r$. Since Ψ is canonical, $\overline{\Psi(s)} = \Lambda^{s}_{\mu}(\overline{s})$. But $\overline{\Psi(s)} = \overline{r} = r$ since r is simple. Thus the simple $\lambda_{\mathbf{ct}}$-term $\overline{s} = \Lambda^{s}_{\mathbf{ct}}(r)$ is $((\mathbf{catch}\ \alpha\ \mathbf{throw}\ \beta\ \Lambda^{s}_{\mathbf{ct}}(u)\ \Lambda^{s}_{\mathbf{ct}}(v))$. Now let c be the contractum of $\overline{s}$:

$$c = \mathbf{catch}\ \alpha\ (\mathbf{throw}\ \beta\ \Lambda^{s}_{\mathbf{ct}}(u)\big\{\mathbf{throw}\ \alpha\ (t\ \Lambda^{s}_{\mathbf{ct}}(v))/\mathbf{throw}\ \alpha\ t\big\}\ \Lambda^{s}_{\mathbf{ct}}(v))$$

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

Since Ψ is a morphism, we should have $\Psi(\bar{s}) \to^\star \Psi(c)$ and thus $\overline{\Psi(\bar{s})} \to^\star \overline{\Psi(c)}$ (since the renaming/simplification rules commute with any other rule). Finally, notice that $r = \overline{\Psi(s)} = \overline{\Psi(\bar{s})}$ but we do not have in general, $r \to^\star \mu\alpha[\alpha](\mu\delta[\beta]u\{[\alpha](t\ v)/t\}\ v)$ (for instance if the contractum of r which is $\mu\alpha[\beta]u\{[\alpha](t\ v)/t\}$ is a normal form). Whence the contradiction. $\square$

4 The typed λ_{ct}-calculus

Typing control operators is strongly related to classical logic. This striking fact has been first noticed by Griffin (1990) and has been widely investigated since, for instance, by Murthy (1990; 1991), Barbanera and Berardi (1994*a*; 1994*b*), Rehof and Sørensen (1994), De Groote (1995; 1994), Krivine (1994), Nakano (1994*b*; 1994*a*; 1995), Sato and Kamayema (1997; 1998; 1997) and Parigot (1992; 1993).

As far as the author knows, Parigot's $\lambda\mu$-calculus is the only λ-calculus with control operators for which strong normalization has been proved in the second order framework.

We first recall the typing rules of this calculus (for more details see Parigot (1992)). Then we derive the typing rules of the **catch** and **throw** operators: they correspond respectively to right contraction and weakening rules of classical natural deduction. The subject reduction property and strong normalization are straightforward consequences.

4.1 The typed $\lambda\mu$-calculus

Axiom

$$x : A^x \vdash A$$

Rules for $\to$

$$\frac{t : \Gamma, A^x \vdash \Delta; B}{\lambda x.t : \Gamma \vdash \Delta; A \to B} \qquad \frac{u : \Gamma \vdash \Delta; A \to B \qquad v : \Gamma \vdash \Delta; A}{(u\ v) : \Gamma \vdash \Delta; B}$$

Rules for $\forall$ (where x does not occur free in Γ, Δ in the introduction rule)

$$\frac{u : \Gamma \vdash \Delta; A}{u : \Gamma \vdash \Delta; \forall x A} \qquad \frac{u : \Gamma \vdash \Delta; \forall x A}{u : \Gamma \vdash \Delta; A\{t/x\}}$$

Rules for $\forall^2$ (where X does not occur free in Γ, Δ in the introduction rule)

$$\frac{u : \Gamma \vdash \Delta; A}{u : \Gamma \vdash \Delta; \forall X A} \qquad \frac{u : \Gamma \vdash \Delta; \forall X A}{u : \Gamma \vdash \Delta; A\{T/X\}}$$

Contraction and weakening rules (where Π is either empty or a single formula B)

$$\frac{t : \Gamma, A^x, A^y \vdash \Delta; \Pi}{t\{x/y\} : \Gamma, A^x \vdash \Delta; \Pi} \qquad \frac{t : \Gamma \vdash \Delta; \Pi}{t : \Gamma, A^x \vdash \Delta; \Pi}$$

$$\frac{t : \Gamma \vdash \Delta, A^\alpha, A^\beta; \Pi}{t\{\beta/\alpha\} : \Gamma \vdash \Delta, A^\beta; \Pi} \qquad \frac{t : \Gamma \vdash \Delta; \Pi}{t : \Gamma \vdash \Delta, A^\alpha; \Pi}$$

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

Remark. As usual in natural deduction, these explicit contraction and weakening rules are not actually needed if we allow for 'generalized axioms':

$$x : \Gamma, A^x \vdash \Delta; A$$

Naming rules

These are the rules of the $\lambda\mu$-calculus that allow for multiple conclusions:

$$\frac{t : \Gamma \vdash \Delta; A}{[\alpha]t : \Gamma \vdash \Delta, A^\alpha;} \qquad \frac{t : \Gamma \vdash \Delta, A^\alpha;}{\mu\alpha.t : \Gamma \vdash \Delta; A}$$

4.2 Typing the **catch** *and* **throw** *operators*

Let us now use the naming rules to derive type judgments for the $\lambda\mu ct$-terms *throw* α t and *catch* α t.

