Object-Level Reasoning with Logics Encoded in HOL Light

OPEN ACCESS

Object-Level Reasoning with Logics Encoded in HOL Light

Petros Papapanagiotou

School of Informatics
University of Edinburgh
Edinburgh, United Kingdom

ppapapan@inf.ed.ac.uk

Jacques Fleuriot

School of Informatics
University of Edinburgh
Edinburgh, United Kingdom

jdf@inf.ed.ac.uk

We present a generic framework that facilitates object level reasoning with logics that are encoded within the Higher Order Logic theorem proving environment of HOL Light. This involves proving statements in any logic using intuitive forward and backward chaining in a sequent calculus style. It is made possible by automated machinery that take care of the necessary structural reasoning and term matching automatically. Our framework can also handle type theoretic correspondences of proofs, effectively allowing the type checking and construction of computational processes via proof. We demonstrate our implementation using a simple propositional logic and its Curry-Howard correspondence to the λ-calculus, and argue its use with linear logic and its various correspondences to session types.

1 Introduction

Higher order logic (HOL) proof assistants, such as HOL Light [13], provide the means to encode other languages and prove conjectures specified within them (*object-level reasoning*). This can help understand how the objects of an encoded logic behave or develop practical applications using that logic, such as the development of correct-by-construction programs, type checking, or verification of specific terms or programs (such as the work on linear logic in Coq by Power et al.[31]).

In practice, performing proofs within an encoding of a custom logic in HOL typically requires the development of specialised tools and tactics, in what can be seen as *a theorem prover within a theorem prover*. These include, for example, mechanisms for fine-grained structural manipulation of the encoded formulae, customised application of the inference rules, seamless backward and forward chaining, etc. Users often need to develop such tools in an ad-hoc way in order to reason specifically about the logic they are interested in, and this drastically increases the effort and time required to obtain a useful encoding. Moreover, such tools may not scale across other logics, leading to a replication of effort.

We present a generic framework and toolset for object-level reasoning with custom logics encoded in HOL Light. It aims to facilitate the exploration of different logic theories by minimizing the effort required between encoding the logic and obtaining a proof. Assuming a sequent calculus style encoding of the logic's inference rules, it allows their direct and intuitive application both backwards and forwards, while the system automates most of the structural reasoning required. The implemented tactics can also be used programmatically to construct automated proof procedures.

An important part of our framework, which sets it apart from similar systems like Isabelle (see Section 5.1), is the handling of type theoretical correspondences between encoded logics and programs. These are inspired by the *propositions-as-types* paradigm (or *Curry-Howard correspondence*) between intuitionistic logic proofs and the λ-calculus [16]. They enable the construction of executable terms in some calculus via proof, also referred to as *computational construction*. Terms are attached on each logical formula, so that the formula represents the type of the term. The application of inference rules on

such a term annotated with its type, results in the construction of a more complex computational expression with some guaranteed correctness properties. Typically, cut-elimination in the proof corresponds to a reduction or execution step in the constructed term.

Our framework allows the user to easily construct computational terms and test and compare different logics and the behaviour of their correspondences (or even multiple correspondences of the same logic at the same time), merely by manipulating the encoding of the inference rules rather than having to rebuild or repurpose the implemented tools each time. More specifically, our framework allows 2 types of proofs:

- *Type checking* proofs involve conjectures whose computational translation is given and the proof can only succeed if the conjecture is provable and the translation is correct. Such proofs effectively verify the types of programs.

- *Construction* proofs involve conjectures whose computational translation is unknown and is constructed as the proof progresses. Such proofs allow us to construct programs that match a given type via proof.

One of the main motivations of our work lies in the variety of computational translations between linear logic and session types, which we discuss briefly in Section 5.

2 Example: Simple Logic

In order to provide simple examples of the functionality of our system, we focus on a subset of propositional logic involving only conjunction ($\times$) and implication ($\rightarrow$). More specifically, the terms of our example propositional logic can be defined as follows, using True (**T**) as a constant/bottom object:

$$Prop ::= \mathbf{T} \mid Prop \times Prop \mid Prop \rightarrow Prop \tag{1}$$

Our system is tailored to sequent calculus formulations, mainly because sequent calculus deduction is easier to encode as an inductive definition within HOL (and particularly in HOL Light). Note, however, that we use an intuitionistic sequent calculus which is equivalent to natural deduction [10].

The inference rules of our simple logic are shown in Figure 1. Note that Γ and Δ represent *contexts*, i.e. finite sequences of formulae that are not affected by the application of the rule.

In the next sections we investigate how this example logic can be encoded in HOL Light and then how our framework enables object-level proofs automatically without any further coding.

2.1 Encoding

A standard way of encoding such a logic in HOL Light starts by defining the syntax of the terms, in our case as shown in (1), as an inductive type. Note that, for clarity, we typeset definitions and formulas using the standard logical symbols for the various connectives instead of the harder to read ASCII-based syntax of HOL Light. The actual HOL Light implementation is available in the code base (see Section 4) and we also point out mappings between standard logic and HOL Light syntax where necessary.

We then need to provide the means to describe a sequent based on logical consequence (denoted by $\vdash$). A sequent may contain multiple formulas, including multiple copies of the same formula (such as in the contraction rule C in Figure 1), and is thus often represented using a list of terms.

Like most logics, our example includes an *Exchange* rule (see Figure 1). This means that the order of the terms in the sequent is not important, so that $A, B \vdash C$ and $B, A \vdash C$ are semantically identical.

<div align="center">Identity & Cut</div>

$$\frac{}{A \vdash A} \; Ax \qquad\qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C} \; Cut$$

<div align="center">Structural Rules</div>

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} \; Exchange \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C} \; W \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \; C$$

<div align="center">Logical Rules</div>

$$\frac{\Gamma, A \vdash C}{\Gamma, A \times B \vdash C} \; L1\times \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \times B \vdash C} \; L2\times \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \times B} \; R\times$$

$$\frac{\Gamma, B \vdash C \quad \Delta \vdash A}{\Gamma, \Delta, A \to B \vdash C} \; L\to \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \; R\to$$

Figure 1: The inference rules of the subset of propositional logic used as an example.

For this reason, in paper proofs the *Exchange* rule is almost always used implicitly, especially in complex proofs which may require a large number of applications of *Exchange* to manipulate the sequents in the right form. We avoid their use altogether in our framework by replacing lists of formulas in a sequent with multisets[1]. In this case, the mechanism of bringing the sequents in the appropriate form relies on reasoning about and matching multisets, and these tasks can be automated more efficiently (see Section 4.2)[2].

