Mechanisation of Model-theoretic Conservative Extension for HOL with Ad-hoc Overloading

Arve Gengelbach
Uppsala University, Uppsala, Sweden
arve.gengelbach@it.uu.se

Johannes Åman Pohjola
CSIRO's Data61, Sydney, Australia
University of New South Wales, Sydney, Australia
johannes.amanpohjola@data61.csiro.au

Tjark Weber
Uppsala University, Uppsala, Sweden
tjark.weber@it.uu.se

Definitions of new symbols merely abbreviate expressions in logical frameworks, and no new facts (regarding previously defined symbols) should hold because of a new definition. In Isabelle/HOL, definable symbols are types and constants. The latter may be ad-hoc overloaded, i. e. have different definitions for non-overlapping types. We prove that symbols that are independent of a new definition may keep their interpretation in a model extension. This work revises our earlier notion of model-theoretic conservative extension and generalises an earlier model construction. We obtain consistency of theories of definitions in higher-order logic (HOL) with ad-hoc overloading as a corollary. Our results are mechanised in the HOL4 theorem prover.

1 Introduction

Isabelle/HOL enriches higher-order logic with ad-hoc overloading. While other theorem provers of the HOL family support overloaded syntax through enhancements of parsing and pretty printing, in Isabelle/HOL overloading is a feature of the logic. The user-defined symbols are types and constants, and in Isabelle/HOL the latter may have multiple definitions for non-overlapping types. For instance, $+_{\alpha\to\alpha\to\alpha}$ is an overloaded constant with different definitions for different type instances of commutative monoids such as the natural numbers or the integers.

Overloaded definitions need further care as the defined symbols may be used prior to their definition, which if treated improperly may lead to cyclic definitions, i. e. unfolding of definitions might not terminate.

For a logic to be useful it should have unprovable statements. A logic is *consistent* if a contradiction cannot be deduced from any of its theories. HOL with user-defined types and constants without overloading is consistent. This can be proved by an argument based on *standard semantics* [16], where Booleans, function types and the equality constant are interpreted as expected, and type variables are interpreted as elements of a fixed universe of sets. The consistency story for HOL with overloading is a long one [20, 15, 12, 17]. Åman Pohjola and Gengelbach [17] prove HOL with overloading consistent in a machine-checked proof, by constructing models of theories of definitions through a construction that originates from Kunčar and Popescu [12]. Kunčar and Popescu introduce a *dependency relation* between the symbols of a theory to track dependencies of defined symbols on their definiens. Under an additional syntactic restriction on overloading [10], they construct a model for any (finite) theory of definitions for which the dependency relation is terminating.

Apart from consistency, a definitional mechanism should be *model-theoretically conservative*: any model of a theory can be extended to a model of the extended theory with new definitions, keeping interpretations of formulae that are independent of the new symbols intact. Informally, at least symbols that are independent of a theory extension may keep their interpretation in a model extension.

For HOL without overloading, model-theoretic conservativity holds unconditionally [8]. With overloading, model-theoretic conservativity holds for symbols that are independent of new definitions, as Gengelbach and Weber prove [4]. However, their proof was based on inherited wrong assumptions from Kunčar and Popescu [12], that Åman Pohjola and Gengelbach in their mechanised model construction [17] uncover and correct. Additionally, the mechanisation supports theory extension by the more expressive constant specification [2], which is a definitional mechanism also used in the theorem provers ProofPower and HOL4 to simultaneously introduce several new constants that satisfy some property.

This paper joins these two lines of work in mechanising that the definitional mechanisms of types and overloaded constants are model-theoretically conservative. The result holds for models that interpret constants introduced by constant specification equal to their witnesses and replaces the earlier monolithic model construction with an iterative one. An interpretation of this result is that the definitional mechanisms of Isabelle/HOL are semantically speaking robustly designed: at least symbols that are independent of an update may keep their interpretation in a model extension.

Even more generally, the syntactic counterpart of model-theoretic conservativity, *proof-theoretic (syntactic) conservativity* shall hold [20]. Informally, a definitional mechanism is *proof-theoretically conservative* if the definitional extension entails no new properties, except those that depend on the symbols which the extension introduces.

For Isabelle/HOL conservativity has been studied in an absolute manner, i. e. any definitional theory is a conservative extension of *initial HOL* [11], the theory of Booleans with Hilbert-choice and infinity axiom. Gengelbach and Weber [5] prove conservativity of any definitional extension above initial HOL, by translating a model-theoretic conservativity for a generalised semantics into its syntactic counterpart. As their semantics are similar to ours, this paper adds to the reliability of their result.

We describe the syntax and (lazy ground) semantics of HOL with ad-hoc overloading in Section 2. Subsequently, in Section 3 we recapitulate the *independent fragment* as the part of a theory that is independent of an extension by a new definition. This fragment is crucial in the iterative model construction, i. e. model-theoretic conservativity in Section 4. We discuss related work in Section 5. The definitions and theorems in this paper are formalised in the HOL4 theorem prover as part of the CakeML project.[1]

Contributions We make the following contributions

- We adapt and formalise the previously introduced *independent fragment* [4] (i. e. a theory's syntax fragment that is independent of a theory update) to support a more general definitional mechanism for constants: *constant specification* [2].
- We use the independent fragment to prove a notion of model-theoretic conservativity [4] in a new setting for the *lazy ground semantics* [17], which delays type variable instantiation and does not instantiate the type of term variables. To the best of our knowledge, this is the first mechanised conservativity result for a logic with overloaded definitions.
- Our work generalises and replaces the earlier monolithic model construction of Åman Pohjola and Gengelbach [17], and obtains consistency of HOL with ad-hoc overloading as a corollary.

[1]`https://code.cakeml.org/tree/master/candle/overloading/semantics/`

2 Background

In this section we introduce the syntax and semantics of HOL with ad-hoc overloading, which we inherit from the earlier work of Åman Pohjola and Gengelbach [17]. Their formalisation makes use of infrastructure from the formalisation of HOL Light (without overloading) by Kumar et al. [9], and the theoretical work on the consistency of HOL with ad-hoc overloading by Kunčar and Popescu [12].

2.1 Types and terms

Types Types, described by the grammar type = Tyvar string | Tyapp string (type list), are rank-1 polymorphic. Type variables Tyvar can be instantiated by a type substitution (ranged over by Θ), which extends homomorphically to type constructors Tyapp. For a type ty and a type substitution Θ, we call $\Theta\ ty$ a *(type) instance* of ty, denoted by $ty \geq \Theta\ ty$. Two types ty_1 and ty_2 are *orthogonal*, denoted by $ty_1 \ \#\ ty_2$, if they have no common instance. *Ground* types are those that contain no type variables, hence remain unchanged under any type substitution. A type substitution is *ground* if it maps every type to a ground type. As ground types only have trivial instances, any two ground types are either equal or orthogonal.

Terms Terms are simply typed λ-expressions, described by the grammar

$$\mathsf{term} \ = \ \mathsf{Var\ string\ type} \mid \mathsf{Const\ string\ type} \mid \mathsf{Comb\ term\ term} \mid \mathsf{Abs\ term\ term}$$

We only consider well-formed terms, that is, λ-abstractions must be of the form Abs $(\mathsf{Var}\ x\ ty)\ t$, i. e. have a term variable as first argument representing the binder. A closed term t, denoted closed t, contains only bound term variables. A term is welltyped if it has a type by the following rules (wherein $\rightarrow$ abbreviates the later introduced function type):

$$\frac{}{\mathsf{Var}\ n\ ty\ \mathsf{has\_type}\ ty} \qquad \frac{}{\mathsf{Const}\ n\ ty\ \mathsf{has\_type}\ ty}$$

$$\frac{s\ \mathsf{has\_type}\ (dty \rightarrow rty) \quad t\ \mathsf{has\_type}\ dty}{\mathsf{Comb}\ s\ t\ \mathsf{has\_type}\ rty} \qquad \frac{t\ \mathsf{has\_type}\ rty}{\mathsf{Abs}\ (\mathsf{Var}\ n\ dty)\ t\ \mathsf{has\_type}\ (dty \rightarrow rty)}$$

A well-typed term tm has a unique type which we denote typeof tm. Applying a type substitution Θ to a term means to apply Θ to the types within, e. g. $\Theta\ (\mathsf{Const}\ c\ ty) = \mathsf{Const}\ c\ (\Theta\ ty)$ (which we call *constant instance*) for a constant Const $c\ ty$. Orthogonality extends from types to constant instances:

$$\mathsf{Const}\ c\ ty_1 \ \#\ \mathsf{Const}\ d\ ty_2 \overset{\mathsf{def}}{=} c \neq d \vee ty_1 \ \#\ ty_2.$$

A user may introduce (non-built-in) types and constants by theory extension, as described in Section 2.3. For types and constants we generally say *symbols*.

Built-ins We abbreviate

Bool	for	Tyapp «bool» [ ]
$x \rightarrow y$	for	Tyapp «fun» $[x;\ y]$
Equal ty	for	Const «=» $(ty \rightarrow ty \rightarrow \mathsf{Bool})$
$s === t$	for	Comb (Comb (Equal (typeof s)) s) t

Any type with any of these type constructors at the top level is *built-in*, as is the constant Equal. These are the only symbols which are not user-defined. A *formula* is a term of type Bool.

For a set of types *tys* we consider its *built-in closure*, written $\mathsf{builtin\_closure}\,tys$:

$$\frac{}{\mathsf{Bool} \in \mathsf{builtin\_closure}\,tys} \qquad \frac{ty \in tys}{ty \in \mathsf{builtin\_closure}\,tys} \qquad \frac{ty_1 \in \mathsf{builtin\_closure}\,tys \quad ty_2 \in \mathsf{builtin\_closure}\,tys}{(ty_1 \to ty_2) \in \mathsf{builtin\_closure}\,tys}$$

Non-built-ins We define operators to collect the non-built-in types of terms and types, and also the non-built-in constants of terms. The list $x^\bullet$ consists of the outermost non-built-in types of a type or term x.

$$\begin{aligned} \mathsf{Bool}^\bullet &\stackrel{\mathsf{def}}{=} [\,] \\ (dom \to rng)^\bullet &\stackrel{\mathsf{def}}{=} dom^\bullet \mathbin{++} rng^\bullet \\ ty^\bullet &\stackrel{\mathsf{def}}{=} [ty] \quad \text{otherwise} \end{aligned} \qquad \begin{aligned} (\mathsf{Var}\; v_0\; ty)^\bullet &\stackrel{\mathsf{def}}{=} ty^\bullet \\ (\mathsf{Const}\; v_1\; ty)^\bullet &\stackrel{\mathsf{def}}{=} ty^\bullet \\ (\mathsf{Comb}\; a\; b)^\bullet &\stackrel{\mathsf{def}}{=} a^\bullet \mathbin{++} b^\bullet \\ (\mathsf{Abs}\; a\; b)^\bullet &\stackrel{\mathsf{def}}{=} a^\bullet \mathbin{++} b^\bullet \end{aligned}$$

As an example, the outermost non-built-in types of $map_{(\alpha \to \mathsf{Bool}) \to \alpha\ \mathsf{list} \to \mathsf{Bool}\ \mathsf{list}}$ over a polymorphic unary list type α list are:

$$\begin{aligned} &(\mathsf{Const}\ «\mathrm{map}»\ ((\mathsf{Tyvar}\ \alpha \to \mathsf{Bool}) \to (\mathsf{Tyvar}\ \alpha)\ \mathsf{list} \to \mathsf{Bool}\ \mathsf{list}))^\bullet = \\ &\ [\mathsf{Tyvar}\ \alpha;\ (\mathsf{Tyvar}\ \alpha)\ \mathsf{list};\ \mathsf{Bool}\ \mathsf{list}] \end{aligned}$$

Any type *ty* can be recovered from built-in types and the type's outermost non-built-in types:

$$\forall ty.\ ty \in \mathsf{builtin\_closure}\ (ty^\bullet)$$

For terms t, we define the list t° to contain all non-built-in constants of t:

$$\begin{aligned} (\mathsf{Comb}\; a\; b)^\circ &\stackrel{\mathsf{def}}{=} a^\circ \mathbin{++} b^\circ \\ (\mathsf{Abs}\ \_\ a)^\circ &\stackrel{\mathsf{def}}{=} a^\circ \end{aligned} \qquad \begin{aligned} (\mathsf{Var}\; x\; ty)^\circ &\stackrel{\mathsf{def}}{=} [\,] \\ (\mathsf{Equal}\; ty)^\circ &\stackrel{\mathsf{def}}{=} [\,] \\ (\mathsf{Const}\; c\; ty)^\circ &\stackrel{\mathsf{def}}{=} [\mathsf{Const}\; c\; ty] \quad \text{otherwise} \end{aligned}$$

2.2 Inference system

A *signature* is a pair of functions that assign type constructor names their corresponding arity and constant names their corresponding type. A *theory* is a pair (s,a) of a signature s and a set of terms (axioms) a. Gengelbach and Weber [4] consider a fixed signature, that is all symbols are initially declared, and a fixed set of axioms. Here instead, both the signature and the (possibly non-definitional) axioms may be extended (see Section 2.3). The functions axsof, tysof and tmsof return the respective components of a theory or signature.

Derivability of *sequents* is defined inductively as a ternary relation $(thy,hyps) \vdash p$ between a theory *thy*, a list of terms (hypotheses) *hyps* and a term (conclusion) p. We display three of the standard inference rules of higher-order logic, with their syntactic well-formedness constraints. The condition $\mathsf{type\_ok}\ (\mathsf{tysof}\ ctxt)\ ty$ requires that *ty* is either a type variable or a type constructor applied to the correct number of arguments, as indicated by its arity in the signature, and that these arguments are also type_ok. Similarly, $\mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ p$ requires that p is a well-typed term, and that its types and constants are instances from the given signature. Finally, $\mathsf{theory\_ok}\ ctxt$ requires that in the context *ctxt* all axioms are well-formed formulae, the theory has well-typed types and contains at least the built-in symbols.

$$\frac{\mathsf{theory\_ok}\ \mathit{thy} \quad p\ \mathsf{has\_type}\ \mathsf{Bool} \quad \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{thy})\ p}{(\mathit{thy},[p]) \vdash p}\ \mathsf{ASSUME}$$

$$\frac{\mathsf{theory\_ok}\ \mathit{thy} \quad \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy})\ \mathit{ty} \quad \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{thy})\ t}{(\mathit{thy},[\,]) \vdash \mathsf{Comb}\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ t)\ (\mathsf{Var}\ x\ \mathit{ty}) === t}\ \mathsf{ABS}$$

$$\frac{(\mathit{thy},h_1) \vdash l_1 === r_1 \quad (\mathit{thy},h_2) \vdash l_2 === r_2 \quad \mathsf{welltyped}\ (\mathsf{Comb}\ l_1\ l_2)}{(\mathit{thy},h_1 \cup h_2) \vdash \mathsf{Comb}\ l_1\ l_2 === \mathsf{Comb}\ r_1\ r_2}\ \mathsf{MK\_COMB}$$

2.3 Theory extensions

A theory is obtained from the empty theory by incremental *updates*. A list of updates is a *context*, and the function thyof returns the context's theory.

```
update =
   NewAxiom term
  | NewType string num
  | NewConst string type
  | TypeDefn string term string string
  | ConstSpec bool ((string × term) list) term
```

NewAxiom adds its argument formula to the theory's set of axioms. NewType and NewConst are type and constant *declarations*; they extend the theory's signature. The remaining TypeDefn and ConstSpec are *definitions* of a type and of constants, respectively. Definitions may extend both the signature and the set of axioms, and we defer their discussion to Sections 2.4 and 2.5.

The updates relation specifies when an update is a valid extension of a context:

$$\frac{\begin{array}{c}\mathit{prop}\ \mathsf{has\_type}\ \mathsf{Bool}\\ \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{ctxt})\ \mathit{prop}\end{array}}{\mathsf{NewAxiom}\ \mathit{prop}\ \mathsf{updates}\ \mathit{ctxt}} \qquad \frac{\begin{array}{c}\mathit{name} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty}\end{array}}{\mathsf{NewConst}\ \mathit{name}\ \mathit{ty}\ \mathsf{updates}\ \mathit{ctxt}} \qquad \frac{\mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt})}{\mathsf{NewType}\ \mathit{name}\ \mathit{arity}\ \mathsf{updates}\ \mathit{ctxt}}$$

The rule for NewAxiom requires that an axiom is a formula over the context's signature. The rule for NewConst requires that the constant's name is new for the context and that its type is from the context's signature. Similarly, the rule for NewType requires that the type name is new for the context.

The reflexive relation ctxt_2 extends ctxt_1 expresses that a context ctxt_2 is obtained from a context ctxt_1 by a sequence of updates. The context init_ctxt contains the built-ins, i. e. the types Bool and Fun and the equality constant. Its extension hol_ctxt also contains a type of individuals, the theory of Booleans, a Hilbert-choice constant with its characteristic axiom, and the axioms of extensionality and infinity.

2.4 Type definitions

A type definition $\mathsf{TypeDefn}\ \mathit{name\ pred\ abs\ rep}$ introduces a new type constructor *name* defined by its characteristic, closed predicate *pred* as a subset of a host type. It makes available the type $\mathsf{Tyapp}\ \mathit{name}\ l$ where the argument list l corresponds to the distinct type variables of *pred*. A proof that the predicate is satisfiable is a prerequisite, as in HOL types are non-empty. Additionally, abstraction and representation bijections between the new type and the subset of the host type are axiomatically introduced.

