Compiling Linear Logic using Continuations

Jean-Philippe Bernardy, Dan Rosén, and Nicholas Smallbone
Chalmers University of Technology and University of Gothenburg

Abstract. As System F is the logical foundation of functional programming, it has long been expected that Classical Linear Logic (CLL) is the logical foundation of concurrent programming. In particular, thanks to an involutive negation, its language of propositions correspond to protocols. This means that CLL provides the principles to integrate concurrency into functional programming languages.

Aiming towards this integration, we translate the concurrency features of CLL into continuations, essentially via a negation-model of CLL into System F. Practically, the translation can be used to embed CLL programs into functional languages such as Haskell. We explain the properties of the interface between the two languages. In particular, using an example, we illustrate how the par (\(\&\)) operator can solve practical functional programming problems.

Keywords: CPS, Classical Linear Logic, Concurrency, System F

1 Introduction

System F, the polymorphic lambda calculus, is the theoretical underpinning of functional programming. It is not, however, resource-aware. Resource-aware relatives of System F are substructural; such systems are often based on intuitionistic linear logic, which gives strong control over resource use.

While ILL is popular in programming language design, many logicians prefer classical linear logic. CLL is more symmetric than ILL: every proposition \(\mathcal{A}\) has a dual \(\mathcal{A}^\perp\), the dual of \(\mathcal{A}^\perp\) is \(\mathcal{A}\) again (akin to classical negation) and every connective has a dual. This symmetry makes CLL simpler than ILL. We believe that it also makes CLL the right basis for concurrent functional programming:

- CLL is an ideal logic for describing protocols. Imagine programming a protocol between two processes. The two processes see the same protocol from opposite sides. In CLL, the two sides’ types will be dual; this means that we can seamlessly connect the two halves. Furthermore, since \(\mathcal{A}^{\perp\perp} = \mathcal{A}\), both sides have equal standing; in ILL, one side’s type would be the negation of the other, and not vice versa.
- By taking the dual of conjunction, we obtain a new connective, par (\(\&\)). This corresponds to parallel execution in programming.
- All proofs of ILL propositions can easily be embedded in CLL. Functional programming is therefore contained in CLL.
Despite these advantages, CLL has not been used much in functional programming, with the notable exception of Wadler [2012], who embeds session types into CLL. This is because, to us as functional language implementors, CLL is something of a mystery. A sequent can have many terms on the right hand side; we can move terms between left and right hand sides, which blurs the distinction between input and output; the $\forall$ connective is completely new. We do not even have a function type: $A \rightarrow B$ is shorthand for $A \perp \forall B$, which is both classical and makes use of the strange $\forall$ connective. None of this is familiar from System F. In short, ILL is a lambda calculus; CLL is not.

Our aim is to make CLL a foundation for concurrent functional programming languages. Towards this goal we give a model of CLL$_2$, second-order classical linear logic, in System F$_2$. Our model uses continuations to encode the concurrent aspects of CLL$_2$, and is effectively a compilation scheme from CLL$_2$ to System F$_2$. The model makes CLL$_2$’s power available to languages based on System F$_2$; in particular, we can embed CLL$_2$ programs in functional languages such as ML or Haskell. It also serves to explain CLL$_2$’s more exotic features, like $\forall$, to someone familiar with System F$_2$. Thus we hope to connect CLL to ordinary functional programming and to demystify it for the programming language designer. Our contributions are as follows:

- We assign a new term representation to CLL$_2$ (Sec. 2), which we call Concurrent Programming Linear Language (CPLL).
- We give a new model for CLL$_2$ by translating it to System F$_2$, and show this model sound. (Sec. 3)
- We show that, if the translation is restricted to the linear fragment of System F$_2$, the model is also complete (Thm. 3).
- We use the model to explain the concurrent features of CLL$_2$ by relating them to System F$_2$ (Sec. 4, Sec. 5).
- We show how to extend the model with the more exotic control primitives Mix and BiCut (Sec. 6).

1.1 Motivation and running example

To motivate CLL$_2$ as a foundation of concurrency, we take as a running example a supply of unique numbers. Such a supply might be used, for example, to generate fresh variable names in a compiler. In a functional setting, we must explicitly thread the supply through the program. The typical interface to a unique supply is a function next : $S \rightarrow S \times \text{Int}$, which produces a number Int from the current state of the supply $S$, and returns a new state.

This interface can easily be used incorrectly: the old state remains accessible, and if it is accidentally reused, two identical streams of numbers will be generated. This problem can be prevented if we give next the linear type $S \rightarrow S \otimes \text{Int}$. This type enforces that the input supply cannot be reused: arguments of a linear arrow ($\rightarrow$) are “consumed” by the act of calling the function.

Programming solely with the next combinator can be cumbersome, as we must completely sequentialize the part of the program that uses the supply,
for example by using a unique supply monad. To ease this pain, many unique supplies offer a `split` combinator, with type \( S \rightarrow S \times S \); one example is the Glasgow Haskell Compiler (GHC). The trouble with `split` is that it requires a crafty implementation: the states of the sub-supplies must somehow interact with each other, even though the type says they do not. (GHC’s implementation of `split` uses global variables in embedded C code, and is introduced with the comment “here comes THE MAGIC” [The GHC Team, 2013].) By using CLL’s `\$` operator, we can give a more precise type to `split`, and keep the implementation simple, as we will see in the next section. Concretely, we have `split : S \rightarrow S \, \$ \, S`, which expresses that the two supplies must be used independently.

The application to unique supplies is only a simple example of the advantages of CLL over ILL. The same idea can be applied to any resource which must be used by independent parts of the program, such as random number generators, preemptible file handles, etc.

## 2 Classical Linear Logic as a Programming Language

We now introduce CPLL, our programming language based on CLL, and show how to use it to implement the `next` and `split` combinators from the previous section. While many languages based on linear logic are logic programming languages, in that the operational semantics of the program are given by a proof search, our language is a functional language, in the sense that the operational semantics corresponds to reduction to normal form.

In CPLL, as in CLL, every type \( A \) has a dual \( A^\perp \), and producing a value of type \( A \) is equivalent to receiving an argument of type \( A^\perp \). Therefore, there is no clear distinction between inputs and outputs, and we say that having an argument \( x : A \) corresponds to communicating over channel \( x \) using protocol \( A \).

Likewise, we need not distinguish arguments from results, and CPLL programs have only arguments, not results. Concretely, the typing judgement takes the form \( \Gamma \vdash \cdot \), meaning that the program consumes every element of \( \Gamma \) and terminates. \(^1\) For instance, to define `next`, instead of producing a value of type \( S \rightarrow S \otimes \text{Int} \), we should consume its dual, \( S \otimes (S^\perp \, \&\, \text{Int}^\perp) \); that is, consume an \( S \) and produce an \( S \) and an \( \text{Int} \).

We might pick \( S = \text{Int} \); this makes it easy to implement `next` but, as we remarked above, makes it very hard to implement `split`. We would have to divide the supply of numbers in two every time and, if our numbers were 32-bit, after 32 nested `splits` we would have run out of numbers.

CLL gives us the tool to solve the problem. Choose \( S = \text{Int} \otimes \text{Int}^\perp \): we receive an \( \text{Int} \) telling us the next number to use, as before, but must also use the \( \text{Int}^\perp \) to communicate the smallest unused number once we have used the supply. We implement `split` by threading the final state of the first supply to the initial state of the second one. As before, our program must consume the dual of \( S \rightarrow S \otimes S \),

\(^1\) In the literature, right-handed presentations of CLL are more common, where the judgement \( \vdash \Gamma \) means to produce every element of \( \Gamma \) concurrently. We choose left-handed sequents as they are more similar to System F.
split's type, which expands to $Int \otimes Int^\perp \otimes (Int^\perp \nabla Int) \otimes (Int^\perp \nabla Int)$. That is, we are given an input state, and channels that consume the output state and the two sub-supplies we will produce, and have to connect them up. We can implement split as follows.

\[
\begin{align*}
&\text{split} : Int \otimes Int^\perp \otimes (Int^\perp \nabla Int) \otimes (Int^\perp \nabla Int) \vdash \\
&\quad \text{let } supply, subsupplies = \text{split} \\
&\quad \text{let } input, output = supply \\
&\quad \text{let } subsupply_1, subsupply_2 = subsupplies \\
&\quad \text{connect subsupply}_1 \text{ to} \\
&\quad \quad \text{input}_1 \leftrightarrow \text{input}_1 \\
&\quad \quad \text{output}_1 \leftrightarrow \text{connect subsupply}_2 \text{ to} \\
&\quad \quad \quad \text{input}_2 \leftrightarrow \text{output}_1 \leftrightarrow \text{input}_2 \\
&\quad \quad \quad \text{output}_2 \leftrightarrow \text{output}_2 \leftrightarrow \text{output}
\end{align*}
\]

This program uses three features of CPLL: let, connect and $\leftrightarrow$. First, let splits a pair into its components. Second, connect pattern matches on a value of type $A \nabla B$: we give two branches, the first binds a variable of type $A$, the second a variable of type $B$, and each variable in the context must be used in exactly one branch. We might think of the $\nabla$ as a pair and the branches as running concurrently. Finally, $\leftrightarrow$ takes a type and its dual and connects them: it is the basic way of terminating in CPLL. All the typing rules are found in Fig. 1.

In the program above, we first use let to split the context into its four parts. We then connect the first sub-supply's input and output: the input to the input state in our context, the output to the input of the second sub-supply. Finally, we make the output state be the output of the second sub-supply.

This definition of split is awkward, but we only intend CPLL to be the core of a real language based on CLL. In reality, instead of $\nabla$ and connect, we would use $\rightarrow$ and function application, like a normal functional language. Instead of $\leftrightarrow$, we would simply return a result.

We now describe CPLL more formally. Types fall into two categories, positive and negative. We use the variables $P$ and $Q$ to range over positive types and $N$ and $M$ for negative types, while $A$ and $B$ stand for any type, regardless of polarity. We have the following types:

\[
\begin{align*}
P, Q & ::= 0 \mid 1 \mid A \oplus B \mid A \otimes B \mid !A \mid \forall \alpha.A[\alpha] \mid \alpha \\
N, M & ::= \top \mid \bot \mid A \& B \mid A \nabla B \mid ?A \mid \exists \alpha.A[\alpha] \mid \alpha^\perp
\end{align*}
\]

As is customary, we write $\alpha^\perp$ for the dual of a type variable $\alpha$, and the function dualising a type $A$ to its linear negation $A^\perp$ is defined by induction on the structure of the type.

