Encoding Product Types

SAM LINDLEY, The University of Edinburgh, UK

Can product types be encoded in simply-typed lambda calculus with base types and function types? In this paper we demonstrate that the answer is more nuanced than one might expect. For instance, it depends on whether the encoding is allowed to be global or not, whether the encoding is allowed to be type-indexed or not, the number of base types, whether the encoding is allowed to use η-conversion or not, and whether the base types include constants or not.

1 SIMPLY-TYPED LAMBDA CALCULUS

We begin by considering simply-typed lambda calculi whose types are constructed from base types (X) and function types ($A \to B$).

$$\begin{array}{ll} \text{Type contexts} & \Delta ::= \cdot \mid \Delta, X \\ \text{Types} & A, B ::= X \mid A \to B \\ \text{Term contexts} & \Gamma ::= \cdot \mid \Gamma, x : A \\ \text{Terms} & M, N ::= x \mid \lambda x^A.M \mid M\ N \end{array}$$

We track the base types in a type context Δ. We frequently omit type annotations on bound variables. We identify terms up to renaming of bound variables. We write $A[B/X]$ for the substitution of type B for X in A. Similarly, we write $M[N/x]$ for the (capture-avoiding) substitution of term N for x in M. Well-kinded types are constructed from base types and the function type constructor.

$\boxed{\Delta \vdash A : \star}$

$$\textsc{Base}\ \frac{X \in \Delta}{\Delta \vdash X : \star} \qquad \textsc{Fun}\ \frac{\Delta \vdash A : \star \qquad \Delta \vdash B : \star}{\Delta \vdash A \to B : \star}$$

Well-typed terms are constructed from variables, lambda abstraction, and application.

$\boxed{\Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Lam}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

We assume here that all types appearing in typing rules and term contexts are well-kinded with respect to a fixed Δ. There are two rewrite rules for the core calculus with function types ($\to.\beta$ and $\to.\eta$).

$$\begin{array}{rcll} (\lambda x.M)\ N & \rightsquigarrow & M[N/x] & (\to.\beta) \\ M & \rightsquigarrow & \lambda x.M\ x & (\to.\eta) \end{array}$$

(We have chosen to orient the η-rule as an expansion [5], but in this paper we focus on the conversion relation so the choice of orientation of the underlying rewrite relation has no material impact on our results.) Later we will add further β and η rewrite rules for product types. We write $\rightsquigarrow_\beta$ for β-reduction and $\rightsquigarrow_\eta$ for η-reduction.

We use the rewrite relations to generate corresponding conversion relations. We write $\sim$ for the compatible, transitive, reflexive, symmetric closure of $\rightsquigarrow$ (i.e. the relation that allows the reduction rules to be applied backwards or forwards anywhere in a term any number of times). We write $\sim_\beta$ and $\sim_\eta$ for the corresponding closure of $\rightsquigarrow_\beta$ and $\rightsquigarrow_\eta$ respectively.

Author's address: Sam Lindley, The University of Edinburgh, Edinburgh, UK, Sam.Lindley@ed.ac.uk.

Definition 1.1. A β-normal form is a simply-typed lambda calculus term whose shape matches the syntactic category of normal forms given below.

$$\begin{array}{ll} \text{Normal forms} & M ::= N \mid \lambda x^A.M \\ \text{Neutral forms} & N ::= x \mid N\ M \end{array}$$

It is standard that every well-typed term is β-convertible to a unique β-normal form.

Proposition 1.2. *Given $\Gamma \vdash M : A$ there exists a unique β-normal form $\Gamma \vdash M' : A$ such that $M \sim_\beta M'$.*

This result follows, for instance, from confluence and termination for simply-typed lambda-calculus.

1.1 Products

The extension of simply-typed lambda calculus with product types is standard. The syntax of types is extended with the product type and the syntax of terms with pairs and first and second projections.

$$\begin{array}{ll} \text{Types} & A, B ::= \cdots \mid A \times B \\ \text{Terms} & M, N ::= \cdots \mid \mathbf{pair}\ M\ N \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \end{array}$$

The kinding and typing rules are extended as follows.

$\boxed{\Delta \vdash A : \star}$

$$\cdots \qquad \textsc{Prod}\ \frac{\Delta \vdash A : \star \qquad \Delta \vdash B : \star}{\Delta \vdash A \times B : \star}$$

$\boxed{\Gamma \vdash M : A}$

$$\cdots \qquad \textsc{Pair}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \mathbf{pair}\ M\ N : A \times B} \qquad \textsc{Fst}\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathbf{fst}\ M : A} \qquad \textsc{Snd}\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathbf{snd}\ M : B}$$

For products there are two β-rewrite rules, one for the first projection ($\times.\beta_2$) and another for the second projection ($\times.\beta_2$), and one η-rewrite rule ($\times.\eta$).

$$\begin{array}{rcll} \mathbf{fst}\ (\mathbf{pair}\ M\ N) & \rightsquigarrow & M & (\times.\beta_1) \\ \mathbf{snd}\ (\mathbf{pair}\ M\ N) & \rightsquigarrow & N & (\times.\beta_2) \\ M & \rightsquigarrow & \mathbf{pair}\ (\mathbf{fst}\ M)\ (\mathbf{snd}\ M) & (\times.\eta) \end{array}$$

When working with the extended calculus $\rightsquigarrow$ and $\sim$ and their subscripted variants are extended in the obvious way. As well as working with the full β-conversion relation and full $\beta\eta$-conversion relation, we will also sometimes work with the relation that is closed under all of the β-rules and the $\rightarrow.\eta$-rule but not $\times.\eta$. We will denote this convertibility relation by $\sim_{\beta\eta^{\rightarrow}}$.

This paper studies encodings of product types in terms of base types and function types.

1.2 Compositional Encodings

As all of the encodings we consider in this paper target syntax, we choose to give a syntactic characterisation of what it means to be a compositional encoding.