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt},[\,]) \vdash \mathsf{Comb}\ \mathit{pred\ witness}\\ \mathsf{closed}\ \mathit{pred}\\ \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt})\\ \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathit{abs} \neq \mathit{rep}\end{array}}{\mathsf{TypeDefn}\ \mathit{name\ pred\ abs\ rep}\ \mathsf{updates}\ \mathit{ctxt}}$$

2.5 Constant specification

Constant specification $\mathsf{ConstSpec}\ \mathit{ov\ eqs\ prop}$ defines possibly several constants by one axiom *prop*. For $(c_i,t_i) \in \mathit{eqs}$, each of the constants c_i is introduced by a closed witness term t_i, that is, the predicate *prop* holds assuming all equalities

$$(thy, [\mathsf{Var}\ c_1\ (\mathsf{typeof}\ t_1) === t_1; \ldots ; \mathsf{Var}\ c_n\ (\mathsf{typeof}\ t_n) === t_n]) \vdash prop.$$

Each of the variables $\mathsf{Var}\ c_i$ serves as a placeholder for $\mathsf{Const}\ c_i$.

If the constant specification is marked as overloading, i. e. if *ov* is true, the mechanism allows to introduce instances of already declared constants. Non-overloading constant specifications need to introduce constants with fresh names.

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt},\mathsf{map}\ (\lambda\ (s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t) === t)\ \mathit{eqs}) \vdash \mathit{prop}\\ \mathsf{every}\ (\lambda\, t.\ \mathsf{closed}\ t \wedge \forall v.\ v \in \mathsf{tvars}\ t \Rightarrow v \in \mathsf{tyvars}\ (\mathsf{typeof}\ t))\ (\mathsf{map\ snd}\ \mathit{eqs})\\ \forall x\ ty.\ \mathsf{VFREE\_IN}\ (\mathsf{Var}\ x\ ty)\ \mathit{prop} \Rightarrow (x,ty) \in \mathsf{map}\ (\lambda\ (s,t).\ (s,\mathsf{typeof}\ t))\ \mathit{eqs}\\ \mathsf{constspec\_ok}\ \mathit{ov\ eqs\ prop\ ctxt}\end{array}}{\mathsf{ConstSpec}\ \mathit{ov\ eqs\ prop}\ \mathsf{updates}\ \mathit{ctxt}}$$

Here $\mathsf{VFREE\_IN}\ x\ tm$ denotes that x is a free term variable in tm. The predicate constspec_ok imposes two important restrictions on constant specifications: the context resulting from the update needs to be orthogonal (no two defined symbols have a common type instance), and any introduced overloading of previously declared constants must not allow cycles through the definitions. We discuss how the latter is avoided with a dependency relation and define orthogonality of contexts in Section 2.6.

Constant specification generalises the introduction of new constants via equational axioms, as considered in [4], by allowing implicit definitions.[2] For further discussion of its advantages we refer to [2].

2.6 Non-cyclic theories

Cycles in theories with overloaded symbols can be avoided by restricting possible definitions in two ways that we define in this section. First, dependencies introduced by definitions and declarations need to be terminating, which is achieved by Kunčar and Popescu through a dependency relation that Åman Pohjola and Gengelbach [17] extend to its present form. Secondly, declared or defined symbols need to be orthogonal [15], that is any pair of constants or any pair of types that originates from distinct definitions is orthogonal.

[2]For instance, Euler's number *e* can be implicitly defined as the real-valued solution of a particular differential equation.

We write $u \equiv t$ for definitional updates, to mean that either u is introduced by a type definition with predicate t or otherwise u is one of the constants introduced by a constant specification with the witness t. For a context *ctxt* and types or terms u and v the dependency relation $u \rightsquigarrow_{ctxt} v$ holds whenever:

1. There is a definition $u \equiv t$ in the context *ctxt* and $v \in t^\bullet \cup t^\circ$, or
2. $u = \mathsf{Const}\ \_\ ty$ is a constant of type ty and $v \in ty^\bullet$, or
3. $u = \mathsf{Tyapp}\ \_\ l$ is a type and $v \in l$.

The first rule applies only to symbols defined by TypeDefn or ConstSpec, whereas the other rules apply also to symbols declared with NewType and NewConst. Formally $\rightsquigarrow$ is a relation on $\mathsf{type} + \mathsf{term}$, a disjoint union with canonical injections INL and INR.

The *(type-)substitutive closure* $\mathscr{R}^{\downarrow}$ of a binary relation $\mathscr{R}$ relates $\Theta\ t_1$ and $\Theta\ t_2$ if $t_1\ \mathscr{R}\ t_2$. A relation $\mathscr{R}$ is *terminating* if there is no sequence $(x_i)_{i\in\mathbb{N}}$ such that $x_i\ \mathscr{R}\ x_{i+1}$ for all $i \in \mathbb{N}$. If a binary relation $\mathscr{R}$ is terminating, its inverse $(\lambda x y.\, y \mathscr{R} x)$ is well-founded.

A context is *orthogonal* if any two distinct type definitions and any two distinct constant definitions are orthogonal. Orthogonality ensures that definitional theories have at most one definition for each ground symbol (recall *ground* means type-variable free).

Åman Pohjola and Gengelbach prove that a model exists for each orthogonal context with overloaded definitions whose substitutive closure of the dependency relation is terminating.

2.7 Semantics

In this section we introduce the semantics, which we inherit from Åman Pohjola and Gengelbach [17].

Zermelo-Fraenkel set theory The semantics is parametrised on a universe where the axioms of Zermelo-Fraenkel set theory (ZF) hold. A model of ZF is not constructible within HOL by Gödel's incompleteness argument. This setup is not new [17]. The existence of a set-theoretic universe is also an assumption in the mechanised proof of soundness of HOL Light (without overloading) [8], and it originates with Arthan [1].

Although this parametrisation appears as the assumption is_set_theory *mem* in some theorem statements, in the pretty-printed definitions we often omit the additional argument $mem\colon \mathscr{U} \Rightarrow \mathscr{U} \Rightarrow \mathsf{bool}$. Herein, the type variable $\mathscr{U}$ is the universe of sets. We also assume is_infinite *mem indset*, which states that $indset\colon \mathscr{U}$ is an infinite set.

For set membership *mem x s* we write $x \in: s$. One is a singleton set, Boolset is the set of two distinct elements True and False, and $\mathsf{Boolean}\colon \mathsf{bool} \Rightarrow \mathscr{U}$ injects Booleans from HOL into $\mathscr{U}$ in the expected way. Funspace $s\ r$ contains as elements all functions with domain $s\colon \mathscr{U}$ and co-domain $r\colon \mathscr{U}$. Abstract $s\ r\ f$ is the intersection of the graph of $f\colon \mathscr{U} \Rightarrow \mathscr{U}$ with $s \times r$. In the special case that for any $x \in: s$ we have $(x, f\ x) \in: r$, then Abstract $s\ r\ f \in:$ Funspace $s\ r$. For $x \in: s$ and $g =$ Abstract $s\ r\ f$, we write $g\ '\ x$ for $f\ x$, namely the second component of $(x, f\ x)$ from g.

Lazy ground semantics A pillar of the semantics is a *(signature) fragment*, which is a tuple $(tys, consts)$ from a signature *sig* satisfying:

$$
\begin{array}{l}
\mathsf{is\_sig\_fragment}\ sig\ (tys,consts) \overset{\mathsf{def}}{=} \\
\quad tys \subseteq \mathsf{ground\_types}\ sig \wedge tys \subseteq \mathsf{nonbuiltin\_types} \wedge consts \subseteq \mathsf{ground\_consts}\ sig\ \wedge \\
\quad consts \subseteq \mathsf{nonbuiltin\_constinsts}\ \wedge \\
\quad \forall s\ c.\ (s,c) \in consts \Rightarrow c \in \mathsf{types\_of\_frag}\ (tys,consts)
\end{array}
$$

The types *tys* are ground, non-built-in types from the signature *sig*. Each constant from *consts* is non-built-in and has a ground type from the fragment, where types_of_frag $(tys,consts)$ is defined as the built-in type closure builtin_closure *tys*. The *total fragment* is the largest fragment of a signature *sig*.

$$\mathsf{total\_fragment}\, sig \overset{\mathsf{def}}{=} (\mathsf{ground\_types}\, sig \cap \mathsf{nonbuiltin\_types}, \mathsf{ground\_consts}\, sig \cap \mathsf{nonbuiltin\_constinsts})$$

The function $\delta\colon \mathsf{type} \Rightarrow \mathscr{U}$ assigns to each non-built-in type of a fragment a value in the universe. ext δ extends this to built-in types in a standard manner. Similarly, ext γ extends an interpretation of non-built-in constants γ to the built-in constants. A *(fragment) interpretation* is a tuple (δ,γ) such that

$$\begin{array}{l}\mathsf{is\_type\_frag\_interpretation}\, tys\, \delta \overset{\mathsf{def}}{=} \forall ty.\ ty \in tys \Rightarrow \mathsf{inhabited}\ (\delta\ ty)\\ \mathsf{is\_frag\_interpretation}\ (tys,consts)\ \delta\ \gamma \overset{\mathsf{def}}{=}\\ \quad \mathsf{is\_type\_frag\_interpretation}\, tys\ \delta \wedge \forall (c,ty).\ (c,ty) \in consts \Rightarrow \gamma\ (c,ty) \in: \mathsf{ext}\ \delta\ ty\end{array}$$

Ground semantics means that only ground instances of types and constants are interpreted. A *fragment valuation* v assigns to each Var x ty, with Θ ty a (ground) type of the fragment, a value that lies in the interpretation of Θ ty.

$$\begin{array}{l}\mathsf{valuates\_frag}\, frag\ \delta\ v\ \Theta \overset{\mathsf{def}}{=}\\ \quad \forall x\ ty.\ \Theta\ ty \in \mathsf{types\_of\_frag}\, frag \Rightarrow v\ (x,ty) \in: \mathsf{ext}\ \delta\ (\Theta\ ty)\end{array}$$

The term semantics is defined as a continuation of a fragment interpretation, parametrised by a fragment valuation v and a type instantiation Θ.

$$\begin{array}{l}\mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ (\mathsf{Var}\, x\ ty) \overset{\mathsf{def}}{=} v\ (x,ty)\\ \mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ (\mathsf{Const}\, name\ ty) \overset{\mathsf{def}}{=} \gamma\ (name,\Theta\ ty)\\ \mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ (\mathsf{Comb}\, t_1\ t_2) \overset{\mathsf{def}}{=} \mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ t_1\ '\ (\mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ t_2)\\ \mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ (\mathsf{Abs}\ (\mathsf{Var}\, x\ ty)\ b) \overset{\mathsf{def}}{=}\\ \quad \mathsf{Abstract}\ (\delta\ (\Theta\ ty))\ (\delta\ (\Theta\ (\mathsf{typeof}\, b)))\ (\lambda m.\ \mathsf{termsem}\ \delta\ \gamma\ v(\!|(x,ty) \mapsto m|\!)\ \Theta\ b)\end{array}$$

Herein $f(\!|x \mapsto y|\!)$ is the function that at x takes the value y and elsewhere equals f.

The semantics applies type substitutions *lazily*, i. e. as late as possible and never to the type of term variables. This avoids a problem [17] with the eager semantics of Kunčar and Popescu: in HOL's Church-style atoms, variables Var x (Tyvar a) and Var x Bool are distinct and hence should be allowed to have different valuations under all type substitutions. With the lazy ground semantics, for Θ (Tyvar a) = Bool we just have $v\ (x,\mathsf{Tyvar}\ a) \in: \mathsf{ext}\ \delta\ (\Theta\ (\mathsf{Tyvar}\ a)) = \mathsf{Boolset}$ and $v\ (x,\mathsf{Bool}) \in: \mathsf{ext}\ \delta\ (\Theta\ \mathsf{Bool}) = \mathsf{Boolset}$. In contrast, eager ground semantics erroneously identifies $v\ (x,\Theta\ (\mathsf{Tyvar}\ a)) = v\ (x,\Theta\ \mathsf{Bool})$.

We define the satisfaction relation of a fragment interpretation (δ,γ), hypotheses *hyps* and a term p w. r. t. a fragment *frag* and a type substitution Θ. Every fragment valuation v that satisfies all instantiated hypotheses must satisfy the instantiated term Θ p.

$$\begin{array}{l}\mathsf{satisfies}\, frag\ \delta\ \gamma\, \Theta\ (hyps,p) \overset{\mathsf{def}}{=}\\ \quad \forall v.\\ \quad\quad \mathsf{valuates\_frag}\, frag\ \delta\ v\ \Theta \wedge p \in \mathsf{terms\_of\_frag\_uninst}\, frag\ \Theta\ \wedge\\ \quad\quad \mathsf{every}\ (\lambda t.\ t \in \mathsf{terms\_of\_frag\_uninst}\, frag\ \Theta)\ hyps \wedge \mathsf{every}\ (\lambda t.\ \mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ t = \mathsf{True})\ hyps \Rightarrow\\ \quad\quad\quad \mathsf{termsem}\ \delta\ \gamma\ v\ \Theta\ p = \mathsf{True}\end{array}$$

Satisfaction of hypotheses *hyps* and a conclusion p w. r. t. a fragment interpretation (δ,γ) and a signature *sig* is quantified over all ground type substitutions of the signature.

$$\begin{array}{l}
\mathsf{sat}\ sig\ \delta\ \gamma\ (hyps,p) \overset{\mathsf{def}}{=} \\
\quad \forall\Theta. \\
\quad\quad (\forall ty.\ \mathsf{tyvars}\ (\Theta\ ty) = [\,]) \wedge (\forall ty.\ \mathsf{type\_ok}\ (\mathsf{tysof}\ sig)\ (\Theta\ ty)) \wedge \\
\quad\quad \mathsf{every}\ (\lambda\ tm.\ tm \in \mathsf{ground\_terms\_uninst}\ sig\ \Theta)\ hyps \wedge p \in \mathsf{ground\_terms\_uninst}\ sig\ \Theta \Rightarrow \\
\quad\quad\quad \mathsf{satisfies}\ (\mathsf{total\_fragment}\ sig)\ \delta\ \gamma\ \Theta\ (hyps,p)
\end{array}$$

A total fragment interpretation (δ,γ) is a model of a theory *thy* if all of the theory's axioms are satisfied.

$$\begin{array}{l}
\mathsf{models}\ \delta\ \gamma\ thy \overset{\mathsf{def}}{=} \\
\quad \mathsf{is\_frag\_interpretation}\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ thy))\ \delta\ \gamma \wedge \\
\quad \forall p.\ p \in \mathsf{axsof}\ thy \Rightarrow \mathsf{sat}\ (\mathsf{sigof}\ thy)\ (\mathsf{ext}\ \delta)\ (\mathsf{ext}\ (\mathsf{ext}\ \delta)\ \gamma)\ ([\,],p)
\end{array}$$

As the semantic counterpart of derivability (Section 2.2), we define semantic entailment $(thy,hyps) \vDash p$.

$$\begin{array}{l}
(thy,hyps) \vDash p \overset{\mathsf{def}}{=} \\
\quad \mathsf{theory\_ok}\ thy \wedge \mathsf{every}\ (\mathsf{term\_ok}\ (\mathsf{sigof}\ thy))\ (p{::}hyps) \wedge \\
\quad \mathsf{every}\ (\lambda\ p.\ p\ \mathsf{has\_type}\ \mathsf{Bool})\ (p{::}hyps) \wedge \mathsf{hypset\_ok}\ hyps \wedge \\
\quad \forall\ \delta\ \gamma.\ \mathsf{models}\ \delta\ \gamma\ thy \Rightarrow \mathsf{sat}\ (\mathsf{sigof}\ thy)\ (\mathsf{ext}\ \delta)\ (\mathsf{ext}\ (\mathsf{ext}\ \delta)\ \gamma)\ (hyps,p)
\end{array}$$

The inference system is sound w. r. t. this semantics [17].

3 Symbol-independent fragment

After recapitulating the syntax and semantics in the previous section, we are set to discuss our contribution. The convenience that constants may be used prior to their definition comes at the price that interpretations of previously introduced symbols may change in extensions that define previously undefined symbols. For instance, the interpretation may change for defined orderings on lists, lexicographically defined as $\leq_{\alpha\ \mathsf{list}\to\alpha\ \mathsf{list}\to\mathsf{Bool}} \equiv\equiv\equiv \mathsf{lex}(\leq_{\alpha\to\alpha\to\mathsf{Bool}})$, if an update defines any previously undefined instance of $\leq$. In this section we carve out the fragment of all symbols that are unaffected by a theory update.

An *independent fragment* collects constants and types of a host fragment *frag* whose definitions within a theory context *ctxt* are independent of any of the symbols from a set U.

$$\begin{array}{l}
\mathsf{indep\_frag}\ ctxt\ U\ frag \overset{\mathsf{def}}{=} \\
\mathsf{let}\ V\ =\ \{\ x \mid \exists\Theta\ u.\ u \in U \wedge\ x\ (\rightsquigarrow_{ctxt}{}^{\downarrow})^{*}\ \Theta\ u\ \}\ ; \\
\quad V_2\ =\ \{\ (x,ty) \mid \mathsf{INR}\ (\mathsf{Const}\ x\ ty) \in V\ \}\ ;\ V_1\ =\ \{\ x \mid \mathsf{INL}\ x \in V\ \}\ \mathsf{in} \\
\quad (\mathsf{fst}\ frag \setminus V_1, \mathsf{snd}\ frag \setminus V_2)
\end{array}$$

The set U contains the symbols introduced by a theory extension. In contrast to [4], where U is a singleton set, we allow the introduction of several symbols at once, e. g. via constant specification. The set V is the pre-image of type instances $\Theta\ u$ of elements u from U (with Θ a ground type substitution) under the reflexive-transitive, type-substitutive closure of the dependency relation $\rightsquigarrow_{ctxt}$. As host fragment *frag*, we only consider total fragments (over different signatures). An independent fragment of a total fragment is indeed a signature fragment, since constants depend on their types.

$$\begin{array}{l}
\vdash ctxt\ \mathsf{extends}\ \mathsf{init\_ctxt} \Rightarrow \\
\quad \mathsf{is\_sig\_fragment}\ (\mathsf{sigof}\ ctxt)\ (\mathsf{indep\_frag}\ ctxt\ U\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ ctxt)))
\end{array}$$

We prove this claim in script, to give a flavour of the reasoning involved in the mechanisation. Thereby we amend the earlier proof [4] for the case where a type substitution ρ and $^\bullet$ do not commute on a type ς, i. e. $\rho(\varsigma^\bullet) \neq \rho(\varsigma)^\bullet$. (This case had been excluded by a faulty lemma inherited from Kunčar and Popescu.)