The full syntax of the language that we support is shown in Fig. 1. The typing rules are a direct adaptation of the rules of CLL to left-handed sequents, except that we also maintain an intuitionistic context for exponentials. The derivations have the form $\Theta; \Gamma \vdash s$, where the intuitionistic context, $\Theta$, and the linear context, $\Gamma$, are separated by a semicolon which may be omitted if $\Theta$ is empty, and $s$ is a program in our syntax.
\[\Theta; x: P, y: P^+ \vdash x \leftrightarrow y \text{ \(\Delta\)}\]

\[\Theta; \Gamma, x: P \vdash a \quad \Theta; y: P^-, \Delta \vdash b \quad \Theta; \Gamma, \Delta \vdash \text{cut} \{x: P \rightarrow a; y: P^\perp \rightarrow b\}\]

\[\Theta; \Gamma, x: \bot \vdash x \text{ \(\Delta\)}\]

\[\Theta; \Gamma, x: A, y: B \vdash a \quad \Theta; \Gamma, z: A \otimes B \vdash \text{let } x, y = z; a \text{ \(\Delta\)}\]

\[\Theta; \Gamma, x: A \vdash a \quad \Theta; \Gamma, y: B \vdash b \quad \Theta; \Gamma, z: A \otimes B \vdash \text{case } z \text{ of } [\text{inl } x \mapsto a; \text{inr } y \mapsto b] \text{ \(\Delta\)}\]

\[\Theta; \Gamma, x: A \oplus B \vdash \text{let } x = \text{snd } z; a \text{ \(\Delta\)}\]

\[\beta \text{ fresh, } \Theta; \Gamma, z: A[\beta] \vdash a \text{ \(\Delta\)}\]

\[\Theta; \Gamma, z: \exists \alpha.A[\alpha] \vdash \text{let } \beta, x = z; a \text{ \(\Delta\)}\]

\[\Theta; z: A; \Gamma, x: A \vdash a \text{ \(\Delta\)}\]

\[\Theta; z: A; \Gamma \vdash \text{let } x = \text{load } z; a \text{ \(\Delta\)}\]

\[\Theta; x: A \vdash a \quad \Theta; x: A \oplus B \vdash \text{let } x = \text{save } z; a \text{ \(\Delta\)}\]

\[\Theta; x: A \vdash a \quad \Theta; x: A[\alpha] \vdash a \quad \Theta; x: A \vdash a \quad \Theta; x: A \vdash \text{offer } z; a \text{ \(\Delta\)}\]

\[\Theta; \Gamma, x: B \vdash a \quad \Theta; \Gamma, z: A\&B \vdash \text{let } x = \text{snd } z; a \text{ \(\Delta\)}\]

\[\Theta; \Gamma, z: \forall \alpha.A[\alpha] \vdash \text{let } x = x; B; a \quad \Theta; \Gamma, z: A[B] \vdash a \quad \Theta; \Gamma, z: \forall \alpha.A[\alpha] \vdash \text{let } x = x; B; a \text{ \(\Delta\)}\]

\[\Theta; \Gamma, z: A \& B \vdash \text{let } x = \text{save } z; a \quad \text{SAVE}\]

\[\Theta; \Gamma, z: A \oplus B \vdash \text{let } x = \text{load } z; a \quad \text{LOAD}\]

\[\Theta; \Gamma, x: A \vdash a \quad \Theta; x: A \vdash a \quad \Theta; x: A \oplus B \vdash \text{let } x = \text{offer } z; a \quad \text{OFFER}\]

**Fig. 1.** Typing rules of CPLL. \(A[\alpha]\) in the conclusion of a rule stands for a type with potential free occurrences of \(\alpha\), while \(A[B]\) in the premise stands for the same type where \(B\) is substituted for \(\alpha\).

### 3 CPS translation

In this section we give the translation from CPLL to the polymorphic lambda calculus, as an embedding from CLL\(_2\) into F\(_2\). The version of F\(_2\) we use has pairs, sum and units, rather than using impredicative encodings. This choice is motivated by our ultimate goal of executing the code, and also eases the translation. Our variant of F\(_2\) has the following type language (the complete typing rules can be found in the appendix):

\[A, B ::= A \times B \mid A + B \mid A \rightarrow B \mid 1 \mid 0 \mid \forall \alpha.A[\alpha] \mid \alpha\]

**Definition 1.** **Translation of types** Assume an abstract type \(r\). We define the postfix operator \(\ast\) from CLL\(_2\) types to F\(_2\) types as follows.

\[(A \otimes B)^\ast = A^\ast \times B^\ast\]

\[(A \oplus B)^\ast = A^\ast + B^\ast\]

\[1^\ast = 1\]

\[0^\ast = 0\]

\[(!A)^\ast = A^\ast\]

\[\alpha^\ast = \alpha\]

\[N^\ast = N^\ast \rightarrow r\]

\[(\forall \alpha.A[\alpha])^\ast = \forall \alpha.((A[\alpha]^\ast + A[\alpha]^\ast \rightarrow r)\]

Positive types are recursively translated using their corresponding F\(_2\) type (with the exception of quantifiers, which we discuss in Sec. 3.2). To translate a negative
type, we translate its dual (which is positive) and then “negate” the result by having it be the argument to a continuation. Some examples:

\[(A \land B)^* = A^{\perp} \times B^{\perp} \rightarrow r, \quad \land^* = 1 \rightarrow r, \quad (?A)^* = A^{\perp} \rightarrow r, \quad \alpha^{\perp} = \alpha \rightarrow r\]

We lift the type translation to programs. Given a CPLL program of type \(\Theta; \Gamma \vdash r\), its translation into \(F_2\) will have type \(\Theta^*, \Gamma^* \vdash r\).

The translation of proofs relies heavily on the following lemma.

**Lemma 1. Conformance of duals** Suppose that \(A\) is a type. If \(A\) is positive, then \(A^{\perp*} = A^* \rightarrow r\). If \(A\) is negative, then \(A^* = A^{\perp*} \rightarrow r\).

**Proof.** By case analysis.

Now we can state this section’s main theorem:

**Theorem 1. Soundness** If \(\Theta; \Gamma \vdash r\) holds in \(CLL_2\) then \(\Theta^*, \Gamma^* \vdash r\) holds in \(F_2\), for every \(r\).

**Proof.** By induction on the derivation.

In the rest of the section we describe this induction proof in more detail. It is constructive: it can be used as a translation procedure, and we will refer to it as such. We use the notation \((\Theta; \Gamma \vdash s)^*\) for the translation of a \(CLL_2\) derivation \(s\), and its result is \(\Theta^*, \Gamma^* \vdash s^*: r\), where \(s^*\) is a proof in \(F_2\). The postfix dot operator is thus overloaded on types, contexts, and derivations. The translation crucially depends on types: it needs to work on a fully typed derivation tree. Thus a lone \(s^*\) stands for a proof of the sequent \((\Theta; \Gamma \vdash s)^*\).

The conformance of duals, Lem. 1, is vital for both the \textsc{Ax} and the \textsc{Cut} rule. We translate \textsc{Ax} into function application:

\[
(\Theta; x : P, y : P^{\perp} \vdash x \leftrightarrow y)^* = \frac{\Theta^*, x : P^{\perp}, y : P^* \vdash r \quad init}{\Theta^*, x : P^{\perp}, y : P^* \vdash r \quad \rightarrow-elim}
\]

Here \(P\) stands for a positive type. If \(x\)’s type is negative, we flip the application to get \(x\ y\). Notice that \(x\)’s polarity determines the evaluation order: this is a repeated theme in our translation, and arises because polarity is what distinguishes an input from an output in \(CLL\).

Cut elimination corresponds to evaluation, and therefore the \textsc{Cut} rule creates a \(\beta\)-redex. Again, we assume that \(x\)’s type is positive; if not, the application is flipped. The notation \(a^*\) and \(b^*\) stand for expressions from the induction hypothesis (IH) applied to the proofs \(a\) and \(b\).

\[
\left(\Theta; \Gamma, x : P \vdash a, \Theta; y : P^{\perp}, \Delta \vdash b\right)_{\text{cut}} = \frac{\Theta^*, x : P^{\perp}, y : P^* \vdash \beta; r^{\text{IH}}}{\Theta^*, x : P^{\perp}, y : P^* \vdash \beta; r^{\text{IH}} \quad \rightarrow-\text{intro}}\frac{\Theta^*, x : P^{\perp}, \Delta \vdash \lambda x. a^*; r^{\text{IH}}}{\Theta^*, x : P^{\perp}, \Delta \vdash \lambda x. a^*; r^{\text{IH}} \quad \rightarrow-\text{intro}}\frac{\Theta^*, x : P^{\perp}, \Delta \vdash (\lambda y. b^*) (\lambda x. a^*); r}{\Theta^*, x : P^{\perp}, \Delta \vdash (\lambda y. b^*) (\lambda x. a^*); r \quad \rightarrow-\text{elim}}
\]
(We here present it without weakening the context, but fully formally \( I^* \) should be removed from the left side's context and \( \Delta^* \) from the right for the induction hypothesis to properly apply.)

From now on we give only the translated term and not the entire typing derivation, which can be found in the appendix. For the \( \otimes \) rule the pair is unpacked:

\[
\left( \Theta; \Gamma, x : A, y : B \vdash a \quad \Theta; \Gamma, z : A \otimes B \vdash \text{let } x, y = z : a \right)^* = \\
\Theta^*, \Gamma^*, z : A^* \times B^* \vdash (\text{let } ((x : A^*), (y : B^*)) = z \text{ in } a^* : r
\]

The translation of \( A \otimes B \) depends on the polarity of both \( A \) and \( B \). The simplest case is when both are positive, and the translation is as follows.

\[
\left( \Theta; \Gamma, x : P \vdash a \quad \Theta; \gamma y : Q, \Delta \vdash b \\
\Theta; \Gamma, z : P \otimes Q, \Delta \vdash \text{connect } z \text{ to } \{x \mapsto a; y \mapsto b\} \right)^{\otimes} = \\
\Theta^*, \Gamma^*, z : (P^* \to r) \times (Q^* \to r) \to r, \Delta^* \vdash z ((\lambda (x : P^*). a^*), (\lambda (y : Q^*). b^*)) : r
\]

In the above, \( a^* \) and \( b^* \) are passed as continuations to \( z \), which drives the order of execution. We see the parallel nature of \( \otimes \) in the fact that \( z \) is given two continuations and can call both of them. Next we examine the case when exactly one of the operands of the \( \otimes \) type is negative. We show the case for a negative left operand, the other case being symmetric.

\[
\left( \Theta; \Gamma, x : N \vdash a \quad \Theta; \gamma y : P, \Delta \vdash b \\
\Theta; \Gamma, z : N \otimes P, \Delta \vdash \text{connect } z \text{ to } \{x \mapsto a; y \mapsto b\} \right)^{\otimes} = \\
\Theta^*, \Gamma^*, z : N^\perp \times (P^* \to r) \to r, \Delta^* \vdash \\
(\text{let } (x : N^\perp \to r) = \lambda (x' : N^\perp^*). z (x', (\lambda (y : P^*). b^*)) \text{ in } a^*) : r
\]

In this case, \( a^* \) is in control of the execution order. It will eventually call the continuation involving \( z \), which in turn will call \( b^* \). There is no parallelism here: the continuations are called in sequence.

When both operands are negative, we get:

\[
\left( \Theta; \Gamma, x : N \vdash a \quad \Theta; \gamma y : M, \Delta \vdash b \\
\Theta; \Gamma, z : N \otimes M, \Delta \vdash \text{connect } z \text{ to } \{x \mapsto a; y \mapsto b\} \right)^{\otimes} = \\
\Theta^*, \Gamma^*, z : N^\perp \times M^\perp \to r, \Delta^* \vdash \\
(\text{let } (x : N^\perp \to r) = \lambda (x' : N^\perp^*). z (x', y') \text{ in } b^* \text{ in } a^*) : r
\]

It may be surprising that, by merely changing the polarity of the types, we get such different terms, each with a different threading of execution. This behaviour is desirable: \( P \otimes Q \) is a parallel construct, while \( P^\perp \otimes Q \) is simply \( P \to Q \), for which we do not want to introduce parallelism. Furthermore, because the original
calculus has no side-effects, the order of execution is not observable. In Sec. 6 we introduce side-effects, and revisit the issue.

