Some Remarks on Control Categories

Peter Selinger
Department of Mathematics and Statistics,
University of Ottawa,
Ottawa, Ontario K1N 6N5, Canada

June 8, 2003

Abstract

This paper is a collection of remarks on control categories, including answers to some frequently asked questions. The paper is not self-contained and must be read in conjunction with [3]. We clarify the definition of response categories, and show that most of the conditions can be dropped. In particular, the requirement of having finite sums can be dropped, leading to an interesting new CPS translation of the $\lambda\mu$-calculus. We discuss the choice of left-to-right vs. right-to-left evaluation in the call-by-value lambda calculus, an issue which is sometimes misunderstood because it is a purely syntactical issue which is not reflected semantically. We clarify the relationships between various alternative formulations of disjunction types and conjunction types, which coincide in call-by-value but differ in call-by-name. We prove that copyable and discardable maps are not always central, and we characterize those control categories of the form $R^{\mathbf{Set}}$ for which copyable and discardable implies central. We prove that any control category with initial object is a preorder.

1 Response categories and sums

The central construction of [3] is the construction of a control category from a category with finite products and powers of the form R^A, for a single, distinguished object R. This construction is a categorical version of continuation passing style (CPS) semantics of the lambda calculus. In [3], a category $\mathbf{C}$ with a distinguished object R is called a ***response category*** if it satisfies the following conditions:

1. $\mathbf{C}$ has finite products.
2. Exponentials of the form R^A exist, for any object A. This means that there is a natural isomorphism of hom-sets $(B, R^A) \cong_A (B \times A, R)$.
3. $\mathbf{C}$ has finite coproducts (sums).
4. Sums and products distribute, i.e., the canonical morphism $d : (A \times C) + (B \times C) \to (A + B) \times C$ is an isomorphism.
5. $\mathbf{C}$ satisfies the following ***mono requirement***: the canonical morphism $\partial_A : A \to R^{R^A}$ is monic for all A.
6. (Without loss of generality). Exponentials are chosen to be distinct, i.e, $A \neq B$ implies $R^A \neq R^B$.

This definition has occasionally been criticized for being too restrictive. Indeed, of the six conditions, only the first two are strictly necessary. The sixth condition was already stated in the paper to be without loss of generality. Condition 5 was never used except in the uniqueness part of the Second Structure Theorem. Condition 4 can be dropped, because its relevant consequence, $R^{(A\times C)+(B\times C)} \cong R^{(A+B)\times C}$, already follows from the other conditions. Finally, condition 3 can be dropped at the cost of slightly complicating the construction of a category of continuations (and thus also the CPS semantics).

Recall that from a response category $\mathbf{C}$, one defines a control category $R^{\mathbf{C}}$, called the ***category of continuations*** of $\mathbf{C}$, as follows: $R^{\mathbf{C}}$ is the full subcategory of $\mathbf{C}$ consisting of the objects of the form R^A. The control category structure is defined on objects as follows (well-definedness uses condition 6 above):

$$\begin{array}{lcl} 1 & := & R^0, \\ R^A \times R^B & := & R^{A+B}, \\ (R^B)^{R^A} & := & R^{B\times R^A}, \\ \bot & := & R^1, \\ R^A \mathbin{⅋} R^B & := & R^{A\times B}. \end{array} \tag{1}$$

The morphism part of the control category structure is defined in the obvious way, and the requisite equations are easily verified. One has the following pair of theorems, proved in [3]:

Theorem 1.1 (Construction Theorem) *Every response category gives rise to a control category via the above construction.*

Theorem 1.2 (Structure Theorem) *Every control category arises in this way.*

These theorems correspond roughly to soundness and completeness of the control category axioms with respect to the CPS interpretation.

The inclusion of conditions 1–5 in the original article was motivated by the desire to give the strongest possible set of conditions which let the structure theorem go through, rather than giving the weakest set of conditions required to prove the construction theorem. In other words, all the response categories which arise in the proof of the structure theorem happen to satisfy conditions 1–5 (and, without loss of generality, 6). At the time of the publication of [3], I felt that it was important to narrow down the class of response categories as much as possible, rather than giving the most general class. In a certain sense, the uniqueness part of the second structure theorem [3, Thm 4.6] shows that the class of response categories defined by conditions 1–5 is indeed as narrow as possible, because any category in this class will arise from the construction in the proof of the Structure Theorem.

We will now show that the Construction Theorem can be adopted to the case with only conditions 1 and 2.

1.1 Dropping condition 6

Condition 6 can be dropped by making a slight change to the definition of the category $R^{\mathbf{C}}$: rather than taking the objects of $R^{\mathbf{C}}$ to be the objects of $\mathbf{C}$ of the form R^A, one takes the objects of $R^{\mathbf{C}}$ to be the objects of $\mathbf{C}$, with a morphism $A \to B$ in $R^{\mathbf{C}}$ defined as a morphism $R^A \to R^B$ in $\mathbf{C}$. This definition is formally cleaner, but notationally messy, as one is now in the awkward position of defining the control category operations on $R^{\mathbf{C}}$ as:

$$\begin{array}{lcl} 1 & := & 0, \\ A \times B & := & A + B, \\ B^A & := & B \times R^A, \\ \bot & := & 1, \\ A \parr B & := & A \times B, \end{array}$$

where the operations on the left-hand side are control category operations in $R^{\mathbf{C}}$, whereas the operations on the right-hand side are response category operations on $\mathbf{C}$. The situation can be slightly alleviated by one of two tricks: either, we write the left-hand side operations differently from the right-hand side operations (e.g. by using bold-face symbols $\boldsymbol{\times}$ vs. ordinary symbols $\times$). Or else, we use a different notation for an object A when considered as an object of $R^{\mathbf{C}}$ than when considered as an object of $\mathbf{C}$. We could, for instance, write the object A as $\bar{A}$ or $[A]$ when considered as an object of $R^{\mathbf{C}}$.

We adopt the second solution, but we use a more mnemonic notation and write $\mathbf{R}^A$ for the object A, when considered as an object of $R^{\mathbf{C}}$. It is understood that this is a notation, rather than an operation, i.e., $\mathbf{R}^A$ is a formal exponential, rather than an actual exponential.

1.2 Dropping condition 5

Nothing much needs to be said about condition 5, since we already remarked that it was never used except to show uniqueness of $\mathbf{C}$.