For a fixed context, F_U denotes the fragment independent of symbols U, and $\mathsf{GType}^\bullet$ and GInst° are all types and non-built-in constants of the total fragment, respectively.

Proof. For a ground constant instance $c_\sigma \in \mathsf{GInst}^\circ \setminus V$, we show that also its type σ is from the types of F_U. Assume that $\sigma \notin \mathsf{builtin\_closure}(\mathsf{GType}^\bullet \setminus V)$. Thus $\sigma^\bullet \not\subseteq \mathsf{GType}^\bullet \setminus V$ and there is a type $\tau \in \sigma^\bullet \cap V$. Assuming the dependency $c_\sigma \rightsquigarrow^{\downarrow +} \tau$ the contradiction $c_\sigma \in V$ follows. We now show $c_\sigma \rightsquigarrow^{\downarrow +} \tau$ for $\tau \in \sigma^\bullet$:
Let c_ς be a (defined or declared) constant. It holds $c_\varsigma \rightsquigarrow t$ for $t \in \varsigma^\bullet$ and thus for any instance $c_{\rho(\varsigma)} \rightsquigarrow^{\downarrow} \rho(t)$ for $t \in \varsigma^\bullet$. Generally, $\rho(\varsigma)^\bullet \neq \rho(\varsigma^\bullet)$ as Åman Pohjola and Gengelbach notice [17]. If ς is a type variable or a non-built-in type, $\varsigma^\bullet = \{\varsigma\}$, then $c_{\rho(\varsigma)} \rightsquigarrow^{\downarrow} \rho(\varsigma)$ and $\rho(\varsigma) \rightsquigarrow t$ for $t \in \rho(\varsigma)^\bullet$. If on the other hand $\varsigma = a \to b$ is the built-in function type, thus σ is a function type and let ρ be such that $\rho(a \to b) = \sigma$. Any type below σ and above $\tau \in \sigma^\bullet$ is a function type (as $\tau \in \sigma^\bullet \neq \{\sigma\}$). If τ is introduced by a type instantiation, then within $a \to b$ there is a type variable α such that $\rho(\alpha)$ syntactically contains τ. Thus $c_{a\to b} \rightsquigarrow \alpha$ by $\alpha \in (a \to b)^\bullet$ and $\rho(\alpha) \rightsquigarrow^+ \tau$ (as in $\rho(\alpha)$ there are only function types above τ). If τ was not introduced by a type instantiation and τ' is the type within $a \to b$ such that $\rho(\tau') = \tau$, then $c_{a\to b} \rightsquigarrow \tau'$ and consequently $c_{\rho(a\to b)} \rightsquigarrow^{\downarrow} \rho(\tau') = \tau$. □

Symbols introduced by a theory extension Until now, the independent fragment has been defined without regard to the theory extension mechanism, to contain all symbols that are independent of the symbols from an arbitrary set U. The relevant independent fragments are those that are independent of a theory extension, i. e. those for which U contains the constant instances and types that are introduced by a theory update. For an update *upd*, we set $U = \mathsf{upd\_introduces}\ \mathit{upd}$ as the apex of the independent fragment cone.

$$\begin{aligned}
&\mathsf{upd\_introduces}\ (\mathsf{ConstSpec}\ \mathit{ov\ eqs\ prop}) \overset{\mathsf{def}}{=} \mathsf{map}\ (\lambda\ (s,t).\ \mathsf{INR}\ (\mathsf{Const}\ s\ (\mathsf{typeof}\ t)))\ \mathit{eqs}\\
&\mathsf{upd\_introduces}\ (\mathsf{TypeDefn}\ \mathit{name\ pred\ abs\ rep}) \overset{\mathsf{def}}{=}\\
&\quad [\mathsf{INL}\ (\mathsf{Tyapp}\ \mathit{name}\ (\mathsf{map\ Tyvar}\ (\mathsf{mlstring\_sort}\ (\mathsf{tvars}\ \mathit{pred}))))]\\
&\mathsf{upd\_introduces}\ (\mathsf{NewType}\ \mathit{name\ arity}) \overset{\mathsf{def}}{=}\\
&\quad [\mathsf{INL}\ (\mathsf{Tyapp}\ \mathit{name}\ (\mathsf{map\ Tyvar}\ (\mathsf{genlist}\ (\lambda\, x.\ \mathsf{implode}\ (\mathsf{replicate}\ (\mathsf{SUC}\ x)\ \#\text{“a”}))\ \mathit{arity})))]\\
&\mathsf{upd\_introduces}\ (\mathsf{NewConst}\ \mathit{name\ ty}) \overset{\mathsf{def}}{=} [\mathsf{INR}\ (\mathsf{Const}\ \mathit{name\ ty})]\\
&\mathsf{upd\_introduces}\ (\mathsf{NewAxiom}\ \mathit{prop}) \overset{\mathsf{def}}{=} [\,]
\end{aligned}$$

For constant specifications and declarations, $\mathsf{upd\_introduces}$ returns the constants available for use after the theory update. For type definitions, the introduced type constructor has as arguments all type variables of the defining predicate sorted by name. Type declarations introduce a type constructor whose arguments are *arity* many distinct type variables.

In the definition of $\mathsf{upd\_introduces}$ we make two choices:

- The independent fragment of an update defining a type τ by a predicate $t_{\sigma\to\mathsf{Bool}}$ defines $U = \{\tau\}$. For a type substitution ρ either all instances $\rho(\tau)$, $\mathsf{rep}_{\rho(\sigma\to\tau)}$ and $\mathsf{abs}_{\rho(\tau\to\sigma)}$ are in F_U or otherwise in its complement (that we earlier denoted V). Although the proof of said property is non-trivial, the choice of defining $U = \{\tau\}$ instead of $U = \{\tau, \mathsf{rep}_{\sigma\to\tau}, \mathsf{abs}_{\tau\to\sigma}\}$ adds the convenience (for case analysis in some proofs) that any constant introduced by an update (w. r. t. $\mathsf{upd\_introduces}$) does not come from a type definition.

- As non-definitional axioms generally are not conservative, any symbol's interpretation may be affected by such an update, hence we define $\mathsf{upd\_introduces}\ (\mathsf{NewAxiom}\ \mathit{prop}) \overset{\mathsf{def}}{=} [\,]$.

We henceforth only regard independent fragments related to theory updates.

$$\mathsf{indep\_frag\_upd}\ \mathit{ctxt}\ \mathit{upd}\ \mathit{frag} \overset{\mathsf{def}}{=} \mathsf{indep\_frag}\ \mathit{ctxt}\ (\mathsf{upd\_introduces}\ \mathit{upd})\ \mathit{frag}$$

The independent fragment of a theory *ctxt* extended by *upd* is carved out from the total fragment over the extended signature, but factually any symbols introduced by the update are not within the fragment:

$$\begin{aligned}&\vdash \mathsf{let}\ \mathit{idf}\ = \mathsf{indep\_frag\_upd}\ (\mathit{upd}{::}\mathit{ctxt})\ \mathit{upd}\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ (\mathit{upd}{::}\mathit{ctxt})))\ \mathsf{in}\\ &\quad \mathit{upd}{::}\mathit{ctxt}\ \mathsf{extends}\ \mathsf{init\_ctxt} \Rightarrow\\ &\quad\ \mathsf{fst}\ \mathit{idf} \subseteq \mathsf{fst}\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ \mathit{ctxt})) \wedge \mathsf{snd}\ \mathit{idf} \subseteq \mathsf{snd}\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ \mathit{ctxt}))\end{aligned}$$

Hereby, the *upd*-independent fragments over the signatures *ctxt* and *upd*::*ctxt* are equal, as each symbol introduced by the extension by *upd* depends on a symbol in $\mathsf{upd\_introduces}\ \mathit{upd}$.

4 Model-theoretic Conservativity

In this section we discuss how we construct a model of an extended theory while keeping parts of a model from the theory prior to extension. With the properties of the construction and an extra assumption on the given models we prove model-theoretic conservativity.

4.1 Model construction

From a model (Δ,Γ) of a theory *ctxt* we construct a model (δ,γ) of the extension *upd*::*ctxt*. Supported theory extensions are either extensions by definition or declaration of constants or a type, or otherwise admissible non-definitional axioms. A model of the extended theory is constructed by recursion over part of the $\rightsquigarrow^{\downarrow}$ relation, based on the model (Δ,Γ). In contrast, the model construction in [12] obtains a model from the ground up, by recursion over the entire $\rightsquigarrow^{\downarrow}$ relation, without reference to any previous interpretation.

A model is constructed by two mutually recursive functions $\mathsf{type\_interpretation\_ext}\ \mathit{ind}\ \mathit{ctxt}\ \mathit{upd}\ \Delta\ \Gamma\ \mathit{ty}$ and $\mathsf{term\_interpretation\_ext}\ \mathit{ind}\ \mathit{ctxt}\ \mathit{upd}\ \Delta\ \Gamma\ c\ \mathit{ty}$ that return the interpretation of a type or constant instance, respectively. As arguments these functions take the model (Δ,Γ) of the theory *ctxt*, the update *upd* and an infinite type *ind*. The model construction for a definitional theory extension *upd*::*ctxt* is guarded with a check: if the symbol to interpret lies in the independent fragment

$$\mathsf{indep\_frag\_upd}\ (\mathit{upd}{::}\mathit{ctxt})\ \mathit{upd}\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ \mathit{ctxt}))$$

of a definitional update *upd*, the symbol may be interpreted w. r. t. the model (Δ,Γ). Otherwise the symbol's interpretation is constructed as discussed in earlier work [17, 12, 4].

Our amendments to the model construction are a few lines each (here the four lines of the second if branch) in type_interpretation_ext and term_interpretation_ext. The inherited tedious parts are elided.

```
type_interpretation_ext $ind\ upd\ ctxt\ \Delta\ \Gamma\ ty \overset{\text{def}}{=}$
 if $\neg$wellformed $(upd::ctxt)$
 then One
 else if
  $(\forall tm.\ upd \neq$ NewAxiom $tm) \wedge$
  $ty \in$ fst (indep_frag_upd $(upd::ctxt)$ $upd$ (total_fragment (sigof $ctxt$)))
 then $\Delta\ ty$
 else ...
```

Requirements for Constant Specification The differing constant definition mechanism entails that the model construction yields no model, but only a fragment interpretation of the theory's total fragment.

For a theory *ctxt* that has a model (Δ,Γ) we need to prove that any axiom from *ctxt* holds in a model (δ,γ) of a valid theory extension $upd::ctxt$. In its proof we are presented with a sub-case that occurs due to the different definitional mechanism for constants, as compared to Gengelbach and Weber. We illustrate the problem by an example theory:

Let *ctxt* be a theory where by constant specification two constants d_{Bool} and e_{Bool} are defined to be distinct by the axiom $d_{\mathsf{Bool}} \neq e_{\mathsf{Bool}}$, that holds for the witnesses $d_{\mathsf{Bool}} = \mathsf{False}$ and $e_{\mathsf{Bool}} = (c_{\mathsf{Bool}} \Rightarrow \mathsf{True})$ for a declared-only constant c_{Bool}. Let an update *upd* define $c_{\mathsf{Bool}} = \mathsf{True}$. For the fragment of *ctxt* that is independent of this update of c_{Bool} we write F. For a model (Δ,Γ) of *ctxt* we have to show that the axiom $d_{\mathsf{Bool}} \neq e_{\mathsf{Bool}}$ holds in the model extension (δ,γ) for $upd::ctxt$ as obtained from the above model construction. With $d_{\mathsf{Bool}} \in F$ and $e_{\mathsf{Bool}} \notin F$, it is impossible to prove that $\gamma(d_{\mathsf{Bool}}) \neq \gamma(e_{\mathsf{Bool}})$ as we only know $\gamma(d_{\mathsf{Bool}}) = \Gamma(d_{\mathsf{Bool}})$ and $\gamma(e_{\mathsf{Bool}}) = \mathsf{true}$.

In the example two constants are simultaneously introduced and defined in terms of another, and only one lies in the independent fragment. In the iterative model construction, information is lost on how constants are interpreted that are dependencies of a symbol. We choose to only extend models where each defined constant is interpreted as its witness.

We require that all constants, defined by constant specifications in a context *ctxt*, are interpreted equal to their witness in a model (Δ,Γ).

```
models_witnesses $\Delta\ \Gamma\ ctxt \overset{\text{def}}{=}$
 $\forall ov\ cl\ prop\ c\ cdefn\ ty\ \Theta.$
   ConstSpec $ov\ cl\ prop \in ctxt \wedge (c,cdefn) \in cl \wedge ty =$ typeof $cdefn\ \wedge$
   $(c,\Theta\ ty) \in$ ground_consts (sigof $ctxt$) $\wedge$ $(c,\Theta\ ty) \in$ nonbuiltin_constinsts $\Rightarrow$
    $\Gamma\ (c,\Theta\ ty) =$ termsem (ext $\Delta$) (ext (ext $\Delta$) $\Gamma$) empty_valuation $\Theta\ cdefn$
```

This added requirement is preserved by the model construction.

```
$\vdash$ is_set_theory $mem \Rightarrow$
   $\forall upd\ ctxt\ \Delta\ \Gamma.$
     $upd::ctxt$ extends init_ctxt $\wedge$ inhabited $ind\ \wedge$
     is_frag_interpretation (total_fragment (sigof $ctxt$)) $\Delta\ \Gamma\ \wedge$
     models_witnesses $\Delta\ \Gamma\ ctxt \Rightarrow$
      models_witnesses (type_interpretation_ext $ind\ upd\ ctxt\ \Delta\ \Gamma$)
       (term_interpretation_ext $ind\ upd\ ctxt\ \Delta\ \Gamma$) $(upd::ctxt)$
```

The restriction to models of theories that satisfy models_witnesses keeps the expressivity of Arthan's constant specification and is conservative w. r. t. constant definition.

Alternatively, the problem as depicted in the example can be circumvented through extending the dependency relation with *cross-dependencies*. For simultaneously introduced constants d and e, any dependency x of e (i. e. $e \rightsquigarrow x$) also becomes a dependency of d (i. e. $d \rightsquigarrow x$) and likewise with d and e swapped. Any constants that are introduced together would thereby be assumed to be related, which reduces expressivity. For two declared constants f_α and g_{Bool} the otherwise legitimate simultaneous definition of $f_\alpha = g_\alpha$ and $g_{\mathsf{Bool}} = \mathsf{True}$ becomes impossible, as it is cyclic: $g_{\mathsf{Bool}} \rightsquigarrow^{\downarrow} g_{\mathsf{Bool}}$. Instead each conjunct would need to be a theory extension on its own.

4.2 Model-theoretic conservativity

In this subsection we introduce our main result. The mechanism to extend theories by definitions or declarations is model-theoretically conservative if for any theory *ctxt* with a model (Δ,Γ) that interprets any constant witness pair from constant specification equal, any theory extension *upd*::*ctxt* (where *upd* is a definition or declaration) has a model (δ,γ) that also satisfies the property:

$$\begin{array}{c}\mathsf{let}\ idf\ =\ \mathsf{indep\_frag\_upd}\ (upd{::}ctxt)\ upd\ (\mathsf{total\_fragment}\ (\mathsf{sigof}\ ctxt))\ \mathsf{in}\\ (\forall ty.\ ty \in \mathsf{fst}\ idf \Rightarrow \delta\ ty = \Delta\ ty) \wedge \forall c\ ty.\ (c,ty) \in \mathsf{snd}\ idf \Rightarrow \gamma\ (c,ty) = \Gamma\ (c,ty)\end{array}$$

If this property holds, it naturally extends to any ground term that is built from built-in types and symbols from the independent fragment *idf*. Hence any such term's interpretation in the new model (δ,γ) equals its interpretation in the old model (Δ,Γ). With the restriction to models that interpret constants as their witnesses we derive that the construction in Section 4.1 yields a model.

$$\begin{array}{l}\vdash \mathsf{is\_set\_theory}\ mem \Rightarrow\\ \quad \forall upd\ ctxt\ \Delta\ \Gamma.\\ \quad\quad ctxt\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \mathsf{inhabited}\ ind \wedge upd\ \mathsf{updates}\ ctxt \wedge\\ \quad\quad \mathsf{axioms\_admissible}\ mem\ ind\ (upd{::}ctxt) \wedge \mathsf{models}\ \Delta\ \Gamma\ (\mathsf{thyof}\ ctxt) \wedge \mathsf{models\_witnesses}\ \Delta\ \Gamma\ ctxt \Rightarrow\\ \quad\quad\quad \mathsf{models}\ (\mathsf{type\_interpretation\_ext}\ ind\ upd\ ctxt\ \Delta\ \Gamma)\ (\mathsf{term\_interpretation\_ext}\ ind\ upd\ ctxt\ \Delta\ \Gamma)\\ \quad\quad\quad\quad (\mathsf{thyof}\ (upd{::}ctxt))\end{array}$$

This constructed model trivially satisfies the given property that interpretations from the *upd*-independent fragment are kept if the *upd* is a declaration or a definition.