The polarity of the operands does not matter in the translation of ⊕:

\[(\Theta; \Gamma, x : A \vdash a \quad \Theta; \Gamma, y : B \vdash b) \oplus = \Theta^*; \Gamma^*, z : A^* + B^* \vdash \text{case } z \text{ of } \{ \text{inl } x \mapsto a^*; \text{inr } y \mapsto b^* \} : r\]

For the & rule we must do case analysis on the polarity of the chosen operand. We only the rules where the left operand are chosen; making the opposite choice yields a symmetric translation.

\[(\Theta; \Gamma, x : P \vdash a) = \Theta^*; \Gamma^*, z : ((P^* \rightarrow r) + (B^* \rightarrow r)) \rightarrow r \vdash z (\lambda (x : P^*). a^*)) : r\]

\[(\Theta; \Gamma, x : N \vdash a) = \Theta^*; \Gamma^*, z : (N^\perp + (B^* \rightarrow r)) \rightarrow r \vdash (\lambda (x : N^\perp). z (\lambda x'. a^*) \text{ in } a^*) : r\]

The translation of the units is straightforward and omitted.

### 3.1 Exponentials

Because we use an intuitionistic context for exponentials, saving and loading turn into nothing. Translating the OFFER rule requires discrimination on the exponentiated type’s polarity. If it is positive, the continuation is called, otherwise the variable is bound with a let:

\[(\Theta; x : P \vdash a) = \Theta^*; z : (P^* \rightarrow r) \rightarrow r \vdash z (\lambda (x : P^*). a^*) : r\]

\[(\Theta; x : N \vdash a) = \Theta^*; z : N_{\perp} \rightarrow r \vdash (\lambda (x : N_{\perp}. z \text{ in } a^*)) : r\]

### 3.2 Quantifiers

Though the translation of quantifiers may sound simple, it has in fact proved the trickiest part. In this section we describe our exploration of the design space, showing that some obvious approaches are dead ends. The crux is that type variables, which themselves have a polarity, may be instantiated by negative or
positive types, and instantiating a negative type variable with a negative type without introducing a double negation is hard.

Let us start with a naive translation: \((\forall \alpha. A[\alpha])^* = \forall \alpha. A[\alpha]^*\). Now consider the type \(\forall \alpha. \alpha \otimes \alpha^+\). Its translation would be \(\forall \alpha. \alpha \times (\alpha \rightarrow r)\). What happens when we instantiate the CLL2 type with \(\bot\)? We get \(\bot \otimes 1\), whose translation is \((1 \rightarrow r) \times 1\), since \(\bot^\ast = 1 \rightarrow r\). But if we instead instantiate the F2 type, we get \((1 \rightarrow r) \times ((1 \rightarrow r) \rightarrow r)\). We cannot easily remove this double negation because there is no intuitionistic equivalence between \(A\) and \(((A \rightarrow r) \rightarrow r)\). Trying to instantiate negative types with their dual is also fruitless.

Instead, we make two copies of the type, one where the quantified variable is positive and one where it is negative. This is possible due to the classical linear isomorphism between \(\forall \alpha. A[\alpha]\) and \(\forall \alpha. A[\alpha^+]\).

A first attempt is to translate \(\forall \alpha. A[\alpha]\) to \(\forall \alpha. A[\alpha]^* \times A[\alpha^+]^*\), and necessarily \(\exists \alpha. A[\alpha]\) to \((\forall \alpha. A[\alpha]^* \times A[\alpha^+]^*) \rightarrow r\). Then a problem appears in the translation of the \(\exists\) rule, when \(A\) is negative. Let us see what happens.

\[
\Theta^*, I^*, z : (\forall \alpha. A[\alpha]^* \times A[\alpha^+]^*) \rightarrow r \vdash z (\Lambda \beta. ?)
\]

To construct an \(x\) with a valid type we need to access the \(\beta\) type variable, which can only be bound by a type-abstraction applied to \(z\). It then remains to supply the body of the abstraction, indicated with a question mark above. The type of the question mark is \(A[\beta]^* \times A[\beta^+]^*\). To use \(a^*\) we must provide \(x\) with type \(A[\beta]^* \rightarrow r\). This requires removing a double negation, which is not possible.

Hence, we add a level of explicit negation, by dualising the inner type, obtaining \((\forall \alpha. A[\alpha])^* = \forall \alpha. (A[\alpha]^* \rightarrow r) \times (A[\alpha^+]^* \rightarrow r)\). By a standard isomorphism, the type simplifies to the following final version:

\[
(\forall \alpha. A[\alpha])^* = \forall \alpha. ((A[\alpha]^* + A[\alpha^+]^*) \rightarrow r)
\]

\[
(\exists \alpha. A[\alpha])^* = (\forall \alpha. ((A[\alpha]^* + A[\alpha^+]^*)) \rightarrow r)) \rightarrow r
\]

Now, when eliminating existentials, the sum type reflects the intended behaviour that only one of the components should be used. The isomorphism between \(\forall \alpha. A[\alpha]\) and \(\forall \alpha. A[\alpha^+]\) is used to change the abstract type variable to its linear negation in the right branch of the case, using the notation \(A[\alpha^+/\alpha]\):

\[
\left(\begin{array}{c}
\beta \text{ fresh}, \Theta; \Gamma, x : A[\beta] \vdash a \\
\Theta; \Gamma, z : \exists \alpha. A[\alpha] \vdash \text{let } \beta, x = z ; a
\end{array}\right)^* = \Theta^*, I^*, z : (\forall \alpha. ((A[\alpha]^* + A[\alpha^+]^*)) \rightarrow r)) \rightarrow r \\
\vdash z (\Lambda \beta. \lambda (x') : A[\beta]^* + A[\beta^+]^*). \text{ case } x' \text{ of } \{ \text{inl } (x : A[\beta]^*) \mapsto a^* ; \text{ inr } (x : A[\beta^+]^*) \mapsto a[\beta^+/\beta]^* \} : r
\]

When eliminating a universal quantifier, say of \(\forall \alpha. A[\alpha]\), into \(A[B]\), both the polarity of \(A[\alpha]\) and the polarity of of \(B\) matter, giving us four different cases,
ensuring that in each case the type variable is instantiated with a positive type. We show two of the cases here; the other two are similar.

1. Both are positive: we instantiate the type and invoke the left continuation.

\[
\Theta, \Gamma, x : P(Q) \vdash a
\]
\[
\Theta, \Gamma, z : \forall \alpha. P[\alpha] \vdash \text{let } x = z \cdot Q; a
\]
\[
\Theta^*, \Gamma^*, z : \forall \alpha.((P[\alpha] \rightarrow r) + (P[\alpha^\perp] \rightarrow r)) \rightarrow r)
\]
\[
z @ Q^* (\text{inl} (\lambda (x : P(Q)^* \cdot a^*)) : r)
\]

2. Both are negative: we bind the right continuation using a let (this is essentially (A + B \rightarrow C) \rightarrow B \rightarrow C).

\[
\Theta, \Gamma, x : N[M] \vdash a
\]
\[
\Theta, \Gamma, z : \forall \alpha. N[\alpha] \vdash \text{let } x = z \cdot M; a
\]
\[
\Theta^*, \Gamma^*, z : \forall \alpha.((N[\alpha]^\perp \perp + N[\alpha^\perp]^\perp) \rightarrow r)
\]
\[
(\text{let } (x : N[M]^\perp \rightarrow r) = (\lambda x' : N[M]^\perp \rightarrow r) \cdot z @ M^\perp \cdot (\text{inr} x' \text{ in } a^*) : r)
\]

Because the translation of quantifiers creates a type whose size is double the size of its argument, the translation of heavily polymorphic types is exponential. This problem is difficult to avoid because of the expressiveness of CLL: in particular, because whether a type variable is instantiated with a positive or a negative type can affect the evaluation order.

## 4 Calling translated programs

The proof presented in the previous section is constructive: it gives a translation from programs in CPLL to the polymorphic lambda calculus. In this section we show how to call such translated programs from a functional language. In particular, we discuss how to call CPLL programs returning a \( \mathcal{Y} \)-type, using an example. This example hopefully also helps to explain the meaning of \( \mathcal{Y} \).

As a first example, suppose we have an atomic type \( \text{Int} \) translated to \( \text{Int} \), a context \( \Gamma \) translated to \( \Gamma(r) \), and a CPLL program of type \( \Gamma \rightarrow \text{Int} \), that is, a proof of the sequent \( \Gamma \otimes \text{Int}^\perp \). The translation gives us a function \( f \) of type \( (\Gamma \times (\text{Int} \rightarrow r)) \rightarrow r \). We should rather have a function of type \( \Gamma \rightarrow \text{Int} \); by instantiating \( r \) with \( \text{Int} \), and the continuation with the identity function, we get exactly this. Thus calling a program that does not have \( \mathcal{Y} \) in its result is simple.

Now imagine that our CPLL program has the type \( \Gamma \rightarrow \text{Int} \mathcal{Y} \text{Int} \), that is, it proves \( \Gamma \otimes \text{Int}^\perp \otimes \text{Int}^\perp \vdash \), and our function \( f \) has type \( (\Gamma(r) \times (\text{Int} \rightarrow r) \times (\text{Int} \rightarrow r)) \rightarrow r \). There are now two continuations, and by linearity our function must invoke both of them. Thus \( \Gamma(r) \) must contain some way of combining two \( r \)s into one: for example a function of type \( r \times r \rightarrow r \). To access the two results, we can let \( r \) be a list of \( \text{Int} \), our combination function be append, and each continuation return a singleton list: \( f \gamma (\lambda x \rightarrow [x]) (\lambda x \rightarrow [x]) \). We revisit the issue of getting several results in Sec. 6.
In essence, the type \( r \), which is abstract from the point of view of the CPLL program, can be seen as a type of side effects, which can be freely chosen on the lambda calculus side. The concrete effects provided by the functional program can be run in any order: the order in which we call the subcontinuations depends on the details of the CPLL program. In our example, this means that the numbers in the resulting list might be produced in any order.

In general, for any value of type \( A \Rightarrow B \), both components must be independent of the other, so no ordering of side-effects is guaranteed. Therefore, in our example, the continuations should attach a tag to each number to recover the identity of each result.

Even though the order is unknown, it is guaranteed that each continuation is called exactly once. Indeed, the translation preserves the linearity properties of the original program. To make this result formal, we show that the target of the translation remains valid in a linear version of \( \text{F}_2 \), namely intuitionistic linear logic (ILL\(_2\)). Its typing rules can be found in the appendix.

To target ILL we modify the type translation to use linear connectives, i.e. using \( \otimes \) instead of \( \times \), and \( \oplus \) instead of \( + \). We need to account for exponentials but, perhaps surprisingly, simply translating \( ! \) into \( ! \) causes a problem in the translation of the Offer rule. An extra layer of negation is needed. One simple remedy is to introduce double negations:

\[
(!A)^* = !\neg\neg A^* \quad (\?A)^* = !\neg\neg A^\perp \Rightarrow r
\]

(The notation \( \neg\neg A \) stands for the double negation \( (A \rightarrow r) \rightarrow r \). This notation can also be used on whole contexts, meaning that each element of the context is doubly negated.) Furthermore, the Save rule fails with this translation of exponentials since it needs to remove a double negation. To resolve this we doubly negate the intuitionistic context.

We can now state the preservation of linearity precisely:

**Theorem 2.** Linear soundness If \( \Theta; \Gamma \vdash \) in \( \text{CLL}_2 \) then \( \neg\neg\Theta^*; \Gamma^* \vdash r \) holds for every type \( r \) in ILL\(_2\).