First let us consider the encoding of types. A compositional encoding of types is determined by three open type parameters.

$$\begin{array}{l}\Delta, \mathcal{X} \vdash \mathsf{Base} : \star \\ \Delta, \mathcal{X}, \mathcal{Y} \vdash \mathsf{Fun} : \star \\ \Delta, \mathcal{X}, \mathcal{Y} \vdash \mathsf{Prod} : \star\end{array}$$

Here we extend the fixed Δ of the object language with type variables $\mathcal{X}, \mathcal{Y}$ in order to define type operators which we will use to interpret base types, function types, and products. We will not use these open types directly, but instead always substitute their free type variables with closed types using the following syntactic sugar.

$$\begin{array}{rcl}\mathsf{Base}[A] & \equiv & \mathsf{Base}[A/\mathcal{X}] \\ \mathsf{Fun}[A, B] & \equiv & \mathsf{Fun}[A/\mathcal{X}, B/\mathcal{Y}] \\ \mathsf{Prod}[A, B] & \equiv & \mathsf{Prod}[A/\mathcal{X}, B/\mathcal{Y}]\end{array}$$

An encoding $[\![-]\!]$ on types is compositional if it factors through suitable definitions of Base, Fun, and Prod.

$$\begin{array}{rcl}[\![X]\!] & = & \mathsf{Base}[X] \\ [\![A \to B]\!] & = & \mathsf{Fun}[[\![A]\!], [\![B]\!]] \\ [\![A \times B]\!] & = & \mathsf{Prod}[[\![A]\!], [\![B]\!]]\end{array}$$

We fix encodings of contexts to be fully determined by an encoding of types. The encoding of type contexts is the identity (we assume the same base types are available in the source and the target).

$$[\![\Delta]\!] = \Delta$$

The encoding on term contexts is pointwise.

$$\begin{array}{rcl}[\![\cdot]\!] & = & \cdot \\ [\![\Gamma, x : A]\!] & = & [\![\Gamma]\!], x : [\![A]\!]\end{array}$$

Just as we characterise a compositional encoding of types by factoring it through an open type parameter for each type constructor, we characterise a compositional encoding of terms by factoring it through an open term parameter for each term constructor.

$$\begin{array}{l}x : A \vdash \mathsf{var}_A : A \\ f : A \to B \vdash \mathsf{lam}_{A,B} : \mathsf{Fun}[A, B] \\ f : \mathsf{Fun}[A, B], x : A \vdash \mathsf{app}_{A,B} : B \\ x : A, y : B \vdash \mathsf{pair}_{A,B} : \mathsf{Prod}[A, B] \\ p : \mathsf{Prod}[A, B] \vdash \mathsf{fst}_{A,B} : A \\ p : \mathsf{Prod}[A, B] \vdash \mathsf{snd}_{A,B} : B\end{array}$$

Again, we do not use these open terms directly, but rather define syntactic sugar for substituting for the free variables.

$$\begin{array}{rcl}\mathsf{var}_A[M] & \equiv & \mathsf{var}_A[M/x] \\ \mathsf{lam}_{A,B}[M] & \equiv & \mathsf{lam}_{A,B}[M/f] \\ \mathsf{app}_{A,B}[M, N] & \equiv & \mathsf{app}_{A,B}[M/f, N/x] \\ \mathsf{pair}_{A,B}[M, N] & \equiv & \mathsf{pair}_{A,B}[M/x, N/y] \\ \mathsf{fst}_{A,B}[M] & \equiv & \mathsf{fst}_{A,B}[M/p] \\ \mathsf{snd}_{A,B}[M] & \equiv & \mathsf{snd}_{A,B}[M/p]\end{array}$$

An encoding $[\![-]\!]$ on terms is compositional if it factors through suitable definitions of var, lam, app, pair, fst, and snd.

$$\begin{array}{rcl}
[\![x^A]\!] & = & \mathsf{var}_{[\![A]\!]}[x] \\
[\![\lambda x^A.M^B]\!] & = & \mathsf{lam}_{[\![A]\!],[\![B]\!]}[\lambda x^{[\![A]\!]}.[\![M]\!]] \\
[\![M^{A\to B}\ N^A]\!] & = & \mathsf{app}_{[\![A]\!],[\![B]\!]}[[\![M]\!],[\![N]\!]] \\
[\![\mathbf{pair}\ M^A\ N^B]\!] & = & \mathsf{pair}_{[\![A]\!],[\![B]\!]}[[\![M]\!],[\![N]\!]] \\
[\![\mathbf{fst}\ M^{A\times B}]\!] & = & \mathsf{fst}_{[\![A]\!],[\![B]\!]}[[\![M]\!]] \\
[\![\mathbf{snd}\ M^{A\times B}]\!] & = & \mathsf{snd}_{[\![A]\!],[\![B]\!]}[[\![M]\!]]
\end{array}$$

The most basic sanity check for any translation between typed calculi is that it preserves typing judgements. Thus we we will always ensure that whenever $\Gamma \vdash M : A$ then $[\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$. Indeed we consider a translation to be ill-defined if it does not preserve typing (and if we were to mechanise our development in a system like Agda or Coq we would use an intrinsically typed representation in which type preservation holds by construction).

Formally one can view a translation on terms as being defined on typed terms, that is typing derivations, rather than on untyped terms. Often such translations are parametric in type information so we can write them down as if they were defined on untyped terms. Sometimes, however, our translations will depend at least to some extent on type information in the derivation, which we abbreviate by placing annotations on subterms. Correspondingly, we choose to include or omit type subscripts on the open term and type parameters depending on whether an encoding is parametric or type-indexed.