An existing theory in HOL Light already has many useful formalised properties of multisets, and we added some more in our own small library extension. A variety of methods and tools, such as multiset normalisation, needed for our particular tasks were also implemented. Finally, we introduced HOL Light abbreviations for multiset sums (^) and singleton multisets ('), in order to obtain a cleaner syntax. In this paper though, for readability, we use the ⊎ symbol to denote a multiset sum, whereas enumerated multisets are enclosed in curly brackets $\{\cdot\}$.

Based on the above, the sequent $\Gamma, A \vdash B$ can be represented using $\vdash$ as an (infix) boolean function in the term $(\Gamma \uplus \{A\}) \vdash B$. Note that a context Γ is represented by a *multiset variable*, i.e. a variable of type $(Prop)multiset$, whereas formulas A and B are term variables of type $Prop$. Also note that other logics may have additional or different types of arguments. For example, a regular (non-intuitionistic) sequent calculus will need 2 multisets of terms (left and right side) and some formulations of linear logic may need 4 multisets of terms (to distinguish linear from non-linear contexts). Our framework can deal with any of these situations with no additional specification other than the definition of $\vdash$ as a function.

Using this representation of sequents, the inference rules of the logic can be encoded as an inductive definition of logical consequence $\vdash$ as a function |-- in HOL Light. Note that this is different from HOL Light's syntax for entailment in proven HOL theorems |-, which we omit here altogether to prevent

---

[1]Sets may also be used to eliminate the need for weakening and contraction rules, but we prefer multisets as a more general approach that allows us to encode Linear Logic (see Section 5).

[2]However rare, non-commutative (or non-associative) substructural logics do exist, such as Lambeck's calculus of syntactic types [18]. These can still be expressed in our framework using lists and will work with our tactics, but will simply not take advantage of the specialised multiset matching.

any ambiguity. In the encoding, HOL implication ($\Longrightarrow$) expresses valid derivations between sequents, making HOL the effective meta-logic. As an example, the $R\times$ rule from Figure 1 can be specified in the definition of $\vdash$ as follows:

$$\Gamma \vdash A \Longrightarrow \Delta \vdash B \Longrightarrow (\Gamma \uplus \Delta) \vdash A \times B \tag{2}$$

2.2 Object-level proofs

So far we covered a straightforward encoding of a logic in HOL Light, with tools that are already available. Given such an encoding, one would expect to be able to do some object-level proofs. Let us take the commutativity of conjunction ($\times$), i.e. $\vdash X \times Y \to Y \times X$ as an example of such a proof:

$$\frac{\dfrac{\dfrac{\overline{X \vdash X} \ Ax}{X \times Y \vdash X} L1\times \quad \dfrac{\overline{Y \vdash Y} \ Ax}{X \times Y \vdash Y} L2\times}{\dfrac{X \times Y, \ X \times Y \vdash Y \times X}{X \times Y \vdash Y \times X} R\times}}{\vdash X \times Y \to Y \times X} \ C}{R \to} \tag{3}$$

Using vanilla HOL Light, proof (3) can be accomplished using the interactive steps shown in Figure 2. The actual proof script is available online and executed interactively (see link in Section 4).

Step	Tactic	Goal(s)
	Initial state	$\varnothing \vdash X \times Y \to Y \times X$
1	`MATCH_MP_TAC` ($R\to$)	$\varnothing \uplus \{X \times Y\} \vdash Y \times X$
2	`MATCH_MP_TAC` (C)	$\varnothing \uplus (\{X \times Y\} \uplus \{X \times Y\}) \vdash Y \times X$
3	`REWRITE_TAC[MUNION_EMPTY2]`	$\{X \times Y\} \uplus \{X \times Y\} \vdash Y \times X$
4	`MATCH_MP_TAC` ($R\times$)	$\{X \times Y\} \vdash Y \wedge \{X \times Y\} \vdash X$
5	`CONJ_TAC`	$\{X \times Y\} \vdash Y \qquad \{X \times Y\} \vdash X$
6	`SUBGOAL_THEN` ($\varnothing \uplus \{X \times Y\} \vdash Y$)...	$\varnothing \uplus \{X \times Y\} \vdash Y \qquad \{X \times Y\} \vdash X$
7	`MATCH_MP_TAC` ($L2\times$)	$\varnothing \uplus \{Y\} \vdash Y \qquad \{X \times Y\} \vdash X$
8	`REWRITE_TAC[MUNION_EMPTY2]`	$\{Y\} \vdash Y \qquad \{X \times Y\} \vdash X$
9	`MATCH_ACCEPT_TAC` (ID)	$\{X \times Y\} \vdash X$
10	`SUBGOAL_THEN` ($\varnothing \uplus \{X \times Y\} \vdash X$)...	$\varnothing \uplus \{X \times Y\} \vdash X$
11	`MATCH_MP_TAC` ($L1\times$)	$\varnothing \uplus \{X\} \vdash X$
12	`REWRITE_TAC[MUNION_EMPTY2]`	$\{X\} \vdash X$
13	`MATCH_ACCEPT_TAC` (ID)	

Figure 2: Interactive steps in HOL Light performing proof (3).

Although there are 7 rule applications in the original proof, the HOL Light script contains almost double the steps even for such a simple, straightforward proof.

The main tactic that allows the application of inference rules is `MATCH_MP_TAC`, which matches the consequent of a HOL implication in a theorem with the goal and sets the antecedent as the new goal. Our rules are expressed using implication, so `MATCH_MP_TAC` fits well for this use. However, the consequent and goal must match exactly for the tactic to work. This forces us to constantly manipulate the structure of the goal to ensure the rule is applied correctly.

Step 1 is straightforward. Observing the $R \to$ rule in Figure 1 and the initial state, it is clear that Γ in the rule is matched to the left-hand side of the turnstile, i.e. $\varnothing$. The new goal is produced by the

antecedent of the rule $\Gamma, A \vdash B$, where the left-hand side becomes $\Gamma \uplus \{A\}$, i.e. $\varnothing \uplus \{X \times Y\}$. The empty multiset is therefore carried forward in the new goal.

This causes a problem after Step 2, where we want to apply the $R\times$ rule. If we were to immediately apply `MATCH_MP_TAC`, it would yield the instantiation $\Gamma = \varnothing$ and $\Delta = \{X \times Y\} \uplus \{X \times Y\}$. However, we would rather have a different context split, i.e. one where $\Gamma = \{X \times Y\}$ and $\Delta = \{X \times Y\}$. In this particular case, this is easily solved by eliminating the empty multiset via rewriting with the theorem `MUNION_MEMPTY2`[3], i.e. the property $\forall M. \varnothing \uplus M = M$. In other cases, this is not as simple to solve. For example, in Steps $6-7$ we need to reintroduce an empty multiset to match with Γ in the $L2\times$ rule through a new subgoal (using `SUBGOAL_TAC`).

