DOI-TR-176

Technical Report

Combinatory Logic and λ-Calculus for Classical Logic

by

Kensuke Baba, Yukiyoshi Kameyama, Sachio Hirokawa

August 18, 2000

Department of Informatics
Kyushu University
Fukuoka 812-81, Japan

Email: baba@i.kyushu-u.ac.jp Phone: +81-92-641-3131(8806)

Combinatory Logic and λ-Calculus for Classical Logic

Kensuke Baba[1], Yukiyoshi Kameyama[2], and Sachio Hirokawa[3]

[1] Graduate School of Information Science and Electrical Engineering, Kyushu University
[2] Department of Information Science, Kyoto University
[3] Computer Center, Kyushu University

Abstract. Since Griffin's work in 1990, classical logic has been an attractive target for extracting computational contents. However, the classical principle used in Griffin's type system is the double-negation-elimination rule, which prevents one to analyze the intuitionistic part and the purely classical part separately. By formulating a calculus with **J** (for the elimination rule of falsehood) and **P** (for Peirce formula which is concerned with purely classical reasoning) combinators, we can separate these two parts. This paper studies the $\lambda\mathbf{PJ}$ calculus with **P** and **J** combinators and the $\lambda\mathbf{C}$ calculus with **C** combinator(for the double-negation-elimination rule). We also propose two λ-calculi which correspond to $\lambda\mathbf{PJ}$ and $\lambda\mathbf{C}$. We give four classes of reduction rules for each calculus, and systematically study their relationship by simulating reduction rules in one calculus by the corresponding one in the other. It is shown that, by restricting the type of **P**, simulation succeeds for several choices of reduction rules, but that simulating the full calculus $\lambda\mathbf{PJ}$ in $\lambda\mathbf{C}$ succeeds only for one class. Some programming examples of our calculi such as encoding of conjunction and disjunction are also given.

1 Introduction

The λ-calculus possesses several features of programming language and their implementations. In spite of its very simple syntax, the λ-calculus is strong enough to describe all mechanically computable functions. Therefore the λ-calculus can be viewed as a paradigmatic programming language. For example, a term $\lambda x.M$ represents the function whose parameter is x, and MN is considered the result of applying function M to argument N. The value of function $\lambda x.M$ at an argument N is calculated by substituting N for x in M, so $(\lambda x.M)N$ can be simplified to $M[x := N]$. The term rewriting procedure

$$(\lambda x.M)N \to M[x := N]$$

is called β-reduction. In order to deal with functions with domain and range, the notion of type is used for λ-calculus. Types are assigned to λ-terms as follows.

$$\frac{M : \beta}{\lambda x^{\alpha}.M : \alpha \to \beta} \qquad \frac{M : \alpha \to \beta \quad N : \alpha}{MN : \beta}$$

If the type of parameter x is α and the type of M is β, the type of function $\lambda x.M$ is $\alpha \to \beta$. If the type of function M is $\alpha \to \beta$ and the type of argument N is α, the type of MN is β. If we regard types as logical formulae, they are consistent with inference rules on implicational fragment $\to$. Namely, types can be considered formulae and terms can be considered proofs, respectively. Moreover, normalizations of proof figure correspond to reduction rules of λ-term. For example, the elimination of inference rules corresponds to the β-reduction as follows.

$$\dfrac{\dfrac{\begin{array}{c}[x:\alpha]\\ \vdots\ P\\ M:\beta\end{array}}{\lambda x.M : \alpha \to \beta} \quad \begin{array}{c}\vdots\ Q\\ N:\alpha\end{array}}{(\lambda x.M)N : \beta} \qquad \text{reduces to} \qquad \begin{array}{c}\vdots\ Q\\ N:\alpha\\ [x:\alpha]\\ \vdots\ P\\ M[x := N] : \beta\end{array}$$

This correspondence between logic and typed λ-calculi is the Curry-Howard Isomorphism[5].

Historically the isomorphism had been restricted to intuitionistic logic, since no constructive interpretation was (believed to be) possible for classical logic. A breakthrough was brought by T. Griffin [2] in 1990. He assigned a consistent type to Felleisen's $\mathcal{C}$-operator, and showed that it corresponds to the double-negation-elimination rule (the $\neg\neg$-elim-rule), hence the system gives computational meaning to classical proofs. After him, many researchers formulated typed λ-calculi which corresponds to classical logic, such as Parigot's $\lambda\mu$-calculus[8], its call-by-value variant[7], de Groote's `exception` calculus, and classical catch/throw calculus $\mathsf{NK}_{c/t}$[10]. These are extension of the isomorphism to classical logic from a computational viewpoint, but we extend it from a logical one. The purpose of our research is to find the reduction rule (or the normalization of proof, in a logical sense) peculiar to classical logic. In order to make the difference between various logic clear, we restrict logical connectives to implicational fragment and falsehood $\bot$ in this paper.

Although Griffin's type system for the $\mathcal{C}$-operator is really striking, there are at least two unpleasant points in his type system.

Firstly, the $\neg\neg$-elim-rule is so powerful that adding the rule to minimal logic (intuitionistic logic without the $\bot$-elimination rule) yields classical logic, hence, we cannot distinguish which parts of reduction rules come from intuitionistic logic, and which parts are purely classical one. Instead of introducing the $\neg\neg$-elim-rule, we can choose the combination of the $\bot$-elimination rule and the so-called Peirce formula $((\alpha \to \beta) \to \alpha) \to \alpha$ as the basis of our classical calculi. The situation is illustrated as follows:

$$\begin{aligned} \text{classical logic} &= \text{minimal logic} + \quad \neg\neg\text{-elim-rule} \\ &= \text{minimal logic} + \bot\text{-elim-rule} + \text{Peirce formula} \\ &= \quad \text{intuitionistic logic} \quad + \text{Peirce formula} \end{aligned}$$

Then, we may be able to distinguish reduction rules for intuitionistic logic (concerning the $\bot$-elimination rule), and reduction rules for the purely classical part (concerning Peirce formula).

The second unpleasant point of Griffin's type system is well known; in order to apply the following Felleisen's reduction, the whole term must have the type $\bot$.

$$E[\mathcal{C}N] \to N(\lambda x.\mathcal{A}(E[x]))$$

Since the $\bot$ type is the empty type, we can never use this reduction rule. Different solutions to this problem have already been proposed by Griffin and Felleisen; yet we can avoid this problem by simply going back to the **call/cc**-operator which has the following reduction rule:

$$E[\mathbf{call/cc}N] \to E[N(\lambda x.\mathcal{A}(E[x]))]$$

In this reduction rule, the whole term $E[\mathbf{call/cc}N]$ may have arbitrary type β, in that case the most general type of **call/cc** is $((\alpha \to \beta) \to \alpha) \to \alpha$, Peirce formula.