At different stages in the proof of model-theoretic conservativity, case analysis occurs of how an update *upd* may extend a theory *ctxt* by *upd* updates *ctxt*. As an example, proof obligations similar to the following reoccur frequently in the formalisation. To show that a symbol x keeps its interpretation in a model extension w. r. t. an update *upd*, one has to show that x is independent of the update *upd* by proving that all dependencies of x are on symbols from the *upd*-independent fragment.

Future work could investigate if the model construction may be conservative even w. r. t. NewAxiom updates of admissible axioms from hol_ctxt.

4.3 Consistency

As a consequence of the model construction from the previous section, we obtain consistency of *definitional* extensions of hol_ctxt, that is extensions that do not contain NewAxiom.

A theory is *consistent* if there is a provable and an unprovable sequent. We inherit the following definition from Kumar et al. [9].

$$\begin{aligned}&\mathsf{consistent\_theory}\ thy \overset{\mathsf{def}}{=}\\ &\quad (thy,[\,]) \vdash \mathsf{Var}\ «\mathrm{x}»\ \mathsf{Bool} === \mathsf{Var}\ «\mathrm{x}»\ \mathsf{Bool} \wedge \neg((thy,[\,]) \vdash \mathsf{Var}\ «\mathrm{x}»\ \mathsf{Bool} === \mathsf{Var}\ «\mathrm{y}»\ \mathsf{Bool})\end{aligned}$$

As a corollary of our work, the existence of a model of init_ctxt combined with the incremental model construction yields consistency of definitional extensions of hol_ctxt [12, 17]. The restriction on the interpretations of constants as their witnesses trivially holds in init_ctxt and is an invariant in the induction.

$$\begin{aligned}&\vdash \mathsf{is\_set\_theory}\ mem \wedge \mathsf{is\_infinite}\ mem\ ind \Rightarrow\\ &\quad \forall ctxt.\ \mathsf{definitional\_extension}\ ctxt\ \mathsf{hol\_ctxt} \Rightarrow \mathsf{consistent\_theory}\ (\mathsf{thyof}\ ctxt)\end{aligned}$$

This work thus generalises and replaces the earlier non-incremental model construction [17].

5 Related Work

For untyped first-order logic, extension by definition of predicate and function symbols is discussed by Shoenfield [19, § 4.6]. A definitions by a predicate extends a theory with an equivalence that contains the predicate only on the left-hand side; a definition by a function symbol requires the proof that the function symbol indeed is a mathematical function. These mechanisms are proof-theoretically conservative, and each model of the original theory has one unique corresponding model of the extended theory. In consequence, both definitional mechanisms are model-theoretically conservative.

Farmer [3] defines an extension of a theory to be a super-set that is a model-theoretic conservative extension, hence keeps model interpretation and consistency. By example of simply-typed first-order logic with extension by algebraic datatypes and constant definitions, the author discusses also weaker notions of semantic conservativity and its properties w. r. t. theory embeddings, so called theory instantiation.

In their formalisation of HOL Light without overloading [9], Kumar et al. also make model-theoretic conservativity a requirement for theory extension by definitions or declarations. They denote this property sound_update *ctxt upd* of each such extension of *ctxt* by *upd*, and prove consistency by an inductive argument. As the definition mechanism for constants they use constant specification that allows to introduce multiple constants at once, given witnesses for which the defining axiom is derivable. Constant specification was first introduced by Arthan [2], and is is implemented in HOL4 [14] and ProofPower.

The study of theoretical foundations of overloaded definitions (together with type classes in higher-order logic) dates back to Wenzel [20]. For Wenzel an extension mechanism for deductive logics needs to be syntactically conservative, which he proves for constant definition where all instances are defined at once. In addition, the considered constant definitions can be unfolded, which is called *realisability*. In this discussion the interplay of overloaded constants and type definitions is not considered.

To avoid inconsistencies Obua [15] remarks that the unfolding of definitions needs to terminate for both type and constant definitions. Further Obua discusses that termination is not semi-decidable for overloaded definitions that recurse through types. The proof sketch of conservativity of overloading in Isabelle, he misses that inconsistencies may be introduced by dependencies through types.

For the Isabelle framework with its Haskell-style type classes, Wenzel and Haftmann [7] state requirements on overloading definitions without discussing if these suffice for acyclic dependencies.

Kunčar and Popescu [12] aim to close the consistency gap for definitional theories in Isabelle, in showing that every definitional theory has a model, by a model construction that recurses into the dependencies of definitions. Fixable gaps in their result are closed in the mechanisation of the model construction by Åman Pohjola and Gengelbach [17]. Instead of constant definition their mechanisation considers Arthan's constant specification, and gives the above discussed *lazy fragment-ground* semantics. We base on their implementation work and generalise their monolithic model construction.

In two works, Kunčar and Popescu study consistency of definitional theories by syntactic arguments. They encode formulas through an unfolding of definitions into a richer logic HOL with comprehension types (HOLC) and prove that provability is preserved [13]. Ultimately, definitional theories are consistent by the consistency of HOLC.

In another paper, they use an unfolding that stays in the logic of HOL [11] by relativising defined types in formulas to a predicate on the defined type's host type. The proof-theoretic conservativity result holds for any definitional theory unfolded into initial HOL, and motivates a dual model-theoretic conservativity result where any model of initial HOL can be extended to a model of a given definitional theory. Our paper proves model-theoretic conservativity of two arbitrary definitional theories.

In recent work Gengelbach and Weber [5] prove model-theoretic conservativity of definitional theories [4] for semantics that do not require full function spaces in order to derive their syntactic counterparts. A definitional extension of a theory is proof-theoretically conservative, that is, if a formula's types and constants are unchanged by a theory update, and the formula is derivable after the update, then it is also derivable from the theory before the update. Their proof-theoretical result holds for constant definition and it is unclear how that result is transferable to constant specification with regard to the additional restriction on models models_witnesses in our proof.

Mizar is a theorem prover that supports overloading of symbols even for overlapping sub-types [6], where either the interpretation w. r. t. a definition may be specified or the most recently introduced definition is chosen for interpretation. Despite mentions of consistency of this sophisticated mechanism (e. g. [18]) there is no proof for consistency or conservativity of Mizar.

6 Conclusion

We established that type definitions and constant specifications in HOL with ad-hoc overloading of arbitrary theories above init_ctxt with fixed admissible axioms from hol_ctxt are model-theoretically conservative. The result holds for models that interpret each constant introduced by constant specification equal to the constant's witness. An interpretation of this result is that the definitional mechanisms of Isabelle/HOL are semantically speaking robustly designed: at least symbols that are independent of an update may keep their interpretation in a model extension.

Model-theoretic conservativity has a proof-theoretic (syntactic) counterpart. Roughly, an extension is *proof-theoretically conservative* if it entails no new theorems in the original language. In other words, every formula of the original language that is a theorem in the extension is already provable in the original theory.

In earlier work, Kunčar and Popescu [11] show that any definitional theory is a proof-theoretically conservative extension of *initial HOL*, i. e. hol_ctxt. The semantic counterpart is that any definitional theory is model-theoretically conservative above initial HOL. Comparably, our semantic conservativity is stronger as it holds for arbitrary theories above hol_ctxt, which we achieved by utilising the independent fragment, i. e. a subset of the signature that is independent of a theory extension.

We conjecture that the syntactic counterpart of our result holds: if D' is an extension of D such that $D' \vdash \varphi$, where φ is a formula whose non-built-in constant instances and types are independent of symbols defined in $D' \setminus D$, then $D \vdash \varphi$. Gengelbach and Weber recently proved this conjecture for constant definition through equality axioms [5]. We leave its study for the more general constant specification mechanism [2] to future work.

References

[1] Rob Arthan: *HOL Formalised: Semantics*. Available at `http://www.lemma-one.com/ProofPower/specs/spc002.pdf`.

[2] Rob Arthan (2014): *HOL Constant Definition Done Right*. In: Interactive Theorem Proving, Springer International Publishing, pp. 531–536, doi:`10.1007/978-3-319-08970-6_34`.

[3] William M. Farmer: *A General Method for Safely Overwriting Theories in Mechanized Mathematics Systems*. Available at `http://imps.mcmaster.ca/doc/overwriting-theories.pdf`.

[4] Arve Gengelbach & Tjark Weber (2017): *Model-Theoretic Conservative Extension for Definitional Theories*. In Sandra Alves & Renata Wasserman, editors: 12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017, Electronic Notes in Theoretical Computer Science 338, Elsevier, pp. 133–145, doi:`10.1016/j.entcs.2018.10.009`.

[5] Arve Gengelbach & Tjark Weber (2020): *Proof-theoretic Conservativity for HOL with Ad-hoc Overloading*. In Violet Ka I Pun, Volker Stolz & Adenilso da Silva Simão, editors: Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings, Lecture Notes in Computer Science 12545, Springer, pp. 23–42, doi:`10.1007/978-3-030-64276-1_2`.

[6] Adam Grabowski, Artur Kornilowicz & Adam Naumowicz (2010): *Mizar in a Nutshell*. J. Formalized Reasoning 3(2), pp. 153–245, doi:`10.6092/issn.1972-5787/1980`.

[7] Florian Haftmann & Makarius Wenzel (2006): *Constructive Type Classes in Isabelle*. In Thorsten Altenkirch & Conor McBride, editors: Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers, Lecture Notes in Computer Science 4502, Springer, pp. 160–174, doi:`10.1007/978-3-540-74464-1_11`.

[8] Ramana Kumar, Rob Arthan, Magnus O. Myreen & Scott Owens (2014): *HOL with Definitions: Semantics, Soundness, and a Verified Implementation*. In: Interactive Theorem Proving, Springer, Cham, pp. 308–324, doi:`10.1007/978-3-319-08970-6_20`.

[9] Ramana Kumar, Rob Arthan, Magnus O. Myreen & Scott Owens (2016): *Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation*. J. Autom. Reasoning 56(3), doi:`10.1007/s10817-015-9357-x`.

[10] Ondrej Kuncar (2015): *Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants*. In Xavier Leroy & Alwen Tiu, editors: Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015, ACM, pp. 85–94, doi:`10.1145/2676724.2693175`.

[11] Ondrej Kuncar & Andrei Popescu (2018): *Safety and conservativity of definitions in HOL and Isabelle/HOL*. PACMPL 2(POPL), pp. 24:1–24:26, doi:`10.1145/3158112`.

[12] Ondrej Kuncar & Andrei Popescu (2019): *A Consistent Foundation for Isabelle/HOL*. J. Autom. Reasoning 62(4), pp. 531–555, doi:`10.1007/s10817-018-9454-8`.

[13] Ondřej Kunčar & Andrei Popescu (2017): *Comprehending Isabelle/HOL's Consistency*. In Hongseok Yang, editor: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Lecture Notes in Computer Science 10201, Springer, pp. 724–749, doi:`10.1007/978-3-662-54434-1_27`.

[14] Michael Norrish & Konrad Slind (2014): *The HOL System LOGIC*. Available at `http://downloads.sourceforge.net/project/hol/hol/kananaskis-10/kananaskis-10-logic.pdf`.

[15] Steven Obua (2006): *Checking Conservativity of Overloaded Definitions in Higher-Order Logic*. In Frank Pfenning, editor: Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA,

USA, August 12-14, 2006, Proceedings, *Lecture Notes in Computer Science* 4098, Springer, pp. 212–226, doi:10.1007/11805618_16.

[16] Andrew M. Pitts (1993): *The HOL Logic*. In M.J.C. Gordon & Tom Melham, editors: *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*, Cambridge University Press, pp. 191–232.

[17] Johannes Åman Pohjola & Arve Gengelbach (2020): *A Mechanised Semantics for HOL with Ad-hoc Overloading*. In Elvira Albert & Laura Kovács, editors: *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, *EPiC Series in Computing* 73, EasyChair, pp. 498–515, doi:10.29007/413d. Available at https://easychair.org/publications/paper/9Hcd.

[18] Piotr Rudnicki (1992): *An Overview of the Mizar Project*. In Bengt Nordström, Kent Petersson & Gordon Plotkin, editors: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pp. 311–332. Available at http://mizar.org/project/MizarOverview.pdf.

[19] Joseph R. Shoenfield (1967): *Mathematical Logic*. A.K. Peters, Natick, Mass.

[20] Markus Wenzel (1997): *Type Classes and Overloading in Higher-Order Logic*. In Elsa L. Gunter & Amy P. Felty, editors: *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, *Lecture Notes in Computer Science* 1275, Springer, pp. 307–322, doi:10.1007/BFb0028402.

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 ($\to$). More specifically, the terms of our example propositional logic can be defined as follows, using True (**T**) as a constant/bottom object:

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

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.

Identity & Cut

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

Structural Rules

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

Logical Rules

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

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

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 $\uplus$ 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:

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

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	$\varnothing \vdash X \times Y \to Y \times X$	
1	`ruleseq` $(R\to)$	$\{X \times Y\} \vdash Y \times X$	
2	`ruleseq` (C)	$\{X \times Y\} \uplus \{X \times Y\} \vdash Y \times X$	
3	`ruleseq` $(R\times)$	$\{X \times Y\} \vdash Y$	$\{X \times Y\} \vdash X$
4	`ruleseq` $(L2\times)$	$\{Y\} \vdash Y$	$\{X \times Y\} \vdash X$
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.

Identity & Cut

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

Structural Rules

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

Logical Rules

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

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

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 $R\rightarrow$ 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:

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

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:

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

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 $R\rightarrow$ rule:

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

The backwards application of the rule (using `ruleseq`) tries to match the conclusion of the rule $\Gamma \vdash \lambda x.y : A \rightarrow B$ to the current goal $\vdash f : X \times Y \rightarrow Y \times X$. 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 $\mathit{Var}\ x : X \times Y \vdash y : Y \times X$.

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

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

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

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

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

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

The rest of the proof yields further instantations of the metavariables x' and y', eventually resulting in the final instantation 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

We would like to thank the attendees of Logical Frameworks and Meta Languages: Theory and Practice (LFMTP) 2020 and the anonymous reviewers for their constructive feedback.

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 n-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.

Deductive Systems and Coherence for Skew Prounital Closed Categories

Tarmo Uustalu
Reykjavik University, Reykjavik, Iceland
Tallinn University of Technology, Tallinn, Estonia
tarmo@ru.is

Niccolò Veltri
Tallinn University of Technology, Tallinn, Estonia
niccolo@cs.ioc.ee

Noam Zeilberger
École Polytechnique, Palaiseau, France
noam.zeilberger@lix.polytechnique.fr

In this paper we develop the proof theory of skew prounital closed categories. These are variants of the skew closed categories of Street where the unit is not represented. Skew closed categories in turn are a weakening of the closed categories of Eilenberg and Kelly where no structural law is required to be invertible. The presence of a monoidal structure in these categories is not required. We construct several equivalent presentations of the free skew prounital closed category on a given set of generating objects: a categorical calculus (Hilbert-style system), a cut-free sequent calculus and a natural deduction system corresponding to a variant of planar (= non-commutative linear) typed lambda-calculus. We solve the coherence problem for skew prounital closed categories by showing that the sequent calculus admits focusing and presenting two reduction-free normalization procedures for the natural deduction calculus: normalization by evaluation and hereditary substitutions. Normal natural deduction derivations ($\beta\eta$-long forms) are in one-to-one correspondence with derivations in the focused sequent calculus. Unexpectedly, the free skew prounital closed category on a set satisfies a left-normality condition which makes it lose its skew aspect. This pitfall can be avoided by considering the free skew prounital closed category on a skew multicategory instead. The latter has a presentation as a cut-free sequent calculus for which it is easy to see that the left-normality condition generally fails.

The whole development has been fully formalized in the dependently typed programming language Agda.

1 Introduction

Proof theory and category theory have gone hand in hand since the pioneering works of Lambek [17, 18, 19] and a number of researchers that followed immediately, like Lawvere [20], Szabo [31, 32], Mann [24], Mints [26]. Category theory helps the proof theorist with mathematical models for logical proof systems, which should help tackling problems like analysis of the connections between different types of proof systems, e.g., sequent calculi and natural deduction [41]. On the other hand, proof theory provides the category theorist with a toolbox for identifying the internal language of categories and for solving problems of combinatorial nature such as Mac Lane's coherence problem [22, 14].

Given a certain notion of category with structure, it is natural to ask whether there exists deductive systems (with good proof-theoretic properties) presenting the "canonical" category with that structure. For Cartesian closed categories, such systems are given by, e.g., the sequent calculus of intuitionistic logic and its natural deduction system, which we also know as typed lambda-calculus. For symmetric monoidal closed categories, some such systems are the sequent calculus of intuitionistic linear logic (with

C. Sacerdoti Coen, A. Tiu (Eds.): 15th International Workshop
on Logical Frameworks: Theory and Practice (LFMTP 2020)
EPTCS 332, 2021, pp. 35–53, doi:10.4204/EPTCS.332.3

$\mathsf{I}, \otimes, \multimap$) and the linear variant of typed lambda-calculus [7, 34]. Formally, the "canonical" category with structure arises from a *free* construction. E.g., simply-typed lambda-calculus (with $1, \times, \Rightarrow$) and with atomic types taken from a set At, is a presentation of the free Cartesian monoidal closed category on At.