Even through this simple example, it is clear that performing such object-level proofs with an encoded logic can be tedious. We need to be able to manipulate the goal state to allow the rules to match, which often requires multiple steps using rewritting or appropriate rule instantiations. This effort put into the management of multiset-based context is also encountered in related work [31].

Our framework facilitates this process, alleviating the need for such fine-grained manipulation. More specifically, it provides both forward and backward reasoning with simple procedural tactics. It also performs intelligent multiset matching automatically, performing all the necessary manipulations (such as adding empty multiset as required) in the background. Using our framework, the proof script from Figure 2 is simplified as shown in Figure 3. Notice how each rule application now only requires a single proof step in HOL Light, using our `ruleseq` command described in Section 4.

Step	Tactic	Goal(s)
	Initial state	$\varnothing \vdash X \times Y \rightarrow Y \times X$
1	`ruleseq` $(R\rightarrow)$	$\{X \times Y\} \vdash Y \times X$
2	`ruleseq` (C)	$\{X \times Y\} \uplus \{X \times Y\} \vdash Y \times X$
3	`ruleseq` $(R\times)$	$\{X \times Y\} \vdash Y \qquad \{X \times Y\} \vdash X$
4	`ruleseq` $(L2\times)$	$\{Y\} \vdash Y \qquad \{X \times Y\} \vdash X$
5	`ruleseq` (ID)	$\{X \times Y\} \vdash X$
6	`ruleseq` $(L1\times)$	$\{X\} \vdash X$
7	`ruleseq` (ID)	

```
let TIMES_COMM = prove_seq ( '∅ ⊢ X×Y → Y×X',
  ETHENL (ETHEN (ruleseq R→) (ETHEN (ruleseq C) (ruleseq R×)))
    [ ETHEN (ruleseq L2×) (ruleseq ID) ;
      ETHEN (ruleseq L1×) (ruleseq ID) ] );;
```

Figure 3: Interactive (top) and packaged (bottom) steps performing proof (3) using our framework.

3 Example: The Curry-Howard isomorphism

Howard's seminal paper describes the use of first order logic as a system for simply typed λ-calculus and studies the correspondence of cut elimination to function evaluation in what came to be known as the Curry-Howard correspondence [16]. Howard uses natural deduction for his formulation, noting how it is "inappropriate" for Gentzen's sequent calculus. Girard delves deeper into this distinction between

---

[3]`https://github.com/PetrosPapapa/hol-light-tools/blob/91f3f1b030728e0edf3ec86732d185207839a8ad/Library/multisets.ml#L75`

the two proof systems and provides a translation from natural deduction to intuitionistic sequent calculus (i.e. sequents with a single conclusion) [10].

To simplify the discussion, we focus on a subset of this type theory involving only function and product types as primitives, to mirror the simple logic presented in the previous section, but also in the spirit of Wadler's discussion of the same topic [34]. The corresponding λ-calculus is defined with the following syntax:

$$Lambda ::= Var\ A \mid Lambda\ Lambda \mid (Lambda, Lambda) \mid \lambda A.\ Lambda$$

This includes variables (*Var*), function application ($f\ x$), products (x, y), and functions ($\lambda x.\ y$). Note that this type definition is polymorphic with respect to the type of variables A. This allows us to describe the (simple) types of this calculus using any datatype (though typically one would use strings).

We also define two projection functions for products fst and snd so that $fst\ (x, y) = x$ and $snd\ (x, y) = y$. Normally, projections are added as a primitive constructor of the inductive type. Cut elimination can then be used to prove what we give by definition here. Since we will not be performing cut elimination proofs, we employ these definitions as a more pragmatic approach.

Using the above syntax and mirroring the rules shown in Figure 1, the rules of this type theory are shown in Figure 4.

<div align="center">

Identity & Cut

$$\dfrac{}{x{:}A \vdash x{:}A}\ Ax \qquad\qquad \dfrac{\Gamma \vdash z{:}A \quad \Delta, z{:}A \vdash x{:}C}{\Gamma, \Delta \vdash x{:}C}\ Cut$$

Structural Rules

$$\dfrac{\Gamma, x{:}A, y{:}B, \Delta \vdash z{:}C}{\Gamma, y{:}B, x{:}A, \Delta \vdash z{:}C}\ Exchange \qquad \dfrac{\Gamma \vdash z{:}C}{\Gamma, x{:}A \vdash z{:}C}\ W \qquad \dfrac{\Gamma, x{:}A, x{:}A \vdash z{:}C}{\Gamma, x{:}A \vdash z{:}C}\ C$$

Logical Rules

$$\dfrac{\Gamma, fst\ x{:}A \vdash z{:}C}{\Gamma, x{:}A \times B \vdash z{:}C}\ L1\times \qquad \dfrac{\Gamma, snd\ x{:}B \vdash z{:}C}{\Gamma, x{:}A \times B \vdash z{:}C}\ L2\times \qquad \dfrac{\Gamma \vdash x{:}A \quad \Delta \vdash y{:}B}{\Gamma, \Delta \vdash (x,y){:}A \times B}\ R\times$$

$$\dfrac{\Gamma, fy{:}B \vdash z{:}C \quad \Delta \vdash y{:}A}{\Gamma, \Delta, f{:}A \to B \vdash z{:}C}\ L \to \qquad\qquad \dfrac{\Gamma, Var\ x{:}A \vdash y{:}B}{\Gamma \vdash \lambda x.y{:}A \to B}\ R \to$$

</div>

Figure 4: The inference rules of the Curry-Howard correspondence for our given subset of propositional logic.

It is worth mentioning the explicit distinction between any lambda term and specifically variables. The $R \to$ rule enforces that only variables can be entered in a λ expression, a property which is usually implicit in such formulations.

In the next sections, we describe how this encoding can be built on top of the previous one in HOL Light, and explain what kind of object-level proofs we would like to perform.

3.1 Encoding

The encoding of such a type theory can be very similar to the one we described in Section 2.1 for the same logic without computational components. The main difference is that in this case, our logical terms are augmented with lambda terms whose types they are describing. For this reason, we define the operator : (represented as :: in HOL Light to avoid a conflict with its own type annotation) as an infix term constructor.