1.3 Local Encoding of Products

An encoding of products is local if all other features are encoded directly as is. Concretely, this means that the open type and term parameters are defined as follows.

$$\begin{array}{rcl}
\mathsf{Base} & = & X \\
\mathsf{Fun} & = & X \to Y
\end{array}$$

$$\begin{array}{rcl}
\mathsf{var} & = & x \\
\mathsf{lam} & = & f \\
\mathsf{app} & = & f\ x
\end{array}$$

The encoding itself is then partially defined as follows.

$$\begin{array}{rcl}
[\![X]\!] & = & X \\
[\![A \to B]\!] & = & [\![A]\!] \to [\![B]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.M]\!] & = & \lambda x.[\![M]\!] \\
[\![M\ N]\!] & = & [\![M]\!]\ [\![N]\!]
\end{array}$$

To complete the local encoding we just need to describe how to encode product types (Prod), pairs (pair), and projections (fst and snd).

2 GLOBAL ENCODING WITH CONTINUATION-PASSING STYLE

One way of encoding products hinges on the isomorphism between uncurried and curried functions:

$$(\mathsf{curry}, \mathsf{uncurry}) : (A \times B) \to C \simeq A \to B \to C$$

$$\begin{aligned}\mathsf{curry} &= \lambda f.\lambda x.\lambda y.f\ (\mathbf{pair}\ x\ y)\\ \mathsf{uncurry} &= \lambda g.\lambda p.g\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p)\end{aligned}$$

If we can ensure that product types appear only on the left of an arrow then we can use this isomorphism to translate them away. A CPS translation ensures that all types apart from the final return type do indeed appear to the left of an arrow.

2.1 A Call-by-Name CPS translation

A call-by-name CPS translation [8] generalises a double-negation translation, where the empty type is replaced by an arbitrary return type R.

$$\mathcal{N}[\![A]\!] = (A^\star \to R) \to R$$

$$\begin{aligned}X^\star &= X\\ (A \times B)^\star &= \mathcal{N}[\![A]\!] \times \mathcal{N}[\![B]\!]\\ (A \to B)^\star &= \mathcal{N}[\![A]\!] \to \mathcal{N}[\![B]\!]\end{aligned}$$

The translation on types compositionally inserts a "double-negation" around each type constructor. Inlining the definition of $(-)^\star$ makes it apparent that product types only occur to the left of arrows (assuming we choose an R that is not a product type).

$$\begin{aligned}\mathcal{N}[\![X]\!] &= (X \to R) \to R\\ \mathcal{N}[\![A \to B]\!] &= ((\mathcal{N}[\![A]\!] \to \mathcal{N}[\![B]\!]) \to R) \to R\\ \mathcal{N}[\![A \times B]\!] &= ((\mathcal{N}[\![A]\!] \times \mathcal{N}[\![B]\!]) \to R) \to R\end{aligned}$$

The translation on term contexts is pointwise.

$$\begin{aligned}\mathcal{N}[\![\cdot]\!] &= \cdot\\ \mathcal{N}[\![\Gamma, x : A]\!] &= \mathcal{N}[\![\Gamma]\!], x : \mathcal{N}[\![A]\!]\end{aligned}$$

The translation on terms is as follows.

$$\begin{aligned}\mathcal{N}[\![x]\!] &= \lambda k.x\ k\\ \mathcal{N}[\![\lambda x.M]\!] &= \lambda k.k\ (\lambda x.\mathcal{N}[\![M]\!])\\ \mathcal{N}[\![M\ N]\!] &= \lambda k.\mathcal{N}[\![M]\!]\ (\lambda f.f\ \mathcal{N}[\![N]\!]\ k)\\ \mathcal{N}[\![\mathbf{pair}\ M\ N]\!] &= \lambda k.k\ (\mathbf{pair}\ \mathcal{N}[\![M]\!]\ \mathcal{N}[\![N]\!])\\ \mathcal{N}[\![\mathbf{fst}\ M]\!] &= \lambda k.\mathcal{N}[\![M]\!]\ (\lambda p.(\mathbf{fst}\ p)\ k)\\ \mathcal{N}[\![\mathbf{snd}\ M]\!] &= \lambda k.\mathcal{N}[\![M]\!]\ (\lambda p.(\mathbf{snd}\ p)\ k)\end{aligned}$$

Proposition 2.1. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_\beta N$ *iff* $\mathcal{N}[\![M]\!] \sim_\beta \mathcal{N}[\![N]\!]$.

The original proof of this proposition (adapted to the untyped lambda calculus) is due to Plotkin [8]. It is somewhat complicated by the fact that the translation introduces *administrative-redexes* which means that the β-rule sometimes has to be applied in reverse. Danvy and Filinski remedy this complication with a more efficient higher-order one-pass CPS translation [3].

2.2 Curried CPS Translation

As in the image of the CPS translation products appear only on the left of arrows we can replace all such instances with curried functions.

$$\begin{array}{rcl} C[\![X]\!] & = & (X \to R) \to R \\ C[\![A \to B]\!] & = & ((C[\![A]\!] \to C[\![B]\!]) \to R) \to R \\ C[\![A \times B]\!] & = & (C[\![A]\!] \to C[\![B]\!] \to R) \to R \end{array}$$

$$\begin{array}{rcl} C[\![x]\!] & = & \lambda k.x\ k \\ C[\![\lambda x.M]\!] & = & \lambda k.k\ (\lambda x.C[\![M]\!]) \\ C[\![M\ N]\!] & = & \lambda k.C[\![M]\!]\ (\lambda f.f\ C[\![N]\!]\ k) \\ C[\![\mathbf{pair}\ M\ N]\!] & = & \lambda k.k\ C[\![M]\!]\ C[\![N]\!] \\ C[\![\mathbf{fst}\ M]\!] & = & \lambda k.C[\![M]\!]\ (\lambda x.\lambda y.x\ k) \\ C[\![\mathbf{snd}\ M]\!] & = & \lambda k.C[\![M]\!]\ (\lambda x.\lambda y.y\ k) \end{array}$$

The changes to the translation are highlighted.