1.3 Dropping condition 4

Condition 4 is interesting, because it is almost redundant, but not quite. In any cartesian-closed category with sums, distibutivity holds simply because the functor $(-) \times C$ is a left adjoint and thus preserves colimits. In symbols, this is calculated as follows:

$$\begin{array}{ll} & ((A \times C) + (B \times C), D) \\ \cong_D & (A \times C, D) \times (B \times C, D) \\ \cong_D & (A, D^C) \times (B, D^C) \\ \cong_D & (A + B, D^C) \\ \cong_D & ((A + B) \times C, D) \end{array} \tag{2}$$

Since all the isomorphisms shown are natural in D, it follows that $(A + B) \times C \cong (A \times C) + (B \times C)$.

Interestingly, when one only has a single exponentiable object R, this argument does not work, and in fact distributivity may fail. One can still write the above chain of isomorphism, but with the fixed object R instead of arbitrary D. A counterexample is any category with finite products and coproducts, if one takes $R = 1$ the terminal object. Clearly, 1 is exponentiable as the definition $1^A = 1$ will work. However, distributivity in general fails (e.g., the opposite of the category of sets).

However, the following condition, which might be called "distributivity in the exponent", follows from axioms 1–3:

$$R^{(A \times C)+(B \times C)} \cong R^{(A+B) \times C} \tag{3}$$

The proof is as follows. First note that if R is exponentiable, then so is R^D, for any D. Therefore, one can apply (2) to get

$$((A \times C) + (B \times C), R^D) \cong_D ((A + B) \times C, R^D),$$

naturally in D. We also have the natural isomorphism $(A, R^B) \cong_{A,B} (B, R^A)$, and thus

$$(D, R^{(A \times C)+(B \times C)}) \cong_D (D, R^{(A+B) \times C}).$$

Since this is natural in D, we have (3). By scutinizing the argument, one finds that the isomorphism is indeed the same as the canonical morphism $R^d : R^{(A+B) \times C} \to R^{(A \times C)+(B \times C)}$. This suffices to ensure that $R^{\mathbf{C}}$ is a control category.

1.4 Dropping condition 3

Sums are needed in a response category in order to have products in the category of continuations, as was shown in equation (1). In the absense of sums, one can add them freely. This works nicely categorically, but it is almost meaningless syntactically, as the resulting CPS translation is not into the original target language (without sums), but into a new target language obtained from the old language by "adding" sums - i.e., the target language is the same as if one had assumed sums in the first place. Actually, this is not quite true, as sums are only needed at the top level. However, fortunately, there is a direct CPS translation which works and which does not need sums.

The essential modification one must make to the definition of a category of continuations is as follows: instead of considering the full subcategory of objects of the form R^A, we now define $R^{\mathbf{C}}$ to be the full subcategory of $\mathbf{C}$ of objects of the form

$$R^{A_1} \times \ldots \times R^{A_n},$$

where $n \geq 0$. (More formally, when using the convention under "dropping condition 6" above, we can define the objects to be tuples $(A_1, \ldots, A_n)$, written as *formal* expressions $\mathbf{R}^{A_1} \times \ldots \times \mathbf{R}^{A_n}$.)

In the case where $\mathbf{C}$ has sums, this definition coincides, up to equivalence of categories, with the old one, as every object of the form $R^{A_1} \times \ldots \times R^{A_n}$ is isomorphic to one of the form $R^{A_1 + \ldots + A_n}$ in this case. However, the new definition also works in the absense of sums.

First, let us observe that $R^{\mathbf{C}}$ is cartesian-closed. Clearly, as a subcategory of $\mathbf{C}$, it is closed under finite products. Also, if $A = \prod_i R^{A_i}$ and $B = \prod_j R^{B_j}$, then the object

$$B^A := \prod_j R^{B_j \times \prod_i R^{A_i}}$$

is an exponential in $\mathbf{C}$, thus $R^{\mathbf{C}}$ is closed under exponentiation. Next, we need to define the premonoidal structure on $R^{\mathbf{C}}$:

$$\begin{array}{lcll} \bot & := & R^1, & \text{(as before)} \\ (\prod_i R^{A_i}) \parr (\prod_j R^{B_j}) & := & \prod_{ij} R^{A_i \times B_j}. & \end{array}$$

Since the object $\prod_{ij} R^{A_i \times B_j}$ is isomorphic to $\prod_i (\prod_j R^{B_j})^{A_i}$ in the category $\mathbf{C}$, it follows that the expression for $\parr$ is functorial in its right argument, and similarly for the left argument. Thus, we have a binoidal structure.

Consider an exponentiable object S in $\mathbf{C}$, two indexing sets $I = 1, \ldots, n$ and $J = 1, \ldots, m$, and two objects $\prod_i S^{A_i}$ and $\prod_j S^{B_j}$. Consider a map $\sigma : J \to I$, and a tuple $f = (f_1, \ldots, f_m)$ such that $f_j : B_j \to A_{\sigma(j)}$ is a morphism in $\mathbf{C}$. Then define $S^{(\sigma,f)} : \prod_i S^{A_i} \to \prod_j S^{B_j}$ to be the morphism whose jth component is the canonical projection $\prod_i S^{A_i} \to S^{A_{\sigma(j)}}$, followed by $S^{f_j} : S^{A_{\sigma(j)}} \to S^{B_j}$. Then it is easy to see that the morphism $S^{(\sigma,f)}$ is natural in exponentiable objects S. It follows immediately that $R^{(\sigma,f)}$ is central with respect to the binoidal structure on $R^{\mathbf{C}}$.

The other properties of symmetric premonoidal structures are now easily checked, for instance, the associativity and unit maps are all of the form $R^{(\sigma,f)}$, for suitable (σ, f). Codiagonals are given by $\nabla = R^{(\sigma,f)} : \prod_{ij} R^{A_i \times A_j} \to \prod_k R^{A_k}$, where $\sigma = \Delta : I \to I \times I$ and $f_i = \Delta : A_i \to A_i \times A_i$ are all diagonal maps. Weakly initial morphisms are defined analogously. One checks that central maps of the form $R^{(\sigma,f)}$ are indeed focal.