The inference rules are defined inductively in the same way, simply using annotated terms instead of propositions. As an example, the $R\times$ rule is defined as follows:

$$\Gamma \vdash x\!:\!A \Longrightarrow \Delta \vdash y\!:\!B \Longrightarrow (\Gamma \uplus \Delta) \vdash (x,y)\!:\!A \times B \tag{4}$$

It is worth noting that in other types of computational correspondence, such as those involving linear logic (see Section 5), calculus terms can be attached to the entire sequent (as opposed to formulas within it). We accomplish this by adding an additional argument to the consequence function (i.e. to $\vdash$).

3.2 Object-level proofs

As mentioned in Section 1, there are 2 types of proofs that can be performed at the object-level with such a logic: *type checking* and *construction proofs*.

The example proof of commutativity of $\times$ from Section 2.2 can be used to type check the function $\lambda x.(snd(Var\,x), fst(Var\,x))$ as follows:

$$\cfrac{\cfrac{\cfrac{\overline{fst(Var\,x)\!:\!X \vdash fst(Var\,x)\!:\!X}\ Ax}{Var\,x\!:\!X \times Y \vdash fst(Var\,x)\!:\!X}\ L1\times \quad \cfrac{\overline{snd(Var\,x)\!:\!Y \vdash snd(Var\,x)\!:\!Y}\ Ax}{Var\,x\!:\!X \times Y \vdash snd(Var\,x)\!:\!Y}\ L2\times}{\cfrac{Var\,x\!:\!X \times Y,\ Var\,x\!:\!X \times Y \vdash (snd(Var\,x), fst(Var\,x))\!:\!Y \times X}{\cfrac{Var\,x\!:\!X \times Y \vdash (snd(Var\,x), fst(Var\,x))\!:\!Y \times X}{\vdash \lambda x.(snd(Var\,x), fst(Var\,x))\!:\!X \times Y \to Y \times X}\ R\to}\ C}\ R\times} \tag{5}$$

Notice how, as far as the logical derivation is concerned, proofs (3) and (5), i.e. the proof with and without computational annotations, are identical (although type checking a different function of the same type would require a different proof). Our framework reflects this property, since the proof script from Figure 3 can be used *exactly as is* to perform proof (5).

The equivalent construction proof aims to construct the term corresponding to the type $X \times Y \to Y \times X$ from the proof instead of knowing it a priori. We can express such proofs using existential quantification in the meta-logic (HOL) to set the following goal:

$$\exists f.\ \vdash f\!:\!X \times Y \to Y \times X$$

Although type checking proofs can be performed in a relatively straightforward way, using appropriate rule applications that match not only the logical but also the computational terms, *construction proofs* can be challenging to do even on paper. Although the logical derivation relies only on the logical terms, the construction of the computational term requires careful tracking of metavariables across the entire proof. For example, the above proof can only be completed by (iteratively) instantiating f to $\lambda x.(snd(Var\,x), fst(Var\,x))$.

In our framework, the same proof script from Figure 3 can be used for the construction proof. The construction is performed automatically in the background, as described in Section 4.4, and the constructed term can be extracted once the proof is complete with a simple instantiation of f.

4 Implementation

In this Section, we describe the key implemented features of our framework. These include the following:

1. Adapted procedural tactics for forward and backward reasoning inspired from Isabelle (Section 4.1).

2. Smarter matching of sequents using multiset matching (Section 4.2) and metavariable unification (Section 4.4).

3. Updated functions for proof state management and tactic application to allow arbitrary extensions of the proof state (Section 4.3).

4. Better management of metavariables, ensuring they are carried forward in the extended proof state for each subgoal (Section 4.3).

5. Functions that facilitate the extraction of constructed components (Section 4.4).

The full implementation, including the examples included in this paper in full detail, can be found online. The code base is separated into a library of general purpose tools[4] (for instance including the extended tactic system described in Section 4.3 and the multiset theorems used in Section 4.2), and the logic encoding library itself[5]. An online tutorial providing a more hands-on guide to encoding the logics described in this paper and the use of our framework is also provided[6].

4.1 Procedural tactics

Traditional use of HOL systems, such as HOL4 and HOL Light, dictates the use of tactics for backwards reasoning and so called *rules* for forward reasoning [13]. Tactics are applied on a goal and produce a (possibly empty) set of new subgoals, whereas *rules* are functions that combine one or more theorems or assumptions to derive new facts. One of the main problems with this approach, particularly in the context of an encoded logic, is that every inference rule needs to be expressed both as a new tactic that applies the inference backwards (as we would typically want a lot more flexibility than what is offered by `MATCH_MP_TAC`) and as a new rule/function that applies the inference forwards. In our example, this would require 9 new tactics and 9 new rules/functions for the primitive inference rules of our simple logic in Figure 1 (except the *Exchange* rule), plus a new tactic and a new function for each derived rule. Our Curry-Howard encoding from Figure 4 would require a different implementation of another 9 tactics and 9 rules.

An alternative, more flexible approach can be found in the procedural proof tactics for natural deduction in Isabelle [29], namely `rule`, `erule`, `drule` and `frule`. These enable the usage of any arbitrary theorem in a proof, either as a forward reasoning step (manipulating assumptions – `drule` and `frule`), as a backwards reasoning step (breaking down the goal – `rule`), or simultaneous forward and backwards reasoning (`erule`). They essentially separate the mechanism of matching a rule to a particular goal state and the rule itself. There are also the four alternatives `rule_tac`, `erule_tac`, `drule_tac`, and `frule_tac`, which can be used to partially instantiate a rule before applying it (for example in order to resolve ambiguities when a rule can be matched to a particular proof state in more than one way). Using these tactics, one can manipulate custom inference rules from any encoded logic.

For this reason, we make use of our *Isabelle Light* framework [23], which emulates the aforementioned Isabelle tactics in HOL Light. It also includes a few tactics for managing metavariables, which are key to this work (see Section 4.3).

---

[4] `https://github.com/PetrosPapapa/hol-light-tools`
[5] `https://github.com/PetrosPapapa/hol-light-embed`
[6] `https://petrospapapa.github.io/hol-light-embed/CurryHoward.html`

The extension of Isabelle Light with the new features described in this work resulted in a new set of procedural tactics tailored to object level reasoning with encoded sequent calculus logics. The new tactics with the additional functionality are marked with a "seq" tag in their name. For example, `ruleseq` is the extension of `rule`, whereas `rule_seqtac` is the extension of `rule_tac` (and similarly for `erule`, `drule`, and `frule`).

4.2   Multiset matching

In Section 2.1, we discussed the use of multisets to represent sequents in order to avoid the *Exchange* rule, followed by examples of the structural manipulation needed in proofs in Section 2.2. When using inference rules in our system we aim to match the multisets describing the involved formulas appropriately, with minimum effort from the user.