Proposition 2.2. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_\beta N$ *iff* $C[\![M]\!] \sim_\beta C[\![N]\!]$.

2.3 Localising CPS

The curried CBN CPS translation relies on CPS to enable currying of functions. If we want to make it local then an obvious thing to try is to only apply CPS at product types and attempt to leave everything else in direct style. The idea is to encode a pair using a term that expects a continuation, and a projection by supplying a continuation such that the value returned by the continuation is the projected value. Indeed we can define a suitable term translation as the homomorphic extension of the following equations.

$$\begin{array}{rcl} \mathcal{H}[\![\mathbf{pair}\ M\ N]\!] & = & \lambda s.s\ \mathcal{H}[\![M]\!]\ \mathcal{H}[\![N]\!] \\ \mathcal{H}[\![\mathbf{fst}\ M]\!] & = & \mathcal{H}[\![M]\!]\ (\lambda x.\lambda y.x) \\ \mathcal{H}[\![\mathbf{snd}\ M]\!] & = & \mathcal{H}[\![M]\!]\ (\lambda x.\lambda y.y) \end{array}$$

2.4 Homogeneous Products

The above local translation is only typeable in a simply-typed lambda-calculus if both components of the product have the same type. The reason is that the type of the selector function s (the counterpart of the continuation in the global CPS translation) must be fixed. Specifically, it must have type $A \to A \to A$ for some A. Consequently, we have a local encoding of $A \times A$ for every type A, but no local encoding of $A \times B$ for distinct types A and B. This local encoding $\mathcal{H}[\![-]\!]$ is defined on simply-typed lambda calculus with homogeneous products.

$$\mathcal{H}[\![A \times A]\!] = (\mathcal{H}[\![A]\!] \to \mathcal{H}[\![A]\!] \to \mathcal{H}[\![A]\!]) \to \mathcal{H}[\![A]\!]$$

$$\begin{array}{rcl} \mathcal{H}[\![\mathbf{pair}\ M^A\ N^A]\!] & = & \lambda s^{\mathcal{H}[\![A]\!] \to \mathcal{H}[\![A]\!] \to \mathcal{H}[\![A]\!]}.s\ \mathcal{H}[\![M]\!]\ \mathcal{H}[\![N]\!] \\ \mathcal{H}[\![\mathbf{fst}\ M^{A \times A}]\!] & = & \mathcal{H}[\![M]\!]\ (\lambda x^{\mathcal{H}[\![A]\!]}.\lambda y^{\mathcal{H}[\![A]\!]}.x) \\ \mathcal{H}[\![\mathbf{snd}\ M^{A \times A}]\!] & = & \mathcal{H}[\![M]\!]\ (\lambda x^{\mathcal{H}[\![A]\!]}.\lambda y^{\mathcal{H}[\![A]\!]}.y) \end{array}$$

Proposition 2.3. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_\beta N$ *iff* $\mathcal{H}[\![M]\!] \sim_\beta \mathcal{H}[\![N]\!]$.

2.5 Untyped Encoding

If we discard all types then the above local encoding degenerates to the well-known Church encoding of pairs in untyped lambda calculus [1, 2], which works for pairs of terms that would be ascribed different types by a simply-typed lambda calculus.

$$\begin{array}{rcl} \mathcal{U}[\![\mathbf{pair}\ M\ N]\!] & = & \lambda s.s\ \mathcal{U}[\![M]\!]\ \mathcal{U}[\![N]\!] \\ \mathcal{U}[\![\mathbf{fst}\ M]\!] & = & \mathcal{U}[\![M]\!]\ (\lambda x.\lambda y.x) \\ \mathcal{U}[\![\mathbf{snd}\ M]\!] & = & \mathcal{U}[\![M]\!]\ (\lambda x.\lambda y.y) \end{array}$$

Proposition 2.4. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_\beta N$ *iff* $\mathcal{U}[\![M]\!] \sim_\beta \mathcal{U}[\![N]\!]$.

2.6 Polymorphic Encoding

If we extend the target language to a polymorphic lambda-calculus, then the above translation can be adapted to use a polymorphic return type, thus supporting heterogeneous pairs.

$$\mathcal{F}[\![A \times B]\!] = \forall Z.(\mathcal{F}[\![A]\!] \to \mathcal{F}[\![B]\!] \to Z) \to Z$$

$$\begin{array}{rcl} \mathcal{F}[\![\mathbf{pair}_{A,B}\ M\ N]\!] & = & \Lambda Z.\lambda s^{\mathcal{F}[\![A]\!]\to\mathcal{F}[\![B]\!]\to Z}.s\ \mathcal{F}[\![M]\!]\ \mathcal{F}[\![N]\!] \\ \mathcal{F}[\![\mathbf{fst}_{A,B}\ M]\!] & = & \mathcal{F}[\![M]\!]\ \mathcal{F}[\![A]\!]\ (\lambda x^{\mathcal{F}[\![A]\!]}.\lambda y^{\mathcal{F}[\![B]\!]}.x) \\ \mathcal{F}[\![\mathbf{snd}_{A,B}\ M]\!] & = & \mathcal{F}[\![M]\!]\ \mathcal{F}[\![B]\!]\ (\lambda x^{\mathcal{F}[\![A]\!]}.\lambda y^{\mathcal{F}[\![B]\!]}.y) \end{array}$$

Proposition 2.5. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_\beta N$ *iff* $\mathcal{F}[\![M]\!] \sim_\beta \mathcal{F}[\![N]\!]$.

3 NON-EXISTENCE OF LOCAL ENCODINGS

In simply-typed lambda calculi, at least one base type is necessary in order for types to be well-defined. In our presentation, the base types are *abstract* in the sense that there are no typing rules that explicitly mention base types and hence no closed terms of base type. Some common variations of simply-typed lambda calculus such as Gödel's System T [4] and Plotkin's PCF [9] include a concrete base type of natural numbers and no abstract base types.