- We recall that *catch* α $t = \mu\alpha[\alpha]t$:

$$\cfrac{\cfrac{t : \Gamma \vdash \Delta, A^\alpha; A}{[\alpha]t : \Gamma \vdash \Delta, A^\alpha;}}{\mu\alpha[\alpha]t : \Gamma \vdash \Delta; A}$$

- We recall that *throw* α $t = \mu\beta[\alpha]t$ where β does not occur free in t:

$$\cfrac{\cfrac{\cfrac{t : \Gamma \vdash \Delta; A}{[\alpha]t : \Gamma \vdash \Delta, A^\alpha;}}{[\alpha]t : \Gamma \vdash \Delta, A^\alpha, B^\beta;}}{\mu\beta[\alpha]t : \Gamma \vdash \Delta, A^\alpha; B}$$

Hence, we are now able to type the native **throw** and **catch** operators.

The catch rule

$$\frac{t : \Gamma \vdash \Delta, A^\alpha; A}{\textbf{catch}\ \alpha\ t : \Gamma \vdash \Delta; A}$$

The throw rule

$$\frac{t : \Gamma \vdash \Delta; A}{\textbf{throw}\ \alpha\ t : \Gamma \vdash \Delta, A^\alpha; B}$$

Remark. As for the $\lambda\mu$-calculus, there is no need for explicit rules for contracting and weakening 'named' conclusions. Consequently, one can see these catch and throw rules respectively as explicit right-hand contraction and weakening rules for classical natural deduction.

Proposition 4.2.1
A $\lambda_{\mathbf{ct}}$-term t is typable of type $\Gamma \vdash \Delta; A$ if and only if $\lambda\mu ct$-term $\Lambda^\iota_\mu(t)$ is typable of type $\Gamma \vdash \Delta; A$.

Example. The $\lambda_{\mathbf{ct}}$-term $\lambda y.$**catch** α $(y\ \lambda x.$**throw** α $x)$, which represents the famous **call/cc** of the Scheme language just as the corresponding $\lambda\mu ct$-term (Parigot, 1992),

https://doi.org/10.1017/S0956796899003512 Published online by Cambridge University Press

can be typed by Peirce's axiom:

$$\dfrac{\dfrac{y : ((A \to B) \to A)^y \vdash (A \to B) \to A \qquad \dfrac{\dfrac{x : A^x \vdash A}{\textbf{throw}\ \alpha\ x : A^x \vdash A^\alpha ; B}}{\lambda x.\textbf{throw}\ \alpha\ x \vdash A^\alpha ; A \to B}}{\dfrac{(y\ \lambda x.\textbf{throw}\ \alpha\ x) : ((A \to B) \to A)^y \vdash A^\alpha ; A}{\textbf{catch}\ \alpha\ (y\ \lambda x.\textbf{throw}\ \alpha\ x) : ((A \to B) \to A)^y \vdash A}}}{\lambda y.\textbf{catch}\ \alpha\ (y\ \lambda x.\textbf{throw}\ \alpha\ x) \vdash ((A \to B) \to A) \to A}$$

4.3 Subject reduction property

The subject reduction property holds for the second order $\lambda\mu$ -calculus: if a $\lambda\mu$-term t is typable of the type $\Gamma \vdash \Delta; A$ and $t \to_{\lambda\mu} t'$ then t' is also typable of the type $\Gamma \vdash \Delta; A$. This property extends directly to the $\lambda_{\mathbf{ct}}$-calculus:

Proposition 4.3.1

Given a $\lambda_{\mathbf{ct}}$-term t, if t is typable of type $\Gamma \vdash \Delta; A$ and $t \to_{\mathbf{ct}} t'$ then t' is also typable of the type $\Gamma \vdash \Delta; A$.