Let us take the $L1\times$ rule as an example:

$$\frac{\Gamma, A \vdash C}{\Gamma, A \times B \vdash C} \; L1\times$$

When matching the context of the conclusion, i.e. $\Gamma \uplus \{A \times B\}$, we need to find matches for the Γ multiset and the $\{A \times B\}$ term.

Let us consider a goal with context $\{x \times y\}$ for some x, y. Our goal here is to obtain the instantiation $\{A/x, \; B/y\}$ in order to split the conjunction. It is obvious, however, that the multisets do not match directly, as the goal is not a sum $\uplus$. Although $\{A \times B\}$ matches with $\{x \times y\}$, there is no component to match with Γ. Instead, we need to introduce an empty multiset so that the goal's context becomes $\varnothing \uplus \{x \times y\}$.

If, instead, the context of the goal was $\{x \times y\} \uplus \{z\}$ for some z, the structure would match the expected sum. However, if we match the multisets directly we would obtain the instantiation $\{\{x \times y\}/\Gamma\}$ and then our match would fail as z and $A \times B$ do not match.

Finally, if the context of the goal was a larger sum, such as $\{x \times y\} \uplus \{z\} \uplus \{w\}$ for some w, then we would want Γ to be matched to the whole of $\{z\} \uplus \{w\}$.

We therefore implement an algorithm that properly matches multiset parts of sequents by incorporating commutativity and associativity of multiset sum $\uplus$. Performing this type of *AC-matching* is a well studied problem [17]. Our particular case incolves matching of terms with no shared variables (eliminating the need for occurs checks) and no nested AC functions (we only have a single flat level of multisets), making this problem more tractable than the general case and solvable in P. Our algorithm splits the multisets into their elements, which are either singleton multisets (formulas) or multiset variables (environments). Then it performs the following matches:

1. First it tries to match elements of the multiset taken from the inference rule that do not contain free variables, i.e. constants and terms pre-instantiated by the user.

2. Then it tries to match elements that are not variable multisets. In our example, $\{A \otimes B\}$ is a singleton set that will be matched first.

3. Multiset variables (such as Γ in our example) are left for last because they can match any part of the target. If the target does not have enough elements for all such multiset variables, they are matched to the empty multiset, whereas if there are more elements left in the goal than the available variables, they are combined into a single multiset sum.

If a match is found for all elements of the rule, the rule is instantiated accordingly. We then apply multiset normalisation via rewriting with the properties of multiset sum, which leads to the rule and the target (goal and assumption) to match exactly, thus allowing the appropriate LCF style justification of the rule application.

As an extra step, we eliminate any remaining empty multisets from the resulting subgoals. These are viewed as part of the internal mechanics of structural manipulation and outside of the encoded logic. They should not be of interest to the user, instead making the proof goal more confusing.

It is worth noting that our algorithm does not currently support backtracking when matching multiset variables. It merely attempts to find one possible match, under the assumption that if the user needs a different match, they will explicitly instantiate the rule (see Section 4.1) to guide the algorithm accordingly. Supporting backtracking is non-trivial as it requires an explicit interaction with the proof state and the tactic application mechanisms, but we are considering it as future work as it can prove particularly useful in automated proof procedures.

4.3 Metavariables

Metavariables in a proof are variables whose instantiations are deferred to later stages in the proof or until the proof is finished. They are particularly useful in a *construction* proof, where the computational translation (i.e. the λ-calculus term in our example) is initially unknown and constructed during the proof (see Section 3.2). It is worth noting that, although metavariables exist as a feature in HOL Light, they are rarely used and are seen as "a bit of a historical accident"[7]. However, in our case they are essential in order to construct computational translations.

Metavariables are shared by all subgoals of a proof, and thus HOL Light stores them beyond the scope of any single subgoal. In contrast, HOL Light tactics are functions that apply to a single subgoal. As a result, tactics have no access to any information on *already existing* metavariables. This causes problems during the application of our tactics, since they need to (a) be able to freely instantiate metavariables to anything that matches and (b) ensure that metavariables that are newly introduced during construction are fresh.

We address this issue by enhancing HOL Light so that tactics can handle subgoals extended with additional information about the proof state (as a state monad). We call these extended tactics `etactics` and their definition, contrasted to the definition of a regular HOL Light `tactic` is shown below:

```
type tactic = goal -> goalstate ;;
type 'a etactic = 'a -> goal -> (goalstate * 'a) ;;
```

Note that `'a` is a type variable that allows the state to be of any type. In our case, this extension allows for our tactics to be given (a) an integer counter that ensures freshness of new variables (similarly to the `genvar` mechanism of HOL Light) and (b) the list of currently used metavariables, in addition to the target subgoal.

The functions `eseq`, `prove_seq`, `ETHEN`, `ETHENL`, `EORELSE`, etc. shown in our examples are part of this extension, are therefore applicable to `etactics`, and correspond to the HOL Light functions `e` (apply a tactic interactively), `prove` (prove a lemma with a composite tactic), `THEN` (sequential application of tactics), `THENL` (apply a tactic and then apply a list of tactics, each to one of the produced subgoals), `ORELSE` (try to apply a tactic and if it fails apply a different tactic), etc. Note that, the original HOL Light tactics can be used within this extension with the `ETAC` keyword:

---

[7]Personal communication with John Harrison, 2009.

```
let (ETAC: tactic -> 'a etactic) = fun tac s g -> tac g,s ;;
```

Another challenge in the effective use of metavariables lies in the justification of every rule application. The LCF approach adopted in HOL Light requires that every tactic application can formally reproduce the original goal from its generated subgoals. Metavariables make this more challenging as the original goal and subgoals may mutate at any point based on the instantiation of their metavariables. The justification mechanisms for `ruleseq` and the other tactics successfully deal with this issue. Moreover, our efforts on this front uncovered a 20 year old bug with metavariable instantiations in HOL Light[8].

4.4   Construction

Computational construction in HOL Light proofs is enabled through the use of metavariables. Unknown computational components (i.e. λ-calculus annotations in our example) are treated as metavariables that are gradually instantiated through the application of the inference rules. Note that when using an inference rule, any variables that are not matched to any (sub)terms of the assumptions or the goal are added as metavariables in the new subgoals. At the end of the proof, applying the resulting instantiations to the initial goal's metavariables gives us the computational terms resulting from that proof.