In summary, the calculus with the $\bot$-elimination rule and Peirce formula is worth studying in detail. With this motivation in mind, we propose in this paper two combinatory-logic-style calculi $\lambda\mathbf{PJ}$ and $\lambda\mathbf{C}$. The former has two

combinators **P** (for Peirce formula) and **J** (for the $\bot$-elim-rule), and the latter has one combinator **C** (for the $\neg\neg$-elim-rule). Besides β- and η-reductions, we classify classical reduction rules into four classes; the logical reduction, simplification reduction, base-case reduction, and η-like reduction. By appropriately choosing a certain class of reduction rules, our calculi have close connection to classical calculi proposed in the literature, for example, Felleisen's ($\mathcal{C}_{left}$) rule is close to our simplification rule.

We then study the relationship between $\lambda\mathbf{PJ}$ and $\lambda\mathbf{C}$ by defining translation from one to the other, and check if a class of reduction rules in one calculus can be simulated by the corresponding class of reduction rules in the other calculus. It turned out that, the simulation mostly works well if we restrict the type of **P**, but does not work so well for the full **P**.

We shall also present λ-calculi $\overline{\lambda}\mathbf{J}$ and $\underline{\lambda}$ which corresponds to these calculi. This λ-calculus-style is more convenient to use, since the notation of these calculi is more compact than the combinatory-logic style. We also study the relation between the λ-calculus style and the combinatory-logic style by giving translations from one to the other. As expected, this translation works well.

The four calculi above and translations between them can be summarized as the following diagram.

$$\begin{array}{ccc} \lambda\mathbf{C} & \overset{\circ}{\underset{\bullet}{\rightleftarrows}} & \lambda\mathbf{PJ} \\ \sharp\downarrow\uparrow\flat & & \sharp\downarrow\uparrow\flat \\ \underline{\lambda} & \overset{\circ}{\underset{\bullet}{\rightleftarrows}} & \overline{\lambda}\mathbf{J} \end{array}$$

The main contribution of this paper is (1) to give two calculi each of which has combinators for classical inference rules, (2) to analyze various classes of reduction rules systematically, and (3) to consider the λ-calculus-style presentation and give the precise relation to combinatory-logic-style presentation.

The rest of this paper is organized as follows: Sections 2 and 3 presents combinatory-logic-style calculi and λ-calculus-style calculi for classical logic, respectively, and study translations between them. Section 4 briefly mentions properties of our calculi such as confluency. Section 5 presents programming examples

of our calculi. Comparison with other works including Parigot's $\lambda\mu$-calculus as well as concluding remarks are given in Section 6.

2 Combinatory Logic for Classical Logic

In this section we propose two calculi $\lambda\mathbf{PJ}$ and $\lambda\mathbf{C}$ as extensions of the simply typed λ-calculus, then study their relationship.

2.1 Syntax

The starting system of all our calculi in this paper is the implicational fragment of minimal logic, (or the simply typed λ-calculus, under the Curry-Howard Isomorphism). We believe that the implicational fragments are the essence of various logics, and it is relatively easy to introduce other connectives to the calculi once after we understand the behavior of implicational fragments.

Types are defined as the following grammar where $\mathtt{A}$ is a metavariable for atomic types other than $\bot$ (falsehood).

$$\alpha ::= \mathtt{A} \mid \bot \mid \alpha \to \alpha$$

Negation is defined in a standard way by $\neg\alpha \stackrel{\triangle}{=} \alpha \to \bot$.

Preterms of $\lambda\mathbf{C}$ and $\lambda\mathbf{P}$ are given as follows where x^α is a metavariable for variables of type α.

$$(for\ \ \lambda\mathbf{C}) \quad M ::= x^\alpha \mid \lambda x^\alpha.M \mid (MM) \mid \mathbf{C}$$
$$(for\ \ \lambda\mathbf{PJ}) \quad M ::= x^\alpha \mid \lambda x^\alpha.M \mid (MM) \mid \mathbf{P} \mid \mathbf{J}$$

As usual, λ is the only binding operator, and bound/free occurrences of variables are defined as standard. The set of free variables in M is denoted as $FV(M)$. We may sometimes omit the type α in a variable x^α. We then give typing rules.

Typing Rules for Ordinary λ-terms

$$\frac{}{x^\alpha : \alpha} \qquad \frac{M : \beta}{\lambda x^\alpha.M : \alpha \to \beta} \qquad \frac{M : \alpha \to \beta \quad N : \alpha}{MN : \beta}$$

Typing Rules for Combinators

$$\overline{\mathbf{C} : \neg\neg\alpha \to \alpha} \qquad \overline{\mathbf{P} : ((\alpha \to \beta) \to \alpha) \to \alpha} \qquad \overline{\mathbf{J} : \bot \to \alpha}$$

Terms are preterms which are typed by the above typing rules. Types are sometimes written as a superscript, for instance, $(\mathbf{J}^{\bot\to\alpha}M^{\bot})^{\alpha}$. Substitution $M[x^{\alpha} := N^{\alpha}]$ is defined as usual. We use lowercase letters such as x, y, z, u, v for variables, uppercase letters such as L, M, N for terms, and Greek letters such as α, β, γ for types. If the $\mathbf{P}$ combinator has type $(\neg\alpha \to \alpha) \to \alpha$, namely the type β is $\bot$, it is also denoted as $\mathbf{P}_{\bot}$. This finishes the definition of the languages of $\lambda\mathbf{C}$ and $\lambda\mathbf{PJ}$. Note that $\lambda\mathbf{C}$ and $\lambda\mathbf{PJ}$ only denote the languages; reduction rules will be defined later.

The following theorem is well-known.

Theorem 1. *Let α be a type. Then the following three are equivalent:*

(1) α is provable in classical logic.
(2) there exists a closed $\lambda\mathbf{C}$-term M such that $M : \alpha$ is provable in $\lambda\mathbf{C}$.
(3) there exists a closed $\lambda\mathbf{PJ}$-term N such that $N : \alpha$ is provable in $\lambda\mathbf{PJ}$.

This theorem shows that the two calculi are logically equivalent and both are classical calculi.

2.2 Reduction Rules

We define four classes of reduction rules for $\lambda\mathbf{C}$ and $\lambda\mathbf{PJ}$ other than the standard β and η reductions:

$$\begin{array}{ll} (\lambda x^{\alpha}.M^{\beta})N^{\alpha} \to M^{\beta}[x^{\alpha} := N^{\alpha}] & (\beta) \\ \lambda x^{\alpha}.M^{\alpha\to\beta}x^{\alpha} \to M^{\alpha\to\beta} & (\eta) \end{array}$$

In the η-reduction the side-condition $x \notin FV(M)$ must be satisfied.

Our design policy for reduction rules is that, we do not insist on making confluent calculi. Rather, we try to formulate reduction rules as natural as possible even if they are not confluent. It is because, in the case of classical calculus, we have no universal principle on which two terms should be equated. After defining calculi and studying relationship, we can consider how confluent subcalculus can

be obtained in different settings.