3.1 Multiple Abstract Base Types

We now prove that if there are multiple abstract base types then there can be no local encoding of products. In order to prove that no such encoding can exist in the presence of two distinct abstract base types X and Y we need only consider the encoding of projections on a variable $p : X \times Y$. Because we seek a local encoding, we must have the following translations on judgements.

$$\begin{array}{l} [\![p : X \times Y \vdash \mathbf{fst}\ p : X]\!] = p : [\![X \times Y]\!] \vdash \mathsf{fst}_{X,Y} : X \\ [\![p : X \times Y \vdash \mathbf{snd}\ p : Y]\!] = p : [\![X \times Y]\!] \vdash \mathsf{snd}_{X,Y} : Y \end{array}$$

(In contrast, the global CPS translations of Sections 2.2 and 2.1 have that $\mathcal{N}[\![X]\!] = \mathcal{C}[\![X]\!] = (X \to R) \to R$ so $\mathsf{fst}_{X,Y}$ and $\mathsf{snd}_{X,Y}$ never feature in the encoding.) Without loss of generality we can assume that $\mathsf{fst}_{X,Y}$ and $\mathsf{snd}_{X,Y}$ are in β-normal form. But a β-normal form of base type must be a variable applied to a sequence of β-normal forms. Thus we must have $m, n, M_1, ..., M_m, N_1, ..., N_n$ such that:

$$\begin{array}{rcl} \mathsf{fst}_{X,Y} & = & p\ M_1\ \ldots\ M_m \\ \mathsf{snd}_{X,Y} & = & p\ N_1\ \ldots\ N_n \end{array}$$

The typing rule for application means that we also have

$$A_1 \to \cdots \to A_m \to X \quad = \quad [\![X \times Y]\!] \quad = \quad B_1 \to \cdots \to B_n \to Y$$

where

$$\begin{array}{l} (p : [\![X \times Y]\!] \vdash M_i : A_i)_{1 \le i \le m} \\ (p : [\![X \times Y]\!] \vdash N_j : B_i)_{1 \le j \le n} j \end{array}$$

But these equations could only hold if X and Y were the same type (and of course incidentally that $m = n$, $A_1 = B_1, \ldots, A_m = B_m$). Thus no such encoding can exist.

Proposition 3.1. *There exists no local encoding $[\![-]\!]$ of products in simply-typed lambda calculus with multiple abstract base types.*

3.2 A Single Abstract Base Type

We now show that a similar argument to the one above can be used to prove that even with a single abstract base type there can be no local encoding of product types. However, as we shall see later the proof is somewhat fragile: small changes to the underlying assumptions invalidate it.

The non-existence proof of the previous subsection depends on the assumption that an encoding should be type-preserving (i.e. well-defined), without even considering soundness. The following proof depends also on an assumption that the encoding is sound with respect to β-conversion.

Consider the encoding of $X \times (X \to X)$. We require:

$$[\![p : X \times (X \to X) \vdash \mathbf{fst}\ p : X]\!] = p : [\![X \times (X \to X)]\!] \vdash \mathsf{fst}_{X,X\to X} : X$$
$$[\![p : X \times (X \to X) \vdash \mathbf{snd}\ p : X \to X]\!] = p : [\![X \times (X \to X)]\!] \vdash \mathsf{snd}_{X,X\to X} : X \to X$$

As before, $\mathsf{fst}_{X,X\to X}$ must be of the form

$$p\ M_1\ \ldots\ M_m$$

and hence:

$$[\![X \times (X \to X)]\!] = A_1 \to \cdots \to A_m \to X$$

But this time there are two choices for $\mathsf{snd}_{X,X\to X}$. Either it can be of the form

$$p\ N_1\ \ldots\ N_{m-1} \qquad (1)$$

in which case

$$A_m = X$$

or it can be of the form:

$$\lambda z.N' \qquad (2)$$

But we can immediately rule out (2) as it implies that

$$[\![\mathbf{snd}\ (\mathbf{pair}\ x\ y)]\!] = \lambda z.N'[[\![\mathbf{pair}\ x\ y]\!]/p]$$

which cannot be β-converted to y whatever the definitions of N' and pair are (meaning that $[\![-]\!]$ would be unsound with respect to β-conversion).

We can also rule out (1) as M_m must be a normal form of base type and the only free variable in scope is p, so M_m must again be the application of p to m normal forms, and the last of those must itself be p applied to m normal forms, and so on. Thus no finite snd can exist.

Proposition 3.2. *There exists no local encoding $[\![-]\!]$ of products in simply-typed lambda calculus with functions and a single abstract base type such that: if $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$ and $M \sim_\beta N$ then $[\![M]\!] \sim_\beta [\![N]\!]$.*

4 EXISTENCE OF LOCAL ENCODINGS

We now describe two distinct small changes to our underlying assumptions, each of which invalidates the proof in Section 3.2 and moreover enables a local encoding of products.