To demonstrate how such construction proofs work in our framework, let us return to the proof of commutativity of $\times$ discussed in Section 3.2:

$$\exists f. \ \vdash f : X \times Y \to Y \times X \tag{6}$$

First, we use the HOL Light `META_EXISTS_TAC` tactic to eliminate the existential quantifier and add the variable f to the list of metavariables, which allows it to be instantiated gradually in the attempt to find the correct witness. We then perform a backwards proof, by applying the associated inference rules. The first step, involves the $R \to$ rule:

$$\frac{\Gamma, \ Var\, x : A \vdash y : B}{\Gamma \vdash \lambda x.y : A \to B} \ R \to$$

The backwards application of the rule (using `ruleseq`) tries to match the conclusion of the rule $\Gamma \vdash \lambda x.y : A \to B$ to the current goal $\vdash f : X \times Y \to Y \times X$. Knowing that f is a metavariable, the framework uses *unification* instead of matching. This yields the following instantiation:

$$\{\lambda x.y/f, \ X \times Y/A, \ Y \times X/B, \ \varnothing/\Gamma\} \tag{7}$$

Note that x and y are variables in the rule that are not matched to any subterms in the goal, so they are added as metavariables in the new goal $Var\, x : X \times Y \vdash y : Y \times X$.

The contraction (C) step simply yields the goal $Var\, x : X \times Y, \ Var\, x : X \times Y \vdash y : Y \times X$. We then want to apply the $R\times$ rule (with all variables primed for freshness):

$$\frac{\Gamma' \vdash x' : A' \quad \Delta' \vdash y' : B'}{\Gamma', \ \Delta' \vdash (x',y') : A' \times B'} \ R\times$$

Following the same unification mechanism as before, we obtain the new instantiation:

$$\{(x',y')/y, \ Y/A', \ X/B', \ \{Var\, x : X \times Y\}/\Gamma', \ \{Var\, x : X \times Y\}/\Delta'\} \tag{8}$$

---

[8] https://github.com/jrh13/hol-light/pull/52

At this point, we can observe the gradual instantiation of f, by composing the 2 instantiations (7) and (8), yielding $\{\lambda x.(x', y')/f\}$.

The rest of the proof yields further instantiations of the metavariables x' and y', eventually resulting in the final instantiation for f, namely $\lambda x.(snd(Var\ x), fst(Var\ x))$. It is worth noting that x is still a metavariable in this final result, having never been instantiated to anything else in the proof.

This process demonstrates a fruitful use of metavariable unification and instantiation to allow construction proofs. This is only achieveable thanks to the multiset matching and metavariable management functionality described in the previous sections.

One of the key benefits of this approach is that the user never deals with any of the computational terms explicitly. In fact, one can install a custom term printer in HOL Light to hide the λ terms completely from the proof, whilst the proof script remains unaffected. This is in-line with the natural expectation that the logical proof should not be affected by the computational annotations.

In addition to this, we have introduced a `constr_prove` command to further facilitate construction proofs. As previously mentioned, construction proofs produce existenatially quantified lemmas, such as (6). These are not very practical, as the constructed term is not visible in the lemma, but can only extracted through the metavariables in the finished proof state. The `constr_prove` command proves the existentially quantified goal, then strips the quantifiers and replaces the variables with the constructed terms. This results in a proven theorem with all the constructed components instantiated.

For instance, consider the following command, which proves lemma (6) using a packaged version of the proof script from Figure 3:

```
constr_prove ( '∃f. ∅ ⊢  f :X × Y → Y × X ',
    ETHENL (ETHEN (ruleseq R →) (ETHEN (ruleseq C) (ruleseq R×)))
      [ ETHEN (ruleseq L2×) (ruleseq ID) ;
        ETHEN (ruleseq L1×) (ruleseq ID) ] );;
```

This command will yield the theorem $\varnothing \vdash \lambda x.(snd(Var\ x), fst(Var\ x)) {:} X \times Y \to Y \times X$ directly.

In addition to making the user workflow for producing constructed lemmas easier, this facilitates the development of libraries of lemmas for different correspondences of the same logic, because the computational terms associated with each lemma do not need to be explicit. For example, if one chose to use a different correspondence of our simple logic, other than the Curry-Howard correspondence to λ-calculus, they would need to change the original encoding, by adapting the inductive definition of $\vdash$. However, the proof involving `constr_prove` shown above would still be valid and it would automatically generate the correct theorem based on the new correspondence.

4.5 Usage and Integration with HOL Light

The implemented generic tactics integrate well within the (extended) tactic system of HOL Light. More specifically, they can be used at different levels, such as the following:

1. **Interactively**: They allow the application of inference rules of the encoded logic in a step-by-step interactive proof setting, using the `eseq` command.

2. **In packaged proofs**: HOL Light proofs are traditionally packaged in a `prove` statement by combining the tactics that achieve the proof using the so-called *tacticals* such as `THEN` and `EVERY`. Our introduced tactics can also be combined in the same way, with the extended tacticals `ETHEN`, `EEVERY`, etc. For example, the packaged verification proof of the commutativity of $\times$ is shown at the bottom half of Figure 3.

3. **Programmatically**: Our procedural tactics facilitate the construction of advanced procedures for proofs in the encoded logics. For instance, they can be directly used within proof search algorithms, helping to reduce the overhead of manipulating the structure of the sequents. Our implementation includes an example of such an automated proof procedure of a type of Linear Logic proofs (see Section 5).

4. **Visually**: In related research, our tactics have proven useful in the context of diagrammatic reasoning [26]. In this, the user performs gestures in a purely graphical interface. Each gesture triggers a reasoning task in our encoded Linear Logic, and this is accomplished via the framework described in the current work.

5   Further Examples: Linear Logic

So far, our discussion revolved around a simple subset of propositional logic and its well studied correspondence to λ-calculus. The usefulness of our framework becomes more apparent when one considers that it can work in the exact same way with the encoding of any logic or correspondence, without the need for any further configuration or metadata of any kind.

A particular logic, which has multiple evolving correspondences, is linear logic [9], a substructural logic with no weakening or contraction rules. In the 90s, Abramsky, Bellin and Scott developed the *proofs-as-processes* paradigm [1, 2], introducing a correspondence between Classical Linear Logic and the π-calculus [21] and forming a type system for deadlock-free concurrent processes. We have used this paradigm in conjunction with our reasoning framework extensively for the specification and composition of workflow processes using Classical Linear Logic [25], with real-world applications in the modelling of clinical pathways in the healthcare domain [27, 24].

As the use of concurrent systems has scaled up dramatically in the past decade, reasoning about their properties, including attempts to ensure deadlock freedom and session fidelity, has been a major research track in concurrency theory for the past years and is an ongoing effort. This effort has brought forth the emergence of new correspondences of linear logic to *session types* [15], which are used to provide richer semantics for the types of communicated values and session-based protocols in mobile processes [8].

