The Transformation Calculus

Jacques Garrigue

Research Institute for Mathematical Sciences
Kyoto University, Kitashirakawa-Oiwakecho
Sakyo-ku, Kyoto 606-01 JAPAN
Tel ++81-75-753-7211
Fax ++81-75-753-7272
E-mail garrique@kurims.kyoto-u.ac.jp

Abstract. The lambda-calculus, by its ability to express any computable function, is theoretically able to represent any algorithm. However, notwithstanding their equivalence in expressiveness, it is not so easy to find a natural translation for algorithms described in an imperative way.

The transformation calculus, which only extends the notion of currying in lambda-calculus, appears to be able to correct this flaw, letting one implicitly manipulate a state through computations.

This calculus remains very close to lambda-calculus, and keeps most of its properties. We proved confluence of the untyped calculus, and strong-normalization in presence of a typing system.

1 Introduction

Currying is as old as lambda calculus. For the simple reason that, in raw lambda calculus—without pairing or similar built-in constructs—, this is the only way to represent multi-argument functions. This just means that we will write

$$\lambda x.\lambda y.M[x, y]$$

in place of

$$(x, y) \mapsto M[x, y].$$

At this stage appears a first asymmetry: while in the pair $(x, y)$ the two variables play symmetrical roles, in $\lambda x.\lambda y.M$ they don’t. An implicit order was introduced. Materially this means that we can partially apply our function directly on $x$ but not on $y$.

We now look at types. There, currying can be seen as isomorphism of types [3]:

$$(A \times B) \rightarrow C \simeq A \rightarrow B \rightarrow C.$$ 

Here comes another asymmetry: why don’t we get any similar isomorphism for $A \rightarrow (B \times C)$.

The calculus we will present here generalizes currying to these two kinds of symmetries: between arguments, and between input and output. For the first one, we are just taking over the mechanism of label-selective currying developed previously [1, 11].

For the second one we develop a new notion of composition, which, contrary to the usual one, is compatible with currying.
The resulting system, *transformation calculus*, is a conservative extension of lambda calculus. Why such a name? Because this essentially syntactic extension—semantics remain very similar—provides us with a new way of representing state transformations, *i.e.* state being represented by labeled input parameters, that may get returned by our term. Handling state as a supplementary parameter that gets returned with the result is not new. But by extending currying we get more flexibility, in two ways. First, since a part of the state is no more than a labeled parameter, we can dynamically extend it by simply adding a new parameter at some point in our term. Second, selective currying lets a transformation ignore parts of the state it doesn't need. They will just be left unmodified.

To demonstrate our point, we introduce *scope-free variables*, which are trivially encoded in the transformation calculus, and can be used in place of usual scoped mutable variables, in the Algol tradition. Since they have no syntactic scope, scope-free variables respect dynamic binding rather than static binding; but they are more flexible than Algol variables, while simulating blocks and stack discipline.

The rest of this paper is composed as follows. In Section 2 we introduce progressively the different features which form the transformation calculus. Section 3 is devoted to the formal definition of the transformation calculus. Sections 4 and 5 respectively define and give the fundamental properties of scope-free variables and a simply typed transformation calculus. Related works are presented in Section 6. Finally, Section 7 concludes. For lack of space, no proofs are given in this paper.

## 2 Composition and streams

We first introduce informally and progressively the features of our calculus. We start from the classical pure lambda-calculus, that is

\[ M ::= x \mid \lambda x. M \mid (M).M \]

with \(\beta\)-reduction

\[ (N).\lambda x. M \rightarrow_{\beta} [N/x] M \]

and where terms are considered modulo \(\alpha\)-conversion (renaming of bound variables).

### 2.1 Implicit currying

Currying is the fundamental transformation by which multi-argument functions are encoded in the lambda-calculus. It can appear in abstractions as well as applications. For instance \(f(a, b)\) will be encoded as \((b).(a).f\), and \(\lambda x, y. M\) becomes \(\lambda x.\lambda y. M\).

This operation does not modify the nature of calculations, since clearly \((a, b).\lambda x, y. M\) and \((b).(a).\lambda x.\lambda y. M\) reduce to the same \([a/x, b/y] M\) (provided \(x\) and \(y\) are distinct variables).

---

1 They can be found, together with denotational semantics for the simply typed calculus, in the report version [9].

2 Application, denoted by a dot, is written postfix, and is left associative.