Logical Reduction Rules The first class of classical reduction rules are called logical reduction rules, which originate from the first author and others' **P**-reduction [4]. It has a quite simple form, and moreover, the type assignment of the **P**-reduction naturally induces the type $((\alpha \to \beta) \to \alpha) \to \alpha$, Peirce formula. The logical reduction rules are defined as follows:

$$M^{\neg\alpha}(\mathbf{C}^{\neg\neg\alpha\to\alpha}N^{\neg\neg\alpha}) \to N^{\neg\neg\alpha}M^{\neg\alpha} \qquad \texttt{(C)}$$
$$M^{\alpha\to\beta}(\mathbf{P}^{((\alpha\to\beta)\to\alpha)\to\alpha}N^{(\alpha\to\beta)\to\alpha}) \to M^{\alpha\to\beta}(N^{(\alpha\to\beta)\to\alpha}M^{\alpha\to\beta}) \qquad \texttt{(P)}$$
$$M^{\neg\alpha}(\mathbf{J}^{\bot\to\alpha}N^{\bot}) \to N^{\bot} \qquad \texttt{(J)}$$

These reduction rules may be rewritten in the style of proof-conversions as follows:

$$\dfrac{\vdots\ M:\neg\alpha \qquad \dfrac{\overline{\mathbf{C}:\neg\neg\alpha\to\alpha} \quad \vdots\ N:\neg\neg\alpha}{\mathbf{C}N:\alpha}}{M(\mathbf{C}N):\bot} \quad \text{reduces to} \quad \dfrac{\vdots\ N:\neg\neg\alpha \quad \vdots\ M:\neg\alpha}{NM:\bot}$$

$$\dfrac{\vdots\ M:\alpha\to\beta \qquad \dfrac{\overline{\mathbf{P}:((\alpha\to\beta)\to\alpha)\to\alpha} \quad \vdots\ N:(\alpha\to\beta)\to\alpha}{\mathbf{P}N:\alpha}}{M(\mathbf{P}N):\beta} \quad \text{reduces to}$$

$$\dfrac{\vdots\ M:\alpha\to\beta \qquad \dfrac{\vdots\ N:(\alpha\to\beta)\to\alpha \quad \vdots\ M:\alpha\to\beta}{NM:\alpha}}{M(NM):\beta}$$

$$\dfrac{\vdots\ M:\neg\alpha \qquad \dfrac{\overline{\mathbf{J}:\bot\to\alpha} \quad \vdots\ N:\bot}{\mathbf{J}N:\alpha}}{M(\mathbf{J}N):\bot} \quad \text{reduces to} \quad \vdots\ N:\bot$$

From these proof conversions, it is clear that the types are preserved by the reductions.

Although the logical reduction rules in this style are natural, unrestricted use of `(P)` or `(C)` immediately causes non-confluency. An example of collapse by `(P)`

is given in [4]. Let M be the following term.

$$(\lambda z^{\alpha}.x^{\alpha\to\alpha}((\lambda u^{\alpha}.u)z))(P(\lambda v^{\alpha\to\alpha}.vy^{\alpha}))$$

Then we have the following reductions using (β) and (P).

$$M \overset{(\mathtt{P})}{\to} \overset{(\beta)}{\to} x(xy)$$
$$M \overset{(\beta)}{\to} \overset{(\mathtt{P})}{\to} \overset{(\beta)}{\to} xy$$

To make the comparison with other calculi easier, we shift to another formulation of logical reduction rules here. An applicative context is a context where the hole is surrounded by no λ-abstraction:

$$E[\,] ::= [\,] \mid (E[\,]\,M) \mid (M\,E[\,])$$

The term $E[M]$ denotes the term $E[\,]$ where the hole $[\,]$ is replaced by the term M (provided it is well-typed). The definition of logical reductions using an applicative context is given below:

$$E^{\perp}[\mathbf{C}^{\neg\neg\alpha\to\alpha}N^{\neg\neg\alpha}] \to N^{\neg\neg\alpha}(\lambda x^{\alpha}.E^{\perp}[x]) \qquad \mathtt{(EC)}$$
$$E^{\beta}[\mathbf{P}^{(\alpha\to\beta)\to\alpha\to\alpha}N^{(\alpha\to\beta)\to\alpha}] \to E^{\beta}[N^{(\alpha\to\beta)\to\alpha}(\lambda x^{\alpha}.E^{\beta}[x])] \qquad \mathtt{(EP)}$$
$$E^{\perp}[\mathbf{J}^{\perp\to\alpha}N^{\perp}] \to N^{\perp} \qquad \mathtt{(EJ)}$$

The two sets of logical reductions are equivalent modulo the $\beta\eta$-equality, since, for instance, we have

$$M(\mathbf{C}N) \overset{(\mathtt{EC})}{\to} N(\lambda x.Mx) \overset{(\eta)}{\to} NM$$

and

$$E[\mathbf{C}N] \overset{(\beta')}{\to} (\lambda x.E[x])(\mathbf{C}N) \overset{(\mathtt{C})}{\to} N(\lambda x.E[x])$$

where (β') is the inverse of β reduction. The latter set of logical reduction rules are also not confluent, but restricting them to obtain confluent subcalculus is easier than the former. In the following, we use the latter set of reductions (EC), (EP) and (EJ) as logical reduction rules.

The logical reduction rules are close to the evaluation rules of Felleisen's $\mathcal{C}$-operator and the **call/cc** operator in Scheme[1]:

$$
\begin{aligned}
E'[\mathcal{C}N] &\to N(\lambda x.\mathcal{A}(E'[x])) \\
E'[\mathbf{call/cc}N] &\to E'[N(\lambda x.\mathcal{A}(E'[x]))] \\
E'[\mathcal{A}N] &\to N
\end{aligned}
$$

where E' is an evaluation context, which is defined depending on which evaluation strategy is adopted, such as call-by-name. The differences of our logical reduction rules and these evaluation rules are (1) we allow arbitrary applicative contexts, while in the latter case, the evaluation context is restricted, and (2) the latter reductions need the $\mathcal{A}$-operator inserted after the computation. The role of the $\mathcal{A}$-operator is "escape", so it is similar to our **J**-combinator. But the $\mathcal{A}$-operator is assigned the type $\bot\to\bot$ in Griffin's typing, so it is not useful. On the other hand, **J** is of type $\bot\to\alpha$ for an arbitrary type α.

Simplification Reduction Rules The second class of classical reduction rules are simplification reduction rules defined by:

$$
\begin{aligned}
(\mathbf{C}M)N &\to \mathbf{C}(\lambda z.M(\lambda u.z(uN))) && \texttt{(Csimp)} \\
(\mathbf{P}M)N &\to \mathbf{P}(\lambda z.M(\lambda u.z(uN))N) && \texttt{(Psimp)} \\
\mathbf{P}M &\to \mathbf{P}_\bot(\lambda z.M(\lambda u.\mathbf{J}(zu))) && \texttt{(P}_\bot\texttt{simp)} \\
(\mathbf{J}M)N &\to \mathbf{J}M && \texttt{(Jsimp)}
\end{aligned}
$$

Since the righthand sides are quite complex, we omitted the types of terms in the above definition, but they can be easily recovered from the following proof-conversions. In the following, α is $\alpha_1 \to \alpha_2$.