Distributivity holds trivially by the definition of $\parr$, and the distributivity map, as well as the cartesian projections, are clearly focal. To prove that $R^{\mathbf{C}}$ is a control category, it remains to be seen that the exponential strength

$$s : B^A \parr C \to (B \parr C)^A$$

is a natural and coherent isomorphism. In the category $R^{\mathbf{C}}$, let $A = \prod_i R^{A_i}$, $B = \prod_j R^{B_j}$, and $C = \prod_k R^{C_k}$. The map s takes the form

$$s : \prod_{jk} R^{B_j \times \prod_i R^{A_i} \times C_k} \to \prod_{jk} R^{B_j \times C_k \times \prod_i R^{A_i}},$$

and coherence and naturality follow easily. Thus, we have the following theorem:

Theorem 1.3 (New Construction Theorem) *Let $\mathbf{C}$ be a "new" response category in the sense that it only satisfies conditions 1 and 2. Then $R^{\mathbf{C}}$, as defined in this subsection, is a control category.*

1.5 Sum-free CPS translation

Note that the proof of the New Construction Theorem can be regarded as an "indexed version" of the proof given in [3]. The more complex definition of $R^{\mathbf{C}}$ added only notational complexity, but no essential complexity, to the construction. In fact, the proof becomes somewhat simpler since distributivity follows more easily than in the case with sums.

It is now a natural question whether the new construction of control categories gives rise to a new CPS translation of the disjunctive $\lambda\mu$-calculus into a target lambda calculus without sums. As one would expect, this is indeed the case, and we now spell out the CPS translation.

In the usual CPS translation with sums, a computation is represented by a term of type R^A, i.e., as a function from continuations to results. In the new translation, a computation is going to be represented by a term of type $R^{A_1} \times \ldots \times R^{A_n}$, for $n \geq 0$, i.e., by a *tuple* of functions from continuations to results. Consequently, the type of a continuation is no longer a simple type like A, but rather a tuple of types, such as $\langle A^1, \ldots, A^n \rangle$.

We introduce some notation for manipulating tuples and indices. In general, the elements of a tuple $\langle A^i \rangle_{i \in I}$ need not be indexed by the natural numbers, but we allow an arbitrary finite index set I. On index sets I and J, we have the operations of disjoint union $I + J = \{\mathit{in}_1 i \mid i \in I\} \cup \{\mathit{in}_2 j \mid j \in J\}$ and cartesian product $I \times J = \{\langle i, j \rangle \mid i \in I, j \in J\}$. We occasionally identify a finite index set with an initial segment $\{1, \ldots, n\}$ of the natural numbers. This identification is arbitrary, but fixed for each index set I.

If $A = \langle A^i \rangle_{i \in I}$ is a tuple, then we write A^i for its ith component, and we write $\mathit{Ind}(A)$ for its index set I. In particular, $|\mathit{Ind}(A)|$ is the length of the tuple. We write $\langle\rangle$ for the empty tuple. If $A = \langle A^i \rangle_{i \in I}$ and $B = \langle B^j \rangle_{j \in J}$ are tuples, we define their *concatenation* $A \mathbin{@} B$ to be the tuple $C = \langle C^k \rangle_{k \in I+J}$, where $C^{\mathit{in}_1 i} = A^i$ and $C^{\mathit{in}_2 j} = B^j$. If C is a type and B is a tuple of types, we write $C \otimes B$ for the tuple $\langle C \times B^j \rangle_{j \in J}$. If A and B are tuples of types, we define their *pairwise product* $A \otimes B$ to be the tuple $\langle A^i \times B^j \rangle_{\langle i,j \rangle \in I \times J}$.

If σ is a function, we write $\sigma[x \mapsto y]$ for a new function which maps x to y and otherwise behaves like σ.

Definition 1.4 (Call-by-name sum-free CPS translation) We begin as usual by assuming that the target calculus has a type constant $\tilde{\sigma}$ for each type constant σ of the $\lambda\mu$-calculus. To each type A of the $\lambda\mu$-calculus, we inductively associate K_A and C_A, where K_A is a tuple of types, and C_A is a type of the target language. K_A is called the type tuple of *continuations* and C_A is called the type of *computations*. We write K_A^i

$$
\begin{array}{lcll}
\underline{x}^i_\sigma & = & \lambda k^{K^i_A}.(\pi_i\tilde{x})k & \text{where } x : A \\
\underline{\langle M,N\rangle}^{in_1 i}_\sigma & = & \lambda k^{K^{in_1 i}_{A\wedge B}}.\underline{M}^i_\sigma k & \text{where } M : A,\ N : B \\
\underline{\langle M,N\rangle}^{in_2 j}_\sigma & = & \lambda k^{K^{in_2 j}_{A\wedge B}}.\underline{N}^j_\sigma k & \text{where } M : A,\ N : B \\
\underline{\pi_1 M}^i_\sigma & = & \lambda k^{K^i_A}.\underline{M}^{in_1 i}_\sigma k & \text{where } M : A\wedge B \\
\underline{\pi_2 M}^j_\sigma & = & \lambda k^{K^j_B}.\underline{M}^{in_2 j}_\sigma k & \text{where } M : A\wedge B \\
\underline{MN}^i_\sigma & = & \lambda k^{K^i_B}.\underline{M}^i_\sigma\langle \underline{N}_\sigma, k\rangle & \text{where } M : A\to B,\ N : A \\
\underline{\lambda x^A.M}^i_\sigma & = & \lambda\langle\tilde{x},c\rangle^{K^i_{A\to B}}.\underline{M}^i_\sigma c & \text{where } M : B \\
\underline{[\alpha]M}^1_\sigma & = & \lambda k^{K^1_\bot}.\underline{M}^{\sigma(\alpha)}_\sigma\tilde{\alpha} & \text{where } M : A \\
\underline{\mu\alpha^A.M}^i_\sigma & = & \lambda\tilde{\alpha}^{K^i_A}.\underline{M}^1_{\sigma[\alpha\mapsto i]} * & \text{where } M : \bot \\
\underline{[\alpha,\beta]M}^1_\sigma & = & \lambda k^{K^1_\bot}.\underline{M}^{\langle\sigma(\alpha),\sigma(\beta)\rangle}_\sigma\langle\tilde{\alpha},\tilde{\beta}\rangle & \text{where } M : A\vee B \\
\underline{\mu(\alpha^A,\beta^B).M}^{\langle i,j\rangle}_\sigma & = & \lambda\langle\tilde{\alpha},\tilde{\beta}\rangle^{K^{\langle i,j\rangle}_{A\vee B}}.\underline{M}^1_{\sigma[\alpha\mapsto i,\beta\mapsto j]} * & \text{where } M : \bot
\end{array}
$$

