

Discharging sub-derivations: A  
proof-theoretic Curry-Howard  
correspondence for a  $\lambda$ -calculus with  
patterns

Nissim Francez  
Computer Science dept.,  
Technion-IIT, Haifa, Israel  
francez@cs.technion.ac.il

Work in progress ...

## Introduction

- From the perspective of the Curry-Howard correspondence (CH), *abstraction over a variable* in the simply-typed  $\lambda$ -calculus  $TA_\lambda$  corresponds to the proof-theoretic notion of *discharge of an assumption* in implication-introduction within natural-deduction (ND) proof-systems.
- Various (propositional) logics and  $\lambda$ -calculi can be obtained by side-conditions on the discharge/abstraction (e.g., side-conditions on the number of occurrences of the free abstracted variable in the body of the expression, Gabbay and de Queiroz 92). Similarly, application in the  $\lambda$ -calculus corresponds by CH to implication-elimination in ND.
- Schroeder-Heister makes a distinction is made between *specific* and *non-specific* assumptions. The former are introduced in an ND proof/derivation “according to their meanings”, while the latter are mere “placeholders” (for a closed proof). While he focuses on differences in *introducing* the two kinds of assumptions, I focus on differences in the *discharge* of the two kinds.

## Aim

- In the standard (simply-typed)  $\lambda$ -calculus, abstraction is based on non-specific assumptions: the term  $\lambda x.M$  does not impose any restrictions on the “role” of  $x$  within  $M$ . This view is corroborated by the standard  $\beta$ -reduction, that allows the (capture free) substitution of an *arbitrary* term  $N$  for  $x$  in  $M$ , as the result of applying  $\lambda x.M$  to  $N$ .
- I study a generalization of the  $\lambda$ -calculus that *does allow* imposing restrictions on the form of  $N$  as a *precondition to a reduction step*.
- This results from *specific abstraction* over assumptions producing a term, with which  $N$  has to *match* for the reduction to take place.
- Formally, a new term is introduced, having the form  $\lambda[M'].M$ , where  $M'$  is itself a term, *not necessarily a variable*.
- This term is interpreted proof-theoretically as a *discharge of a sub-derivation* (with all its open assumptions).
- By identifying  $\lambda[x].M$  with  $\lambda x.M$  we recover the usual  $\lambda$ -calculus as a sub-calculus.

## Simultaneous Abstraction

– Technically, the proposed generalization amounts to an *interdependent simultaneous abstraction* of a collection of related variables: when well-typed,  $\lambda[M'].M$  binds *simultaneously* the *free* variables of  $M'$ , imposing a mutual-dependence relation over them. The role of the *bound* variables in  $M'$  is an auxiliary means for expressing the required dependency relation.

– A simple example: suppose it is desired to have an abstraction over  $M$  to be applicable *only* to terms  $N$  that are themselves applications, say  $(PQ)$ . This is facilitated by abstracting over a “generic application”, say  $(uv)$ , forming  $\lambda[(uv)].M$ ; during a reduction-step,  $(uv)$  has to *match*  $(PQ)$ , producing a substitution  $s = [u/P, v/Q]$ , and the result of the reduction-step is the term  $sM = M[u/P, v/Q]$ , i.e., the simultaneous substitution of  $P, Q$  for all free occurrences of  $u, v$ , respectively, in  $M$ .

- Note that this renders reduction as a *partial* operation: if  $N$  is *not* of the form  $(PQ)$ , the reduction-step cannot take place.

-The term  $\lambda x.M$  is the special case where no restriction is imposed on  $N$  in a reduction-step.

## Relation to Previous Work

– A similar calculus,  $\lambda\phi$ -calculus, was presented in van Oostrom (TR, 1990), with the aim of *pattern matching* in functional programming. The focus there is on establishing *confluence* of the induced reduction

Later, Barthe, Cirstea, Kirshner and Liquori (POPL 03) studied typing in such functional language. None of them considers the proof-theoretical interpretation of the calculus, in particular not sub-proof discharge.

– In *Logic Programming* (e.g., PROLOG) the same effect is obtained via *term-unification*.

## Generalized Terms

The set  $GLTerm$  of *generalized  $\lambda$ -terms* is defined below, with  $Free(.)$  the set of free variables.