$$
\dfrac{\dfrac{\dfrac{\vdots}{M:\neg\neg\alpha}}{\mathbf{C}M:\alpha} \quad \vdots\, N:\alpha_1}{(\mathbf{C}M)N:\alpha_2}
\qquad \text{reduces to} \qquad
\dfrac{\dfrac{\dfrac{\vdots}{M:\neg\neg\alpha} \quad \dfrac{\dfrac{\vdots}{z:\neg\alpha_2} \quad \dfrac{\dfrac{\vdots}{u:\alpha} \quad \vdots\, N:\alpha_1}{uN:\alpha_2}}{z(uN):\bot}}{\lambda u.z(uN):\neg\alpha}}{M(\lambda u.z(uN)):\bot}}{\lambda z.M(\lambda u.z(uN)):\neg\neg\alpha_2}}{\mathbf{C}(\lambda z.M(\lambda u.z(uN))):\alpha_2}
$$

$$
\dfrac{\dfrac{\dfrac{\vdots}{M:(\alpha\to\beta)\to\alpha}}{\mathbf{P}M:\alpha} \quad \vdots\, N:\alpha_1}{(\mathbf{P}M)N:\alpha_2}
\qquad \text{reduces to}
$$

$$\dfrac{\dfrac{M : (\alpha \to \beta) \to \alpha \qquad \dfrac{\dfrac{z : \alpha_2 \to \beta \qquad \dfrac{u : \alpha \quad N : \alpha_1}{uN : \alpha_2}}{z(uN) : \beta}}{\lambda u.z(uN) : \alpha \to \beta}}{M(\lambda u.z(uN)) : \alpha} \qquad N : \alpha_1}{\dfrac{M(\lambda u.z(uN))N : \alpha_2}{\dfrac{\lambda z.M(\lambda u.z(uN))N : (\alpha_2 \to \beta) \to \alpha_2}{\mathbf{P}(\lambda z.M(\lambda u.z(uN))N) : \alpha_2}}}$$

$$\dfrac{M : (\alpha \to \beta) \to \alpha}{\mathbf{P}M : \alpha} \quad \text{reduces to} \quad \dfrac{\dfrac{\dfrac{M : (\alpha \to \beta) \to \alpha \qquad \dfrac{\dfrac{z : \neg\alpha \quad u : \alpha}{zu : \bot}}{\mathbf{J}(zu) : \beta}}{\lambda u.\mathbf{J}(zu) : \alpha \to \beta}}{M(\lambda u.\mathbf{J}(zu)) : \alpha}}{\lambda z.M(\lambda u.\mathbf{J}(zu)) : (\alpha \to \beta) \to \alpha}}{\mathbf{P}(\lambda z.M(\lambda u.\mathbf{J}(zu))) : \alpha}$$

$$\dfrac{\dfrac{M : \bot}{\mathbf{J}M : \alpha} \quad N : \alpha_1}{(\mathbf{J}M)N : \alpha_2} \quad \text{reduces to} \quad \dfrac{M : \bot}{\mathbf{J}M : \alpha_2}$$

Note that there are type restrictions on the simplification rules, and not all the terms in the form $(\mathbf{C}M)N$, $(\mathbf{P}M)N$, $\mathbf{P}M$ and $(\mathbf{J}M)N$ can be reduced by these rules. The motivation of simplification rules are to simplify the type of combinators. For example, in the `(Psimp)` rule, the type of $\mathbf{P}$ in the lefthand side is $((\alpha \to \beta) \to \alpha) \to \alpha$, and that in the righthand side is $((\alpha_2 \to \beta) \to \alpha_2) \to \alpha_2$. Since α is $\alpha_1 \to \alpha_2$, the type of $\mathbf{P}$ was simplified.

There are two simplification rules for $\mathbf{P}$; `(Psimp)` and `(P`$_\bot$`simp)`. Let $\mathbf{P}$ be of type $((\alpha \to \beta) \to \alpha) \to \alpha$. The former simplifies α (provided α is not an atomic type) and the latter simplifies β.

Base-case Reduction Rules The following reduction rules are what we call base-case reduction rules[1].

[1] We do not know if any *standard* names exist for these reduction rules.

$$\mathbf{C}^{\neg\neg\perp\to\perp}(\lambda x^{\neg\perp}.M^{\perp}) \to M^{\perp} \qquad (\mathtt{C}_0)$$
$$\mathbf{P}^{(\neg\alpha\to\alpha)\to\alpha}(\lambda x^{\neg\alpha}.M^{\alpha}) \to M^{\alpha} \qquad (\mathtt{P}_0)$$
$$\mathbf{J}^{\neg\perp}M^{\perp} \to M^{\perp} \qquad (\mathtt{J}_0)$$

In the first two rules, we assume $x \notin FV(M)$. These reductions eliminate combinators in a certain situation.

η-like Reduction Rules The following reduction rules are what we call base-case reduction rules.

$$\mathbf{C}^{\neg\neg\alpha\supset\alpha}(\lambda x^{\neg\alpha}.xM^{\alpha}) \to M^{\alpha} \qquad (\mathtt{C}\eta)$$
$$\mathbf{P}^{(\neg\alpha\to\alpha)\to\alpha}(\lambda x^{\neg\alpha}.\mathbf{J}^{\perp\supset\alpha}(xM^{\alpha})) \to M^{\alpha} \qquad (\mathtt{P}\eta)$$
$$\mathbf{C}^{\neg\neg\alpha\to\alpha}(\lambda x^{\neg\alpha}.x(\mathbf{C}^{\neg\neg\alpha\to\alpha}(\lambda y^{\neg\alpha}.xM^{\alpha}))) \to M^{\alpha} \qquad (\mathtt{C}_{\Delta})$$

In these rules, we assume $x \notin FV(M)$ and in the last rule, $y \notin FV(M)$. We call the first two rules $(\mathtt{C}\eta)$ and $(\mathtt{P}\eta)$, since it looks like the η-reduction $\lambda x.Mx \to M$. The last reduction rule is taken from λ_{Δ} calculus[9], hence we call it $(\mathtt{C}_{\Delta})$. The $(\mathtt{C}_{\Delta})$ reduction rule is apparently complex. However, it seems impossible to derive this rule by other simple rules.

2.3 Relation to Other Classical Calculi

The $\lambda\mathbf{C}$ and $\lambda\mathbf{PJ}$ are closely connected to other classical calculi by choosing appropriate classes of reduction rules. If we take the reduction rule (EC), then the calculus $\lambda\mathbf{C}$ is close to Felleisen-Griffin's system. The simplification rule (Csimp) is the same as the rule (C_{left}) in Felliesen's axiomatization, although he did not treat the operator as a combinator.

The calculus $\lambda\mathbf{PJ}$ is close to the **call/cc**-operator in the programming languages Scheme[1], and similar calculi have been studied by several researchers, for instance, Nishizaki[6] used (EP) and (EJ). The simplification rule (Psimp) comes from the first author's previous work[4].

To our knowledge, the base-case reduction rules and η-like reduction rules are not intensively studied, though it seems that they are used from time to time.

2.4 Simulation

In this subsection, we study the relationship between $\lambda\mathbf{PJ}$ and $\lambda\mathbf{C}$ for four classes of reduction rules. We shall give two translations $\bullet$ from $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$, and $\circ$ from $\lambda\mathbf{C}$ to $\lambda\mathbf{PJ}$, and study how each reduction rule is simulated by the corresponding reduction rule.