4.1 Allowing η-Conversion

The non-existence proof in Section 3.2 relies on ruling out the second form for N, which we do by observing that no lambda abstraction can ever be β-converted to a variable. If however, we relax our requirements just slightly by looking for an encoding in which we admit η-conversion in addition to β-conversion, then it turns out that we can in fact encode $X \times (X \to X)$ as follows:

$$\begin{array}{rcl} \mathcal{E}[\![X \times (X \to X)]\!] & = & (X \to (X \to X) \to X) \to X \\ \\ \mathcal{E}[\![\mathbf{pair}\ M^X\ N^{X \to X}]\!] & = & \lambda f.f\ \mathcal{E}[\![M]\!]\ \mathcal{E}[\![N]\!] \\ \mathcal{E}[\![\mathbf{fst}\ M^{X \times (X \to X)}]\!] & = & \mathcal{E}[\![M]\!]\ (\lambda x\ y.x) \\ \mathcal{E}[\![\mathbf{snd}\ M^{X \times (X \to X)}]\!] & = & \lambda z.\mathcal{E}[\![M]\!]\ (\lambda x\ y.y\ z) \end{array}$$

Now we have

$$\begin{array}{l} \mathcal{E}[\![\mathbf{fst}\ (\mathcal{E}[\![\mathbf{pair}\ x\ y]\!])]\!] \ \sim_\beta \ x \\ \mathcal{E}[\![\mathbf{snd}\ (\mathcal{E}[\![\mathbf{pair}\ x\ y]\!])]\!] \ \sim_\beta \ \lambda z.y\ z \ \sim_\eta \ y \end{array}$$

as required.

Indeed if we have a single abstract base type X and admit η-conversion then we can encode *all* product types.

$$\begin{array}{rcl} \mathcal{E}[\![A \times B]\!] & = & (\mathcal{E}[\![A]\!] \to \mathcal{E}[\![B]\!] \to X) \to X \\ \\ \mathcal{E}[\![\mathbf{pair}\ M\ N]\!] & = & \lambda f.f\ \mathcal{E}[\![M]\!]\ \mathcal{E}[\![N]\!] \\ \mathcal{E}[\![\mathbf{fst}\ M^{A_1 \to \ldots A_n \to X, B}]\!] & = & \lambda z_1\ \ldots\ z_n.\mathcal{E}[\![M]\!]\ (\lambda x\ y.x\ z_1\ \ldots\ z_n) \\ \mathcal{E}[\![\mathbf{snd}\ M^{A, B_1 \to \ldots B_n \to X}]\!] & = & \lambda z_1\ \ldots\ z_n.\mathcal{E}[\![M]\!]\ (\lambda x\ y.y\ z_1\ \ldots\ z_n) \end{array}$$

As there is a single abstract base type X, we can always obtain a term of type X from any p whatever its type by applying it to enough arguments. We can take advantage of η-conversion to ensure that arguments $(z_1, \ldots, z_n)$ of the required types are in scope.

Proposition 4.1. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_{\beta\eta^\to} N$ *iff* $\mathcal{E}[\![M]\!] \ \sim_{\beta\eta} \ \mathcal{E}[\![N]\!]$.

Soundness (left-to-right) follows by induction on typing derivations. Completeness (right-to-left) follows from soundness, existence of unique normal forms, and the observation that normal forms are encoded as normal forms.

4.2 Adding Constants

Adding a constant of type X is another way to subvert the proof that there is no encoding of $X \times (X \to X)$ in Section 3.2. If we introduce a single constant $c_X : X$ (or equivalently interpret top-level terms in a context extended with a distinguished free variable $c_X : X$), then we can encode $X \times (X \to X)$ without the need for η-conversion. The introduction of the constant removes the obstacle to N having the form (1), and we can simply set $M_m = c_X$ (with $m = 2$) as follows.

$$\begin{array}{rcl} \mathcal{L}[\![X \times (X \to X)]\!] & = & ((X \to X) \to (X \to X) \to (X \to X)) \to (X \to X) \\ \\ \mathcal{L}[\![\mathbf{pair}\ M\ N]\!] & = & \lambda f.f\ (\lambda z.\mathcal{E}[\![M]\!])\ \mathcal{E}[\![N]\!] \\ \mathcal{L}[\![\mathbf{fst}\ M^{X \times (X \to X)}]\!] & = & \mathcal{L}[\![M]\!]\ (\lambda x\ y.x)\ c_X \\ \mathcal{L}[\![\mathbf{snd}\ M^{X \times (X \to X)}]\!] & = & \mathcal{L}[\![M]\!]\ (\lambda x\ y.y) \end{array}$$

Now that X is inhabited (by the constant c_X), we can embed an X value into $X \to X$ in such a way that we can re-extract the original X value as required (by application to c_X), which means we can

simulate the heterogeneous product $X \times (X \to X)$ by a homogeneous product $(X \to X) \times (X \to X)$.

$$x : X \vdash \lambda z^X.x : X \to X$$
$$f : X \to X \vdash f\ c_X : (X \to X) \to X$$

Without c_X it would not be possible to map a function of type $X \to X$ to a term of type X.

The same idea generalises to enable us to encode all product types. The constant c_X ensures that X is inhabited, but this also means that every other type is inhabited and for any type we can define a default closed term $\mathsf{default}_A$ as follows.

$$\begin{array}{rcl} \mathsf{default}_X & = & c_X \\ \mathsf{default}_{A \to B} & = & \lambda z^A.\mathsf{default}_B \end{array}$$

The property that enables us to factor heterogeneous products through homogeneous products is that for any pair of types A and B there exists a least-upper bound $A \sqcup B$ such that we can embed both A and B into $A \sqcup B$, formally both A and B are *definable retracts* of $A \sqcup B$.

Definition 4.2. Given types A and B, we say that A is a *definable retract* of B if there exist terms $x : A \vdash N : B$ and $x : B \vdash M : A$ such that $M[N/x] \sim_{\beta\eta} x$. We say that (M, N) is a *retraction pair* for the *retraction* $A \triangleleft B$, or more concisely $(M, N) : A \triangleleft B$.