Caires et al. use Intuionistic Linear Logic terms to describe session types and attach them via a proofs-as-processes style to π-calculus channels [4]. Subsequent published papers describe further developments of this theory, including a version for asynchronous communication [7], a comparison to a Classical Linear Logic based version [5], and the use of dependent session types to describe properties about the information being communicated [32, 30]. In parallel to this track, Wadler developed a *propositions-as-sessions* theory [33]. In it, he chooses to loosen the connection to the original π-calculus by introducing a new process calculus named *CP*. Reductions in CP are *defined* based on cut-elimination steps in Classical Linear Logic, instead of being defined separately and then having their correspondence to cut-elimination proven. Further developments in the correspondence between linear logic and session types are being produced to this day [14].

Our framework has the capacity to work hand-in-hand with the meta-theoretic efforts in these strands of work, by providing a formal setting to produce object-level proofs. This can help generate actual process instances using linear logic proofs and examine their properties. For instance, this can shine light into how new session type languages, such as Wadler's CP, can behave in practice.

To demonstrate this, our implementation[9] includes 2 example encodings of linear logic. The first, is an encoding of Intuitionistic Linear Logic with no correspondence. The second is an encoding of

---

[9]`https://github.com/PetrosPapapa/hol-light-embed/blob/master/Examples/`

the *propositions-as-sessions* paradigm, i.e. a correspondence of Classical Linear Logic to CP. The latter includes an automated tactic that can prove a type of linear logic sequents within this encoding.

It is worth noting that both these examples and the Curry-Howard encoding described in this paper can all be loaded in HOL Light at the same time, and managed by the same set of tactics we discussed.

5.1 Related Work

There are strong similarities between our system and the core design of Isabelle [28, 29], which provides support for a variety of formal theories, such as Higher Order Logic (*Isabelle/HOL*) and Zermelo-Fraenkel set theory (*Isabelle/ZF*). Essentially, it exposes an intuitionistic meta-logic (*Isabelle/Pure*) that enables the encoding of different theories, and the development of sophisticated proof tactics and procedures for both interactive and automated theorem proving within those theories. Our work draws inspiration from this approach and enables similar functionality (for instance our encoding of Intuitionistic Linear Logic[10] strongly resembles the one in Isabelle[11]). The key difference lies in the ability to construct an initially unknown computation of a given type using any given set of primitive logic/type rules annotated with their computational translation. Simple synthesis using Constructive Type Theory has been shown to be possible in Isabelle[12], but has not been investigated to any depth since its formalisation almost 30 years ago by Paulson. Moreover, we use HOL Light's Higher Order Logic as the meta-logic which is more expressive than Isabelle, but narrows the kinds of logics whose use we can automate.

We employ a similar perspective in terms of multiset sequents to embed logics to that of Dawson and Goré's implementation in Isabelle/HOL [6]. While their work is further evidence of the need and usefulness of frameworks for arbitrary encoded logics, it focuses on reasoning about meta-theoretic properties. Instead, our framework is designed to tackle object-level reasoning and computational construction at that level.

Other systems that are known to rely on the propositions-as-types paradigm such as Coq [3] and Agda [22] have efficient, specialised procedures to deal with computational construction, but these are exclusive to their particular underlying type systems. They are not generic and so cannot be used to reason with different logics or correspondences.

Another similar approach is that of λProlog, a logic programming-based system aimed at hosting encodings of different logics and calculi [20]. For example, Forum is a linear logic based meta-logic built on top of λProlog [19]. Although λProlog allows more expressive encodings and provides proof search by default (based on its logic programming backend), it cannot support construction proofs because of its lack of support for metavariables (as argued by Guidi et al. who suggest potential extensions to achieve this [12]). Moreover, λProlog does not provide the guarantees of correctness of modern theorem provers, e.g. via the LCF approach [11], as is the case here with HOL Light.

In principle, other logical frameworks, such as Isabelle and Coq, would allow a similar implementation, and we believe some of the ideas and techniques presented here could transfer over. We chose HOL Light because it is a powerful system that allows interaction at a low level, resulting in much flexibility and programmability. For instance, it allows direct access to the proof structure and involved terms, which facilitates the development of sophisticated custom tactics or even an entire extension of the proof system as described in Section 4.3.

---

[10] https://github.com/PetrosPapapa/hol-light-embed/blob/master/Examples/ILL.ml
[11] https://isabelle.in.tum.de/dist/library/Sequents/Sequents/ILL.html
[12] https://isabelle.in.tum.de/dist/library/CTT/CTT

6   Conclusion

In summary, we presented a generic framework for object-level reasoning with encoded logics within HOL Light. We assume the inference rules of an encoded logic are encoded in a sequent calculus style, through the definition of logical inference as a function of logical terms.We focus on sequents that consist of multisets of terms, allowing their arbitrary ordering, though this is not necessarily a requirement.

The framework then exposes tactics, originally inspired by Isabelle's procedural natural deduction tactics, that allow intuitive forward and backward application of these rules. They automatically take care of structural reasoning and appropriate construction of the computational component. The latter provides the means for extracting correct-by-construction terms via any correspondence in the style of Curry-Howard. The tactics rely on a complex implementation involving multiset matching, unification, and metavariables. Although they work within an extended tactic system, they can integrate with the HOL Light proof environment, and can be used interactively, in packaged proofs, programmatically (for example to construct automated proof procedures), or even visually.

We demonstrated the functionality of the tactics through a simple propositional logic and its λ-calculus correspondence. Encodings of linear logic and its correspondence to the π-calculus or to session types, two of which are included as examples in the framework, showcase the usefulness of this work towards the study and development of deadlock-free software with session types.

Future goals include optimisations in terms of efficiency and metavariable management. We will also study the encoding of quantified object logics, as part of a more in-depth consideration of the full capabilities and limitations of the framework. Moreover, we are exploring ways to support natural deduction style rules in addition to the currently supported sequent calculus style.

We believe our implementation provides the basis for facilitated encodings of logics within HOL Light, without the need to reimplement an entire proof engine from scratch. This creates the potential to easily and effectively test the behaviour of different logics and their computational translations. Based on this, our framework can prove to be powerful tool for research in type theory and programming languages, in addition to its already successful application in process specification and composition and the healthcare domain.

Acknowledgement

References

[1] S. Abramsky (1994): *Proofs as processes*. *Theoretical Computer Science* 135(1), pp. 5–9, doi:10.1016/0304-3975(94)00103-0.

[2] G. Bellin & PJ Scott (1994): *On the π-calculus and linear logic*. *Theoretical Computer Science* 135(1), pp. 11–65, doi:10.1016/0304-3975(94)00104-9.