Table 1: The sum-free CPS translation of the call-by-name $\lambda\mu$-calculus

for the ith component of the tuple K_A.

$$
\begin{array}{lcll}
K_\sigma & = & \langle\tilde{\sigma}\rangle, & \text{where } \sigma \text{ is a type constant,} \\
K_\top & = & \langle\rangle, & \\
K_{A\wedge B} & = & K_A \mathbin{@} K_B, & \\
K_{A\to B} & = & C_A \otimes K_B, & \\
K_\bot & = & \langle 1\rangle, & \\
K_{A\vee B} & = & K_A \otimes K_B, & \\
C_A & = & (K^1_A \to R)\times\ldots\times(K^n_A\to R), & \text{where } K_A = \langle K^1_A,\ldots,K^n_A\rangle.
\end{array}
$$

We further assume that for each variable x and name α of the $\lambda\mu$-calculus, we are given a variable $\tilde{x}$ or $\tilde{\alpha}$ of the target calculus. Let $\Delta = \alpha_1{:}A_1,\ldots,\alpha_m{:}A_m$ be a control context. A *choice function* for Δ is a function σ from the set of names $\{\alpha_1,\ldots,\alpha_m\}$ to indices such that $\sigma(\alpha_i)\in \mathit{Ind}(K_{A_i})$, for all i.

The call-by-name sum-free CPS translation $\underline{M}_\sigma$ of a typed term $\Gamma\vdash M : A \mid \Delta$ is defined relative to a choice function σ for Δ. By definition, $\underline{M}_\sigma$ is an I-tuple of terms, where $I = \mathit{Ind}(K_A)$. The translation is defined in Table 1. As before, we write $\underline{M}^i_\sigma$ for the ith component of $\underline{M}_\sigma$, and it is understood that this is an operation of the meta-language. However, when we write $\underline{N}_\sigma$ without the superscript, as in the right-hand side of the clause for $\underline{MN}^i_\sigma$, then we mean the literal tuple $\langle\underline{N}^1_\sigma,\ldots,\underline{N}^n_\sigma\rangle$ of the target language.

The CPS translation respects typing in the following sense:

$$\frac{x_1{:}B_1,\ldots,x_n{:}B_n\vdash M : A \mid \alpha_1{:}A_1,\ldots,\alpha_m{:}A_m}{\tilde{x}_1{:}C_{B_1},\ldots,\tilde{x}_n{:}C_{B_n},\tilde{\alpha}_1{:}K^{\sigma(\alpha_1)}_{A_1},\ldots,\tilde{\alpha}_m{:}K^{\sigma(\alpha_m)}_{A_m}\vdash \underline{M}_\sigma : C_A}.$$

Remarks: Note that there is no rule for the unit term $* : 1$. This is because $\mathit{Ind}(K_1) = \emptyset$ and thus $\underline{*}_\sigma = \langle\rangle$ is, by definition, the empty tuple. Also note that

the rules for pairing are equivalent (modulo administrative β-reductions) to a simple concatenation:

$$\underline{\langle M,N\rangle}_\sigma = \underline{M}_\sigma \mathbin{@} \underline{N}_\sigma$$

Similarly, the rules for projection amount to taking a certain sub-tuple.

1.6 Discussion of the CPS transform

We have succeeded in removing sum types from the target language of the Hofmann-Streicher call-by-name CPS transform. This, however, comes at the price of introducing a lot of indices in the meta-language, as the sums, with their associated injections and case distinctions, are pushed into the meta-language (distributing over some other types along the way).

One the one hand, this is interesting. It means that a lot of the analysis which goes into case distinctions can actually be done at compile-time, rather than at run-time. However, if the translation is implemented carelessly, this can lead to an exponential blow-up in code size. For instance, in the rule for application $\underline{MN}_\sigma$, the term $\underline{N}_\sigma$ is replicated many times on the right-hand side, once for every index i. Also, the fact that the translation of code depends on a choice function σ means that, potentially, an exponential amount of code needs to be generated, as there are many such σ to consider. It therefore seems that the sum-free CPS translation is not very useful in practice, at least not in the worst case. However, note that the translation of a closed term does not depend on σ, and in principle, only a small number of values of σ need to be encountered during the recursive translation of a term.

It remains to be seen whether the sum-free CPS translation can be useful in special cases. Many real-world programs probably do not contain a large number of simultaneous control variables of product types, and thus the above-mentioned exponential blowup may not occur in the average case. Also, it would be interesting to know whether the indexing technique can be used, even if the target language has sum types, to do certain compile-time optimizations.

From a pure lambda calculus point of view, sum types are somewhat undesirable and traditionally do not occur in low-level target languages, and one might therefore be glad to be rid of them. However, in practice, the lambda calculus is only used as an intermediate abstraction on the way towards a more low-level implementation, for instance, Krivine's abstract machine. And in terms of Krivine's abstract machine, sum types have a perfectly natural and efficient realization via symbolic tags on the program stack (see e.g. [4]). Removing sum types does not seem to lead to a useful improvement of the abstract machine; in fact, it appears to degrade it. In effect, we are using the type of a program to predict statically how many symbolic tags would be contained in the current stack, and we then store the actual tag information in the instruction pointer, rather than on the stack. One would assume that this leads to a massive duplication of code and no noticeable improvement in performance in general. However, it is still possible that the method can give rise to some interesting static optimizations in special cases.

Another point to note is that the "sum-free" CPS translation will not work once recursive types are introduced into the language. Sums just can't be distributed to the "top level" meaningfully in a recursive type such as $L = 1 + (N \times L)$.

2 Left-to-right vs. right-to-left evaluation

In Section 7 of "Control Categories and Duality" [3], we define call-by-value evaluation in such a way that in an application MN or a pair $\langle M, N\rangle$, the left term M is evaluated before the right term N. We also point out that "the opposite choice would have been equally plausible".

This point is often misunderstood. Some readers get the impression that one ends up with a fundamentally different lambda calculus if one changes the evaluation order from left-to-right to right-to-left, and ask why this apparent asymmetry is not reflected in the categorical semantics. One author even goes as far as claiming that we overlooked this issue [2, Sec. 5.2].