*Proof.* Same as before, except no weakenings are necessary, and the above changes to the exponential fragment. The full proof is in the appendix.

As a bonus, the linear version of the model is complete. (The model is not complete in \( \text{F}_2 \), because the type may not respect linearity. For example the empty type \( \alpha \otimes \alpha \rightarrow \alpha \) translates to \( (\alpha \times \alpha \times (\alpha \rightarrow r)) \rightarrow r \), which is inhabited.)

**Theorem 3.** Completeness If, for every \( r \), \( \neg\neg\Theta^*; \Gamma^* \vdash r \) holds in ILL\(_2\), then \( \Theta ; \Gamma \vdash \) holds in \( \text{CLL}_2 \).

*Proof.* By the straightforward embedding of ILL\(_2\) into \( \text{CLL}_2 \) we get the \( \text{CLL}_2 \) sequent \( \neg\neg\Theta^*; \Gamma^* , r^\perp \vdash \). Let \( r \) be \( \perp \). The result then follows from the classical equivalence \( A^* \leadsto r \rightarrow A \), which can be proved by induction on the size of \( A \), and the classical equivalence \( \neg\neg B \rightarrow B \) (for the intuitionistic context).
5 Operational Semantics

The operational semantics of CPLL is given by importing the cut-elimination procedure from CLL. The reduction of two elimination rules connected by a cut on the variables being eliminated corresponds to communication between processes, and we call them ready cuts. For example, a cut between the & and ⊕ rules reduces into a simpler cut which corresponds operationally to sending one bit of information and continuing with the appropriate branch. The reduction rules for CPLL can be found in the appendix.

Reduction steps also include structural rules and commuting conversions. Structural rules eliminate cuts where one of the branches is the Ax rule, and rebalance nested cuts. A commuting conversion is necessary when the rule after a cut does not involve the variable introduced by the cut. This rule is then moved to before the cut.

As we saw in Sec. 3, cuts are translated to redexes in F₂. One might expect that the translation transports these operational behaviour from CLL₂ to F₂, formally expressed as follows.

**Proposition 1.** Operational equivalence for any CPLL programs $a$ and $b$, if a step of a cut-elimination procedure reduces $a$ to $b$, then $a^*$ is equal to $b^*$.

This proposition holds if the reduction step is a ready cut or a structural cut, considering $\alpha$-, $\beta$-, $\eta$-, and $\iota$-reductions. The proof for this is in the appendix.

However, the proposition fails if the reduction step in CLL₂ is a commuting conversion step, because these conversions change the order of side effects in the CLL₂ proof. This is harmless, as there is no side-effectful operation in CLL₂. However, the translation reifies the ordering of side-effects, and F₂ is sensitive to this ordering, making the proposition false.

6 Extensions: Halt, Mix and BiCut

Even though CPLL can be understood as a concurrent programming language, the constructions that we have seen so far contain just enough information to be able to run CPLL programs purely sequentially, without ever having to combine the concurrent effects of various sub-computations. However, CLL can be made truly concurrent with the Mix and BiCut rules.
Mix and Halt. The Mix rule has been proposed by Girard [1987], and is a sound extension of CLL. It can be thought of as a Cut with no communication channel between the sub-processes. The Mix rule allows us to implement $A \otimes B \rightarrow A \triangleright B$, as well as $\perp \rightarrow 1$.

It is possible to interpret Mix in our framework, but then we must assume that $r$ is equipped with a binary operator $\oplus$, corresponding to the composition of effects. Given its concurrent interpretation, Mix is generally assumed to be commutative and associative. Consequently, the operator $\oplus$ must enjoy the same properties for the model to be consistent with this interpretation. Concretely, this ensures that the result of calling a program is not affected by scheduling.

$$
(\Theta; \Gamma \vdash a \quad \Theta; \Delta \vdash b \\
\Theta; \Gamma, \Delta \vdash \text{mix}\{a; b\})^\text{Mix} = \Theta^\bullet, \Gamma^\bullet, \Delta^\bullet \vdash a^\bullet \oplus b^\bullet : r
$$

In practice, Mix is essential to any serious use of CPLL. For example, in the context of our unique supplies example, one would need a function to discard a supply which is no longer needed. This means we need discard of type $S \rightarrow 1$. This can only be implemented using Mix: using the Ax rule $S = \text{Int} \otimes \text{Int}^\perp$ can only be converted to $\perp$.

A cousin of Mix is Halt, which allows us to implement $1 \rightarrow \perp$. The Halt rule is also sound, and its translation requires a designated value of type $r$.

$$
(\vdash \text{halt}^\text{HALT})^\bullet = \vdash \text{mempty} : r
$$

This value corresponds to the absence of side effects, and thus should be the unit of the $\oplus$ operator. Thanks to Halt, we can implement the initial unique supply of type $S$.

Bicut. We now consider the BiCut rule [Abramsky et al., 1996], which can be thought of as a Cut with two communication channels, and corresponds to the principle $A \triangleright B \rightarrow A \otimes B$. In general, it is not safe: the processes spawned may wait for each others’ input on either channel, and deadlock.

If we have a value of type $A \triangleright B$, each component $A$ and $B$ must be processed independently, throughout the life of the program. In the context of our example, once a supply is split, the trees generated using either sub-supply can never be recombined, severely limiting the use of split. The BiCut rule allows such a recombination, and is therefore useful in practice. Fortunately, there is a case where BiCut cannot cause a deadlock: when both $A$ and $B$ are data (by definition, when both $A^\bullet$ and $B^\bullet$ make no mention of $r$). In that case, the flow of data is strictly in one direction, and BiCut indeed cannot create a deadlock. Our interpretation of BiCut is a generalisation of the technique presented in Sec. 4. The idea is to first run $a$, which produces the data. The data is stored into a memory cell, part of the $r$ type, by means of continuations passed to $a$. Once $a$ terminates, the data can be read and passed to the consumer.

This translation makes additional requirements on $r$: it must be a storage which supports references, and support a non-commutative (sequential) composition of effects. The sequential composition ensures that the reads occur after
the writes. Because of linearity, the writes are guaranteed to happen. The relative order of execution of the two writes do not matter, because they refer to different cells. In Haskell, choices for \( r \) include monads such as IO or ST.

\[
\begin{align*}
\left( \Theta; \Gamma, x : A^\perp, y : B^\perp \vdash a; \Theta; u : A, v : B, \Delta \vdash b \right) \text{ of } \text{BiCut}
\end{align*}
\]

\[
\Theta^*, \Gamma^*, \Delta^* \vdash x_r \leftarrow \text{newRef};
\]

\[
y_r \leftarrow \text{newRef};
\]

\[
\text{let } (x : A^* \rightarrow r) = \text{writeRef } x_r \text{ in}
\]

\[
\text{let } (y : B^* \rightarrow r) = \text{writeRef } y_r \text{ in}
\]

\[
a^*[x, y];
\]

\[
u \leftarrow \text{readRef } x_r;
\]

\[
v \leftarrow \text{readRef } y_r;
\]

\[
b^*[u, v]
\]

These extensions preserve our proofs of (linear) soundness, but not completeness, because it is no longer possible to instantiate \( r \) with \( \perp \).

7 Related Work

Abramsky [1993] gave the first computational interpretations of CLL, in terms of a variant of the chemical abstract machine. Here we give another interpretation in terms of \( F_2 \), the standard computational model.

Implementing concurrency by continuations is a well-known technique [Wand, 1980]. The technique is popular in the LISP community, but has so far not been applied to languages based on CLL. On the logical side of Curry-Howard, translation to continuations manifests itself as a translation from classical to intuitionistic logics (via polarisation). This is not a new idea either: it has been identified by Girard [1991] for linear logic, but can be traced far back to Glivenko [1929] (Zeilberger [2009] gives a good survey). Chaudhuri [2010] translates, among others, focused CLL into focused ILL. Our encodings of types are almost the same; the difference is that we deal with unfocused linear logic containing quantifiers, MIX and BiCut and that we focus on programming with CLL and give all of the above a computational interpretation.

8 Further Work and Conclusion

Alternative translations. Many variations on the type translation schema are possible, as long as they satisfy Lem. 1. For example, linear functions could be translated specially:

\[
(N \land B)^* = N^\perp \rightarrow B^* \quad (N \otimes B)^* = (N^* \rightarrow B^\perp)^{\rightarrow r}
\]

The cost of such a change is a more complicated term translation. The reward is a more useful type translation and perhaps more efficient programs, but more importantly: it reduces the number of negations, which makes more types be data, in turn making BiCut usable in more cases.
Performance. In this paper, we have not paid particular attention to performance, and have focused on the clarity of the translation from CLL to $F_2$. On the other hand, our encodings are fairly lightweight—for example, we encode $P \rightarrow Q$ as $P^* \times (Q^* \rightarrow r) \rightarrow r$, just as a CPS-based compiler would. Preliminary experiments show respectable performance. This is consistent with the folklore that functional language compilers thrive on continuation-heavy code.

Foundations. A largely adopted view is that CLL is more foundational than System F, and that one of Girard’s goals for CLL is to “break down” the intuitionistic implication into a linear implication and an exponential. Hence, it seems counterintuitive to compile from a low-level logic ($CLL_2$) to a higher-level one ($F_2$). But decades of work has gone into efficiently compiling $F_2$, and we can take advantage of that work.

In time, CLL might end up being used as the basis for core languages in the compiler for functional languages. Until then, the translation presented in this paper brings the benefits of a principled language for concurrency (CPLL) to the functional programming community. Our contribution is to bridge the gap between the logical foundations of CLL and its programming applications.

Acknowledgements We thank Fabien Renaud for useful feedback. This work is partially funded by SSF, under grant RAWFP.

Bibliography

The GHC Team. The unique supply implementation of the glasgow haskell compiler, 2013. retrieved Nov. 2013.
9 Appendix

9.1 Dualisation of types

\[
\begin{align*}
0^\perp &= T \quad & T^\perp &= 0 \\
1^\perp &= \bot \quad & \bot^\perp &= 1 \\
(A \oplus B)^\perp &= A^\perp \& B^\perp \quad & (A \& B)^\perp &= A^\perp \oplus B^\perp \\
(A \otimes B)^\perp &= A^\perp \uplus B^\perp \quad & (A \uplus B)^\perp &= A^\perp \otimes B^\perp \\
(!A)^\perp &= \neg A \quad & (?A)^\perp &= !A \\
(\forall \alpha.A[\alpha])^\perp &= \exists \alpha.A[\alpha] \quad & (\exists \alpha.A[\alpha])^\perp &= \forall \alpha.A[\alpha] \\
\alpha^\perp &= \alpha \quad & (\alpha^\perp)^\perp &= \alpha
\end{align*}
\]

9.2 Unique supply example: CPLL implementation

initial : S

CPLL implementation
zero : Int, ignore : Int \uplus \bar{1}; initial : Int \uplus \bar{1} Int \uplus

connect initial to
  start \to let zero' = load zero
  zero' \to start
  res \to let ignore' = load ignore
  connect ignore' to
    tmp \to res \to tmp
    tmp \to let \phi = tm
    halt