[3] Y. Bertot, P. Castéran, G. Huet & C. Paulin-Mohring (2004): *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, doi:10.1007/978-3-662-07964-5.

[4] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & Francois Laroussinie, editors: *CONCUR 2010 - Concurrency Theory*, *Lecture Notes in Computer Science* 6269, Springer Berlin Heidelberg, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[5] Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Towards concurrent type theory*. In: *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, ACM, New York, NY, USA, pp. 1–12, doi:10.1145/2103786.2103788.

[6] Jeremy E. Dawson & Rajeev Goré (2010): *Generic Methods for Formalising Sequent Calculi Applied to Provability Logic*. In Christian G. Fermüller & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 263–277, doi:10.1007/978-3-642-16242-8_19.

[7] Henry DeYoung, Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication*. In Patrick Cégielski & Arnaud Durand, editors: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, *Leibniz International Proceedings in Informatics (LIPIcs)* 16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 228–242, doi:10.4230/LIPIcs.CSL.2012.228.

[8] Simon J. Gay, António Ravara & Vasco T. Vasconcelos (2003): *Session Types for Inter-Process Communication*. Technical Report TR-2003-133, Department of Computing Science, University of Glasgow.

[9] Jean-Yves Girard (1995): *Linear Logic: its syntax and semantics*. In Jean-Yves Girard, Yves Lafont & Laurent Regnier, editors: *Advances in Linear Logic*, *London Mathematical Society Lecture Notes Series* 222, Cambridge University Press, doi:10.1017/CBO9780511629150.

[10] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press, New York, NY, USA.

[11] M. Gordon, R. Milner & C. P. Wadsworth (1979): *Edinburgh LCF: A Mechanized Logic of Computation (Lecture Notes in Computer Science)*, first edition. 78, Springer-Verlag, doi:10.1007/3-540-09724-4_3.

[12] Ferruccio Guidi, Claudio Sacerdoti Coen & Enrico Tassi (2016): *Implementing Type Theory in Higher Order Constraint Logic Programming*. Working paper or preprint.

[13] J. Harrison (1996): *HOL Light: A tutorial introduction*. *Lecture Notes in Computer Science*, pp. 265–269, doi:10.1007/BFb0031814.

[14] Bas van den Heuvel & Jorge A. Pérez (2020): *Session Type Systems based on Linear Logic: Classical versus Intuitionistic*. *Electronic Proceedings in Theoretical Computer Science* 314, p. 1–11, doi:10.4204/eptcs.314.1.

[15] Kohei Honda (1993): *Types for dyadic interaction*, pp. 509–523. Springer, Berlin, Heidelberg, doi:10.1007/3-540-57208-2_35.

[16] William A. Howard (1980): *The formulas-as-types notion of construction*. In J. P. Seldin & J. R. Hindley, editors: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, pp. 479–490. Reprint of 1969 article.

[17] Deepak Kapur & Paliath Narendran (1992): *Complexity of unification problems with associative-commutative operators*. *Journal of Automated Reasoning* 9(2), pp. 261–288, doi:10.1007/BF00245463.

[18] Joachim Lambek (1961): *On the calculus of syntactic types*. *Structure of language and its mathematical aspects* 166, p. C178, doi:10.1090/psapm/012.

[19] Dale Miller (1996): *Forum: A multiple-conclusion specification logic*. *Theoretical Computer Science* 165(1), pp. 201 – 232, doi:10.1016/0304-3975(96)00045-X.

[20] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*, 1st edition. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9781139021326.

[21] Robin Milner (1991): *The polyadic π-calculus: a tutorial*. *Logic and Algebra of Specification* 94, pp. 91–180, doi:10.1007/978-3-642-58041-3_6.

[22] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

[23] Petros Papapanagiotou & Jacques Fleuriot (2010): *An Isabelle-Like Procedural Mode for HOL Light*. In Christian G. Fermüller & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science 6397, Springer, pp. 565–580, doi:10.1007/978-3-642-16242-8_40.

[24] Petros Papapanagiotou & Jacques Fleuriot (2015): *Modelling and Implementation of Correct by Construction Healthcare Workflows*, pp. 28–39. Springer International Publishing, Cham, doi:10.1007/978-3-319-15895-2_3.

[25] Petros Papapanagiotou & Jacques Fleuriot (2019): *A Pragmatic, Scalable Approach to Correct-by-Construction Process Composition Using Classical Linear Logic Inference*. In Fred Mesnard & Peter J. Stuckey, editors: *Logic-Based Program Synthesis and Transformation*, Springer International Publishing, Cham, pp. 77–93, doi:10.1007/978-3-030-13838-7_5.

[26] Petros Papapanagiotou, Jacques Fleuriot & Sean Wilson (2012): *Diagrammatically-Driven Formal Verification of Web-Services Composition*. In Philip Cox, Beryl Plimmer & Peter Rodgers, editors: *Diagrammatic Representation and Inference*, Lecture Notes in Computer Science 7352, Springer Berlin Heidelberg, pp. 241–255, doi:10.1007/978-3-642-31223-6_25.

[27] Petros Papapanagiotou & Jacques D. Fleuriot (2013): *Formal verification of collaboration patterns in healthcare*. Behaviour & Information Technology, pp. 1–16, doi:10.1080/0144929X.2013.824506.

[28] Lawrence C. Paulson (1990): *Isabelle: The Next 700 Theorem Provers*. Logic and Computer Science 31, p. 361–386.

[29] Lawrence C. Paulson (1994): *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science 828, Springer, doi:10.1007/BFb0030541.

[30] Frank Pfenning, Luis Caires & Bernardo Toninho (2011): *Proof-Carrying Code in a Session-Typed Process Calculus*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs*, Lecture Notes in Computer Science 7086, Springer Berlin Heidelberg, pp. 21–36, doi:10.1007/978-3-642-25379-9_4.

[31] J. Power, C. Webster, C. Maynooth & I. Kildare (1999): *Working with Linear Logic in Coq*. In: *12th International Conference on Theorem Proving in Higher Order Logics*, Work-in-Progress Report.

[32] Bernardo Toninho, Luís Caires & Frank Pfenning (2011): *Dependent session types via intuitionistic linear type theory*. In: *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP '11, ACM, New York, NY, USA, pp. 161–172, doi:10.1145/2003476.2003499.

[33] Philip Wadler (2014): *Propositions as sessions*. Journal of Functional Programming 24(2-3), p. 384–418, doi:10.1017/S095679681400001X.

[34] Philip Wadler (2015): *Propositions As Types*. Commun. ACM 58(12), pp. 75–84, doi:10.1145/2699407.