In recent work, we have investigated the deductive systems associated to *skew monoidal categories* [35, 37]. These are a weakening, first studied by Szlachányi [33], of *monoidal categories* [6, 22] in which the unitors and associator are not required to be invertible, they are merely natural transformations in a particular direction. These categories are not uncommon, e.g., they appear in the study of relative monads [3] and quantum categories [16]. The free skew monoidal category on a set At can be constructed as a sequent calculus with sequents of the form $S \mid \Gamma \longrightarrow C$, where the antecedent is split into an optional formula S, called the *stoup*, and a list of formulae Γ, the *context*. This sequent calculus has some peculiarities: left rules apply only to the formula in the special stoup position, while the tensor right rule forces the formula in the stoup of the conclusion to be the formula in the stoup of the first premise. This sequent calculus enjoys cut elimination and a focused subsystem, defining a root-first proof search strategy attempting to build a derivation of a sequent. The focused calculus finds exactly one representative of each equivalence class of derivations and is thus a concrete presentation of the free skew monoidal category, as such solving the coherence problem for skew monoidal categories.

In this paper, we perform a similar proof-theoretic analysis of *skew prounital closed categories* [36]. These are the skew variant of *prounital closed category* of Shulman [28, Rev. 49], which in turn is a relaxation of the notion of *closed category* by Eilenberg and Kelly [11, 21]. Intuitively, a prounital closed category is a category with an internal hom object $A \multimap B$ for any two objects A and B. There is no requirement for a unit object I, nor for a tensor product $\otimes$. But the unit is implicitly present to a degree thanks to the presence of a functor $J : \mathbb{C} \to \mathbf{Set}$, with JA playing the role of the set of maps from the non-represented unit to the object A. In other words, for a closed category $\mathbb{C}$, we have $JA = \mathbb{C}(\mathsf{I}, A)$. Related categories where the unit is "half-there" appear in the study of categorical models of classical linear logic [12]. Yet weaker are Lambek's *residuated categories* [17] (with one implication) where the unit is completely absent. *Skew closed categories*, a variant of closed categories where no structural law is required to be invertible, were first considered by Street [30].

We present several deductive systems giving different but equivalent presentations of the free skew prounital closed category on a set At: a categorical calculus (Hilbert-style system), a cut-free sequent calculus and a natural deduction calculus. Similarly to the skew monoidal case [35], sequents in sequent calculus have the form $S \mid \Gamma \longrightarrow C$ and the left implication rule only applies to the formula in the stoup position. The natural deduction calculus is, under Curry-Howard correspondence, a variant of planar typed lambda-calculus [1, 39]. Lambda-terms in this calculus have all free and bound variables used exactly once and in the order of their declaration.

We give two equivalent calculi of normal forms: a focused sequent calculus and normal natural deduction derivations, corresponding to canonical representatives of $\beta\eta$-equality. We show three reduction-free (in the sense that we do not use techniques from rewriting theory) normalization procedures: focusing [5], sending a sequent calculus derivation to a focused derivation; normalization by hereditary substitutions [38, 13], sending a natural deduction derivation to a focused derivation; normalization by evaluation [8, 4], sending a natural deduction derivation to a normal natural deduction derivation.

Using our sequent calculus, it is possible to show that the free skew prounital closed category on a set satisfies a *left-normality* condition. The structural law $\widehat{j}$, that we are not asking to be invertible, turns out to be invertible anyway. This degeneracy implies that our sequent calculus and natural deduction calculus admit a stoup-free presentation. In particular, the natural deduction calculus is equivalent, under Curry-Howard correspondence, to (non-skew) planar typed lambda-calculus. From a category-theoretic point of view, there is no reason to construct the free skew prounital closed category on a *set* instead of a more interesting category. We conclude this paper by discussing two equivalent presentations of the free skew prounital closed category on a *skew multicategory* [10] instead of a set: a Hilbert-style calculus and a cut-free sequent calculus. These constructions generalize the free construction on a set and do not generally entail the left-normality condition.

As explained by Shulman [28, Rev. 49], prounital closed categories are the natural notion of category with internal hom as the only required connective (when we do not have/do not want a monoidal structure $\mathsf{I},\otimes$ in the category), since they form an essential, in the sense of minimal, class of models for planar typed lambda-calculus. This is the case because, formally, they are equivalent to *closed multicategories* [23]. Multicategories are models of deductive systems with only identity and composition as basic operations, while closed multicategories are also able to model implication. Standard approaches to denotational semantics of typed lambda-calculus, adapted to the planar case, would exclude prounital closed categories as valid models. This is because these approaches usually require the presence of a Cartesian monoidal (just monoidal in the planar case) structure complementing the closed structure. We believe there is no good reason to discard models not interpreting the non-existing connectives $\mathsf{I},\otimes$ and prounital closed categories are the right notion of categorical model for planar typed lambda calculus. Analogously, skew prounital closed categories are the correct notion of category with skew internal hom as the only required connective, since they are equivalent to *closed skew multicategories* [10], a skew variant of closed multicategories.

It is worth mentioning that there is another way of weakening closed categories by simply dropping all references to the unit altogether, that is, by only asking for internal hom objects equipped with the extranatural transformation L and pentagon equation (c5) described below. These may be called *nonunital closed categories,* or *semi-closed categories* after Bourke [9], and are of some interest in providing interpretations for planar lambda terms with no closed subterms (a condition analogous to that of *bridgelessness* in graph theory [39]). We do not treat nonunital closed categories explicitly here, although we expect that our results may be adapted from the prounital to the nonunital case in a straightforward way.

We fully formalized the results presented in the paper in the dependently typed programming language Agda. The formalization uses Agda version 2.6.0. and it is available at `https://github.com/niccoloveltri/skew-prounital-closed-cats`.

2 Skew Prounital Closed Categories

A *skew prounital closed category* [36] is a category $\mathbb{C}$ equipped with functors $J : \mathbb{C} \to \mathbf{Set}$ (the *element set* functor) and $\multimap : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbb{C}$ (the *internal hom* functor) and (extra)natural transformations j, i, L typed

$$j_A \in J\,(A \multimap A) \qquad i_{A,B} : J\,A \to \mathbb{C}(A \multimap B, B) \qquad L_{A,B,C} \in \mathbb{C}(B \multimap C, (A \multimap B) \multimap (A \multimap C))$$

satisfying the following equations where we write $\circ_0 : \mathbb{C}(A,B) \times J\,A \to J\,B$ for J as a left action, i.e., $f \circ_0 e = J\,f\,e$:

(c1) $e = i_{A,A}\,e \circ_0 j_A \in J\,A$ for $e \in J\,A$;

(c2) $i_{A\multimap A, A\multimap C}\,j_A \circ L_{A,A,C} = \mathsf{id}_{A\multimap C} \in \mathbb{C}(A \multimap C, A \multimap C)$;

(c3) $L_{A,B,B} \circ_0 j_B = j_{A\multimap B} \in J\,((A \multimap B) \multimap (A \multimap B))$;

(c4) $i_{A,B}\,e \multimap C = ((A \multimap B) \multimap i_{A,C}\,e) \circ L_{A,B,C} \in \mathbb{C}(B \multimap C, (A \multimap B) \multimap C)$ for $e \in J\,A$;

(c5) $(B \multimap C) \multimap L_{A,B,D} \circ L_{B,C,D} = L_{A,B,C} \multimap ((A \multimap B) \multimap (A \multimap D)) \circ L_{A\multimap B, A\multimap C, A\multimap D} \circ L_{A,C,D}$ $\in \mathbb{C}(C \multimap D, (B \multimap C) \multimap ((A \multimap B) \multimap (A \multimap D)))$.

We typically write $\mathbb{C}(-,B)$ for $J\,B$. Let S be an *optional object*, i.e., S is either nothing (denoted $S=-$) or it is an object of $\mathbb{C}$. We define $\mathbb{C}(S,B)$ as $J\,B$ if $S=-$ and as $\mathbb{C}(A,B)$ if $S=A$. The use of this "enhanced" notion of homset with an optional object as domain allows the unification of the two notions of composition $\circ$ and $\circ_0$. We overload the composition symbol $\circ$: given $f\in\mathbb{C}(S,B)$ and $g\in\mathbb{C}(B,C)$, we write $g\circ f\in\mathbb{C}(S,C)$, which is equal to $g\circ_0 f$ when $S=-$ and it is equal to the usual composition of maps $g\circ f$ when $S=A$. With the new notation, the types of structural laws j and i become

$$j_A\in\mathbb{C}(-,A\multimap A)\qquad i_{A,B}:\mathbb{C}(-,A)\to\mathbb{C}(A\multimap B,B)$$

We note that it is not strictly necessary to require that $\multimap$ is a functor $\mathbb{C}^{\mathsf{op}}\times\mathbb{C}\to\mathbb{C}$. It suffices to require that $A\multimap\ :\mathbb{C}\to\mathbb{C}$ is a functor for every A, since the functorial action $\multimap B$ can for any B be defined from the rest of the structure as $f\multimap B = i_{A\multimap A',A\multimap B}\,((A\multimap f)\circ j_A)\circ L_{A,A',B}$ for $f:A\to A'$ and proved to preserve identity and composition and be natural in B from the equations governing it. Under such an alternative definition of skew prounital closed category, the equations (c4)–(c5) above have to be suitably adjusted and an equation for bifunctoriality of $\multimap$ added.

Skew prounital closed categories differ from Shulman's prounital closed categories in that the derivable map

$$\begin{aligned}&\widehat{j}_{A,B}:\mathbb{C}(A,B)\to\mathbb{C}(-,A\multimap B)\\&\widehat{j}_{A,B}\,f=(A\multimap f)\circ j_A\end{aligned}\tag{1}$$

is not required to be a natural isomorphism. A skew prounital closed category in which $\widehat{j}$ is invertible is called *left-normal*.

We should note that Shulman's prounital closed categories, although more normal than skew prounital closed categories, are nonetheless partially skew. One could also require *right-normality* and *associative-normality*, corresponding to invertibility of certain derivable maps $\widehat{\imath}$ and $\widehat{L}$ [36]. Eilenberg and Kelly's closed categories are partially skew in that they are not associative-normal.

Example. A simple example of a prounital closed category (adapted from de Schipper [29]) is obtained by taking a suitable full subcategory of the skeletal version $\mathbb{F}$ of the Cartesian monoidal closed category of finite sets and functions (i.e., exactly one set of each finite cardinality). Namely, we keep only cardinalities from $M\subseteq\mathbb{N}$ given inductively by: $3\in M$ and $n^m\in M$ for all $m,n\in M$. Now $1\notin M$ and $3\times 3=9\notin M$, so we have lost the original unit $\mathsf{I}=1$ and the tensor $p\otimes m=p\times m$, but we still have the internal hom given by $m\multimap n=n^m$. No other candidate unit or tensor can work since we need to have $m\cong\mathsf{I}\multimap m$ and $\mathbb{C}(p\otimes m,n)\cong\mathbb{C}(p,m\multimap n)$. Nevertheless, this category is prounital with the J functor given by the composition of inclusions $M\hookrightarrow\mathbb{F}\hookrightarrow\mathbf{Set}$, and j_m and $i_{m,n}$ defined in the evident way. This category is left-normal, but neither right-normal nor associative-normal.

To skew a closed category, one can use any left-strong monad on it [36]; the same construction works for a prounital closed category (the concept of left-strength of a monad has to be adjusted to this setting). We consider the reader monad given by $T\,m=m^k$ for some fixed $k\in M$. The Kleisli category is skew prounital closed with the internal hom defined by $m\multimap^T n=m\multimap T\,n=n^{k\times m}$. This category is neither left-normal nor right-normal or associative-normal.

Alternatively, we can begin with $\mathbb{F}$ and take the full subcategory corresponding to $M=\mathbb{N}\setminus\{0,1\}$. This time we get a prounital monoidal closed category. We can skew it as before and we still get a skew prounital nonmonoidal closed category since the Kleisli construction destroys the tensor.

A *strict prounital closed functor* between skew prounital closed categories $\mathbb{C}$ and $\mathbb{D}$ consists of a functor $F:\mathbb{C}\to\mathbb{D}$ such that $F\,(A\multimap B)=F\,A\multimap F\,B$, $F\,(f\multimap g)=F\,f\multimap F\,g$, and the structural laws j, i and L are preserved on the nose. In particular, the functor F is asked to map $\mathbb{C}(S,B)$ to $\mathbb{D}(F\,S,F\,B)$, with

$F\,S = -$ if $S = -$ and $F\,S = F\,A$ if $S = A$, and preserve the enhanced notion of composition $\circ : \mathbb{C}(B,C) \times \mathbb{C}(S,B) \to \mathbb{C}(S,C)$. Skew prounital closed categories and strict prounital closed functors between them form a category.

3 The Free Skew Prounital Closed Category on a Set

We now look at different presentations of the free skew prounital closed category on a set of generating objects. Let us first make explicit the definition of this free construction.

The *free* skew prounital closed category on a set At is a skew prounital closed category $\mathbf{FSkPCl}(\mathsf{At})$ equipped with an inclusion $\iota : \mathsf{At} \to \mathbf{FSkPCl}(\mathsf{At})$, interpreting elements of At as objects of $\mathbf{FSkPCl}(\mathsf{At})$. For any other skew prounital closed category $\mathbb{C}$ with a function $G : \mathsf{At} \to \mathbb{C}$, there must exist a unique strict prounital closed functor $\bar{G} : \mathbf{FSkPCl}(\mathsf{At}) \to \mathbb{C}$ compatible with ι.

The existence of the free skew prounital closed category $\mathbf{FSkPCl}(\mathsf{At})$ entails the existence of a left adjoint to the forgetful functor between the category of skew prounital closed categories and strict prounital closed functors and the category of sets and functions.

3.1 Categorical Calculus

The first presentation consists of a deductive system that we call the *categorical calculus* since it is directly derived from the definition of skew prounital closed category. (We could also think of it as a Hilbert-style calculus of sorts, or under the Curry-Howard correspondence, a combinatory logic.) Objects are *formulae* inductively generated as follows: a formula is either an element X of At (an *atomic* formula) or of the form $A \multimap B$, where A, B are formulae. We write Fma for the set of formulae.

Maps between an optional formula S and a formula C are *derivations* of the sequent $S \Longrightarrow C$, inductively generated by the following inference rules:

$$\frac{}{A \Longrightarrow A}\ \mathsf{id} \qquad \frac{S \Longrightarrow B \quad B \Longrightarrow C}{S \Longrightarrow C}\ \mathsf{comp} \qquad \frac{C \Longrightarrow A \quad B \Longrightarrow D}{A \multimap B \Longrightarrow C \multimap D}\ \multimap$$
$$\frac{}{\Longrightarrow A \multimap A}\ j \qquad \frac{\Longrightarrow A}{A \multimap B \Longrightarrow B}\ i \qquad \frac{}{B \multimap C \Longrightarrow (A \multimap B) \multimap (A \multimap C)}\ L \tag{2}$$

Derivations are *identified* up to a congruence relation $\doteq$ that is inductively generated by the following pairs of derivations:

$$\begin{array}{ll}
\text{(category laws)} & \mathsf{id} \circ f \doteq f \qquad f \doteq f \circ \mathsf{id} \qquad (f \circ g) \circ h \doteq f \circ (g \circ h) \\
(\multimap \text{ functorial}) & \mathsf{id} \multimap \mathsf{id} \doteq \mathsf{id} \qquad (f \circ h) \multimap (k \circ g) \doteq (h \multimap k) \circ (f \multimap g) \\
 & f \multimap \mathsf{id} \circ j \doteq \mathsf{id} \multimap f \circ j \\
(j,i,L \text{ (extra)nat. trans.}) & g \circ i(e) \circ h \multimap \mathsf{id} \doteq i(h \circ e) \circ \mathsf{id} \multimap g \\
 & (f \multimap g) \multimap (\mathsf{id} \multimap h) \circ L \doteq \mathsf{id} \multimap (f \multimap \mathsf{id}) \circ L \circ g \multimap h \\
 & i(e) \circ j \doteq e \qquad i(j) \circ L \doteq \mathsf{id} \\
\text{(c1-c5)} & L \circ j \doteq j \qquad \mathsf{id} \multimap i(e) \circ L \doteq i(e) \multimap \mathsf{id} \\
 & \mathsf{id} \multimap L \circ L \doteq L \multimap \mathsf{id} \circ L \circ L
\end{array} \tag{3}$$

In the term notation for derivations, we write $g \circ f$ for $\mathsf{comp}\, f\, g$ to agree with the standard categorical notation.

The categorical calculus defines the free skew prounital closed category on At in a straightforward way. Given another skew prounital closed category $\mathbb{C}$ with function $G : \mathsf{At} \to \mathbb{C}$, we can easily define mappings $\bar{G}_0 : \mathsf{Fma} \to \mathbb{C}_0$ and $\bar{G}_1 : S \Longrightarrow C \to \mathbb{C}(\bar{G}_0(S), \bar{G}_0(C))$ by induction. These specify a strict prounital closed functor, in fact the only existing one satisfying $\bar{G}_0(X) = G(X)$.

3.2 Cut-Free Sequent Calculus

The second presentation of **FSkPCl**(At) is a sequent calculus. Sequents are triples of the form $S \mid \Gamma \longrightarrow C$. The succedent C is a formula in Fma. The antecedent is split in two parts: the *stoup* S is an optional formula, i.e. it is either empty or it is a single formula; the *context* Γ is a list of formulae.