The least upper bound $A \sqcup B$ of types A and B is defined together with type-indexed open terms for injecting into and projecting out of $A \sqcup B$.

$$\begin{array}{rcl}
x : A & \vdash & \mathsf{inl}_{A,B} : A \sqcup B \\
x : B & \vdash & \mathsf{inr}_{A,B} : A \sqcup B \\
x : A \sqcup B & \vdash & \mathsf{outl}_{A,B} : A \\
x : A \sqcup B & \vdash & \mathsf{outr}_{A,B} : B \\
\\
X \sqcup X & = & X \\
x : X \vdash \mathsf{inl}_{X,X} & = & x : X \\
x : X \vdash \mathsf{inr}_{X,X} & = & x : X \\
x : X \vdash \mathsf{outl}_{X,X} & = & x : X \\
x : X \vdash \mathsf{outr}_{X,X} & = & x : X \\
\\
(A \to B) \sqcup X & = & A \to (B \sqcup X) \\
x : A \to B \vdash \mathsf{inl}_{A \to B,X} & = & \lambda y^A.\mathsf{inl}[x\ y] : A \to (B \sqcup X) \\
x : X \vdash \mathsf{inr}_{A \to B,X} & = & \lambda z^A.\mathsf{inr}[x] : A \to (B \sqcup X) \\
x : A \to (B \sqcup X) \vdash \mathsf{outl}_{A \to B,X} & = & \lambda y^A.\mathsf{outl}[x\ y] : A \to B \\
x : A \to (B \sqcup X) \vdash \mathsf{outr}_{A \to B,X} & = & \mathsf{outr}[x\ \mathsf{default}_A] : X \\
\\
X \sqcup (A \to B) & = & A \to (X \sqcup B) \\
x : X \vdash \mathsf{inl}_{X,A \to B} & = & \lambda z^A.\mathsf{inl}[x] : A \to (X \sqcup B) \\
x : A \to B \vdash \mathsf{inr}_{X,A \to B} & = & \lambda y^A.\mathsf{inr}[x\ y] : A \to (X \sqcup B) \\
x : A \to (X \sqcup B) \vdash \mathsf{outl}_{X,A \to B} & = & \mathsf{outl}[x\ \mathsf{default}_A] : X \\
x : A \to (X \sqcup B) \vdash \mathsf{outr}_{X,A \to B} & = & \lambda y^A.\mathsf{outr}[x\ y] : A \to B \\
\\
(A \to B) \sqcup (A' \to B') & = & (A \sqcup A') \to (B \sqcup B') \\
x : A \to B \vdash \mathsf{inl}_{A \to B,A' \to B'} & = & \lambda y^{A \sqcup A'}.\mathsf{inl}[x\ (\mathsf{outl}[y])] : (A \sqcup A') \to (B \sqcup B') \\
x : A' \to B' \vdash \mathsf{inr}_{A \to B,A' \to B'} & = & \lambda y^{A \sqcup A'}.\mathsf{inr}[x\ (\mathsf{outr}[y])] : (A \sqcup A') \to (B \sqcup B') \\
x : (A \sqcup A') \to (B \sqcup B') \vdash \mathsf{outl}_{A \to B,A' \to B'} & = & \lambda y^A.\mathsf{outl}[x\ (\mathsf{inl}[y])] : A \to B \\
x : (A \sqcup A') \to (B \sqcup B') \vdash \mathsf{outr}_{A \to B,A' \to B'} & = & \lambda y^B.\mathsf{outr}[x\ (\mathsf{inr}[y])] : A' \to B'
\end{array}$$

Just as with the open encoding parameters of Section 1.2 we make use of syntactic sugar for substituting for the free variables in the injection and projection terms.

$$\begin{array}{rcl} \mathsf{inl}[M] & \equiv & \mathsf{inl}[M/x] \\ \mathsf{inr}[M] & \equiv & \mathsf{inr}[M/x] \\ \mathsf{outl}[M] & \equiv & \mathsf{outl}[M/x] \\ \mathsf{outr}[M] & \equiv & \mathsf{outr}[M/x] \end{array}$$

Proposition 4.3. *For all types A and B:*

(1) $(\mathsf{outl}_{A,B}, \mathsf{inl}_{A,B}) : A \triangleleft A \sqcup B$
(2) $(\mathsf{outr}_{A,B}, \mathsf{inr}_{A,B}) : B \triangleleft A \sqcup B$

The proof is by simultaneous induction on the structure of A and B.

We can now encode products using the least-upper bound construction.

$$\mathcal{L}[\![A \times B]\!] \quad = \quad ((A \sqcup B) \to (A \sqcup B) \to (A \sqcup B)) \to (A \sqcup B)$$

$$\begin{array}{rcl} \mathcal{L}[\![\mathbf{pair}\ M^A\ N^B]\!] & = & \lambda f.f\ (\mathsf{inl}_{\mathcal{L}[\![A]\!],\mathcal{L}[\![B]\!]}[\mathcal{L}[\![M]\!]])\ (\mathsf{inr}_{\mathcal{L}[\![A]\!],\mathcal{L}[\![B]\!]}[\mathcal{L}[\![N]\!]]) \\ \mathcal{L}[\![\mathbf{fst}\ M^{A\times B}]\!] & = & \mathsf{outl}_{\mathcal{L}[\![A]\!],\mathcal{L}[\![B]\!]}[\mathcal{L}[\![M]\!]\ (\lambda x\ y.x)] \\ \mathcal{L}[\![\mathbf{snd}\ M^{A\times B}]\!] & = & \mathsf{outr}_{\mathcal{L}[\![A]\!],\mathcal{L}[\![B]\!]}[\mathcal{L}[\![M]\!]\ (\lambda x\ y.y)] \end{array}$$

Alas, there is a slight issue here. In the definition of definable retracts we allow η-conversion. This is necessary for the case that both A and B are function types in the above proposition.