Proof

By proposition 4.2.1, if t is typable of type $\Gamma \vdash \Delta; A$, then $\Lambda_\mu^\iota(t)$ is also typable of type $\Gamma \vdash \Delta; A$. We know that $\Lambda_\mu^\iota(t) \to_{\lambda\mu} \Lambda_\mu^\iota(t')$, and since the subject reduction property holds for the $\lambda\mu$-calculus, $\Lambda_\mu^\iota(t')$ is typable of type $\Gamma \vdash \Delta; A$, and again by proposition 4.2.1, t' is also typable of type $\Gamma \vdash \Delta; A$. $\square$

4.4 Strong normalization of the second order $\lambda_{\mathbf{ct}}$-calculus

The $\lambda\mu$-calculus is strongly normalizing in the second order framework, i.e. if a $\lambda\mu$-term t is typable of type $\Gamma \vdash \Delta; A$ then there is no infinite sequence of reductions starting from t. This property extends directly to the $\lambda_{\mathbf{ct}}$-calculus.

Proposition 4.4.1

Given a $\lambda_{\mathbf{ct}}$-term t, if t is typable of type $\Gamma \vdash \Delta; A$ then there is no infinite sequence of reductions starting from t.

Proof

If t is typable of type $\Gamma \vdash \Delta; A$, then the $\lambda\mu\mathbf{ct}$-term $\Lambda_\mu^\iota(t)$ is also typable of type $\Gamma \vdash \Delta; A$. If there was an infinite sequence of reductions $t_1 \to_{\mathbf{ct}} t_2 \ldots \to_{\mathbf{ct}} t_n \ldots$ then, since Λ_μ^ι preserves one-step reduction (proposition 3.3.3), there would be an infinite sequence of reductions $\Lambda_\mu^\iota(t_1) \to_{\lambda\mu} \Lambda_\mu^\iota(t_2) \ldots \to_{\lambda\mu} \Lambda_\mu^\iota(t_n) \ldots$, which contradicts the strong normalization of the $\lambda\mu$-calculus. $\square$

Remark. The converse is also true: the strong normalization of the $\lambda_{\mathbf{ct}}$-calculus implies the strong normalization of the $\lambda\mu$-calculus. Indeed, the translation $\Lambda_{\mathbf{ct}}^\iota$ is also an injective morphism. Moreover, a $\lambda\mu$-term t is typable of type $\Gamma \vdash \Delta; A$ if and only if $\Lambda_{\mathbf{ct}}^\iota(t)$ is typable of type $\Gamma \vdash \Delta; A$.

5 Conclusion

We have defined a confluent λ-calculus with a catch/throw mechanism. Any $\lambda_{\mathbf{ct}}$-term typable in the second order classical natural deduction is strongly normalizing. We have also seen that the **call/cc** of Scheme can be defined as:

$$\mathbf{call/cc}\ t \equiv \mathbf{catch}\ \alpha\ (t\ \lambda x.\mathbf{throw}\ \alpha\ x)$$

De Groote (1994) has shown in that the $\lambda\mu$-calculus is isomorphic (modulo convertibility) to the C-calculus. Similarly, it would be interesting to study how the λ-calculus with a 'native' **call/cc** is related to the $\lambda_{\mathbf{ct}}$-calculus. Besides, we have only investigated here Parigot's 'call-by-name' $\lambda\mu$-calculus. Ong and Stewart (1996; 1997) have proposed a 'call-by-value' $\lambda\mu$-calculus. It is likely that a 'call-by-value' $\lambda_{\mathbf{ct}}$-calculus can be derived from their work. Notice that De Groote (1994) and Ong (1996; 1997) separate the μ and the [ ] in their $\lambda\mu$-calculus. Nevertheless, this separation does not define a catch/throw mechanism (since in $\mu\alpha.t$ the type of t is $\perp$).

We did not consider tag-abstraction as in the work of Nakano, Kamayema and Sato, since there is no need for tag-abstraction in the classical framework where a tag (μ-variable) α can be reified as the term $\lambda x.\mathbf{throw}\ \alpha\ x$ whose type is $\vdash A^{\alpha}; \neg A$ (first-class continuations are typed by the negation $\neg A \equiv A \rightarrow \perp$). Of course, this is not sound anymore in intuitionistic logic since this type is the excluded-middle. We consider tag-abstraction in a constructive framework in a forthcoming paper, but where subtraction, the connector dual to implication (see Crolard, 1996, 1998) will be used instead of disjunction.

Acknowlegements

We would like to thank Serge Grigorieff for important suggestions and careful technical proofreading. The many discussions with Hugo Herbelin have been essential for this work. Comments of the anonymous referees have also been very useful.

References

Barbanera, F. and Berardi, S. (1994*a*) A symmetric lambda calculus for 'classical' program extraction. *Theoretical Aspects of Computer Software: Lecture Notes in Computer Science 542*, pp. 495–515. Springer-Verlag.