In truth, the call-by-value calculus with right-to-left evaluation is isomorphic to that with left-to-right evaluation. Suppose, for instance, that $\langle M, N\rangle_l$ represents the left-to-right pairing operation, and $\langle M, N\rangle_r$ represents the right-to-left pairing operation. Then each is definable in terms of the other:

$$\begin{array}{rcl} \langle M, N\rangle_l & = & \text{let } x = M \text{ in let } y = N \text{ in } \langle x, y\rangle_r \\ \langle M, N\rangle_r & = & \text{let } y = N \text{ in let } x = M \text{ in } \langle x, y\rangle_l \end{array}$$

As each operator is definable in terms of the other, the question which of the two is included in the syntax, and which one is derived, becomes merely a question of syntactic choice. As there is no semantic difference between the two calculi, there is no resulting asymmetry in the definition of co-control categories.

In fact, it would even be possible to have a syntactically symmetric calculus by requiring, for instance, that only pairs of the form $\langle V, W\rangle$ are allowed, where V and W are values. In this case, the distinction between left-to-right and right-to-left pairing disappears, whereas the general pairing is still definable as

$$\begin{array}{l} \langle M, N\rangle = \text{let } x = M \text{ in let } y = N \text{ in } \langle x, y\rangle \quad \text{or} \\ \langle M, N\rangle = \text{let } y = N \text{ in let } x = M \text{ in } \langle x, y\rangle, \end{array} \tag{4}$$

forcing the programmer to specify the evaluation order each time.

This also explains why this apparent "asymmetry" between the left-to-right and right-to-left evaluation orders has no counterpart in the call-by-name calculus. The dual of a pair of variables $\langle x, y\rangle$ is a term of the form $\lambda\kappa.[x, y]\kappa$. However, while the syntax allows us to replace term variables x, y by arbitrary terms M, N, the same is not true for control variables. There are no "control terms", so the $[x, y]$ notation only applies to variables. Thus, while we can write $\langle M, N\rangle$, forcing us to specify an evaluation order, we cannot write $\lambda\kappa.[M, N]\kappa$. Instead, we are forced to to write either

$$\begin{array}{l} \lambda\kappa.M(\mu x.N(\mu y.[x, y]\kappa)) \quad \text{or} \\ \lambda\kappa.N(\mu y.M(\mu x.[x, y]\kappa)), \end{array} \tag{5}$$

thus specifying the evaluation order explicitly.

The fact that the two variants in (4) are not equivalent in call-by-value is of course due to the fact that pairing defines a premonoidal structure (not a monoidal structure) in call-by-value. Dually, the fact that the two variants in (5) do not coincide in call-by-name is due to the fact that disjunction defines a premonoidal structure there. This is captured in the semantics.

However, the fact that we have a special syntax for one of the two terms in (4), but no special syntax for one of the two terms in (5), is a syntactic artifact and has no semantic meaning.

3 On disjunction types

This section is an edited version of my response to a question by Phil Wadler in early 2003.

3.1 Traditional vs. unified disjunction types

As pointed out in [3], disjunction types can be formulated in two possible ways: the "intuitionistic" (traditional) sum type $A + B$, which is defined via left and right injections and case distinctions:

$$M ::= \ldots \;\big|\; \text{inl } M \;\big|\; \text{inr } N \;\big|\; \text{case } L \text{ of inl } x \Rightarrow M \mid \text{inr } y \Rightarrow N,$$

and the "classical" (unified) disjunction type $A \vee B$, which is defined as:

$$M ::= \ldots \;\big|\; \mu(\alpha^A, \beta^B).M \;\big|\; [\alpha, \beta]M$$

The traditional syntax is characterized by the fact that there are *separate* constructors for left and right injections, whereas the unified syntax uses a *single* constructor. As reported in [3], the two disjunction types are strictly different in call-by-name, but they coincide in call-by-value.

The reason for this is easiest to see in the categorical semantics: in call-by-value, both the "traditional" sum type and the "unified" sum type define a categorical coproduct in a co-control category. Since coproducts are unique up to isomorphism, this implies that the types $A + B$ and $A \vee B$ are isomorphic in call-by-value.

Syntactically, this means that for every term M of type $A + B$ there is a term M' of type $A \vee B$, and for every term N of type $A \vee B$ there is a term N' of type $A + B$, such that $M'' = M$ and $N'' = N$ (modulo the laws of call-by-value equivalence). Specifically, we have

$$\begin{array}{rcl} M' & = & \mu(\alpha, \beta).\text{case } M \text{ of inl } x \Rightarrow [\alpha]x \mid \text{inr } y \Rightarrow [\beta]y \\ N' & = & \mu\gamma.[\gamma]\text{inl } \mu\alpha.[\gamma]\text{inr } \mu\beta.[\alpha, \beta]N \end{array}$$

Then $M'' =_v M$ and $N'' =_v N$ holds in call-by-value. An interesting point is that this works for any terms M, N, and not just for values.

On the other hand, in call-by-name, one does not have $M'' = M$ as above. As a matter of fact, there can be no isomorphism between $A+B$ and $A \vee B$ in call-by-name, which is a consequence of [3, Cor. 3.8].

In fact, in call-by-name, the "traditional" type $A + B$ is provably isomorphic to $\neg\neg(A \vee B)$, or equivalently, to $\neg((\neg A) \times (\neg B))$ (this of course only works in classical logic, i.e., in the presence of continuations). Specifically, for any terms $M : A+B$ and $N : \neg((\neg A) \times (\neg B))$, we define

$$\begin{array}{rcl} M' & = & \lambda p.\text{case } M \text{ of inl } x \Rightarrow (\pi_1 p)x \mid \text{inr } y \Rightarrow (\pi_2 p)y \\ N' & = & \mu\gamma.N(\lambda x.[\gamma]\text{inl } x, \lambda y.[\gamma]\text{inr } y) \end{array}$$

Then the equations $M'' =_n M$ and $N'' =_n N$ hold under the call-by-name laws.

3.2 Traditional vs. unified product types

As Phil Wadler pointed out, one can also formulate conjunction in two ways. There is the traditional formulation, with separate deconstructors:

$$M ::= \ldots \;\big|\; \pi_1 L \;\big|\; \pi_2 L \;\big|\; (M, N).$$

There is also the "unified" formulation, with a single deconstructor:

$$M ::= \ldots \;\big|\; \text{case } L \text{ of } \langle x, y\rangle \Rightarrow M \;\big|\; \langle M, N\rangle.$$

The situation is similar to that of disjunction. Again, the two formulations agree in call-by-value. However, in call-by-name, the situation is trickier. The syntax alone does not uniquely determine the kind of product type we are describing. There are (at least) two possible semantic interpretations of the "unified" syntax (corresponding to different operational semantics, different CPS-translations, different equational theories etc.).