Derivations in the sequent calculus are inductively generated by the following inference rules:

$$\frac{A \mid \Gamma \longrightarrow C}{- \mid A,\Gamma \longrightarrow C}\ \mathsf{pass} \qquad \frac{S \mid \Gamma, A \longrightarrow B}{S \mid \Gamma \longrightarrow A \multimap B}\ {\multimap}\mathsf{R}$$
$$\frac{}{A \mid \ \longrightarrow A}\ \mathsf{ax} \qquad \frac{- \mid \Gamma \longrightarrow A \quad B \mid \Delta \longrightarrow C}{A \multimap B \mid \Gamma,\Delta \longrightarrow C}\ {\multimap}\mathsf{L} \tag{4}$$

(pass for 'passivate', L, R for introduction on the left (in the stoup) resp. right) and identified up to the congruence $\overset{\circ}{=}$ induced by the equations:

$$\begin{array}{ll} (\eta\text{-conversion}) & \mathsf{ax}_{A\multimap B} \overset{\circ}{=} {\multimap}\mathsf{R}\,({\multimap}\mathsf{L}\,(\mathsf{pass}\,\mathsf{ax}_A, \mathsf{ax}_B)) \\ & \\ (\text{commutative conversions}) & \\ \qquad \mathsf{pass}\,({\multimap}\mathsf{R}\,f) \overset{\circ}{=} {\multimap}\mathsf{R}\,(\mathsf{pass}\,f) & (\text{for } f : A' \mid \Gamma, A \longrightarrow B) \\ \qquad {\multimap}\mathsf{L}\,(f, {\multimap}\mathsf{R}\,g) \overset{\circ}{=} {\multimap}\mathsf{R}\,({\multimap}\mathsf{L}\,(f,g)) & (\text{for } f : - \mid \Gamma \longrightarrow A',\ g : B' \mid \Delta, A \longrightarrow B) \end{array} \tag{5}$$

There are no primitive cut rules in this sequent calculus, but two forms of cut are admissible:

$$\frac{S \mid \Gamma \longrightarrow A \quad A \mid \Delta \longrightarrow C}{S \mid \Gamma,\Delta \longrightarrow C}\ \mathsf{scut} \qquad \frac{- \mid \Gamma \longrightarrow A \quad S \mid \Delta_0, A, \Delta_1 \longrightarrow C}{S \mid \Delta_0, \Gamma, \Delta_1 \longrightarrow C}\ \mathsf{ccut} \tag{6}$$

Notice that the left rule ${\multimap}\mathsf{L}$ acts only on the implication $A \multimap B$ in the stoup. Another left rule ${\multimap}\mathsf{C}$ acting on implication in the passive context is derivable from cut:

$$\frac{- \mid \Gamma \xrightarrow{f} A \quad S \mid \Delta_0, B, \Delta_1 \xrightarrow{g} C}{S \mid \Delta_0, A \multimap B, \Gamma, \Delta_1 \longrightarrow C}\ {\multimap}\mathsf{C} \quad = \quad \frac{\dfrac{\dfrac{- \mid \Gamma \xrightarrow{f} A \quad \dfrac{}{B \mid \ \longrightarrow B}\ \mathsf{ax}}{A \multimap B \mid \Gamma \longrightarrow B}\ {\multimap}\mathsf{L}}{- \mid A \multimap B, \Gamma \longrightarrow B}\ \mathsf{pass} \qquad S \mid \Delta_0, B, \Delta_1 \xrightarrow{g} C}{S \mid \Delta_0, A \multimap B, \Gamma, \Delta_1 \longrightarrow C}\ \mathsf{ccut} \tag{7}$$

Soundness. Sequent calculus derivations can be turned into categorical calculus derivations using a function $\mathsf{sound} : (S \mid \Gamma \longrightarrow C) \to (S \Longrightarrow [\![\Gamma|C]\!])$, where the formula $[\![\Gamma|C]\!]$ is inductively defined as

$$[\![\ |C]\!] = C \qquad\qquad [\![A,\Gamma|C]\!] = A \multimap [\![\Gamma|C]\!]$$

Given $f : S \mid \Gamma, A \longrightarrow B$, define $\mathsf{sound}({\multimap}\mathsf{R}\ f)$ simply as $\mathsf{sound}(f)$. Given $f : A \mid \Gamma \longrightarrow C$, define $\mathsf{sound}(\mathsf{pass}\ f)$ as

$$\begin{array}{c} \dfrac{\dfrac{}{\Longrightarrow A \multimap A}\ j \qquad \dfrac{\dfrac{}{A \Longrightarrow A}\ \mathsf{id} \quad A \overset{\mathsf{sound}(f)}{\Longrightarrow} [\![\Gamma|C]\!]}{A \multimap A \Longrightarrow A \multimap [\![\Gamma|C]\!]}\ \multimap}{\Longrightarrow A \multimap [\![\Gamma|C]\!]}\ \mathsf{comp} \\ \hline\hline \Longrightarrow [\![A,\Gamma|C]\!] \end{array}$$

The double-line rule corresponds to the application of the equality $[\![A,\Delta|C]\!] = A \multimap [\![\Delta|C]\!]$. Given $f : - \mid \Gamma \longrightarrow A$ and $g : B \mid \Delta \longrightarrow C$, define $\mathsf{sound}({\multimap}\mathsf{L}(f,g))$ as

$$\frac{\dfrac{\dfrac{}{A \Longrightarrow A}\ \mathsf{id} \quad B \overset{\mathsf{sound}(g)}{\Longrightarrow} [\![\Delta|C]\!]}{A \multimap B \Longrightarrow A \multimap [\![\Delta|C]\!]}\ \multimap \qquad \dfrac{\dfrac{}{A \multimap [\![\Delta|C]\!] \Longrightarrow [\![\Gamma|A]\!] \multimap [\![\Gamma,\Delta|C]\!]}\ L^{\star} \quad \dfrac{\overset{\mathsf{sound}(f)}{\Longrightarrow} [\![\Gamma|A]\!]}{[\![\Gamma|A]\!] \multimap [\![\Gamma,\Delta|C]\!] \Longrightarrow [\![\Gamma,\Delta|C]\!]}\ i}{A \multimap [\![\Delta|C]\!] \Longrightarrow [\![\Gamma,\Delta|C]\!]}\ \mathsf{comp}}{A \multimap B \Longrightarrow [\![\Gamma,\Delta|C]\!]}\ \mathsf{comp}$$

where the operation $L^\star$, defined by induction on Γ, performs iterated applications of the structural law L. The function sound is well-defined, in the sense that it sends $\overset{\circ}{=}$-equivalent derivations to $\doteq$-related derivations.

Completeness. Derivations in the categorical calculus can be turned into sequent calculus derivations via a function $\mathsf{cmplt} : (S \Longrightarrow [\![\Gamma|C]\!]) \to (S \mid \Gamma \longrightarrow C)$. The ax rule models the identity map, while sequential composition is interpreted using scut. Functoriality of $\multimap$ is modelled using ${\multimap}\mathsf{R}$ and ${\multimap}\mathsf{L}$. The function cmplt sends the structural laws j,i,L of skew prounital closed categories to the following derivations in the sequent calculus:

$$(j) \qquad \dfrac{\dfrac{\dfrac{}{A \mid\ \longrightarrow A}\,\mathsf{ax}}{- \mid A \longrightarrow A}\,\mathsf{pass}}{- \mid\ \longrightarrow A \multimap A}\,{\multimap}\mathsf{R}$$

$$(i) \qquad \dfrac{- \mid\ \longrightarrow A \qquad \dfrac{}{B \mid\ \longrightarrow B}\,\mathsf{ax}}{A \multimap B \mid\ \longrightarrow B}\,{\multimap}\mathsf{L}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \mid\ \longrightarrow A}\,\mathsf{ax}}{- \mid A \longrightarrow A}\,\mathsf{pass} \qquad \dfrac{}{B \mid\ \longrightarrow B}\,\mathsf{ax}}{A \multimap B \mid A \longrightarrow B}\,{\multimap}\mathsf{L}}{- \mid A \multimap B, A \longrightarrow B}\,\mathsf{pass} \qquad \dfrac{}{C \mid\ \longrightarrow C}\,\mathsf{ax}}{B \multimap C \mid A \multimap B, A \longrightarrow C}\,{\multimap}\mathsf{L}}{B \multimap C \mid A \multimap B \longrightarrow A \multimap C}\,{\multimap}\mathsf{R}}{B \multimap C \mid\ \longrightarrow (A \multimap B) \multimap (A \multimap C)}\,{\multimap}\mathsf{R} \qquad (L)$$

The function cmplt is well-defined, in the sense that it sends $\doteq$-equivalent derivations to $\overset{\circ}{=}$-related derivations. Moreover it is possible to prove that cmplt is the inverse of sound (up to the equivalence relations $\doteq$ and $\overset{\circ}{=}$). This shows that the sequent calculus is a presentation of the free skew prounital closed category $\mathbf{FSkPCl}(\mathsf{At})$.

3.3 Natural Deduction Calculus

The third presentation of $\mathbf{FSkPCl}(\mathsf{At})$ is a natural deduction calculus. Sequents are triples $S \mid \Gamma \longrightarrow_{\mathsf{nd}} C$ as in the sequent calculus of Section 3.2, but the left rule ${\multimap}\mathsf{L}$ for $\multimap$ is replaced by the elimination rule ${\multimap}\mathsf{e}$.

$$\dfrac{A \mid \Gamma \longrightarrow_{\mathsf{nd}} C}{- \mid A, \Gamma \longrightarrow_{\mathsf{nd}} C}\,\mathsf{pass} \qquad \dfrac{S \mid \Gamma, A \longrightarrow_{\mathsf{nd}} B}{S \mid \Gamma \longrightarrow_{\mathsf{nd}} A \multimap B}\,{\multimap}\mathsf{i}$$

$$\dfrac{}{A \mid\ \longrightarrow_{\mathsf{nd}} A}\,\mathsf{ax} \qquad \dfrac{S \mid \Gamma \longrightarrow_{\mathsf{nd}} A \multimap B \quad - \mid \Delta \longrightarrow_{\mathsf{nd}} A}{S \mid \Gamma, \Delta \longrightarrow_{\mathsf{nd}} B}\,{\multimap}\mathsf{e}$$

The two cut rules in (6) are admissible also in the natural deduction calculus. Derivations are identified by the congruence relation $\overset{\circ}{=}_{\mathsf{nd}}$, a skew ordered variant of the usual $\beta\eta$-equivalence of simply typed lambda-calculus, induced by the equations

$$\begin{array}{lll}
(\beta\text{-conversion}) & {\multimap}\mathsf{e}\,({\multimap}\mathsf{i}\, f, g) \overset{\circ}{=}_{\mathsf{nd}} \mathsf{ccut}\,(g, f) & (\text{for } f : S \mid \Gamma, A \longrightarrow_{\mathsf{nd}} B,\ g : - \mid \Delta \longrightarrow_{\mathsf{nd}} A) \\
(\eta\text{-conversion}) & f \overset{\circ}{=}_{\mathsf{nd}} {\multimap}\mathsf{i}\,({\multimap}\mathsf{e}\,(f, \mathsf{pass}\ \mathsf{ax})) & (\text{for } f : S \mid \Gamma \longrightarrow_{\mathsf{nd}} A \multimap B) \\
(\text{commutative conversions}) & & \\
 & \mathsf{pass}\,({\multimap}\mathsf{i}\, f) \overset{\circ}{=}_{\mathsf{nd}} {\multimap}\mathsf{i}\,(\mathsf{pass}\, f) & (\text{for } f : A' \mid \Gamma, A \longrightarrow_{\mathsf{nd}} B) \\
 & \mathsf{pass}\,({\multimap}\mathsf{e}\,(f, g)) \overset{\circ}{=}_{\mathsf{nd}} {\multimap}\mathsf{e}\,(\mathsf{pass}\, f, g) & (\text{for } f : A' \mid \Gamma \longrightarrow_{\mathsf{nd}} A \multimap B,\ g : - \mid \Delta \longrightarrow_{\mathsf{nd}} A)
\end{array} \qquad (8)$$

This natural deduction calculus corresponds to a variant of the planar fragment of linear typed lambda-calculus [1, 39]. The formulae correspond to types. The derivations correspond to lambda terms in which all free and bound variables are used exactly once and in the order of their declaration. The equations axiomatize the appropriate variant of $\beta\eta$-equivalence.

It is possible to prove that the natural deduction calculus is equivalent to the sequent calculus (up to the equivalences of derivations $\overset{\circ}{=}$ and $\overset{\circ}{=}_{\mathsf{nd}}$), and it is therefore a presentation of $\mathbf{FSkPCl}(\mathsf{At})$. We do not follow this strategy here. Instead we construct reduction-free normalization procedures for the sequent calculus and the natural deduction calculus. The procedures target two calculi of normal forms: a focused subsystem of the sequent calculus and a calculus of $\beta\eta$-long normal forms wrt. $\overset{\circ}{=}_{\mathsf{nd}}$. By showing that the calculi of normal forms are equivalent, we conclude that the sequent calculus and the natural deduction calculus are also equivalent up to the equivalences of derivations $\overset{\circ}{=}$ and $\overset{\circ}{=}_{\mathsf{nd}}$.

3.4 Focused Sequent Calculus Derivations

The congruence relation $\overset{\circ}{=}$ on sequent calculus derivations can be considered as a term rewrite system, by directing every equation in (5) from left to right. The resulting rewrite system is weakly confluent and strongly normalizing, hence confluent with unique normal forms.

Derivations in normal form wrt. $\overset{\circ}{=}$ can be described by a suitable *focused* subcalculus of the full sequent calculus, following the paradigm introduced by Andreoli [5]. Derivations in this subcalculus are inductively generated by the following inference rules:

$$\frac{S \mid \Gamma, A \longrightarrow_{\mathsf{I}} B}{S \mid \Gamma \longrightarrow_{\mathsf{I}} A \multimap B}\multimap\mathsf{R} \qquad \frac{A \mid \Gamma \longrightarrow_{\mathsf{P}} C}{- \mid A, \Gamma \longrightarrow_{\mathsf{P}} C}\mathsf{pass} \qquad \frac{}{A \mid \quad \longrightarrow_{\mathsf{F}} A}\mathsf{ax}$$

$$\frac{S \mid \Gamma \longrightarrow_{\mathsf{P}} X}{S \mid \Gamma \longrightarrow_{\mathsf{I}} X}\mathsf{P2I} \qquad \frac{A \mid \Gamma \longrightarrow_{\mathsf{F}} C}{A \mid \Gamma \longrightarrow_{\mathsf{P}} C}\mathsf{F2P} \qquad \frac{- \mid \Gamma \longrightarrow_{\mathsf{I}} A \quad B \mid \Delta \longrightarrow_{\mathsf{F}} C}{A \multimap B \mid \Gamma, \Delta \longrightarrow_{\mathsf{F}} C}\multimap\mathsf{L} \tag{9}$$

This is a sequent calculus with an additional phase annotation on sequents, for controlling root-first proof search. In phase I (for *inversion*), sequents have the form $S \mid \Gamma \longrightarrow_{\mathsf{I}} C$, where S is a general stoup and C is a general formula. During this phase, we eagerly apply the invertible rule $\multimap$R until the succedent is reduced to an atomic formula. In phase P (for *passivation*), we have the opportunity of applying the pass rule and can only go to the last phase F (for *focusing*) when the stoup has become a formula. During this phase we can finish the derivation using ax, which is now restricted to atomic formulae, or apply the $\multimap$L rule. If we apply the $\multimap$L rule, we are thrown back to the I phase in the first premise. One can observe that, in any I-derivation, the succedent of any P- or F-sequent must actually be an atom, but the generality of allowing any formula in the succedent in these phases (which we can have in derivations of P- or F-sequents) will be useful for us shortly in the discussion of hereditary substitutions below and also in Section 3.5 where we will relate focused sequent calculus derivations to normal natural deduction calculus derivations.

Focusing. The focused rules define a sound and complete root-first proof search strategy for the cut-free sequent calculus of Section 3.2. Soundness of the focused calculus is evident: focused derivations can be embedded into sequent calculus derivations via functions $\mathsf{emb}_k : (S \mid \Gamma \longrightarrow_k C) \to (S \mid \Gamma \longrightarrow C)$ for all phases $k \in \{\mathsf{I}, \mathsf{P}, \mathsf{F}\}$ that just erase all phase annotations and uses of the rules P2I and F2P.

By the normalization property of the rewrite system associated to $\overset{\circ}{=}$, we know that the focused calculus is also complete. This can also be established by constructing a reduction-free normalization function $\mathsf{focus} : (S \mid \Gamma \longrightarrow C) \to (S \mid \Gamma \longrightarrow_{\mathsf{I}} C)$ sending each derivation in the sequent calculus to a canonical representative of its $\overset{\circ}{=}$-equivalence class in the focused calculus. This means in particular that focus maps $\overset{\circ}{=}$-related derivations to equal focused derivations. For the definition of focus, we show that general pass, $\multimap\mathsf{L}$ and ax rules are admissible in phase I:

$$\frac{A \mid \Gamma \longrightarrow_{\mathsf{I}} C}{- \mid A,\Gamma \longrightarrow_{\mathsf{I}} C}\ \mathsf{pass}^{\mathsf{I}} \qquad \frac{- \mid \Gamma \longrightarrow_{\mathsf{I}} A \quad B \mid \Delta \longrightarrow_{\mathsf{I}} C}{A \multimap B \mid \Gamma,\Delta \longrightarrow_{\mathsf{I}} C}\ \multimap\mathsf{L}^{\mathsf{I}} \qquad \frac{}{A \mid \ \longrightarrow_{\mathsf{I}} A}\ \mathsf{ax}^{\mathsf{I}}$$

This makes each sequent calculus inference rule matched by a focused calculus rule. Then the normalization procedure focus can be easily defined by induction on the input derivation. The function focus is the inverse of $\mathsf{emb}_{\mathsf{I}}$ up to $\overset{\circ}{=}$: given a sequent calculus derivation $f : S \mid \Gamma \longrightarrow C$, we have $\mathsf{emb}_{\mathsf{I}}\ (\mathsf{focus}\ f) \overset{\circ}{=} f$; given a focused derivation $f : S \mid \Gamma \longrightarrow_{\mathsf{I}} C$, we have $\mathsf{focus}\ (\mathsf{emb}_{\mathsf{I}}\ f) = f$.