Barbanera, F. and Berardi, S. (1994*b*) Extracting constructive content from classical logic via control-like reductions. *Lecture Notes in Compter Science 662*. Springer-Verlag.

Beus, B. and Streicher, T. (1998) Classical logic, continuation semantics and abstract machines. *J. Functional Programming* (submitted).

Crolard, T. (1996) *Extension de l'isomorphisme de Curry-Howard au traitement des exceptions (application d'une étude de la dualité en logique intuitionniste)*. Thèse de Doctorat, Université Paris 7.

Crolard, T. (1999) *Subtractive logic. Theoretical Comput. Sci.* (to appear).

De Groote, P. (1994) A CPS-Translation of the $\lambda\mu$-calculus. *Lecture Notes in Computer Science 787*, p. 85.

De Groote, P. (1994*b*) On the relation between the lambda-calculus and the syntactic theory of sequential control. *Lecture Notes in Computer Science 822*, p. 31.

De Groote, P. (1995) A simple calculus of exception handling. *2nd Int. Conference on Typed Lambda Calculi and Applications: Lecture Notes in Computer Science*, pp. 201–215. Springer-Verlag.

De Groote, P. (1999) An environment machine for the $\lambda\mu$-calculus. Mathematical Structure of Computer Science (to appear).

Felleisen, M., Friedman, D. P., Kohlbecker, E. and Duba, B. F. (1986) Reasoning with continuations. *First Symp. on Logic and Computer Science*, pp. 131–141.

Felleisen, M., Friedman, D. P., Kohlbecker, E. and Duba, B. F. (1987) A syntactic theory of sequential control. *Theoretical Comput. Sci.* **52**(3), 205–237.

Griffin, T. G. (1990) A formulæ-as-type notion of control. *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 47–58.

Kameyama, Y. (1997) A new formulation of the catch/throw mechanism. In: Ida, T., Ohori, A. and Takeichi, M. (eds.), *Second Fuji International Workshop on Functional and Logic Programming*, pp. 106–122. World Scientific.

Kameyama, Y. and Sato, M. (1998) A classical catch/throw calculus with tag abstraction and its strong normalizability. In: Lin, X. (ed.), *Proc. 4th Australasian Theory Symp.: Australian Computer Science Communications 20-3*, pp. 183–197. Springer-Verlag.

Krivine, J.-L. (1994) Classical logic, storage operators and second order λ-calculus. *Ann. Pure & Appl. Logic*, **68**, 53–78.

Murthy, C. R. (1990) *Extracting constructive content from classical proofs*. PhD thesis, Department of Computer Science, Cornell University.

Murthy, C. R. (1991) Classical proofs as programs: How, when, and why. *Technical Report 91-1215*, Department of Computer Science, Cornell University.

Nakano, H. (1994*a*) A constructive logic behind the catch and throw mechanism. *Ann. Pure & Appl. Logic*, **69**(3), 269–301.

Nakano, H. (1994*b*) The non-deterministic catch and throw mechanism and its subject reduction property. *Logic, Language and Computation: Lecture Notes in Computer Science 592*, pp. 61–72. Springer-Verlag.

Nakano, H. (1995) *The logical structures of the catch and throw mechanism*. PhD thesis, The University of Tokyo.

Ong, C.-H. L. and Stewart, C. A. (1997) A Curry–Howard foundation for functional computation with control. *Conference record of POPL '97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 215–227.

Ong, C.-H. L. (1996) A semantic view of classical proofs: Type-theoretic, categorical, and denotational characterizations (preliminary extended abstract). *Proceedings 11th Ann. IEEE Symp. on Logic in Computer Science, LICS'96*, pp. 230–241. New Brunswick, NJ. IEEE Press.

Parigot, M. (1992) $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. *Proc. Logic Prog. and Autom. Reasoning: Lecture Notes in Computer Science 624*, pp. 190–201. Springer-Verlag.

Parigot, M. (1993) Classical proofs as programs. *Computational Logic and Theory: Lecture Notes in Computer Science 713*, pp. 263–276. Springer-Verlag.

Py, W. (1998) *Propriétés combinatoires du $\lambda\mu$-calcul*. Thèse de Doctorat, Université de Savoie.

Rehof, N. J. and Sørensen, M. H. (1994) The λ_Δ-calculus. *Theoretical Aspects of Computer Software: Lecture Notes in Computer Science 542*, pp. 516–542. Springer-Verlag.

Reynolds, J. C. (1993) The discoveries of continuations. *Lisp and Symbolic Computation*, **6**, 233–248.

Sato, M. (1997) Intuitionistic and classical natural deduction systems with the Catch and the Throw rules. *Theoretical Comput. Sci.* **175**(1), 75–92.