To simplify the discussion, let us use different notations for the two possible product types. Let us write $A \otimes B$ for the type which is found e.g. in Haskell. We will continue to use the following syntax with angled brackets for denoting operations on this type:

$$M ::= \ldots \;\big|\; \text{case } L \text{ of } \langle x, y\rangle \Rightarrow M \;\big|\; \langle M, N\rangle.$$

The intended meaning of "case L of $\langle x, y\rangle \Rightarrow M$" is: evaluate L to something of the form $\langle L_1, L_2\rangle$, then bind x to L_1 and y to L_2 and continue with M.

For the second semantics of the "unified" product type, let us write $A \times B$, and let us use round brackets, so that

$$M ::= \ldots \;\big|\; \text{case } L \text{ of } (x, y) \Rightarrow M \;\big|\; (M, N)$$

Here, the meaning of "case L of $(x, y) \Rightarrow M$" is: bind x to $(\pi_1 L)$ and y to $(\pi_2 L)$, then evaluate M. Unlike with the type $A \otimes B$, we do not evaluate L at all until we need the value of x or y. Thus, this "pattern matching" is really just an abbreviation for the following term in the traditional syntax: $(\lambda x.\lambda y.M)(\pi_1 L)(\pi_2 L)$. Conversely, the traditional destructors $\pi_1 L$ and $\pi_2 L$ can be defined as $\pi_1 L = (\text{case } L \text{ of } (x, y) \Rightarrow x)$ and $\pi_2 L = (\text{case } L \text{ of } (x, y) \Rightarrow y)$. It follows that the second semantics of the "unified" formulation is fact equivalent to the "traditional" formulation.

Let us refer to the type $A \times B$, with pairing (M, N), as the "traditional" product type and to $A \otimes B$ with pairing $\langle M, N\rangle$ as the "unified" type, despite the fact that they can both be written in the unified syntax.

It can be argued that the type $A \times B$ is "more" call-by-name (or demand-driven) than the type $A \otimes B$, in the sense that nothing gets evaluated at all until there is a demand for the variable x or y. The type $A \otimes B$ (and its case construct) introduces some amount of "data-driven" computation (i.e., we evaluate L to $\langle L_1, L_2\rangle$ although this may not be needed in the subsequent computation).

There is an interesting relationship between the types $A \times B$ and $A \otimes B$. Namely, there is a type isomorphism

$$A \otimes B \cong \neg\neg(A \times B).$$

Indeed, for $M : \neg\neg(A \times B)$ and $N : A \otimes B$, define

$$\begin{array}{rcllcl} M' & = & \mu\gamma.M(\lambda p.[\gamma]\langle \pi_1 p, \pi_2 p\rangle) & : & A \otimes B \\ N' & = & \lambda k.\text{case } N \text{ of } \langle x, y\rangle \Rightarrow k(x, y) & : & \neg\neg(A \times B). \end{array}$$

Then one can verify that $M'' =_n M$ and $N'' =_n N$ under the call-by-name laws.

It is worth noting that, when writing isomorphisms such as the above, extreme care must be taken; there are many ways of writing terms of the correct types which do not yield an isomorphism. For instance, the following alternative definition for M' does not work:

$$M' = \langle \mu\alpha.M(\lambda p.[\alpha]\pi_1 p), \mu\beta.M(\lambda p.[\beta]\pi_2 p)\rangle$$

And neither does this alternative definition work for N':

$$N' = \lambda k.k(\text{case } N \text{ of } \langle x, y\rangle \Rightarrow (x, y)).$$

Also note that we do not have an isomorphism between $A \otimes B$ and $A \times B$: while one can find maps going both ways, they are not mutually inverse. For instance, for $M : A \times B$ and $N : A \otimes B$, one naively defines:

$$\begin{array}{rcl} M' & = & \text{case } M \text{ of } (x, y) \Rightarrow \langle x, y\rangle \\ N' & = & \text{case } N \text{ of } \langle x, y\rangle \Rightarrow (x, y) \end{array}$$

Then we have $M'' =_n M$, but we do not have $N'' =_n N$. The reason for this is that $M = (\pi_1 M, \pi_2 M)$ holds for the type $A \times B$, but the corresponding law does not hold for the type $A \otimes B$. Thus, there is no isomorphism between the types $A \times B$ and $A \otimes B$.

Also note that the call-by-name CPS translations for the types $A \times B$ and $A \otimes B$ are different. We have:

$$\begin{array}{l} K_{A\times B} = K_A + K_B \\ K_{A\otimes B} = (C_A \times C_B) \to R \end{array}$$

$$\begin{array}{lcl} \underline{(M, N)} & = & \lambda k.\text{case } k \text{ of inl } k_1 \Rightarrow \underline{M}k_1 \mid \text{inr } k_2 \Rightarrow \underline{N}k_2 \\ \underline{\text{case } M \text{ of } (x, y) \Rightarrow N} & = & \lambda k.\text{let } x = \lambda c.\underline{M}(\text{inl } c) \text{ in let } y = \lambda d.\underline{M}(\text{inr } d) \text{ in } \underline{N}k \\ \underline{\langle M, N\rangle} & = & \lambda k.k\langle \underline{M}, \underline{N}\rangle \\ \underline{\text{case } M \text{ of } \langle x, y\rangle \Rightarrow N} & = & \lambda k.\underline{M}(\lambda p.\text{let } x = \pi_1 p \text{ in let } y = \pi_2 p \text{ in } \underline{N}k) \end{array}$$

From the CPS translation, we can also see that $C_{A\otimes B} = \neg\neg(C_A \times C_B)$ and $C_{A\times B} \cong C_A \times C_B$, so the isomorphism between $A \otimes B$ and $\neg\neg(A \times B)$ also holds with respect to the CPS translation.