Haskell translation:

\[ \lambda \text{zero ignore initial} -> \]
\[ \quad \text{let start =} \]
\[ \quad \quad \lambda \text{start'} -> \]
\[ \quad \quad \quad \text{initial} \]
\[ \quad \quad \quad \quad (\text{start'}, \lambda \text{res} -> \]
\[ \quad \quad \quad \quad \quad \text{let ignore'} = \text{ignore in} \]
\[ \quad \quad \quad \quad \quad \text{let tmp =} \]
\[ \quad \quad \quad \quad \quad \quad \lambda \text{tmp'} -> \]
\[ \quad \quad \quad \quad \quad \quad \quad \text{ignore'} \]
\[ \quad \quad \quad \quad \quad \quad \quad (\text{tmp'}, \lambda \text{tmp}_2 -> \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \text{let () = tm} \quad \text{tm}_2 \text{ in} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{mempty}) \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{in} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{tm res}) \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{in} \]
\[ \quad \quad \quad \quad \quad \quad \text{let zero'} = \text{zero in} \]
\[ \quad \quad \quad \text{start zero'} \]
Normalised Haskell translation:

\[
\text{\textbf{split}}: S \rightarrow S \& S
\]

CPLL implementation

\[
\text{\textbf{split}}: \text{Int} \times \text{Int}^\perp \times (\text{Int}^\perp \& \text{Int}) \times (\text{Int}^\perp \& \text{Int}) \vdash
\]

let \( f, gh = \text{split} \)

let \( g, h = gh \)

let \( i, o = f \)

connect \( g \) to

\[
\text{m} \mapsto \text{connect h to}
\]

\[
\begin{align*}
\text{i}' & \mapsto \text{i}' \leftrightarrow \text{i} \\
\text{m}' & \mapsto \text{m}' \leftrightarrow \text{m} \\
\text{o}' & \mapsto \text{o}' \leftrightarrow \text{o}
\end{align*}
\]

Haskell translation:

\[
\text{\textbf{split}} ->
\]

let \((f, gh) = \text{split} \) in

let \((g, h) = gh \) in

let \((i, o) = f \) in

connect \( g \) to

\[
\text{m} \mapsto \text{connect h to}
\]

\[
\begin{align*}
\text{i}' & \mapsto \text{i}' \leftrightarrow \text{i} \\
\text{m}' & \mapsto \text{m}' \leftrightarrow \text{m} \\
\text{o}' & \mapsto \text{o}' \leftrightarrow \text{o}
\end{align*}
\]

Normalised Haskell translation:

\[
\text{\textbf{getUnique}}: S \rightarrow N \& \text{Int}
\]
CPLL implementation

\( \text{succ} : \text{Int}^+ \Rightarrow \text{Int}, \text{copy} : \text{Int}^+ \Rightarrow \text{Int} \otimes \text{Int}; \text{getUnique} : \text{Int} \otimes \text{Int}^+ \otimes (\text{Int}^+ \Rightarrow \text{Int}^+ \Rightarrow \text{Int}^+) \Rightarrow \)

\begin{align*}
&\text{let } \text{inpout}, \text{triple} = \text{getUnique} \\
&\text{let } \text{inp}, \text{out} = \text{inpout} \\
&\text{let } \text{copy'} = \text{load } \text{copy} \\
&\text{connect } \text{copy'} \text{ to} \\
&\quad \text{inp'} \mapsto \text{inp' } \mapsto \text{inp} \\
&\quad \text{inp} \mapsto \text{let } \text{inp}_1, \text{inp}_2 = \text{inp} \\
&\quad \quad \text{let } \text{succ'} = \text{load } \text{succ} \\
&\quad \quad \text{connect } \text{succ'} \text{ to} \\
&\quad \quad \quad \text{sinp'} \mapsto \text{sinp'} \mapsto \text{inp}_2 \\
&\quad \quad \quad \text{sinp} \mapsto \text{connect } \text{triple} \text{ to} \\
&\quad \quad \quad \quad \text{w} \mapsto \text{u} \mapsto \text{inp}_1 \\
&\quad \quad \quad \quad \text{t} \mapsto \text{connect } \text{t} \text{ to} \\
&\quad \quad \quad \quad \quad \text{m} \mapsto \text{m} \mapsto \text{sinp} \\
&\quad \quad \quad \quad \quad \text{m'} \mapsto \text{m'} \mapsto \text{out} \\
&\end{align*}

Haskell translation:

\[
\text{\textbackslash succ \ copy \ getUnique } \Rightarrow \\
\text{let } (\text{inpout}, \text{triple}) = \text{getUnique} \ \text{in} \\
\text{let } (\text{inp}, \text{out}) = \text{inpout} \ \text{in} \\
\text{let } \text{copy'} = \text{copy} \ \text{in} \\
\text{let } \text{inp'} = \\
\quad \text{\textbackslash (inp', \text{inp''} \mapsto \text{copy'} \text{ in})} \\
\quad , (\text{\textbackslash inps } \mapsto \\
\quad \quad \text{let } (\text{inp}_1, \text{inp}_2) = \text{inps} \ \text{in} \\
\quad \quad \text{let } \text{succ'} = \text{succ} \ \text{in} \\
\quad \quad \text{let } \text{sinp'} = \\
\quad \quad \quad \text{\textbackslash (sinp', \text{sinp''} \mapsto \text{succ'} \text{ in})} \\
\quad \quad \quad , (\text{\textbackslash sinp } \mapsto \\
\quad \quad \quad \quad \text{let } \text{u} = \\
\quad \quad \quad \quad \quad \text{\textbackslash (u' \mapsto \\
\quad \quad \quad \quad \quad \quad \text{let } \text{t} = \\
\quad \quad \quad \quad \quad \quad \quad \text{\textbackslash t' \mapsto \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{\textbackslash triple \ (u', t') } \text{ in} \\
\quad \quad \quad \quad \quad \quad \quad \text{let } \text{m} = \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{\textbackslash m' \mapsto \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{t (m')} \\
\quad \quad \quad \quad \quad \quad \quad \quad , (\text{\textbackslash m' } \mapsto \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{out m'})}
\]
Normalised Haskell translation:

\ succ copy getUnique ->
let (inpout, triple) = getUnique in
let (inp, out) = inpout in
  copy
  (inp,
   (\ inps ->
    let (inp_1, inp_2) = inps in
      succ (inp_2, (\m' -> triple (inp_1, (m', out))))))

\ discard ->
let (inpout, b) = discard in
let (inp, out) = inpout in
out inp 'mappend' b ()

Normalised Haskell translation:

\ discard ->
let (inpout, b) = discard in
let (inp, out) = inpout in
out inp 'mappend' b ()
### 9.3 F2 Typing rules

\[
\begin{align*}
\Gamma \vdash f : A \rightarrow B & \quad \Gamma \vdash a : A \\ 
\Gamma \vdash f \ a : B & \quad \rightarrow\text{elim} \\
\Gamma \vdash e : A \times B & \quad \Gamma, x : A, y : B \vdash e : C \\
\Gamma \vdash (x, y) = e \text{ in } c : C & \quad \times\text{elim} \\
\Gamma \vdash (a, b) : A \times B & \quad \times\text{intro} \\
\Gamma \vdash e : A + B & \quad x : A, \Gamma \vdash a : C \\
\Gamma \vdash y : B, \Gamma \vdash b : C & \quad e : \text{of } \{\text{inr } y \mapsto b\} : C \\
\Gamma \vdash e : A & \quad \rightarrow\text{intro-l} \\
\Gamma \vdash e : B & \quad \text{intro-r} \\
\Gamma \vdash \text{inr } e : A + B & \quad \text{intro} \\
\Gamma \vdash \forall \alpha. A[\alpha] & \quad \forall\text{elim} \\
\Gamma \vdash \alpha \in A & \quad \forall\text{intro} \\
\Gamma \vdash e : A & \quad \text{init} \\
\Gamma \vdash e : A & \quad \text{cut} \\
\end{align*}
\]

### 9.4 ILL Typing rules

The rules for & in ILL are not listed because they are never used in this article.

\[
\begin{align*}
\Theta; \Gamma \vdash f : A \rightarrow B & \quad \Theta; \Delta \vdash a : A \\
\Theta; \Gamma, x : A \vdash b : B & \quad \rightarrow\text{elim} \\
\Theta ; \Gamma \vdash \lambda x. b : A \rightarrow B & \quad \rightarrow\text{intro} \\
\Theta ; \Gamma \vdash e : A \otimes B & \quad \Theta ; \Delta \vdash x : A, y : B \vdash e : C \\
\Theta ; \Gamma, \Delta \vdash (x, y) = e \text{ in } c : C & \quad \otimes\text{elim} \\
\Theta ; \Gamma, \Delta \vdash (a, b) : A \otimes B & \quad \otimes\text{intro} \\
\Theta ; \Gamma \vdash e : A & \quad \otimes\text{intro-l} \\
\Theta ; \Gamma \vdash e : B & \quad \otimes\text{intro-r} \\
\Theta ; \Gamma \vdash \text{inr } e : A \oplus B & \quad \oplus\text{intro} \\
\Theta ; \Gamma \vdash e : \forall \alpha. A[\alpha] & \quad \forall\text{elim} \\
\Theta ; \Gamma \vdash \alpha \in A & \quad \forall\text{intro} \\
\Theta ; \Gamma \vdash e : A & \quad \text{init} \\
\Theta ; \Gamma, \Delta \vdash x = e \text{ in } c : C & \quad \text{cut} \\
\Theta ; \Gamma, \Delta \vdash x : A \vdash \text{load } x : A & \quad \text{init} \\
\Theta ; \Gamma \vdash e : !A & \quad \Theta ; \Gamma \vdash e : A \\
\Theta ; \Gamma, \Delta \vdash z : A, \Theta ; \Delta \vdash c : C & \quad \text{let } z = \text{save } e \text{ in } c : C & \quad \text{load} \\
\Theta ; \Gamma \vdash e : !A & \quad \text{let } e = \text{offer } e : !A & \quad \text{load} \end{align*}
\]
9.5 Type translation rules to System F

\[(A \otimes B)^* = A^* \times B^* \quad (A \sqcap B)^* = A^{\perp*} \times B^{\perp*} \to r\]
\[(A \oplus B)^* = A^* + B^* \quad (A \& B)^* = (A^\perp* + B^\perp*) \to r\]
\[1^* = 1 \quad \perp^* = 1 \to r\]
\[0^* = 0 \quad \top^* = 0 \to r\]
\[(!A)^* = A^* \quad (?A)^* = A^{\perp*} \to r\]
\[\alpha^* = \alpha \quad \alpha^{\perp*} = \alpha \to r\]

\[(\forall \alpha. A[\alpha])^* = \forall \alpha. ((A[\alpha]^{\perp*} + A[\alpha^{\perp*}]) \to r)\]
\[(\exists \alpha. A[\alpha])^* = (\forall \alpha. ((A[\alpha]^{\perp*} + A[\alpha^{\perp*}]) \to r)) \to r\]

9.6 Type translation rules to ILL

\[(A \otimes B)^* = A^* \otimes B^* \quad (A \sqcap B)^* = A^{\perp*} \otimes B^{\perp*} \to \circ r\]
\[(A \oplus B)^* = A^* \oplus B^* \quad (A \& B)^* = (A^{\perp*} \oplus B^{\perp*}) \to \circ r\]
\[1^* = 1 \quad \perp^* = 1 \to \circ r\]
\[0^* = 0 \quad \top^* = 0 \to \circ r\]
\[(!A)^* = !\neg\neg A^* \quad (?A)^* = !\neg\neg A^{\perp*} \to \circ r\]
\[\alpha^* = \alpha \quad \alpha^{\perp*} = \alpha \to \circ r\]

\[(\forall \alpha. A[\alpha])^* = \forall \alpha. ((A[\alpha]^{\perp*} \oplus A[\alpha^{\perp*}]) \to \circ r)\]
\[(\exists \alpha. A[\alpha])^* = (\forall \alpha. ((A[\alpha]^{\perp*} \oplus A[\alpha^{\perp*}]) \to \circ r)) \to \circ r\]