Before going to the concrete translations, we give some terminology. Suppose there are two calculi S_1 and S_2, and ϕ is a translation from S_1 terms to S_2 terms. Then, we say a reduction rule $M \to N$ in S_1 is simulated by reduction rules $R_1, \cdots, R_n$ in S_2 if there is a reduction sequence $\phi(M) \to \cdots \to \phi(N)$ where only $R_1, \cdots, R_n$, β and η reductions are used. In some cases we cannot prove the simulation relation, but we can prove $\phi(M)$ and $\phi(N)$ are equal with respect to the equivalence relation generated by $R_1, \cdots, R_n$, β and η reductions. In this case we say the reduction rule $M \to N$ in S_1 is weakly simulated by reduction rules $R_1, \cdots, R_n$ in S_2[2].

Note that the weak simulation has no meaning if the equivalence relation induced by the reduction rules is collapsed.

Translation from $\lambda\mathbf{C}$ to $\lambda\mathbf{PJ}$ A translation function $\circ$ from $\lambda\mathbf{C}$ terms to $\lambda\mathbf{PJ}$ terms is given below.

$$
\begin{aligned}
x^\circ &= x \\
(\lambda x.M)^\circ &= \lambda x.M^\circ \\
(MN)^\circ &= M^\circ N^\circ \\
\mathbf{C}^\circ &= \lambda x.\mathbf{P}(\lambda y.\mathbf{J}(xy))
\end{aligned}
$$

Then we have the following theorem.

Theorem 2. *For the translation $\circ$, we have the following.*

(1) $\circ$ is sound w.r.t. the type system, namely, if $M : \alpha$ in $\lambda\mathbf{C}$, then $M^\bullet : \alpha$ in $\lambda\mathbf{PJ}$.

(2) `(EC)` *can be simulated by* `(EP)` *and* `(EJ)`.

[2] Plotkin used the word "translation" in this case. We diverge from his terminology in order to use the word "translation" for the mapping function on terms.

(3) `(Csimp)` *can be weakly simulated by* `(Psimp)` *and* `(Jsimp)`.

(4) $(\mathtt{C}_0)$ *can be simulated by* $(\mathtt{P}_0)$ *and* $(\mathtt{J}_0)$.

(5) $(\mathtt{C}\eta)$ *can be simulated by* $(\mathtt{P}\eta)$.

(6) $(\mathtt{C}_\Delta)$ *can be simulated by* `(J)` *and* $(\mathtt{P}\eta)$.

Proofs are easy exercises; we give a proof of (3) only. Each side of the `(Csimp)` reduction rule can be reduced as follows:

$$\begin{array}{rcl}((\mathbf{C}M)N)^\circ & = & ((\lambda x.\mathbf{P}(\lambda y.\mathbf{J}(xy)))M^\circ)N^\circ \\ & \stackrel{(\beta)}{\to} & \mathbf{P}(\lambda y.\mathbf{J}(M^\circ y))N^\circ \\ & \stackrel{\texttt{(Psimp)}}{\to} & \mathbf{P}(\lambda z.(\lambda y.\mathbf{J}(M^\circ y))(\lambda u.z(uN^\circ))N^\circ) \\ & \stackrel{(\beta)}{\to} & \mathbf{P}(\lambda z.\mathbf{J}(M^\circ(\lambda u.z(uN^\circ)))N^\circ) \\ & \stackrel{\texttt{(Jsimp)}}{\to} & \mathbf{P}(\lambda z.\mathbf{J}(M^\circ(\lambda u.z(uN^\circ))))\end{array}$$

$$\begin{array}{rcl}(\mathbf{C}(\lambda z.M(\lambda u.z(uN))))^\circ & = & (\lambda x.\mathbf{P}(\lambda y.\mathbf{J}(xy)))(\lambda z.M^\circ(\lambda u.z(uN^\circ))) \\ & \stackrel{(\beta)}{\to} & \mathbf{P}(\lambda y.\mathbf{J}((\lambda z.M^\circ(\lambda u.z(uN^\circ)))y)) \\ & \stackrel{\texttt{(Jsimp)}}{\to} & \mathbf{P}(\lambda y.\mathbf{J}(M^\circ(\lambda u.y(uN^\circ))))\end{array}$$

Hence, the `(Csimp)` reduction rule is weakly simulated. Since the use of β-reduction is reversed, it is not simulation in the strong sense. Nevertheless, this weak simulation is useful, since we can construct confluent theories which include `(Csimp)`.

As in the theorem, all reduction rules except $(\mathtt{C}_\Delta)$ can be simulated by the corresponding rules. The result for $(\mathtt{C}_\Delta)$ is not desirable, since it needs `(J)`.

First Translation from λPJ to λC The translation from $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$ is problematic compared to the inverse direction, since the type restriction in `(EC)` is too restrictive to simulate `(EP)`. From the computational viewpoint, the reason is clear; Felleisen's operator (under Griffin's typing) can be reduced only when the surrounding evaluation context has the type $\bot$, while the **call/cc** operator can be reduced in general. We are inclined to think that $\mathbf{C}$ corresponds to $\mathbf{P}_\bot$ (with $\mathbf{J}$), and does not correspond to full $\mathbf{P}$.

Here, we shall give two translations from $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$; one is from a subcalculus of $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$ which works well, and the other is from the full $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$ which is not completely successful.

The subcalculus of $\lambda\mathbf{PJ}$ is obtained by restricting $\mathbf{P}$ to $\mathbf{P}_\perp$, namely, we restrict the β-part of $\mathbf{P}$ to $\perp$. The following translation is from $\lambda\mathbf{P}_\perp\mathbf{J}$ to $\lambda\mathbf{C}$.

$$\begin{aligned} x^\bullet &= x \\ (\lambda x.M)^\bullet &= \lambda x.M^\bullet \\ (MN)^\bullet &= M^\bullet N^\bullet \\ \mathbf{P}_\perp{}^\bullet &= \lambda x.\mathbf{C}(\lambda y.y(xy)) \\ \mathbf{J}^\bullet &= \lambda x.\mathbf{C}(\lambda y.x) \end{aligned}$$

Using this translation, we have the following results.

Theorem 3. *For the translation* $\bullet$*, we have the following.*

(1) The translation $\bullet$ *is sound w.r.t. the type system.*

(2) `(EP)` *and* `(EJ)` *can be simulated by* `(EC)`*.*

(3) `(Psimp)` *and* `(Jsimp)` *can be weakly simulated by* `(Csimp)`*.*

(4) $(\mathtt{P}_0)$ *can be simulated by* $(\mathtt{C}\eta)$*.*

(5) $(\mathtt{J}_0)$ *can be simulated by* $(\mathtt{C}_0)$*.*

(6) $(\mathtt{P}\eta)$ *can be simulated by* $(\mathtt{C}_\Delta)$*.*

The simulation of $(\mathtt{P}_0)$ and $(\mathtt{P}\eta)$ is not desirable, but other simulations work well.