The focused calculus solves the *coherence problem* for skew prounital closed categories, in the sense of giving an explicit characterization of the homsets of $\mathbf{FSkPCl}(\mathsf{At})$. It also solves two related algorithmic problems effectively:

- Duplicate-free enumeration of all maps $S \Longrightarrow C$ in the form of representatives of $\dot{=}$-equivalence classes of categorical calculus derivations: For this, find all focused derivations of $S \mid \ \longrightarrow_{\mathsf{I}} C$, which is solvable by exhaustive proof search, which terminates, and translate them to the categorical calculus derivations.
- Finding whether two given maps of type $S \Longrightarrow C$, presented as categorical calculus derivations, are equal, i.e., $\dot{=}$-related as derivations: For this, translate them to focused derivations of $S \mid \ \longrightarrow_{\mathsf{I}} C$ and check whether they are equal, which is decidable.

Monoidal and nonmonoidal closed categories, and prounital closed categories likewise, admit no simple coherence theorem like Mac Lane's for monoidal categories [22] (depending on an easy condition on the domain and codomain, no maps or just one in a homset). Enumeration of (presentations of) maps and equality checking are nontrivial [15, 21, 26, 32]. In our focused calculus, the sequent $(X \multimap Y) \multimap (X \multimap Z) \mid X \multimap Y, X \multimap X, X \longrightarrow_{\mathsf{I}} Z$ has two distinct focused derivations (we learned this example from Anupam Das).

Hereditary Substitutions. Focused sequent calculus derivations can also be used as normal forms for the natural deduction calculus of Section 3.3. We show this by describing a reduction-free normalization procedure that is typically called *normalization by hereditary substitution* [38, 13]. Normal forms for this procedure (at least in the case of simply-typed lambda-calculus) are typically defined in terms of a suitable spine calculus, but we have defined our focused sequent calculus liberally enough to serve this purpose.

The focused calculus defines a sound and complete root-first proof search strategy for the natural deduction calculus of Section 3.3. Focused derivations can easily be embedded into natural deduction derivations: there are functions $\mathsf{emb}^{\mathsf{nd}}_k : (S \mid \Gamma \longrightarrow_k C) \to (S \mid \Gamma \longrightarrow_{\mathsf{nd}} C)$ for all $k \in \{\mathsf{I}, \mathsf{P}, \mathsf{F}\}$.

Similar to focusing completeness, normalization by hereditary substitution is also specified in two steps. First we need to show that ax and $\multimap\mathsf{e}$ rules are admissible in phase I:

$$\frac{}{A \mid \ \longrightarrow_{\mathsf{I}} A}\ \mathsf{ax}^{\mathsf{I}} \qquad \frac{S \mid \Gamma \longrightarrow_{\mathsf{I}} A \multimap B \quad - \mid \Delta \longrightarrow_{\mathsf{I}} A}{S \mid \Gamma,\Delta \longrightarrow_{\mathsf{I}} B}\ \multimap\mathsf{e}^{\mathsf{I}}$$

(We already know that a general pass rule is admissible in phase I[1].) Focused derivations in phase I should correspond to normal forms, i.e., canonical representatives of $\overset{\circ}{=}_{\mathsf{nd}}$-equivalence classes. In particular,

[1] We also already know from focusing that ax^{I} is admissible, but it is defined in terms of $\multimap\mathsf{L}^{\mathsf{I}}$. For normalization by hereditary substitutions, we define ax^{I} differently, avoiding the use of $\multimap\mathsf{L}^{\mathsf{I}}$ since $\multimap\mathsf{L}$ is not a primitive rule of the natural deduction calculus.

they should not contain any redex. This forces the rule $\multimap\mathsf{e}^{\mathsf{l}}$ to be simultaneously defined with 3 pairs of substitution rules (i.e., cut rules) in the focused calculus, one for each phase $k \in \{\mathsf{I},\mathsf{P},\mathsf{F}\}$:

$$\frac{S \mid \Gamma \longrightarrow_k A \quad A \mid \Delta \longrightarrow_k C}{S \mid \Gamma,\Delta \longrightarrow_k C}\;\mathsf{scut}^k \qquad \frac{- \mid \Gamma \longrightarrow_k A \quad S \mid \Delta_0,A,\Delta_1 \longrightarrow_k C}{S \mid \Delta_0,\Gamma,\Delta_1 \longrightarrow_k C}\;\mathsf{ccut}^k$$

The rule $\multimap\mathsf{e}^{\mathsf{l}}$ can then be defined as dictated by the β-conversion equation in the definition of $\stackrel{\circ}{=}_{\mathsf{nd}}$ as

$$\frac{\dfrac{S \mid \Gamma,A \xrightarrow{f}_{\mathsf{l}} B}{S \mid \Gamma \longrightarrow_{\mathsf{l}} A \multimap B}\;\multimap\mathsf{i} \qquad - \mid \Delta \xrightarrow{g}_{\mathsf{l}} A}{S \mid \Gamma,\Delta \longrightarrow_{\mathsf{l}} B}\;\multimap\mathsf{e}^{\mathsf{l}} \quad = \quad \frac{- \mid \Delta \xrightarrow{g}_{\mathsf{l}} A \quad S \mid \Gamma,A \xrightarrow{f}_{\mathsf{l}} B}{S \mid \Gamma,\Delta \longrightarrow_{\mathsf{l}} B}\;\mathsf{ccut}^{\mathsf{l}}$$

Notice that the first premise is forced to be of the form $\multimap\mathsf{i}\ f$. This simultaneous substitution in canonical forms and reduction of redexes that appear from substitution is the main idea behind hereditary substitutions. We can then construct a normalization function $\mathsf{hered} : (S \mid \Gamma \longrightarrow_{\mathsf{nd}} C) \to (S \mid \Gamma \longrightarrow_{\mathsf{l}} C)$ sending each primitive rule of the natural deduction calculus to its admissible counterpart in the focused calculus. In particular, the function hered maps each natural deduction derivation to its normal form as rendered in the focused calculus (which is our spine calculus).

The function hered is the inverse of $\mathsf{emb}^{\mathsf{nd}}_{\mathsf{l}}$ up to $\stackrel{\circ}{=}_{\mathsf{nd}}$: given a natural deduction derivation $f : S \mid \Gamma \longrightarrow_{\mathsf{nd}} C$, we have $\mathsf{emb}^{\mathsf{nd}}_{\mathsf{l}}\ (\mathsf{hered}\ f) \stackrel{\circ}{=}_{\mathsf{nd}} f$; given a focused derivation $f : S \mid \Gamma \longrightarrow_{\mathsf{l}} C$, we get $\mathsf{hered}\ (\mathsf{emb}^{\mathsf{nd}}_{\mathsf{l}}\ f) = f$.

3.5 Normal Natural Deduction Derivations

The congruence relation $\stackrel{\circ}{=}_{\mathsf{nd}}$ on natural deduction calculus derivations also has normal forms, which correspond precisely to $\beta\eta$-long normal forms in the familiar terminology of lambda-calculus.

Derivations in normal form wrt. $\stackrel{\circ}{=}_{\mathsf{nd}}$ can be described by a suitable subcalculus of the full natural deduction calculus. Derivations in this subcalculus are inductively generated by the following inference rules:

$$\frac{S \mid \Gamma,A \longrightarrow_{\mathsf{nf}} B}{S \mid \Gamma \longrightarrow_{\mathsf{nf}} A \multimap B}\;\multimap\mathsf{i} \qquad \frac{A \mid \Gamma \longrightarrow_{\mathsf{p}} C}{- \mid A,\Gamma \longrightarrow_{\mathsf{p}} C}\;\mathsf{pass} \qquad \frac{}{A \mid \quad \longrightarrow_{\mathsf{ne}} A}\;\mathsf{ax}$$

$$\frac{S \mid \Gamma \longrightarrow_{\mathsf{p}} X}{S \mid \Gamma \longrightarrow_{\mathsf{nf}} X}\;\mathsf{p2nf} \qquad \frac{A \mid \Gamma \longrightarrow_{\mathsf{ne}} C}{A \mid \Gamma \longrightarrow_{\mathsf{p}} C}\;\mathsf{ne2p} \qquad \frac{A' \mid \Gamma \longrightarrow_{\mathsf{ne}} A \multimap B \quad - \mid \Delta \longrightarrow_{\mathsf{nf}} A}{A' \mid \Gamma,\Delta \longrightarrow_{\mathsf{ne}} B}\;\multimap\mathsf{e}$$

Derivations are organized in an introduction phase and in an elimination phase [27]. In lambda-calculus jargon, we refer to derivations in these phases as (pure) *normal forms* and *neutrals*. Normal forms are derivations of sequents of the general form $S \mid \Gamma \longrightarrow_{\mathsf{nf}} C$. Similarly to the case of simply-typed lambda-calculus, a normal form is an iteration of λ-abstraction on a neutral term of an atomic type. Neutrals are derivations of sequents of the form $A \mid \Gamma \longrightarrow_{\mathsf{ne}} C$ where the stoup is required to be a formula. Intuitively, they correspond to terms which are stuck for $\stackrel{\circ}{=}_{\mathsf{nd}}$-conversion. A neutral is either a variable (declared in the stoup, in our case) or a function application that cannot compute due to the presence of another neutral in the function position. Due to the skew aspect of our natural deduction calculus, we also add an intermediate third phase p, with sequents of the form $A \mid \Gamma \longrightarrow_{\mathsf{p}} C$, in which we have the choice of applying the structural rule pass. This is analogous to the passivation phase of the focused sequent calculus of Section 3.4.

The normal natural deduction calculus defines a root-first proof search strategy for the natural deduction calculus. This procedure is sound. Normal forms can easily be embedded into natural deduction derivations: there are functions $\mathsf{emb}^{\mathsf{nd}}_{k} : (S \mid \Gamma \longrightarrow_k C) \to (S \mid \Gamma \longrightarrow_{\mathsf{nd}} C)$ for all $k \in \{\mathsf{nf},\mathsf{p},\mathsf{ne}\}$.

Normalization by Evaluation. The completeness of the normal natural deduction calculus, implying that normal natural deduction derivations are indeed $\beta\eta$-long normal forms, is proved via *normalization by evaluation* [8, 4]. This is a reduction-free procedure in which terms are first evaluated into a certain semantic domain, and their evaluations are then reified back into normal forms.

We begin by constructing two discrete categories: **Cxt** and **SCxt**. The category **Cxt** has lists of formulae as objects. It has a strict monoidal structure with the empty list as unit and concatenation of lists as tensor. The category **SCxt** has objects of the form $S \mid \Gamma$, with S an optional formula and Γ a list of formulae. It has a unit object $- \mid$ (in which both components are empty) and there exists an action of the monoidal category **Cxt** on **SCxt**: $(S \mid \Gamma) \cdot \Gamma' = S \mid \Gamma, \Gamma'$. There is a functor $E : \mathbf{Cxt} \to \mathbf{SCxt}$, sending each list Γ to the pair $- \mid \Gamma$.

We consider the two presheaf categories $\mathbf{Set}^{\mathbf{Cxt}}$ and $\mathbf{Set}^{\mathbf{SCxt}}$. The monoidal structure on **Cxt** lifts to the *Day convolution* monoidal closed structure on $\mathbf{Set}^{\mathbf{Cxt}}$:

$$\mathsf{I}_{\mathsf{cxt}}\,\Gamma = (\Gamma = ()) \qquad (P \otimes_{\mathsf{cxt}} Q)\,\Gamma = \textstyle\sum_{\Gamma_0,\Gamma_1} (\Gamma = \Gamma_0,\Gamma_1) \times P\,\Gamma_0 \times Q\,\Gamma_1$$
$$(Q \multimap_{\mathsf{cxt}} P)\,\Gamma = \textstyle\prod_{\Delta}\, Q\,\Delta \to P\,(\Gamma,\Delta)$$

(Here and below () denotes the empty list.)

The unit of **SCxt** lifts to a unit in $\mathbf{Set}^{\mathbf{SCxt}}$ given by $\mathsf{I}_{\mathsf{scxt}}\,(S \mid \Gamma) = (\Gamma = ()) \times (S = -)$. The action of **Cxt** on **SCxt** lifts to an action of $\mathbf{Set}^{\mathbf{Cxt}}$ on $\mathbf{Set}^{\mathbf{SCxt}}$:

$$(P \otimes_{\mathsf{scxt}} Q)\,(S \mid \Gamma) = \textstyle\sum_{\Gamma_0,\Gamma_1} (\Gamma = \Gamma_0,\Gamma_1) \times P\,(S \mid \Gamma_0) \times Q\,\Gamma_1$$

The functor $\otimes_{\mathsf{scxt}} Q$ has a right adjoint $Q \multimap_{\mathsf{scxt}}$ given by:

$$(Q \multimap_{\mathsf{scxt}} P)\,(S \mid \Gamma) = \textstyle\prod_{\Delta}\, Q\,\Delta \to P\,(S \mid \Gamma,\Delta)$$

The first step of normalization by evaluation is the interpretation of syntactic constructs, i.e. formulae and natural deduction derivations, as semantic entities in $\mathbf{Set}^{\mathbf{SCxt}}$. Formulae are modelled as presheaves over **SCxt**. Implication is modelled via the functor $\multimap_{\mathsf{scxt}}$. Notice the composition with the functor E, which is needed for the interpretation to be well-defined. The interpretation of an atomic formula X on an object $S \mid \Gamma$ is the set of normal forms of type $S \mid \Gamma \longrightarrow_{\mathsf{nf}} X$.

$$\{\!\{X\}\!\}\,(S \mid \Gamma) = S \mid \Gamma \longrightarrow_{\mathsf{nf}} X \qquad \{\!\{A \multimap B\}\!\}\,(S \mid \Gamma) = ((\{\!\{A\}\!\} \circ E) \multimap_{\mathsf{scxt}} \{\!\{B\}\!\})\,(S \mid \Gamma)$$

Lists of formulae can be interpreted as presheaves over **Cxt**.

$$\{\!\{\ \}\!\}\,\Delta = \mathsf{I}_{\mathsf{cxt}}\,\Delta \qquad \{\!\{A,\Gamma\}\!\}\,\Delta = ((\{\!\{A\}\!\} \circ E) \otimes_{\mathsf{cxt}} \{\!\{\Gamma\}\!\})\,\Delta$$

Finally, antecedents $S \mid \Gamma$ can be interpreted as presheaves over **SCxt**:

$$\{\!\{- \mid \Gamma\}\!\}\,(S \mid \Delta) = (\mathsf{I}_{\mathsf{scxt}} \otimes_{\mathsf{scxt}} \{\!\{\Gamma\}\!\})\,(S \mid \Delta) = (S = -) \times \{\!\{\Gamma\}\!\}\,\Delta$$
$$\{\!\{A \mid \Gamma\}\!\}\,(S \mid \Delta) = (\{\!\{A\}\!\} \otimes_{\mathsf{scxt}} \{\!\{\Gamma\}\!\})\,(S \mid \Delta)$$

The next step of normalization by evaluation is the interpretation of a derivation $f : S \mid \Gamma \longrightarrow_{\mathsf{nd}} C$ in the natural deduction calculus as a natural transformation between presheaves $\{\!\{S \mid \Gamma\}\!\}$ and $\{\!\{C\}\!\}$. Formally, we define an evaluation function by induction on the input derivation:

$$\mathsf{eval} : (S \mid \Gamma \longrightarrow_{\mathsf{nd}} C) \to \{\!\{S \mid \Gamma\}\!\}\,(S' \mid \Delta) \to \{\!\{C\}\!\}\,(S' \mid \Delta)$$

Subsequently, we extract a normal form from the evaluated term. The reification procedure sends a semantic element in $\{\!\{A\}\!\}\ (S \mid \Gamma)$ to a normal form in $S \mid \Gamma \longrightarrow_{\mathsf{nf}} A$. The latter is defined by mutual induction with a function reflecting neutrals in $A \mid \Gamma \longrightarrow_{\mathsf{ne}} C$ to semantic elements in $\{\!\{C\}\!\}\ (A \mid \Gamma)$.

$$\mathsf{reflect} : (A \mid \Gamma \longrightarrow_{\mathsf{ne}} C) \to \{\!\{C\}\!\}\ (A \mid \Gamma) \qquad\qquad \mathsf{reify} : \{\!\{A\}\!\}\ (S \mid \Gamma) \to (S \mid \Gamma \longrightarrow_{\mathsf{nf}} A)$$

Finally, a normalization procedure $\mathsf{nbe} : (S \mid \Gamma \longrightarrow_{\mathsf{nd}} C) \to (S \mid \Gamma \longrightarrow_{\mathsf{nf}} C)$ is defined as follows. Apply eval to a given derivation $f : S \mid \Gamma \longrightarrow_{\mathsf{nd}} C$ in the natural deduction calculus, obtaining a natural transformation $\mathsf{eval}\ f$ between presheaves $\{\!\{S \mid \Gamma\}\!\}$ and $\{\!\{C\}\!\}$. Take the component of $\mathsf{eval}\ f$ at $S \mid \Gamma$, which is a function of type $\{\!\{S \mid \Gamma\}\!\}\ (S \mid \Gamma) \to \{\!\{C\}\!\}\ (S \mid \Gamma)$. By induction on S and Γ, it is possible to define a canonical element $\gamma : \{\!\{S \mid \Gamma\}\!\}(S \mid \Gamma)$. This allows to obtain an element $\mathsf{eval}\ f\ \gamma : \{\!\{C\}\!\}\ (S \mid \Gamma)$, which can finally be reified into a normal form:

$$\mathsf{nbe}\ f = \mathsf{reify}\ (\mathsf{eval}\ f\ \gamma)$$

We formally verified that the function nbe is well-defined, i.e. it sends $\overset{\circ}{=}_{\mathsf{nd}}$-related derivations to the same normal form. Moreover, nbe is the inverse up to $\overset{\circ}{=}_{\mathsf{nd}}$ of the embedding $\mathsf{emb}^{\mathsf{nd}}_{\mathsf{nf}} : (S \mid \Gamma \longrightarrow_{\mathsf{nf}} C) \to (S \mid \Gamma \longrightarrow_{\mathsf{nd}} C)$ of normal forms into natural deduction derivations: given a natural deduction derivation $f : S \mid \Gamma \longrightarrow_{\mathsf{nd}} C$, we have $\mathsf{emb}^{\mathsf{nd}}_{\mathsf{nf}}\ (\mathsf{nbe}\ f) \overset{\circ}{=}_{\mathsf{nd}} f$; given a normal form $f : S \mid \Gamma \longrightarrow_{\mathsf{nf}} C$, we have $\mathsf{nbe}\ (\mathsf{emb}^{\mathsf{nd}}_{\mathsf{nf}}\ f) = f$.