9.7 CLL2 Reduction rules

We list rules operating with one given orientation of the cut. For each reduction, its composition with a the cut swap shown below must be considered.
Principal reduction rules

In the ?!-reduction, a cut is in fact introduced for every load. For simplicity, we only present the reduction with one load right after the save.
Structural reduction rules

**Associativity**

\[
\begin{array}{c}
\Theta; \Gamma, A \vdash a \\
\Theta; A, \Delta, B \vdash b
\end{array} \quad \text{Cut}
\]

\[
\begin{array}{c}
\Theta; \Gamma, A \vdash a \\
\Theta; A, \Delta, B \vdash b \\
\Theta; B, \Xi \vdash c
\end{array} \quad \text{Cut}
\]

\[
\Theta; \Gamma, A \vdash \Xi
\]

**Ax**

\[
\begin{array}{c}
\Theta; \Gamma, A \vdash a[x] \\
\Theta; A, A \vdash a
\end{array} \quad \text{Ax}
\]

\[
\Theta; \Gamma, A \vdash a
\]

\[
\Theta; \Gamma, A \vdash a[w]
\]
Commuting conversions

\[ \frac{\Theta; A, B, \Delta \vdash b \quad \Theta; C, \Xi \vdash c}{\Theta; \Gamma, A[B], \Delta \vdash c} \]

\[ \frac{\Theta; \Gamma, A \vdash a \quad \Theta; A, B \vdash \beta}{\Theta; \Gamma, A, B \vdash \beta} \]

\[ \frac{\Theta; A, B, \Delta \vdash b}{\Theta; \Gamma, A \vdash a} \]

\[ \frac{\Theta; \Gamma, A \vdash a \quad \Theta; A, B \vdash \beta}{\Theta; \Gamma, A, B \vdash \beta} \]

\[ \frac{\Theta; \Gamma, A \vdash a \quad \Theta; A, B \vdash \beta}{\Theta; \Gamma, A, B \vdash \beta} \]

\[ \frac{\Theta; \Gamma, A \vdash a \quad \Theta; A, B \vdash \beta}{\Theta; \Gamma, A, B \vdash \beta} \]

\[ \frac{\Theta; 
\]
9.8 Reduction correspondence

**Associativity**

\[
\begin{align*}
\left( \Theta; \Gamma, x : A^+ \vdash a &\quad \Theta; \bar{x} : A, \Delta, y : B^+ \vdash b \quad C_{\text{cut}} \right) = \\
\left( \Theta; \Gamma, \Delta, y : B^+ \vdash \text{cut} \{ x : A^+ \mapsto a; \bar{x} : A \mapsto b \} \quad \Theta; \bar{y} : B, \Xi \vdash c \quad C_{\text{cut}} \right) \\
\left( \Theta; \Delta, \Xi \vdash \text{cut} \{ y : B^+ \mapsto \text{cut} \{ x : A^+ \mapsto a; \bar{x} : A \mapsto b \}; \bar{y} : B \mapsto c \} \right)
\end{align*}
\]

\[\Theta; \Gamma^*, A^*, \Xi^* \vdash (\lambda y . (\lambda x . a^*[x]) (\lambda \bar{x} . b^*[\bar{x}, \bar{y}]) (\lambda \bar{y} . c^*[\bar{y}])) : r \quad \rightarrow \quad (\lambda x . a^*[x]) (\lambda \bar{x} . b^*[\bar{x}, \lambda \bar{y} . c^*[\bar{y}]])) : r \quad \rightarrow \quad a^*[\lambda \bar{x} . b^*[\bar{x}, \lambda \bar{y} . c^*[\bar{y}]]) : r\]

**Ax**

\[
\begin{align*}
\left( \Theta; \Gamma, x : A^+ \vdash a[x] &\quad \Theta; y : A, w : A^+ \vdash y \mapsto w \quad \text{Ax} \quad \left( \Theta; \Gamma, w : A^+ \vdash \text{cut} \{ x : A^+ \mapsto a[x]; y : A \mapsto y \mapsto w \} \quad \text{C}_{\text{cut}} \right) \right) = \\
\left( \Theta; \Gamma, w : A^+ \vdash \text{cut} \{ x : A^+ \mapsto a[x]; y : A \mapsto y \mapsto w \} \quad \text{C}_{\text{cut}} \right) \quad \text{Ax}
\end{align*}
\]

\[\Theta; \Gamma^*, w : A^* \rightarrow r \vdash (\lambda x . a^*[x]) (\lambda y . w y) : r \quad \rightarrow \quad a^*[\lambda y . w y] : r \quad \rightarrow \quad a^*[w] : r \]

\[\Theta; \Gamma, w : A^+ \vdash a[w] \quad \rightarrow \quad \Theta; \Gamma, w : A^* \rightarrow r \vdash a^*[w] : r\]
\[
\begin{align*}
\left( \frac{\Theta; \Gamma, \bar{x} : A \vdash a \quad \Theta; \Delta, \bar{y} : B \vdash b}{\Theta; \Gamma, \Delta, \bar{z} : A \otimes B \vdash \text{cut} \ \bar{z} \to (x \mapsto a; \bar{y} \mapsto b)} \right) = \\
\left( \frac{\Theta; x : A, y : B, \Xi \vdash c}{\Theta; z : A \otimes B, \Xi \vdash \text{let } x, y = z; c} \right) \text{ cut}
\end{align*}
\]

\[
\rightarrow \Theta^*; I^*, \Delta^*, \Xi^* \vdash \lambda \bar{x}. \text{ let } \bar{x} = \lambda \bar{x}'. \text{ let } \bar{y} = \\
\lambda \bar{y}'. \bar{z} \ (\bar{x}', \bar{y}') \in b^*[\bar{y}] \in a^*[\bar{x}] \ (\lambda z. \text{ let } (x, y) = z \in c^*[x, y]) : r \\
\rightarrow a^*[\lambda \bar{x}'. \text{ let } \bar{y} = \lambda \bar{y}'. \ (\lambda z. \text{ let } (x, y) = z \in c^*[x, y]) \ (\bar{x}', \bar{y}')] : r
\]

\[
\left( \frac{\Theta; \Gamma, \bar{x} : A \vdash a \quad \Theta; x : A, \Delta, \Xi \vdash \text{cut} \ \{ \bar{y} : B \vdash b; y : B \vdash c \} \text{ cut}}{\Theta; \Gamma, \Delta, \Xi \vdash \text{cut} \ \{ \bar{x} : A \rightarrow a; x : A \rightarrow \text{cut} \ \{ \bar{y} : B \rightarrow b; y : B \rightarrow c \} \text{ cut}} \right)^* = \\
\rightarrow \Theta^*; I^*, \Delta^*, \Xi^* \vdash \lambda \bar{x}. \text{ let } \bar{x} = \lambda \bar{x}'. \text{ let } \bar{y} = \\
a^*[\lambda \bar{x}. \text{ let } \bar{y} = \lambda \bar{y}'. \ (\lambda z. \text{ let } (x, y) = z \in c^*[x, y]) \ (\bar{x}', \bar{y}')] : r \\
\rightarrow a^*[\lambda \bar{x}'. \text{ let } (x, y) = (\bar{x}', \bar{y}')] \ (\lambda y. \bar{c}[x, y]) : r \\
\rightarrow a^*[\lambda x. \ (\lambda \bar{y}. \bar{c}[\bar{x}]) \ (\lambda y. \bar{c}[x, y]) : r \\
\rightarrow a^*[\lambda x. \ (\lambda \bar{y}. \bar{b}[\bar{y}]) \ (\lambda y. \bar{c}[x, y]) : r \\
\rightarrow a^*[\lambda x. \ (\lambda \bar{y}. \bar{b}[\bar{y}]) \ (\lambda y. \bar{c}[x, y]) : r \\
\rightarrow a^*[\lambda x. \ (\lambda \bar{y}. \bar{b}[\bar{y}]) \ (\lambda y. \bar{c}[x, y]) : r
\]
\[
\begin{array}{c}
\begin{array}{c}
\Delta \vdash \overline{x} : A \\
\Delta, \overline{x} : A \vdash \overline{x} \to \overline{a} ; \overline{y} \to \overline{b}
\end{array}
\end{array}
\]
\[
\begin{align*}
\neg\Theta^*; I^*, \Delta^*, \Xi^* \vdash (\lambda \tilde{z} . \text{let } \tilde{x} = \lambda \tilde{x}' . \tilde{z} (\tilde{x}' (\lambda \bar{y} . b^*[\bar{y}])) \in a^*[\tilde{x}]) (\lambda z . \text{let } (x, y) = z \in c^*[x, y]) : r \\
\rightarrow (\text{let } \tilde{x} = \lambda \tilde{x}' . (\lambda z . \text{let } (x, y) = z \in c^*[x, y]) (\tilde{x}' (\lambda \bar{y} . b^*[\bar{y}])) \in a^*[\tilde{x}]) : r \\
\rightarrow a^*[\lambda \tilde{x}' . (\lambda z . \text{let } (x, y) = z \in c^*[x, y]) (\tilde{x}' (\lambda \bar{y} . b^*[\bar{y}]))] : r \\
\rightarrow a^*[\lambda x . (\lambda y . c^*[x, y]) (\lambda \bar{y} . b^*[\bar{y}])] : r \\
\rightarrow a^*[\lambda x . c^*[x, \lambda \bar{y} . b^*[\bar{y}]]) : r \\
\neg\Theta^*; I^*, \Delta^*, \Xi^* \vdash (\lambda \tilde{z} . ((\lambda \bar{x} . a^*[\bar{x}])), (\lambda \bar{y} . b^*[\bar{y}])) (\lambda z . \text{let } (x, y) = z \in c^*[x, y]) : r \\
\rightarrow (\lambda z . \text{let } (x, y) = z \in c^*[x, y]) ((\lambda \bar{x} . a^*[\bar{x}]), (\lambda \bar{y} . b^*[\bar{y}])) : r \\
\rightarrow (\text{let } (x, y) = ((\lambda \bar{x} . a^*[\bar{x}]), (\lambda \bar{y} . b^*[\bar{y}])) \in c^*[x, y]) : r \\
\rightarrow c^*[\lambda \bar{x} . a^*[\bar{x}], \lambda \bar{y} . b^*[\bar{y}]] : r \\
\neg\Theta^*; I^*, \Delta^*, \Xi^* \vdash (\lambda \bar{x} . a^*[\bar{x}]), (\lambda \bar{y} . b^*[\bar{y}]) (\lambda x . (\lambda y . c^*[x, y]) (\lambda \bar{y} . b^*[\bar{y}]))) (\lambda \bar{x} . a^*[\bar{x}]) : r \\
\rightarrow (\lambda y . c^*[\lambda \bar{x} . a^*[\bar{x}], y]) (\lambda \bar{y} . b^*[\bar{y}]) : r \\
\rightarrow c^*[\lambda \bar{x} . a^*[\bar{x}], \lambda \bar{y} . b^*[\bar{y}]] : r
\end{align*}
\]
\[
\begin{align*}
\Theta, \Gamma, x : A^+ \vdash a & \quad \Theta, \bar{x} : A, \Xi \vdash b & \quad \Theta, \bar{y} : B, \Xi \vdash c \\
\Theta, \Gamma, z : A^+ & \& B^+ \vdash \text{let } x = \text{fst } z; a & \quad \Theta, \bar{z} : A \& B, \Xi \vdash \text{case } \bar{z} \text{ of } \{ \text{inl } \bar{x} \mapsto b; \text{inr } \bar{y} \mapsto c \} \\
\Theta, \Gamma, \Xi \vdash \text{cut } \{ z : A^+ & \& B^+ \vdash \text{let } x = \text{fst } z; a; \bar{z} : A \& B \mapsto \text{case } \bar{z} \text{ of } \{ \text{inl } \bar{x} \mapsto b; \text{inr } \bar{y} \mapsto c \} & \quad \Theta^* \vdash I^*, \Xi^* \vdash
\end{align*}
\]
\[
\begin{align*}
\left(\Theta; x : A \vdash a\right) & \quad \left(\Theta; \lambda z : A \vdash b\right) \\
\text{Offer} & \\
\Theta; z : A \vdash \text{offer} z : a & \quad \Theta; \bar{x} : \text{load} \bar{x} : b \\
\text{Save} & \\
\Theta; \Gamma \vdash \text{cut} \{z : A \vdash \text{offer} z ; a; \bar{x} : A \vdash \text{load} \bar{x} : b\} & \quad \Theta; \Gamma \vdash \text{cut} \{z : A \vdash \text{offer} z ; a; \bar{x} : A \vdash \text{load} \bar{x} : b\}
\end{align*}
\]