Second Translation from $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$ To give a translation from full $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$, we need to change the translation for $\mathbf{P}$ as follows. For brevity, we use the same name $\bullet$ for the second translation.

$$\mathbf{P}^\bullet = \lambda x.\mathbf{C}(\lambda y.y(x(\lambda z.\mathbf{C}(\lambda u.yz))))$$

Other definitions are the same. Then we have the following results for this translation.

Theorem 4. *For the new translation* $\bullet$*, we have the following.*

(1) $\bullet$ *is sound w.r.t. the type system.*

(2) (EJ) *can be simulated by* (EC)*.*

(3) (Psimp) *and* (Jsimp) *can be weakly simulated by* (Csimp)*.*

(4) ($\mathtt{P}_\perp$simp) *can be weakly simulated by* (Csimp) *and* ($\mathtt{C}_0$)*.*

(5) ($\mathtt{P}_0$) *can be simulated by* ($\mathtt{C}\eta$)*.*

(6) ($\mathtt{J}_0$) *can be simulated by* ($\mathtt{C}_0$)*.*

(7) ($\mathtt{P}\eta$) *can be simulated by* ($\mathtt{C}_\Delta$) *and* ($\mathtt{C}_0$)*.*

The problem is, as we mentioned, (EP) cannot be simulated by (EC). Consider the following reduction.

$$\begin{aligned}(E[\mathbf{P}N])^\bullet &= E^\bullet[(\lambda x.\mathbf{C}(\lambda y.y(x(\lambda z.\mathbf{C}(\lambda u.yz)))))N^\bullet] \\ &\stackrel{(\beta)}{\to} E^\bullet[\mathbf{C}(\lambda y.y(N^\bullet(\lambda z.\mathbf{C}(\lambda u.yz))))]\end{aligned}$$

We cannot proceed further if the type of $(E[\,])^\bullet$ is not $\perp$.

In either translation, the correspondence is more complex than the translation from $\lambda\mathbf{C}$ to $\lambda\mathbf{PJ}$. For instance, ($\mathtt{C}_0$) is simulated by ($\mathtt{P}_0$) and ($\mathtt{J}_0$) while ($\mathtt{P}_0$) is simulated not by ($\mathtt{C}_0$) but ($\mathtt{C}\eta$). Also the simulation of ($\mathtt{P}\eta$) needs the complex rule ($\mathtt{C}_\Delta$).

Summary of Simulation We can summarize the results of simulation as the following table. In this table, $\Longleftrightarrow$ means that each set of reduction rules can be simulated by the set of reduction rules in the other side and vice versa. The symbol $\leftrightarrow$ means that the simulation is weak simulation. We adopt the first translation as the translation from $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$.

$\lambda\mathbf{C}$		$\lambda\mathbf{P}_\perp\mathbf{J}$
(EC)	$\Longleftrightarrow$	(EP) + (EJ)
(Csimp)	$\leftrightarrow$	(Psimp) + (Jsimp)
(EC) + (Csimp)	$\leftrightarrow$	(EP) + (EJ) + (Psimp) + (Jsimp)

We have no good equivalence result if the reduction set contains the base-case reduction rules or the η-like reduction rules. It follows that, if we use the second

translation as the translation from $\lambda\mathbf{PJ}$ to $\lambda\mathbf{C}$, the only meaningful equivalence is `(Csimp)` $\leftrightarrow$ `(Psimp)` + `(Csimp)`.

Relation of Translations After establishing the results on simulations, a natural question arises: whether the translations $\circ$ and $\bullet$ are inverse mappings to each other, namely, $M^{\bullet\circ} = M$ for any $\lambda\mathbf{C}$ term M, and $N^{\circ\bullet} = N$ for any $\lambda\mathbf{PJ}$ term N ? Unfortunately, it does not hold. The first half holds if we allow the `(C)` reduction rule, however the second half does not hold for adding any reduction rule considered in this paper.

3 λ-calculus for Classical Logic

In this section, we introduce two calculi $\underline{\lambda}$ and $\overline{\lambda}\mathbf{J}$. The two calculi have the λ-calculus style rather than combinators. The calculus $\underline{\lambda}$ corresponds to $\lambda\mathbf{C}$ and the calculus $\overline{\lambda}\mathbf{J}$ corresponds to $\lambda\mathbf{PJ}$.

3.1 Syntax

First we define preterms.

$$(for\ \underline{\lambda}) \quad M ::= x \mid \lambda x.M \mid (M\,M) \mid \lambda\underline{x}.M$$
$$(for\ \overline{\lambda}\mathbf{J}) \quad M ::= x \mid \lambda x.M \mid (M\,M) \mid \lambda\overline{x}.M \mid \mathbf{J}$$

As is seen, we eliminated combinators $\mathbf{C}$ and $\mathbf{P}$, and added new binding mechanisms $\lambda\underline{x}.M$ and $\lambda\overline{x}$. The $\mathbf{J}$ combinator remains in $\overline{\lambda}\mathbf{J}$. For notational convenience, a sequence of binders will be punctuated, for example, $\lambda x\overline{yz}.xyz$ is $\lambda x.\lambda\overline{y}.\lambda\overline{z}.xyz$, and $\lambda x\underline{yz}.xyz$ is $\lambda x.\lambda\underline{y}.\lambda\underline{z}.xyz$.

The typing rules for new terms are as follows:

Typing Rules for Classical λ-terms

$$\frac{\begin{array}{c}[x:\neg\alpha]\\ \vdots\\ M:\bot\end{array}}{\lambda\underline{x}.M:\alpha} \qquad \frac{\begin{array}{c}[x:\alpha\to\beta]\\ \vdots\\ M:\alpha\end{array}}{\lambda\overline{x}.M:\alpha}$$

In these typing rules, the assumptions $x:\neg\alpha$ and $x:\alpha\to\beta$ are discharged at the application of these two rules.

3.2 Reduction Rules

The classical reduction rules are classified into four classes. Since reduction rules for the **J** combinator are the same as before, we give one for $\lambda\underline{x}.M$ and $\lambda\overline{x}.M$ only. Note that we give the same name to the reduction rule as the corresponding rule in combinatory-logic-style calculi.

Logical Reduction Rules

$$E[\lambda\underline{x}.M] \to M[x := \lambda y.E[y]] \qquad \texttt{(EC)}$$
$$E[\lambda\overline{x}.M] \to E[M[x := \lambda y.E[y]]] \qquad \texttt{(EP)}$$

where $E[\,]$ is an applicative context.

Simplification Reduction Rules

$$(\lambda\underline{x}.M)N \to \lambda\underline{z}.M[x := \lambda u.z(uN)] \qquad \texttt{(Csimp)}$$
$$(\lambda\overline{x}.M)N \to \lambda\overline{z}.M[x := \lambda u.z(uN)]N \qquad \texttt{(Psimp)}$$
$$\lambda\overline{x}.M \to \lambda\overline{z}.M[x := \lambda u.\mathbf{J}(zu)] \qquad \texttt{(P}_\perp\texttt{simp)}$$

Base-case Reduction Rules

$$\lambda\underline{x}.M \to M \qquad \texttt{(C}_0\texttt{)}$$
$$\lambda\overline{x}.M \to M \qquad \texttt{(P}_0\texttt{)}$$

where $x \notin FV(M)$.