Let us revisit the setup of Section 3.2, but with a slightly more complicated product. Consider a local encoding of $(X \to X) \times (X \to X) \to X$. We require:

$$\begin{array}{l} [\![p : (X \to X) \times ((X \to X) \to X) \vdash \mathbf{fst}\ p : X \to X]\!] \\ = \\ p : [\![(X \to X) \times ((X \to X) \to X)]\!] \vdash \mathsf{fst}_{X\to X,(X\to X)\to X} : X \to X \end{array}$$

$$\begin{array}{l} [\![p : (X \to X) \times ((X \to X) \to X) \vdash \mathbf{snd}\ p : (X \to X) \to X]\!] \\ = \\ p : [\![X \times (X \to X)]\!] \vdash \mathsf{snd}_{X\to X,(X\to X)\to X} : (X \to X) \to X \end{array}$$

Following similar reasoning to that of Section 3.2 we observe that without η-conversion $\mathsf{fst}_{X\to X,(X\to X)\to X}$ must be of the form

$$p\ M_1\ \ldots\ M_{m-1}$$

and $\mathsf{snd}_{X\to X,(X\to X)\to X}$ must be of the form

$$p\ N_1\ \ldots\ N_{m-1}$$

and hence:

$$\begin{array}{l} A_1 \to \cdots \to A_{m-2} \to X \to X \\ = \\ [\![(X \to X) \times ((X \to X) \to X)]\!] \\ = \\ B_1 \to \cdots \to B_{m-2} \to (X \to X) \to X \end{array}$$

But this cannot be the case as X and $X \to X$ are different types.

Proposition 4.4. *There exists no local encoding $[\![-]\!]$ of products in simply-typed lambda calculus with functions and a single base type with one constant.*

Nevertheless, we have at least constructed an alternative encoding that uses both η-conversion and constants.

Proposition 4.5. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then* $M \sim_{\beta\eta^{\rightarrow}} N$ *iff* $\mathcal{L}[\![M]\!] \sim_{\beta\eta} \mathcal{L}[\![N]\!]$.

Soundness (left-to-right) follows by induction on typing derivations using Proposition 4.3. Completeness (right-to-left) follows from soundness, existence of unique normal forms, and the observation that normal forms are encoded as normal forms.

The same idea can be further adapted to accommodate multiple base types. In order for this to work out we require not only that there is a constant c_X for each base type X, but there must exist a least upper bound $X \sqcup Y$ for every pair of base types X and Y. To construct the latter we would need some additional structure on base types. For instance if X was a type of booleans and Y a type of natural numbers then we might reasonably expect to be able to embed X in Y.

5 DISCUSSION

We proved that there are no local encodings of products in simply-typed lambda calculus if either: there are multiple abstract base types, or there is a single abstract base type and we seek an encoding that may not rely on $\rightarrow.\eta$-conversion.

Despite these non-existence proofs we have also now seen three compositional encodings of products in simply-typed lambda calculus.

- The first (Section 2.2) is the CPS encoding. This is a global encoding. It is not type-indexed, works for any number of base types, and does not rely on η-conversion or any constants.
- The second (Section 4.1) is a local encoding. It is type-indexed, depends on having exactly one base type and $\rightarrow$.η-conversion, but does not require any constants.
- The third (Section 4.2) is also a local encoding. It is type-indexed, depends on having one constant for each base type, a least upper bound for every pair of base types, and $\rightarrow.\eta$-conversion.

Localising the CPS Encoding. Though it is itself a global encoding, we have seen that several local encodings can be derived from the CPS encoding. First of all it gives rise to a local encoding of homogeneous products (Section 2.4) within simply-typed lambda calculus. Our third encoding of heterogeneous products factors through the encoding of homogeneous products. Essentially the same translation on terms also gives rise to the Church encoding of heterogeneous products in untyped lambda calculus (Section 2.5) and in System F (Section 2.6).

Church Encoding of Pairs. Church's original encoding for pairs [2] is in fact given for homogeneous pairs of natural numbers. His encoding of the pair constructor is the familiar one discussed in this paper. However, he gives more complicated encodings of projection than the familiar ones presented in this paper. His encoding of first and second projection explicitly discards the unused component of the pair, and only work for pairs of natural numbers. This is because the encoding is given in a relevant untyped lambda calculus in which every bound variable must be used.

Type-Indexing. It is worth noting that without type-indexing neither of the local encodings would work, and indeed there can be no parametric encoding of **fst** or **snd** irrespective of whether we have η-conversion or constants. The untyped encoding (Section 2.5) is local and parametric. The polymorphic encoding (Section 2.6) is local. Technically it is type-indexed, however, it is parametric in the sense that the type indexes are only used to determine type annotations, so, for instance, an implementation that used type erasure could use a uniform parametric encoding.

Related Work. Local type-indexed encodings of products (and richer data types) in PCF are well-known. For instance, Longley and Normann give a theoretical account of encodings of a range of data types in PCF in Chapter 4 of their textbook on higher-order computability [7], whilst Kiselyov [6] gives concrete implementations of encodings of a range of data types in PCF. Whereas

PCF includes a single decidedly concrete base type, namely natural numbers, we have considered simply-typed lambda calculus with abstract base types.

Conclusion. We have analysed the encoding of a rather basic feature, namely product types, in simply-typed lambda calculus. In doing so we have highlighted the fragility of such encodings and the importance of carefully stating the underlying assumptions before claiming that a particular feature (even a really simple one like product types) is expressible in a given language.

REFERENCES

[1] Hendrik Pieter Barendregt. 1985. *The Lambda Calculus - its Syntax and Semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.
[2] Alonzo Church. 1941. *The Calculi of Lambda-Conversion*.
[3] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Math. Struct. Comput. Sci.* 2, 4 (1992), 361–391.
[4] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press.
[5] C. Barry Jay and Neil Ghani. 1995. The Virtues of Eta-Expansion. *J. Funct. Program.* 5, 2 (1995), 135–154.
[6] Oleg Kiselyov. 2022. Simply-typed encodings: PCF considered as unexpectedly expressive programming language. https://okmij.org/ftp/Computation/simple-encodings.html.
[7] John Longley and Dag Normann. 2015. *Higher-Order Computability*. Springer.
[8] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159.
[9] Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255.