3.3 Disjunctions, conjunctions, and duality

We found that there are two possible formulations of conjunction and disjunction, the "traditional" and "unified" formulations. Also, we found that in call-by-value, the two formulations coincide, whereas in call-by-name, they are different. This is somewhat surprising, as one would have expected, by duality, that if conjunctions coincide in call-by-value, then disjunctions coincide in call-by-name, and vice versa. Let us examine this situation closer. To summarize, in call-by-name, we have the following type isomorphisms (with unified types shown on the left, and traditional ones on the right):

$$\begin{array}{lcl} A \otimes B & \cong & \neg\neg(A \times B) \\ (\neg\neg A) \vee (\neg\neg B) & \cong & A + B \end{array}$$

This suggests that in call-by-name, one should take $A \times B$ as the primitive product type, $A \vee B$ as the primitive disjunction type, and one should treat the types $A \otimes B$ and $A + B$ as derived. Whether or not one provides syntactic sugar for the constructors and deconstructors of the derived types $A \otimes B$ and $A + B$ is a matter of taste, but one clearly needs to provide a syntax for the primitive types $A \times B$ and $A \vee B$. Thus, one ends up with the traditional notation for products, and the unified notation for sums.

It is interesting to note that some real-world call-by-name languages, such as Haskell, provide neither of the primitive types $A \times B$ or $A \vee B$; rather, they provide only the derived types $A \otimes B$ and $A + B$.

Regarding duality, we can remark that the dual of the primitive call-by-name types $A \times B$ and $A \vee B$ are the call-by-value types $A + B$ and $A \times B$, respectively. The question remains what happens to the duals of the derived call-by-name types $A \otimes B$ and $A + B$. By duality and the above type isomorphism, we know that the dual of the type $A \otimes B$ is $\neg\neg(A + B)$, and the dual of the type $A + B$ is $(\neg\neg A) \times (\neg\neg B)$. These types of course exist in call-by-value, but they are usually not given a special name or a special syntax.

The best explanation for this phenomenon is that the duality between call-by-name and call-by-value is a *semantic* duality, not a *syntactic* one. Note that, if $\langle\!| M |\!\rangle$ and $(\!| N |\!)$ are the duality translations, we do not claim that $\langle\!| (\!| N |\!) |\!\rangle = N$ and $(\!| \langle\!| M |\!\rangle |\!) = M$ literally, but only up to type isomorphism and the respective equational theories. Thus, we do not claim that every syntactic construct has an exact equivalent in the dual. Rather, we claim that every syntactic construct is *expressible* in the dual. Whether or not the dual of a particular concept is considered important enough to be given a special syntax in the dual language is a matter of taste and programming style.

3.4 Type isomorphism

Maybe one of the most interesting contributions of category theory to the semantics of programming languages is the emphasis that it puts on type isomorphisms. A type isomorphism between types A and B is more than just a translation from A to B and one from B to A. One requires that the two translations are *mutually inverse*, up the the equational theory of the language.

Type isomorphisms capture quite a bit of information about the implementation of a type. Intuitively, if two types are isomorphic, the compiler will probably implement

them in the same way. Or at least it may try to do so.

4 Focus and center

This section is an edited and expanded version of an email to Carsten Führmann in 1999.

In a control category, the center and focus coincide. In other words, every central map is copyable and discardable. This proved in Lemmas 3.2 and 3.12 in [3], but the proof that centrality implies copyability already appears, for $\otimes\neg$-categories, in Thielecke's thesis [5].

One may ask whether the converse holds, i.e., whether every copyable and discardable map is central. This is not the case, as the following counterexample shows.

Let $R = \{0, 1, 2, 3\}$ and consider the category $R^{\mathbf{Set}}$, i.e. the category of continuations made from $\mathbf{Set}$ with response object R. Consider the following map $f : R^2 \to R^1$:

f	0	1	2	3
0	0	0	2	2
1	1	1	3	3
2	0	0	2	2
3	1	1	3	3

The table shows $f(a, b)$ in the ath row and bth column. Notice that f has the two following properties, for all $a, b, c, d \in R$.

$$\begin{array}{rcl} f(a, a) & = & a, \\ f(f(a, b), f(c, d)) & = & f(a, d). \end{array}$$

To check the second property quickly, notice that $f(a, b) = (a \,\&\, 1 \mid b \,\&\, 2)$, where "&" and "|" denote bitwise "and" and "or", respectively.

The first property is equivalent to discardability, and the second one to copyability of f. However, f is not central, e.g. consider $g : R^1 \to R^1$ with $g(0) = 0$, $g(1) = 2$, $g(2) = 1$, $g(3) = 3$. Then $f(g(1), g(2)) = 0 \neq 3 = g(f(1, 2))$. This implies that f is not central.

This example may be somewhat puzzling, until one realizes that it is a special case of the following construction. The following may also be the key to finding a syntactic programming language example.

Consider a cartesian-closed category $\mathbf{C}$ with two distinguished objects S and T. Notice that for any object A,

$$(S \times T)^A \cong S^A \times T^A.$$

Now consider the continuation category with response object $S \times T$. If $u, v : A \to B$

are any two maps, then we can define a new map $f : (S\times T)^B \to (S\times T)^A$ as follows:

$$\begin{tikzcd}
(S\times T)^B \arrow[r, "f"] \arrow[d, "\cong"'] & (S\times T)^A \arrow[d, "\cong"] \\
S^B \times T^B \arrow[r, "S^u\times T^v"] & S^A \times T^A
\end{tikzcd}$$

Then f is copyable and discardable, because S^u and T^v are copyable and discardable in $S^{\mathbf{C}}$ and $T^{\mathbf{C}}$, respectively, and copyability and discardability work "componentwise" in $(S\times T)^{\mathbf{C}}$. On the other hand, unless $u = v$, f is not in general of the form $(S\times T)^w$, and thus not in general central.

As a special case, it follows that in the control category $R^{\mathbf{Set}}$, there are copyable and discardable maps which are not central, if the cardinality of R is composite, i.e., if R is infinite or $|R| = nm$ for $n, m \geq 2$. Interestingly, the converse also holds: if the cardinality of R is a prime number (or 0 or 1), then copyability and discardability implies centrality in $R^{\mathbf{Set}}$. In case $|R| = 0$ or $|R| = 1$, the category $R^{\mathbf{Set}}$ is a preorder and there is nothing to show (all maps are central).

Theorem 4.1 *Let R be a set of finite cardinality p, where p is a prime number. Let $\star : R^2 \to R$ be a binary operation satisfying*

1. $a \star a = a$,
2. $(a \star b) \star (c \star d) = a \star d$,

for all $a, b, c, d \in R$. Then $\star$ is a projection, i.e., either $a \star b = a$ for all $a, b \in R$, or $a \star b = b$ for all $a, b \in R$.