\[
\begin{align*}
(\lambda z. z(\text{offer} (\lambda x. a^*[x])))(\lambda \bar{x}. \text{let } \bar{x} = \text{save } \bar{x} \text{ in } (\text{load } \bar{x}) (\lambda y. b^*[y])) : r & \quad \rightarrow \\
(\lambda \bar{x}. \text{let } \bar{x} = \text{save } \bar{x} \text{ in } (\text{load } \bar{x}) (\lambda y. b^*[y]))(\text{offer} (\lambda x. a^*[x])) : r & \quad \rightarrow \\
(\text{let } \bar{x} = \text{save } (\text{offer} (\lambda x. a^*[x])) \text{ in } (\text{load } \bar{x}) (\lambda y. b^*[y])) : r & \quad \rightarrow \\
(\lambda x. a^*[x])(\lambda y. b^*[y]) : r & \quad \rightarrow \\
\left(\Theta; \Gamma, y : A \vdash b\right) & \quad \left(\Theta; \Gamma, x : A \vdash a\right) \\
\text{Cut} & \\
\Theta; \Gamma \vdash \text{cut} \{y : A \vdash b; x : A \vdash a\} & \quad \Theta; \Gamma \vdash \text{cut} \{y : A \vdash b; x : A \vdash a\}
\end{align*}
\]
\[ (\Theta; z : L \vdash z) \quad (\Theta; \overline{z} : 1 \vdash \overline{a}) \quad (\Theta; \overline{z} : 1 \vdash \overline{\lambda z.\overline{a}}) \quad (\Theta; z : L \vdash z) \quad (\Theta; \overline{z} : 1 \vdash \overline{\lambda z.\overline{a}}) \quad \text{cut} \] 

\[ \lambda z. \overline{z} \cdot \overline{\lambda z.\overline{a}} \] 

\[ \overline{\lambda z.\overline{a}} \]
$$\exists \forall \rightarrow$$
\[
\begin{align*}
&\left( \Theta, \Gamma, x : A[B] \vdash a \quad \forall b \text{ fresh, } \Theta, \bar{x} : A[\beta]^+, \Xi \vdash \bar{b} \quad \exists \bar{\beta} \text{ in } \Lambda \right)^* = \\
&\left( \Theta, \Gamma, z : \forall \alpha. A[\alpha] \vdash \text{let } x = z \bullet B ; a \quad \Theta, \bar{z} : \exists \alpha. A[\alpha]^+ \vdash \text{let } \beta, \bar{x} = \bar{z}; \bar{b} \quad \text{Cor} \right)
\end{align*}
\]

\[
\neg \Theta^*, \Gamma^*, \Xi^* \vdash (\Lambda \bar{z} . \bar{z} : \Lambda \bar{x} . \Lambda \bar{x}. \text{case } \bar{x}' \text{ of } \{ \text{inl } \bar{x} \mapsto b^*[\bar{x}]; \text{inr } \bar{x} \mapsto b[\beta^+ / \beta^*[\bar{x}]] \}) (\lambda z . z \odot B^* (\text{inl (}\lambda x . a^*[x]\))): r \quad \rightarrow
\]

\[
(\lambda z . z \odot B^* (\text{inl (}\lambda x . a^*[x]\)) (\Lambda \beta . \lambda \bar{x}' . \text{case } \bar{x}' \text{ of } \{ \text{inl } \bar{x} \mapsto b^*[\bar{x}]; \text{inr } \bar{x} \mapsto b[\beta^+ / \beta^*[\bar{x}]] \}) : r \quad \rightarrow
\]

\[
(\Lambda \beta . \lambda \bar{x}' . \text{case } \bar{x}' \text{ of } \{ \text{inl } \bar{x} \mapsto b^*[\bar{x}]; \text{inr } \bar{x} \mapsto b[\beta^+ / \beta^*[\bar{x}]] \} : r \quad \rightarrow
\]

\[
(\lambda x . a^*[x]) \text{ of } \{ \text{inl } \bar{x} \mapsto b^*[\bar{x}]; \text{inr } \bar{x} \mapsto b[\beta^+ / \beta^*[\bar{x}]] \} : r \quad \rightarrow
\]

\[
\left( \Theta, \Gamma, x : A[B] \vdash a \quad \Theta, \bar{x} : A[\beta]^+, \Xi \vdash \bar{b} \quad \text{Cor} \right)^* = \neg \Theta^*, \Gamma^*, \Xi^* \vdash
\]

\[
(\lambda \bar{x} . b^*[\bar{x}]) (\lambda x . a^*[x]) : r \quad \rightarrow
\]

\[
b^*[\lambda x . a^*[x]] : r
\]
There are many commuting conversions that fail, so we only list this last example.
9.9 Translation: Full translation with derivation trees

**Ax**

\[(\Theta; x : P, y : P \vdash x \leftrightarrow y) \] =

\[\vdash \neg \neg \Theta; y : P \rightarrow r \] init \( \vdash \neg \neg \Theta; x : P \rightarrow r \) init \( \vdash \neg \Theta; y : P \rightarrow \perp \) \( \vdash \neg \Theta; x \dashv P \rightarrow \perp \) elim

**Cut**

\[\begin{align*}
(\Theta; \Gamma, x : P \vdash a) \vdash b \\
(\Theta; \Gamma, \Delta \vdash \text{cut} \{x : P \mapsto a; y : P \mapsto b\})
\end{align*}\] =

\[\vdash \neg \Theta; \Delta, \neg \Theta; \Delta \vdash y : P \rightarrow \perp \rightarrow r \] intro \( \vdash \neg \Theta; \Delta \vdash \lambda y. b : P \rightarrow \perp \rightarrow r \) elim \( \vdash \neg \Theta; \Delta \vdash \lambda x. a : P \rightarrow \perp \rightarrow r \) intro \( \vdash \neg \Theta; \Delta \vdash (\lambda y. b) (\lambda x. a) : r \) elim

**L**

\[(\Theta; x : \perp \vdash x) \] =

\[\vdash \neg \Theta; x : 1 \rightarrow r \vdash x : 1 \rightarrow r \] init \( \vdash \neg \Theta; x : 1 \rightarrow r \vdash x : 0 \rightarrow r \) elim

**1**

\[\begin{align*}
(\Theta; \Gamma, x : 1 \vdash a) \\
(\Theta; \Gamma, x : 1 \vdash \text{let } \phi = x \vdash a)
\end{align*}\] =

\[\vdash \neg \Theta; x : 1 \vdash x : 1 \] init \( \vdash \neg \Theta; x : 1 \vdash x : 0 \rightarrow r \) elim

**0**

\[\begin{align*}
(\Theta; \Gamma, x : 0 \vdash \text{dump } \Gamma \text{ in } x^0) \\
(\Theta; \Gamma, x : 0 \vdash \text{dump } \Gamma \text{ in } x^0)
\end{align*}\] =

\[\vdash \neg \Theta; x : 0 \vdash x : 0 \] init \( \vdash \neg \Theta; x : 0 \vdash x : 1 \rightarrow \perp \rightarrow r \) elim

**A \otimes B**

\[\begin{align*}
(\Theta; \Gamma, x : A, y : B \vdash a) \\
(\Theta; \Gamma, z : A \otimes B \vdash \text{let } x, y = z \vdash a)
\end{align*}\] =

\[\vdash \neg \Theta; z : A \otimes B \vdash z : A \otimes B \] init \( \vdash \neg \Theta; \Gamma, x : A \otimes B \vdash y : B \rightarrow a \rightarrow r \) init \( \vdash \neg \Theta; \Gamma, y : B \rightarrow a \rightarrow r \rightarrow \perp \rightarrow r \) init \( \vdash \neg \Theta; \Gamma, x : A \otimes B \vdash \text{let } x, y = z \vdash a \rightarrow r \) \otimes \text{elim}
\( A \oplus B \)

\[
\begin{array}{c}
\Theta; \Gamma, x : A \vdash a \quad \Theta; \Gamma, y : B \vdash b \\
\Theta; \Gamma, z : A \oplus B \vdash \text{case } z \text{ of } \{ \text{inl } x \mapsto a; \text{inr } y \mapsto b \}^\oplus \\
\end{array}
\]

\[
\begin{array}{c}
\vdash \Theta^*; x : A^* \vdash a^* : r^{IH} \\
\vdash \Theta^*; y : B^* \vdash b^* : r^{IH} \\
\vdash \Theta^*, \Gamma^*, z : A^* \oplus B^* \vdash \text{case } z \text{ of } \{ \text{inl } x \mapsto a^*; \text{inr } y \mapsto b^* : r^{\oplus\text{-elim}} \\
\end{array}
\]

\( P & B \)

\[
\begin{array}{c}
\Theta; \Gamma, x : P \vdash a \\
\Theta; \Gamma, z : P & B \vdash \text{let } x = \text{fst } z ; a^k \end{array}
\]

\[
\begin{array}{c}
\vdash \Theta^*; x : P^* \vdash a^* : r^{IH} \\
\vdash \Theta^*, \Gamma^* : \lambda x. a^* : P^* \vdash r^{\lambda\text{-intro}} \\
\vdash \Theta^*, \Gamma^* : \text{inl } (\lambda x. a^*) : (P^* \rightarrow r) \oplus (B^* \rightarrow r)^{\oplus\text{-intro-}} \\
\vdash \Theta^*, \Gamma^* : ((P^* \rightarrow r) \oplus (B^* \rightarrow r)) \rightarrow r \vdash z : ((P^* \rightarrow r) \oplus (B^* \rightarrow r)) \rightarrow r^{\rightarrow\text{-elim}} \\
\end{array}
\]