η-like Reduction Rules

$$\lambda\underline{x}.xM \to M \qquad \texttt{(C}\eta\texttt{)}$$
$$\lambda\overline{x}.\mathbf{J}(xM) \to M \qquad \texttt{(P}\eta\texttt{)}$$
$$\lambda\underline{x}.x(\lambda\underline{y}.xM) \to M \qquad \texttt{(C}_\Delta\texttt{)}$$

where $x \notin FV(M)$.

3.3 Relation to Other Classical Calculi

Classical calculi in the λ-calculus-style proposed so far are not many; the only exception is Rehof and Sϕrensen's λ_Δ-calculus[9], which roughly corresponds to $\underline{\lambda}$ with the reduction rules (Csimp), ($\texttt{C}_0$) and ($\texttt{C}_\Delta$).

3.4 Simulation

As in the case of the combinatory-logic-style presentation, the two calculi $\overline{\lambda}\mathbf{J}$ and $\underline{\lambda}$ can simulate each other. The translations are given as follows:

$$
\begin{aligned}
x^\circ &= x \\
(\lambda x.M)^\circ &= \lambda x.M^\circ \\
(MN)^\circ &= M^\circ N^\circ \\
(\lambda \underline{x}.M)^\circ &= \lambda \overline{x}.\mathbf{J}(M)^\circ
\end{aligned}
$$

$$
\begin{aligned}
x^\bullet &= x \\
(\lambda x.M)^\bullet &= \lambda x.M^\bullet \\
(MN)^\bullet &= M^\bullet N^\bullet \\
(\lambda \overline{z}.M)^\bullet &= \lambda \underline{z}.zM^\bullet \\
\mathbf{J}^\bullet &= \lambda x \underline{y}.x
\end{aligned}
$$

By these translations, we have the same result as Theorems 2, 3, and 4.

3.5 Relationship between Combinatory-logic style and λ-calculus style

The combinatory-logics $\lambda\mathbf{C}$ and $\lambda\mathbf{PJ}$ and the λ-calculi $\underline{\lambda}$ and $\overline{\lambda}\mathbf{J}$ corresponds to each other. We define a translation $\flat$ from combinatory-logic-styles to the λ-calculus-styles and a translation $\sharp$ from the λ-calculus-styles to combinatory-logic-styles.

$$
\begin{aligned}
x^\flat &= x \\
(\lambda x.M)^\flat &= \lambda x.M^\flat \\
(MN)^\flat &= M^\flat N^\flat \\
(\lambda \underline{x}.M)^\flat &= \mathbf{C}(\lambda x.M^\flat) \\
(\lambda \overline{x}.M)^\flat &= \mathbf{P}(\lambda x.M^\flat) \\
\mathbf{J}^\flat &= \mathbf{J}
\end{aligned}
$$

$$
x^\sharp = x
$$

$$(\lambda x.M)^\sharp = \lambda x.M^\sharp$$
$$(MN)^\sharp = M^\sharp N^\sharp$$
$$\mathbf{C}^\sharp = \lambda x\underline{y}.xy$$
$$\mathbf{P}^\sharp = \lambda x\overline{y}.xy$$
$$\mathbf{J}^\sharp = \mathbf{J}$$

Theorem 5. *For the translation $\flat$ and $\sharp$, we have the following:*

(1) $\flat$ and $\sharp$ are sound w.r.t. the type system.

(2) every reduction rule is simulated (in a strong sense) by the reduction rule which has the same name.

(3) The two translations are inverses of the other; namely, $M^{\sharp\flat} = M$ and $N^{\flat\sharp} = N$ for any term M in $\underline{\lambda}$ and any term N in $\overline{\lambda}\mathbf{J}$.

In this sense, we may regard the combinatory logics $\lambda\mathbf{C}$ and $\lambda\mathbf{PJ}$ are exactly the same systems as the λ-calculi $\underline{\lambda}$ and $\overline{\lambda}\mathbf{J}$, respectively.

4 Properties of the Calculi

In this section we briefly give known properties of our calculi.

4.1 Confluency

As mentioned earlier, unrestricted use of the logical reduction causes non-confluency, so we must, for instance, restrict evaluation strategy. This topic is thoroughly studied in the literature, and we do not get into detail. The simplification reduction (Csimp) is well behaved in the sense it produces a confluent calculus with β and η reductions. Moreover, we can safely add the reductions ($\mathtt{C}_0$) and ($\mathtt{C}\eta$) to this calculus without losing the confluency. It seems that we can add ($\mathtt{C}_\Delta$) to this calculus.

By the simulation between $\lambda\mathbf{PJ}$ and $\lambda\mathbf{C}$, these results can be lifted to $\lambda\mathbf{PJ}$, for instance, the system with (Psimp) is confluent, and so on. Adding the ($\mathtt{P}_\perp$simp) rule does not cause a problem, since the order of (Psimp) and ($\mathtt{P}_\perp$simp) can be exchanged.

4.2 Normalization

The first author with others proved that weak normalization holds for (Psimp). Careful inspection of this result shows that, a subcalculus of λ**PJ** which has (Psimp) rule is always weakly normalizing. We do not know if a calculus without (Psimp) (for instance, a calculus with (EP) only) is weakly normalizing or not.

Very little about strong normalization is known until now. We know that the system with (P) is not strongly normalizing [4].

4.3 Normal Forms

Normal forms in $\overline{\lambda}$**J** have an elegant form if we slightly modify the (Psimp) reduction. It was given as the following form:

$$(\lambda \overline{x}.M)N \to \lambda \overline{z}.M[x := \lambda u.z(uN)]N \qquad \texttt{(Psimp)}$$

We can slightly modify it to a more general reduction as follows.

$$\lambda \overline{x}.M \to \lambda y.\lambda \overline{z}.M[x := \lambda u.z(uy)]y \qquad new\texttt{(Psimp)}$$

Then we have the following theorem.

Theorem 6. *Let M be a normal term in $\overline{\lambda}$**J**. Then it is in the form of $\lambda x_1, \cdots, x_n, \overline{y}_1, \cdots \overline{y}_m.zM_1 \cdots M_k$.*

The theorem follows from the fact that, if there is a subterm in the form $\lambda \overline{y}x.M$, then we can always apply (Psimp) rule, so it was not normal.

5 Computation in λPJ

In this section we examine the expressive power of our calculi. Since computational aspects of λ**C** are relatively well known, our target here is λ**PJ** or $\overline{\lambda}$**J** with reduction rules (Pη) and (Jsimp).