Proof. The crucial observation is that for all $a, b, x, y \in R$,

$$a \star x = a \star y \Rightarrow b \star x = b \star y.$$

Because, if $a\star x = a\star y$, then we have $b\star x = (b\star x)\star(a\star x) = (b\star x)\star(a\star y) = b\star y$. It follows that we can define an equivalence relation on R by setting $x \sim y$ if for some $a \in R$, $a\star x = a\star y$. Note that for all $a, x \in R$, $a\star(a\star x) = (a\star a)\star(a\star x) = a\star x$, thus $a \star x \sim x$.

Next, we claim that each equivalence class $[x]_\sim$ has equal cardinality. We define $\phi_{yx} : [x]_\sim \to [y]_\sim$ for every $x, y \in R$ by $\phi_{yx}(z) = z \star y$. We claim that ϕ_{xy} is a bijection with inverse ϕ_{yx}. Namely, for any $z \in [x]_\sim$, we have $\phi_{xy}(\phi_{yx}(z)) = (z\star y)\star x = (z\star y)\star(x\star x) = z\star x = z\star z = z$. Thus, the equivalence relation $\sim$ partitions R into sets of equal cardinality. Since $|R| = p$ is prime, it follows that the cardinality of the equivalence classes is either 1 or p. If the cardinality is 1, then $a\star x \sim x$ implies $a\star x = x$, for all a, x; in this case, $\star$ is the second projection. Else, the cardinality is p, hence $a = a\star a = a\star y$ for all a, y, i.e., $\star$ is the first projection. $\square$

Corollary 4.2 *Let R be a finite set of prime cardinality, I any set, and let $f : R^I \to R$ such that*

$$\begin{tikzcd}
R^I \arrow[r, "f"] & R \\
R \arrow[u, "\Delta"] \arrow[ur, "id"'] &
\end{tikzcd}
\quad \textit{and} \quad
\begin{tikzcd}
R^I \arrow[r, "f"] & R \\
(R^I)^I \arrow[u, "\nabla"] \arrow[r, "f^I"] & R^I, \arrow[u, "f"']
\end{tikzcd}$$

where $\Delta(a)(i) = a$ and $\nabla(g)(i) = g(i)(i)$. Then there exists $i \in I$ such that $f(\vec{a}) = a_i$, for all $\vec{a} = (a_i)_{i\in I}$.

Proof. We think of f as an I-ary operation on R. We say that f *depends* on an index $i \in I$ if there exist elements $\vec{a}, \vec{b} \in R^I$ such that $a_j = b_j$ for all $j \neq i$, but $f(\vec{a}) \neq f(\vec{b})$. Clearly, by the first diagram, f is not a constant function and thus depends on at least one i. Let $S = \{i \in I \mid f \text{ depends on } i\}$. We show that S is a singleton by showing that for any partition of I into two subsets $I = I_1 \dot{\cup} I_2$, $S \subseteq I_1$ or $S \subseteq I_2$. Namely, given such a partition, define $\star : R^2 \to R$ by $a \star b = f(\vec{c})$, where $c_i = a$ if $i \in I_1$ and $c_i = b$ if $i \in I_2$. The commutative diagrams ensure that $\star$ satisfies the conditions of Theorem 4.1, thus it is independent of one of its arguments. This proves that S is a singleton, say $S = \{i\}$. The first diagram ensures that f is the projection onto its ith argument. $\square$

Corollary 4.3 *Let R be a finite set of prime cardinality. Then in the control category $R^{\mathbf{Set}}$, all copyable discardable morphisms are central.*

Proof. Suppose $f : R^I \to R^J$ is discardable and copyable. For each $j \in J$, define $f_j : R^I \to R$ by $f_j(\vec{a}) = f(\vec{a})(j)$. Since f is discardable and copyable, f_j satisfies the hypotheses of Corollary 4.2. Thus, f_j is a projection onto some component i_j. It follows that $f = R^\phi$, where $\phi(j) = i_j$. Thus f is central. $\square$

5 Control categories and coproducts

Many questions about control categories are of the form "does there exists a non-trivial control category with property X". Here, non-trivial means that the category is not a preorder. One such question, posed by Masahito Hasegawa, is whether there exists a non-trivial "local" control category, i.e., one whose slices are control categories. I do not know the answer to this question.

Hasegawa also asked whether there exists a non-trivial control category with finite coproducts. We already know from [3, Cor. 3.8] that $A \parr B$ cannot be a coproduct. However, the question is whether coproducts can exist in addition to the control category structure.

The answer is no, due to the following theorem:

Theorem 5.1 *Any control category with initial object is a preorder.*

Proof. Recall that $0 \times A \cong 0$ and $0^0 \cong 1$ in any cartesian-closed category. Now consider the projections $\pi_1 : 0 \times \bot \to 0$ and $\pi_2 : 0 \times \bot \to \bot$. These maps are central; moreover π_1 is an isomorphism. It follows that the unique map $0 \to \bot$ is central (this map is $\pi_2 \circ \pi_1^{-1}$). There is also a central map $\bot \to 0$. Since the only central map $\bot \to \bot$ is the identity, it follows that $0 \cong \bot$.

By Lemma 3.7 and Corollary 3.14 of [3], it follows that the category is a preorder: suppose that $f : 1 \to C$. By Corollary 3.14, there is a central morphism $g : \bot^\bot \to C$. But $\bot^\bot \cong 0^0 \cong 1$, hence there is a central $h : 1 \to C$. By Lemma 3.7, this implies $C \cong 1$. Thus, for all C, $|(1, C)| \leq 1$. This implies $|(A, B)| = |(1, B^A)| \leq 1$, hence the category is a preorder. □

It immediately follows that the category of sets is not a control category, nor is any presheaf category.

Hasegawa pointed out that the first part of the proof can be simplified because in any cartesian-closed category, the initial object is strict, i.e., if there exists $f : A \to 0$, then A is initial [1]. Since $i : \bot \to 0$ always exists, $\bot$ is automatically initial.

References

[1] J. Lambek and P. J. Scott. *An Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics 7. Cambridge University Press, New York, 1986.

[2] I. Ogata. A proof theoretical account of continuation passing style. In *Proceedings of CSL'02*, Springer LNCS 2471, pages 490–505, 2002.

[3] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2), 2001.

[4] P. Selinger. From continuation passing style to Krivine's abstract machine. Preprint. Available from http://quasar.mathstat.uottawa.ca/˜selinger/papers.html, 2003.

[5] H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.