\( N & B \)

\[
\begin{array}{c}
\Theta; \Gamma, x : N \vdash a \\
\Theta; \Gamma, z : N & B \vdash \text{let } x = \text{fst } z ; a^k \end{array}
\]

\[
\begin{array}{c}
\vdash \Theta^*; x : N^* \vdash a^* : r^{IH} \\
\vdash \Theta^*, x' : N^* \vdash x' : N^*^{\oplus\text{-intro-}} \\
\vdash \Theta^* ; z : (N^* \oplus (B^* \rightarrow r)) \rightarrow r \vdash z : (N^* \oplus (B^* \rightarrow r)) \rightarrow r^{\rightarrow\text{-elim}} \\
\vdash \Theta^* ; z : (N^* \oplus (B^* \rightarrow r)) \rightarrow r \vdash \lambda x'. z \{ \text{inl } x' \} : N^* \rightarrow a^*^{\rightarrow\text{-intro}} \\
\vdash \Theta^*, \Gamma^* ; z : (N^* \oplus (B^* \rightarrow r)) \rightarrow r \vdash \text{let } x = \lambda x'. z \{ \text{inl } x' \} \text{ in } a^* : r^{\text{cut}} \\
\end{array}
\]

Exponentials, in System F

\( !A \)

\[
\begin{array}{c}
x : A, \Theta; \Gamma \vdash a \\
\Theta; \Gamma, z : !A \vdash \text{let } x = \text{save } z ; a^{\text{save}} \end{array}
\]

\[
\begin{array}{c}
\vdash \Theta^* ; x : A^* \vdash a^* : r^{IH} \\
\vdash \Theta^*, \Gamma^*, z : A^* \vdash \text{let } x = z \text{ in } a^* : r^{!\text{-elim}} \\
\end{array}
\]
\[
\frac{\Theta, z : A; \Gamma, x : A \vdash a}{\Theta, z : A; \Gamma \vdash \text{let } x = \text{load } z; a} \quad \text{(Load)}
\]

\[
\frac{\Theta^*, z : A^*, \Gamma^* \vdash z : A^*}{\Theta^*, z : A^*, \Gamma^*, x : A^* \vdash a^* : r} \quad \text{IH}
\]

\[
\frac{\Theta^*, z : A^*, \Gamma^* \vdash \text{let } x = z \text{ in } a^* : r}{\Theta^*, z : A, \Gamma \vdash a}
\]

\[
\frac{\Theta, z : A; \Gamma \vdash a}{\Theta, z \vdash \text{offer } z; a} \quad \text{(Offer)}
\]

\[
\frac{\Theta^*, z : (P^* \rightarrow r) \rightarrow x : P^* \vdash a^* : r}{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda x. a^* : P^* \rightarrow r} \quad \text{→-intro}
\]

\[
\frac{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda(\lambda x. a^*): P^* \rightarrow r}{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda x. a^* : r} \quad \text{→-elim}
\]

\[
\frac{\Theta, x : P \vdash a}{\Theta, z : ?P \vdash \text{offer } z; a}
\]

\[
\frac{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda(\lambda x. a^*): P^* \rightarrow r}{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda x. a^* : r} \quad \text{→-intro}
\]

\[
\frac{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda x. a^* : r}{\Theta^*, z : (P^* \rightarrow r) \rightarrow r \vdash \lambda x. a^* : r} \quad \text{→-elim}
\]

\[
\frac{\Theta, x : N \vdash a}{\Theta, z : ?N \vdash \text{offer } z; a}
\]

Exponentials, in ILL

\[
\frac{\Theta, z : !A \vdash a}{\Theta; z \vdash !A \vdash \text{save } z; a} \quad \text{(Save)}
\]

\[
\frac{\neg\Theta^*, z : !\neg A^* \vdash z : !\neg A^*}{\neg\Theta^*, z : !\neg A^* \vdash \text{let } x = \text{save } z; a} \quad \text{→-intro}
\]

\[
\frac{\neg\Theta^*, z : !\neg A^*, \Gamma^* \vdash a^* : r}{\neg\Theta^*, z : !\neg A^* \vdash \text{let } x = z \text{ in } a^* : r} \quad \text{→-elim}
\]

\[
\frac{\Theta, z : A; \Gamma, x : A \vdash a}{\Theta, z : A; \Gamma \vdash \text{load } z; a} \quad \text{(Load)}
\]

\[
\frac{\neg\Theta^*, z : !\neg A^* \vdash \text{load } z : !\neg A^*}{\neg\Theta^*, z : !\neg A^*, \Gamma^* \vdash \lambda x. a^* : A^* \rightarrow r} \quad \text{→-intro}
\]

\[
\frac{\neg\Theta^*, z : !\neg A^*, \Gamma^* \vdash \lambda x. a^* : A^* \rightarrow r}{\neg\Theta^*, z : !\neg A^* \vdash \text{let } x = z \text{ in } \lambda x. a^* : r} \quad \text{→-elim}
\]
\[
\begin{align*}
\Theta; x : P \vdash a \\
\Theta; z : ? P \vdash x = \text{offer } z; a \quad \text{Offer}^* \quad \Downarrow
\end{align*}
\]

\[
\begin{align*}
\neg\Theta^*; x' : \neg P^* \vdash \text{init}_L & \quad \neg\Theta^*; x' : \neg P^* \vdash \lambda x. a^* : r \quad \text{intro} \\
\neg\Theta^*; x' : \neg P^* \vdash \lambda x. a^* : r & \quad \neg\Theta^*; x' : \neg P^* \vdash \text{init}_L \\
\neg\Theta^*; \text{offer} (\lambda x'. x' (\lambda x. a^*)) : !\neg(P^* \rightarrow r) & \quad \text{intro} \\
\neg\Theta^*; \text{offer} (\lambda x'. x' (\lambda x. a^*)) : !\neg(P^* \rightarrow r) & \quad \text{elim}
\end{align*}
\]

\[
\begin{align*}
\neg\Theta^*; z \vdash \neg(P^* \rightarrow r) & \quad \text{init}_L \\
\neg\Theta^*; z \vdash \neg(P^* \rightarrow r) & \quad \text{elim}
\end{align*}
\]

\[
\begin{align*}
\Theta; x : N \vdash a \\
\Theta; z : ? N \vdash x = \text{offer } z; a \quad \text{Offer}^* \quad \Downarrow
\end{align*}
\]

\[
\begin{align*}
\neg\Theta^*; x : \neg N^* \vdash r & \quad \text{intro} \\
\neg\Theta^*; x : \neg N^* \vdash r & \quad \text{intro} \\
\neg\Theta^*; \text{offer} (\lambda x. a^*) : !\neg N^* & \quad \text{init}_L \\
\neg\Theta^*; \text{offer} (\lambda x. a^*) : !\neg N^* & \quad \text{elim}
\end{align*}
\]

Quantifiers
∀α.P[α], Q

\[
\begin{array}{l}
\frac{\Gamma, x : P[Q] \vdash a}{\Gamma, z : \forall \alpha. P[\alpha] \vdash \text{let } x = z \cdot Q; a} \\
\frac{\neg\neg \Theta; z : \forall \alpha. ((\forall \alpha.((P[\alpha] \rightarrow r) \oplus (P[\alpha^+] \rightarrow r)) \rightarrow r) \vdash z : \forall \alpha. ((\forall \alpha.((P[\alpha] \rightarrow r) \oplus (P[\alpha^+] \rightarrow r)) \rightarrow r) \vdash z : Q^* : ((\forall \alpha.((P[\alpha] \rightarrow r) \oplus (P[\alpha^+] \rightarrow r)) \rightarrow r) \oplus ((P[\alpha^+][Q^*/\alpha] \rightarrow r)) \rightarrow r) \rightarrow r
\end{array}
\]

\[
\begin{array}{l}
\frac{\neg\neg \Theta; \Gamma^*, x : P[Q]^* \vdash a^* : r}{\neg\neg \Theta^*; \Gamma^* \vdash \lambda x. a^* : (P[\alpha]^*[Q^*/\alpha]) \rightarrow r} \\
\frac{\neg\neg \Theta^*; \Gamma^* \vdash \text{inl} (\lambda x. a^*) : ((P[\alpha]^*[Q^*/\alpha]) \rightarrow r) \oplus ((P[\alpha^+][Q^*/\alpha] \rightarrow r)) \rightarrow r}{\neg\neg \Theta^*; \Gamma^* \vdash z : \forall \alpha. ((\forall \alpha.((P[\alpha] \rightarrow r) \oplus (P[\alpha^+] \rightarrow r)) \rightarrow r) \vdash z : Q^* : (\text{inl} (\lambda x. a^*)) : r
\end{array}
\]
∀\(\alpha. P[\alpha], \Theta; \Gamma, x : P[M]\) ⊢ a

∀\(\alpha) • = ¬ ¬ \Theta •; \Gamma •, x : P[M] \vdash a • : r

IH

¬ ¬ \Theta •; \Gamma • ⊢ λ x. a • : (P[\alpha] • M[\alpha]/\alpha) ⊢ r

− intro

¬ ¬ \Theta •; \Gamma • ⊢ inr(λ x. a •) : (P[\alpha] • M[\alpha]/\alpha) ⊢ r ⊢ r

− elim

∀− elim
∀α. N[α], @ Q

\[
\begin{align*}
\text{∀α. } & N[α], \quad \mathbf{Q} \\
\otimes_{\Theta, \Gamma, \mathbf{z}} & : N[\mathbf{Q}] \vdash a
\end{align*}
\]

\[
\begin{array}{c}
\text{Θ; Γ, z : ∀α. N[α] \vdash } \mathbf{let} \ x = z \cdot \mathbf{Q} ; a
\end{array}
\]

\[
\begin{array}{c}
\text{\[α\], } @ Q
\end{array}
\]

\[
\begin{array}{c}
\Theta^*; z : \forall\alpha. ((N[\alpha] \rightarrow N[\alpha]) \rightarrow r) \rightarrow z : \forall\alpha. ((N[\alpha] \rightarrow N[\alpha]) \rightarrow r)
\end{array}
\]

\[
\begin{array}{c}
\text{∀− elim}
\end{array}
\]

\[
\begin{array}{c}
\text{∀\alpha. } N[\alpha], \quad \mathbf{Q} \\
\otimes_{\Theta, \Gamma, \mathbf{z}} & : N[\mathbf{Q}] \vdash a
\end{array}
\]

\[
\begin{array}{c}
\text{Θ}; \Gamma; \mathbf{z} : \forall\alpha. ((N[\alpha] \rightarrow N[\alpha]) \rightarrow r) \rightarrow z : \forall\alpha. ((N[\alpha] \rightarrow N[\alpha]) \rightarrow r) \vdash a^* : r
\end{array}
\]

\[
\begin{array}{c}
\text{IH}
\end{array}
\]

\[
\begin{array}{c}
\text{∀\alpha. } N[\alpha], \quad \mathbf{Q} \\
\otimes_{\Theta, \Gamma, \mathbf{z}} & : N[\mathbf{Q}] \vdash a
\end{array}
\]

\[
\begin{array}{c}
\text{Θ}; \Gamma; \mathbf{z} : \forall\alpha. ((N[\alpha] \rightarrow N[\alpha]) \rightarrow r) \rightarrow z : \forall\alpha. ((N[\alpha] \rightarrow N[\alpha]) \rightarrow r) \vdash a^* : r
\end{array}
\]

\[
\begin{array}{c}
\text{cut}
\end{array}
\]

\[
\begin{array}{c}
\text{∀\alpha. } N[\alpha], \quad \mathbf{Q} \\
\otimes_{\Theta, \Gamma, \mathbf{z}} & : N[\mathbf{Q}] \vdash a
\end{array}
\]
∀ \alpha. N[\alpha], \lambda x. \lambda x'. z @ M \vdash a \iff \lambda x. \lambda x'. z @ M \implies a = \neg \neg \Theta \land \lambda x. \lambda x'. z @ M \proves z @ M \implies a