5.1 Catch/Throw Mechanism

A common usage of continuation is to simulate the catch/throw mechanism in Common Lisp. Define *Catch* and *Throw* as follows:

$$Catch(x, M^{\alpha}) = \lambda \overline{x}^{\neg\beta}.M^{\alpha}$$
$$Throw(x, M^{\alpha}) = \mathbf{J}(x^{\neg\alpha}M^{\alpha})$$

For example, $\texttt{Catch}(x, \texttt{Throw}(x, 0)+1)$ is defined as $\lambda\overline{x}.\mathbf{J}(x0)+1$. Here we assume integers and $+$ are added to the language and we have the reduction $(\mathbf{J}M)+N \to \mathbf{J}M$. Then we have $\texttt{Catch}(x, \texttt{Throw}(x, 0) + 1) \to \lambda\overline{x}.\mathbf{J}(x0) \to 0$ as expected.

5.2 Defining Conjunction and Disjunction

As in other calculi for classical logic, we can define conjunction ($\wedge$) and disjunction ($\vee$) using implication and falsehood.

Let $\alpha \wedge \beta$ be $\neg(\alpha \to \neg\beta)$. Then we can define

$$pair(M^{\alpha}, N^{\beta}) \stackrel{\triangle}{=} \lambda u^{\alpha\to\neg\beta}.\ uMN$$
$$\pi_0(L^{\alpha\wedge\beta}) \stackrel{\triangle}{=} \lambda\overline{x}^{\neg\alpha}.\ \mathbf{J}(L(\lambda y^{\alpha}z^{\beta}.xy)$$
$$\pi_1(L^{\alpha\wedge\beta}) \stackrel{\triangle}{=} \lambda\overline{x}^{\neg\beta}.\ \mathbf{J}(L(\lambda y^{\alpha}z^{\beta}.xz)$$

As expected, $\pi_0(pair(M, N))$ reduces to M by the following reduction sequence:

$$\pi_0(pair(M,N)) \to \lambda\overline{x}.\mathbf{J}((\lambda f.fMN)(\lambda yz.xy)) \to \lambda\overline{x}.\mathbf{J}((\lambda yz.xy)MN) \to \lambda\overline{x}.\mathbf{J}(xM) \stackrel{(\mathtt{P}\eta)}{\to} M$$

Similarly, we obtain $\pi_1(pair(M, N)) \to N$.

Disjunction $\alpha \vee \beta$ can be defined as $\neg\alpha \to \neg\neg\beta$.

$$inj_0(M^{\alpha}) \stackrel{\triangle}{=} \lambda x^{\neg\alpha}y^{\neg\beta}.xM$$
$$inj_1(N^{\beta}) \stackrel{\triangle}{=} \lambda x^{\neg\alpha}y^{\neg\beta}.yN$$
$$case(L^{\alpha\vee\beta}; x^{\alpha}.M^{\gamma}; y^{\beta}.N^{\gamma}) \stackrel{\triangle}{=} \lambda\overline{z}^{\neg\gamma}.\ \mathbf{J}(L(\lambda x^{\alpha}.zM)(\lambda y^{\beta}.zN))$$

As expected, $case(inj_0(L); x^{\alpha}.M^{\gamma}; y^{\beta}.N^{\gamma})$ reduces to $M[x := L]$ by the following reduction sequence:

$$case(inj_0(L); x.M; y.N) \ \to\ \lambda\overline{z}.\mathbf{J}((\lambda xy.xL)(\lambda x.zM)(\lambda y.zN))$$

$$\begin{array}{rl} & \to \quad \lambda\overline{z}.\mathbf{J}((\lambda x.zM)L) \\ & \to \quad \lambda\overline{z}.\mathbf{J}(z(M[x := L])) \\ \overset{(\mathtt{P}\eta)}{} & \to \quad M[x := L] \end{array}$$

Similarly we obtain $case(inj_1(L); x.M; y.N) \to N[y := L]$.

6 Concluding Remarks

Parigot's $\lambda\mu$-calculus is yet another interesting calculus for classical logic[8]. Basically, $\lambda\mu$-calculus corresponds to multiple-consequence logic in a natural deduction style, so it differs from Griffin's type system in the fundamental principle. However, de Groote has shown that the structural reduction rule in $\lambda\mu$ can be simulated by a subset of reduction rules in Griffin-Felleisen's system. If we interpret de Groote's result in our setting, the structural reduction rule in $\lambda\mu$ can be simulated by `(Csimp)` in $\lambda\mathbf{C}$, or `(Psimp)` + `(Jsimp)` in $\lambda\mathbf{PJ}$.

However, we think another, more direct correspondence exists between $\lambda\mu$ and $\lambda\mathbf{PJ}$ as shown in the following diagrams.

$$\begin{array}{ccc} \mathrm{LK} & \to & \mathrm{LJ} + \mathrm{Peirce} \\ \downarrow & & \downarrow \\ \lambda\mu & \to & \lambda\mathbf{P} \end{array}$$

In this diagram, LK means the implicational fragment (without the falsehood) and LJ+Peirce means the implicational fragment of LJ (without the falsehood) with the following rule (Peirce-rule) added.

$$\frac{\Gamma, A \to B \vdash A}{\Gamma \vdash A}$$

It seems that we can translate a proof in LK to a proof in LJ+Peirce; since the only difference is the existence of right weakening/contraction rules, all we have to do is to replace right contraction rules by Peirce-rules and other intuitionistic rules[3]. If this translation succeeds, then it may be straightforward to move the translation to the translation from $\lambda\mu$ to $\lambda\mathbf{P}$, since there is a natural translation from LK to $\lambda\mu$, and $LJ + Peirce$ to $\lambda\mathbf{P}$.

References

1. Clinger, W., and J. Rees (eds.): "Revised^4 Report on the Algorithmic Language Scheme", 1991.

2. Griffin, T.: "A Formulae-as-Types Notion of Control", Conference Record of 17th ACM Symposium on Principles of Programming Languages, pp. 47–58, 1990.
3. Hirokawa, S.: "Right Weakening and Right Contraction in LK", Proc. CATS'96, Australian Computer Science Communications, Vol. 18, No. 3, pp. 168–174, 1996.
4. Hirokawa, S., S. Komori and I. Takeuti: "A Reduction Rule for Peirce Formula", Studia Logica 56, 3, pp. 419–426, 1996.
5. Howard, W. A.: "The Formulae-as-Types Notion of Construction", reprinted in *The Curry-Howard Isomorphism* (de Groote, Ph. ed.), Academia, 1995.
6. Nishizaki, S.: "Programs with Continuations and Linear Logic", TACS'91 Proceedings, *Lecture Notes in Computer Science* **526** (T. Ito and A. R. Meyer eds.), pp. 513-531, 1991.
7. Ong, C.-H. L. and C. A. Stewart: "A Curry-Howard Foundation for Functional Computation with Control", *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.
8. Parigot, M.: "$\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction", Proc. International Conference on Logic Programming and Automated Reasoning (A. Voronkov ed.), *Lecture Notes in Artificial Intelligence* **624**, pp. 190–201, Springer, 1992.
9. Rehof, N. J. and M. H. Sϕrensen: "The λ_Δ-calculus", Theoretical Aspects of Computer Software, *Lecture Notes in Computer Science* **789** (M. Hagiya and J. C. Mitchell eds.), pp. 516-542, 1994.
10. Sato, M.: "Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules", *Theoretical Computer Science*, **175**.1 pp. 75-92, 1997.