1. If  $x \in V$ , then  $x \in GLTerm$ , and  $Free(x) = \{x\}$ .
2. If  $M, N \in GLTerm$ , then  $(MN) \in GLTerm$ , with  $GFree(MN) = Free(M) \cup Free(N)$ .
3. If  $M, M' \in GLTerm$ , then  $\lambda[M].M' \in GLTerm$ , with  $Free(\lambda[M].M') = Free(M') - Free(M)$ .

Examples of generalized  $GLTerms$  are  $\lambda[(uv)].(vu)$ ,  $\lambda[\lambda w.(u(wv))].(u(xv))$ .

## Matching

The *matching* of a term  $M \in LTerm$  with another term  $M' \in LTerm$ , where  $M \sqsubseteq M'$ , (*subsumption*) producing the *induced substitution*  $s = \mu(M, M')$ , is defined by induction on  $M$ .

1. If  $M \equiv x$ , then for every  $M'$ ,  $\mu(M, M') = [x/M']$ .
2. If  $M \equiv (PQ)$ ,  $M' \equiv (P'Q')$ ,  $\mu(P, P') = s_1$  and  $\mu(Q, Q^{prime}) = s_2$ , then  $\mu(M, M') = s_1 \cup s_2$  (implying compatibility of  $s_1, s_2$ ).
3. If  $M \equiv \lambda[P].N$ ,  $M' \equiv \lambda[P'].N'$ , and  $\mu(N, N') = s_1 \cup s_2$ , where  $\mu(P, P') = s_2$ , then  $\mu(M, M') = s_1$

The induced substitution  $\mu(M, M')$  is the most general substitution  $s$  satisfying  $sM \equiv M'$ . Note that matching is invariant under  $\alpha$ -equivalence.

## Examples of Matching and Non-matching

- $M = (uv)$  matches  $(PQ)$  with induced substitution  $[u/P, v/Q]$  (possibly,  $P = Q$ —*untypable*).
- $(uu)$  matches  $(PP)$  with  $[u/P]$ , but does not match  $(PQ)$  (for  $P \neq Q$ ), because  $u$  matches  $P$  with  $[u/P]$  and  $u$  matches  $Q$  with  $[u/Q]$  and those two substitutions *are not compatible*.
- $(uv)$  does not match  $\lambda x.x$ , the latter not of the form of an application.
- $\lambda w.(u(wv))$  matches  $\lambda w'.(P(w'Q))$ , with  $s = [u/P, v/Q]$ .
- $\lambda x.(wx)$  does not match  $\lambda y.((yu)(yv))$  (the latter *untypable*) with  $[w/(yu)]$ , since such a matching would require  $[x/(yv)]$ , not compatible with  $[x/y]$ .
- Matching can “dig” deeper, as with  $((\lambda x.(yy))((zu)y))$ , which matches  $((\lambda w.(PP))((\lambda r.(rr)Q)P))$  (for any  $P, Q$ ), with  $[y/P, z/\lambda r.(rr), u/Q]$ .

## Generalized $\beta$ -Reduction

We now define a generalization of  $\beta$ -reduction, denoted by  $\hat{\beta}$ -reduction, a (binary) relation between generalized terms.

The contextual closure of

$$((\lambda[N].M)P) \rightsquigarrow_{\hat{\beta}} sM$$

where  $s = \mu(N, P)$ .

- Note that the usual  $\beta$ -reduction is a special case of the  $\hat{\beta}$ -reduction, since for  $(\lambda x.MP)$ ,  $x$  matches  $P$  with induced substitution  $s = [x/P]$ , so we get  $[x/P]M$ , the usual result of  $\beta$ -reduction.

**Example:** since  $\mu((uv), (PQ)) = [a/u, v/P]$ , we have

$$(\lambda[(uv)].(uv)(PQ)) \rightsquigarrow_{\hat{\beta}} [u/P, v/Q](uv) = (PQ) \quad (1)$$

## The System $TA_{\hat{\lambda}}$

– We now pass to the proof-theoretic reflection of generalized terms and the reduction among them. The ND-rules below are a *typing rules* for generalized terms, still using the intuitionistic implicational fragment as types.

$$\frac{[\Gamma_1]_i, \Gamma_2 \vdash Q : \tau \quad [\Gamma_1 \vdash P : \sigma]_i}{\Gamma_2 \vdash \lambda[P].Q : (\sigma \rightarrow \tau)} (\rightarrow I_i)$$

$$\frac{\Gamma_1 \vdash \lambda[P].Q : \sigma \rightarrow \tau \quad \Gamma_2 \vdash P' : \sigma}{\Gamma_1 \Gamma_2 \vdash sQ : \tau} (\rightarrow E), \text{ where } s = \mu(P, P')$$

Here  $s$  is the substitution produced by *matching*  $P$  and  $P'$  (defined below). The second premise is called a *licensing derivation*.  $\sigma, \tau$  range over wffs in the implicational fragment of the propositional calculus.

## Abstracting application

– Consider the (simply-typed) identity function  $\lambda x.x : B \rightarrow B$ . Suppose we want to restrict it to terms of type  $B$  that are *applications*, i.e., matching  $(uv)$ : hence,  $u$  is of type  $A \rightarrow B$ , and  $v$  of type  $A$ , for some  $A, B$ . This is achieved by abstracting *simultaneously* two assumptions, by a licensing derivation that establishes the type  $B$  for  $(uv)$ , thereby recording in the term that this type was formed by an application, abstracted over.

$$\frac{\frac{[u : A \rightarrow B]_1 \quad [v : A]_1}{(uv) : B} (\rightarrow E)}{\lambda[(uv)].(uv) : B \rightarrow B} (\rightarrow I_1)$$

First, observe that the conclusion is of the required form. It is of the type of the identity function, and its term restricts application (via  $\hat{\beta}$ -reduction) only to terms in the form of an application. What we did is to abstract simultaneously over  $\{u, v\}$  (by discharging simultaneously a “package” of two different assumptions, indexed **1** in the example), producing a type ( $B$  in the example), that becomes an antecedent of an implication based on a *licensing derivation*. The licensing (sub-)derivation is discharged together with the discharged assumptions.

## Restricting Contexts (Open Assumptions)

Consider another example, showing the effect on context (undischarged (open) assumptions).

$$\frac{\frac{[u : (A \rightarrow B)]_1 \quad \frac{x : (A \rightarrow A) \quad [v : A]_1}{(xv) : A} (\rightarrow E)}{(u(xv)) : B} (\rightarrow E) \quad \left[ \frac{u : (A \rightarrow B) \quad v : A}{(uv) : B} (\rightarrow E) \right]_1}{\lambda[(uv)].(u(xv)) : (B \rightarrow B)} (\rightarrow I_1)$$

Here  $x$  has to be of a “mediating” type  $A \rightarrow A$  for the term to be well-typed in contrast to being of the “mediating” typed  $A \rightarrow B$  in simple typing.

## Abstracting abstraction

Consider the term  $\lambda[\lambda w.(u(wv))].(u(xv))$ . Here too, the abstracted term will have to be well-typed, restricting simultaneously both  $u$  and  $v$ , *but not  $w$* , abstracted over internally (**reflecting an assumption discharge within the licensing derivation**). The restriction is that  $u$  must have a type applicable to a function applied to  $(xv)$ , not to  $v$  itself. Thus, in the body of the generalized term, (*the free!*)  $x$  will reflect an assumption having such a type.

$$\frac{[u : (B \rightarrow C)]_1 \quad \frac{x : (A \rightarrow B) \quad [v : A]_1 (\rightarrow E)}{(xv) : B} (\rightarrow E)}{(u(xv)) : C} (\rightarrow E) \quad \text{see below} \quad (\rightarrow I_1)}{\lambda[\lambda w.(u(wv))].(u(xv)) : (((A \rightarrow B) \rightarrow C) \rightarrow C)} (\rightarrow I_1)$$

$$\left[ \frac{\frac{u : (B \rightarrow C) \quad \frac{[w : (A \rightarrow B)]_2 \quad v : A}{(wv) : B} (\rightarrow E)}{(u(wv)) : C} (\rightarrow E)}{\lambda w.(u(wv)) : ((A \rightarrow B) \rightarrow C)} (\rightarrow I_2) \right]_1$$