Comparing Normal Forms. We conclude this section by showing that focused sequent calculus derivations and normal natural deduction derivations are in one-to-one correspondence. Notice that we have already established a one-to-one correspondence indirectly: the correctness of normalization by hereditary substitution implies that the set of focused calculus derivations $S \mid \Gamma \longrightarrow_{\mathsf{I}} C$ is isomorphic to the set of natural deduction derivations $S \mid \Gamma \longrightarrow_{\mathsf{nd}} C$ quotiented by the equivalence relation $\overset{\circ}{=}_{\mathsf{nd}}$, which is further isomorphic to the set of normal natural deduction derivations $S \mid \Gamma \longrightarrow_{\mathsf{nf}} C$ thanks to the correctness of normalization by evaluation. The goal of this section is to provide a simple direct comparison of the two classes of normal forms.

The crucial step of this comparison is the relation between neutrals and derivations in phase F. This is because normal forms in phase nf have the same primitive inference rules derivations in phase I, and similarly for derivations of the passivation phases of the two calculi. We simultaneously define six translations back and forth between the three pairs of corresponding phases of the two calculi. We only show the constructions of the translations $\mathsf{ne2F}$ and $\mathsf{F2ne}$ between neutrals and derivations in phase F. The functions $\mathsf{nf2I}$ and $\mathsf{I2nf}$ for translating between normal forms and derivations in phase I are trivially defined, similarly for the functions translating between the p and P phases. The definitions of translations $\mathsf{ne2F}$ and $\mathsf{F2ne}$ rely on two auxiliary functions $\mathsf{ne2F}'$ and $\mathsf{F2ne}'$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{ne2F}' : (A \mid \Gamma \longrightarrow_{\mathsf{ne}} B) \to (B \mid \Delta \longrightarrow_{\mathsf{F}} C) \to (A \mid \Gamma, \Delta \longrightarrow_{\mathsf{F}} C)} \\
\mathsf{ne2F}'\ \mathsf{ax} & g = g \\
\mathsf{ne2F}'\ ({\multimap}\mathsf{e}\ (f, a)) & g = \mathsf{ne2F}'\ f\ ({\multimap}\mathsf{L}\ (\mathsf{nf2I}\ a, g))
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{F2ne}' : (A \mid \Gamma \longrightarrow_{\mathsf{ne}} B) \to (B \mid \Delta \longrightarrow_{\mathsf{F}} C) \to (A \mid \Gamma, \Delta \longrightarrow_{\mathsf{ne}} C)} \\
\mathsf{F2ne}'\ f\ \mathsf{ax} & = f \\
\mathsf{F2ne}'\ f\ ({\multimap}\mathsf{L}\ (a, g)) & = \mathsf{F2ne}'\ ({\multimap}\mathsf{e}\ (f, \mathsf{I2nf}\ a))\ g
\end{array}$$

Remember that neutrals are lambda-terms of the form $x\ a_1\ \ldots\ a_n$ with x being a variable (the only one) declared in the stoup. The accumulator g in the definition of $\mathsf{ne2F}'$ is intended to collect the arguments $a_i, \ldots, a_n$ that have already been seen. So when a new argument a appears, which is a normal form, this is immediately translated to an l-phase derivation via $\mathsf{nf2l}$ and then pushed on top of the accumulator using the left rule $\multimap\mathsf{L}$. The accumulator f in the definition of $\mathsf{F2ne}'$ serves a similar purpose. The translations $\mathsf{ne2F}$ and $\mathsf{F2ne}$ are then easily definable:

$$\begin{array}{ll} \mathsf{ne2F} : (A \mid \Gamma \longrightarrow_{\mathsf{ne}} C) \to (A \mid \Gamma \longrightarrow_{\mathsf{F}} C) & \qquad \mathsf{F2ne} : (A \mid \Gamma \longrightarrow_{\mathsf{F}} C) \to (A \mid \Gamma \longrightarrow_{\mathsf{ne}} C) \\ \mathsf{ne2F}\ f = \mathsf{ne2F}'\ f\ \mathsf{ax} & \qquad \mathsf{F2ne}\ f = \mathsf{F2ne}'\ \mathsf{ax}\ f \end{array}$$

These translations form an isomorphism. The crucial lemma for proving this is: given $f : A \mid \Gamma \longrightarrow_{\mathsf{ne}} B$ and $g : B \mid \Delta \longrightarrow_{\mathsf{F}} C$, we have $\mathsf{ne2F}\ (\mathsf{F2ne}'\ f\ g) = \mathsf{ne2F}'\ f\ g$ and $\mathsf{F2ne}\ (\mathsf{ne2F}'\ f\ g) = \mathsf{F2ne}'\ f\ g$.

4 Losing Skewness and How to Restore It

The free skew prounital closed category $\mathbf{FSkPCl}(\mathsf{At})$ on a set of atoms At is left normal, which means that its skew aspect is superfluous. In other words, $\mathbf{FSkPCl}(\mathsf{At})$ is also the free (non-skew) prounital closed category on At. An advantage of our proof theoretic analysis is that left-normality can be proved in any one of the equivalent calculi of Section 3. Left-normality is a simple observation in the sequent calculus, while it is not clear how to derive it directly in the categorical calculus of Section 3.1.

First we notice that left-normality, defined as the invertibility of the derivable map $\widehat{j}$ of (1), is translated to the invertibility of the passivation rule pass in the sequent calculus of Section 3.2. In other words, $\widehat{j}$ is invertible up to $\doteq$ in the categorical calculus if and only if pass is invertible up to $\overset{\circ}{=}$ in the sequent calculus. Then we show that pass has as inverse the admissible rule act:

$$\frac{- \mid A, \Gamma \longrightarrow C}{A \mid \Gamma \longrightarrow C}\ \mathsf{act} \tag{10}$$

This is defined by induction on the given derivation $f : - \mid A, \Gamma \longrightarrow C$. There are only two possible cases: if $f = \mathsf{pass}\ f'$, define $\mathsf{act}\ f = f'$; if $f = \multimap\mathsf{R}\ f'$, define $\mathsf{act}\ f = \multimap\mathsf{R}\ (\mathsf{act}\ f')$.

An important consequence of left-normality is that all calculi described in Sections 3.2–3.5 admit a presentation without the stoup and the pass rule. In particular, the natural deduction calculus of Section 3.3 is equivalent to (non-skew) planar simply-typed lambda-calculus [1, 39]. The categorical calculus of 3.1 also admits a stoup-free version where sequents take the form $\Longrightarrow A$ where A is a formula. The inference rules are

$$\frac{\Longrightarrow B \quad \Longrightarrow B \multimap C}{\Longrightarrow C}\ \mathsf{comp}'$$

$$\frac{}{\Longrightarrow A \multimap A}\ j \qquad \frac{\Longrightarrow A}{\Longrightarrow (A \multimap B) \multimap B}\ i' \qquad \frac{}{\Longrightarrow (B \multimap C) \multimap ((A \multimap B) \multimap (A \multimap C))}\ L' \tag{11}$$

Under the Curry-Howard correspondence, this is the combinatory logic capturing planar lambda-calculus: comp' is application, j is the I-combinator, and L' is the B-combinator, while the operation i' replaces the C-combinator of BCI combinatory logic [25] and is needed in the absence of symmetry.

A natural question arises: why did we bother including the stoup in our calculi in the first place? There are two reasons behind our choice to include the stoup.

First, in the future we plan to extend the skew calculi described in this paper with other connectives, such as unit and tensor. We already know from previous work on the proof theory of skew monoidal categories [35] that the extended calculi will not be left-normal, so we will not be able to discard the stoup. We believe that a thorough investigation of the normalization procedures of Sections 3.4 and 3.5, which work in the presence of the stoup, is a stepping stone towards the development of normalization functions for more involved calculi with additional connectives.

Second, the left-normality of $\mathbf{FSkPCl}(\mathsf{At})$ arises from the fact that we are considering the free skew prounital closed category on a *set*. In other words, it corresponds to a left adjoint to the forgetful functor between the category of skew prounital closed categories and strict prounital closed functors and the category of sets and functions. From a categorical point of view, there is no good reason to privilege the category of sets and functions in this picture. The next subsection is devoted to the study of the free skew prounital closed category $\mathbf{FSkPCl}(\mathbb{M})$ on a skew multicategory $\mathbb{M}$. The category $\mathbf{FSkPCl}(\mathsf{At})$ arises as a particular instance of the latter more general construction. Crucially, $\mathbf{FSkPCl}(\mathbb{M})$ is generally not left-normal.

4.1 The Free Skew Prounital Closed Category on a Skew Multicategory

We start by recollecting Bourke and Lack's notion of skew multicategory [10]. We slightly reformulate Bourke and Lack's definition to make its relationship to the sequent calculus of Section 3.2 more direct. Skew multicategories are similar to the multicategories of Lambek [18] (also known as colored (non-symmetric) operads), but instead use an optional object paired with a list of objects as the domain of a multimap, rather than just a list of objects.

A *skew multicategory* $\mathbb{M}$ consists of a set $\mathbb{M}_0$ of objects and, for any optional object S, list of objects Γ and object C in $\mathbb{M}_0$, a set $\mathbb{M}(S|\Gamma;C)$ of multimaps whereby a multimap is called *loose* if S is empty and *tight* if S is an object. For any object A, there is an identity multimap $\mathsf{id} \in \mathbb{M}(A|\ ;A)$. There are two composition operations $\mathsf{s}\circ : \mathbb{M}(A|\Delta;C) \times \mathbb{M}(S|\Gamma;A) \to \mathbb{M}(S|\Gamma,\Delta;C)$ and $\mathsf{c}\circ : \mathbb{M}(S|\Delta_0,A,\Delta_1;C) \times \mathbb{M}(-|\Gamma;A) \to \mathbb{M}(S|\Delta_0,\Gamma,\Delta_1;C)$ and a loosening operation $\mathsf{loosen} : \mathbb{M}(A|\Gamma;C) \to \mathbb{M}(-|A,\Gamma;C)$ satisfying a large number of equations, expressing unitality of identity wrt. composition, associativity of composition, commutativity of parallel cuts and commutativity of composition and loosening. See the whole list of equations in our previous work [35].

A *skew multifunctor* G between skew multicategories $\mathbb{M}$ and $\mathbb{M}'$ consists of a function G_0 sending objects of $\mathbb{M}$ to objects of $\mathbb{M}'$ and a function $G_1 : \mathbb{M}(S|\Gamma;C) \to \mathbb{M}'(G_0S|G_0\Gamma;G_0C)$ preserving identity, composition and loosening. Here G_0 is extended to optional objects and lists of objects by $G_0 S = -$ if $S = -$ and $G_0 S = G_0 A$ if $S = A$. Similarly, $G_0(A_1,\ldots,A_n) = G_0A_1,\ldots,G_0A_n$. Skew multicategories and skew multifunctors form a category. There exists a forgetful functor U between the category of skew prounital closed categories and the latter category. Given a skew prounital closed category $\mathbb{C}$, we define $U\,\mathbb{C}$ as the skew multicategory with the same objects as $\mathbb{C}$ and with the multihomset $(U\,\mathbb{C})(S|\Gamma;C)$ given by $\mathbb{C}(S,[\![\Gamma|C]\!])$. From the structure of $\mathbb{C}$, using properties of the interpretation $[\![\Gamma|C]\!]$, one defines the identity, composition and loosening of $U\,\mathbb{C}$.

The *free* skew prounital closed category on a skew multicategory $\mathbb{M}$ is then a skew prounital closed category $\mathbf{FSkPCl}(\mathbb{M})$ equipped with a skew multifunctor $\iota : \mathbb{M} \to U(\mathbf{FSkPCl}(\mathbb{M}))$. For any other skew prounital closed category $\mathbb{C}$ with a skew multifunctor $G : \mathbb{M} \to U\,\mathbb{C}$, there must exist a unique strict prounital closed functor $\bar{G} : \mathbf{FSkPCl}(\mathbb{M}) \to \mathbb{C}$ compatible with ι.

Let $\mathbb{M}$ be a skew multicategory. We construct a categorical calculus presenting the free skew prounital closed category on $\mathbb{M}$. We then proceed to describe an equivalent cut-free sequent calculus.

Categorical Calculus The formulae are given by objects $X \in \mathbb{M}_0$ (atomic formulae) and $A \multimap B$ for any formulae A, B. The inference rules are the same as in (2), supplemented with an additional inference rule

$$\frac{\mathbb{M}(T|\Phi;Z)}{T \Longrightarrow [\![\Phi|Z]\!]}\ \iota$$

where T is an optional atom, Φ is a list of atoms and Z is an atom. The equational theory $\doteq$ from (3) is extended with new generating equations expressing the fact that ι is a skew multifunctor between $\mathbb{M}$ and $U(\mathbf{FSkPCl}(\mathbb{M}))$.

Cut-Free Sequent Calculus The inference rules are those given in (4) minus the rules ax and pass plus two new rules

$$\frac{\mathbb{M}(T|\Phi;Z)}{T \mid \Phi \longrightarrow Z}\ \iota \qquad\qquad \frac{- \mid \Gamma \longrightarrow A \quad S \mid \Delta_0, B, \Delta_1 \longrightarrow C}{S \mid \Delta_0, A \multimap B, \Gamma, \Delta_1 \longrightarrow C}\ {\multimap}\mathsf{C}$$

The rule ${\multimap}\mathsf{C}$ was derivable using cut in the sequent calculus of Section 3.2, as we showed in (7). Here it is needed as a primitive rule to achieve cut admissibility. From the presence of a map $f \in \mathbb{M}(X|Y;Z)$ in the base skew multicategory $\mathbb{M}$, we need to be able to derive, e.g., the sequent $X \mid A \multimap Y, A \longrightarrow Z$, which in the categorical calculus is derivable as follows:

$$\frac{\dfrac{\mathbb{M}(X|Y;Z)}{X \Longrightarrow Y \multimap Z}\ \iota \quad \dfrac{}{Y \multimap Z \Longrightarrow (A \multimap Y) \multimap (A \multimap Z)}\ L}{X \Longrightarrow (A \multimap Y) \multimap (A \multimap Z)}\ \mathsf{comp}$$

The equational theory on derivations is obtained from the congruence $\overset{\circ}{=}$ of (5) by adding the following generating equations:

(preservation of id and loosen by ι)

$$\begin{array}{rcll} \mathsf{ax}_X & \overset{\circ}{=} & \iota\ (\mathsf{id}_X) & \\ \mathsf{pass}\ (\iota\ f) & \overset{\circ}{=} & \iota\ (\mathsf{loosen}\ f) & (\text{for } f \in \mathbb{M}(X|\Phi;Z)) \end{array}$$

(commutative conversions of ${\multimap}\mathsf{C}$)

$$\begin{array}{rcll} {\multimap}\mathsf{C}\ (f, {\multimap}\mathsf{R}\ g) & \overset{\circ}{=} & {\multimap}\mathsf{R}\ ({\multimap}\mathsf{C}\ (f,g)) & (\text{for } f : - \mid \Gamma \longrightarrow A', g : S \mid \Delta_0, B', \Delta_1, A \longrightarrow B) \\ \mathsf{pass}\ ({\multimap}\mathsf{C}\ (f,g)) & \overset{\circ}{=} & {\multimap}\mathsf{C}\ (f, \mathsf{pass}\ g) & (\text{for } f : - \mid \Gamma \longrightarrow A, g : A' \mid \Delta_0, B, \Delta_1 \longrightarrow C) \\ \mathsf{pass}\ ({\multimap}\mathsf{L}\ (f,g)) & \overset{\circ}{=} & {\multimap}\mathsf{C}\ (f, \mathsf{pass}\ g) & (\text{for } f : - \mid \Gamma \longrightarrow A, g : B \mid \Delta \longrightarrow C) \\ {\multimap}\mathsf{C}\ (f, {\multimap}\mathsf{L}(g,h)) & \overset{\circ}{=} & {\multimap}\mathsf{L}\ (g, {\multimap}\mathsf{C}\ (f,h)) & (\text{for } f : - \mid \Gamma \longrightarrow A, g : - \mid \Gamma' \longrightarrow A', h : B' \mid \Delta_0, B, \Delta_1 \longrightarrow C) \\ {\multimap}\mathsf{C}\ (f, {\multimap}\mathsf{L}(g,h)) & \overset{\circ}{=} & {\multimap}\mathsf{L}\ ({\multimap}\mathsf{C}\ (f,g), h) & (\text{for } f : - \mid \Gamma \longrightarrow A, g : - \mid \Delta_0, B, \Delta_1 \longrightarrow A', h : B' \mid \Delta \longrightarrow C) \\ {\multimap}\mathsf{C}\ (f, {\multimap}\mathsf{C}(g,h)) & \overset{\circ}{=} & {\multimap}\mathsf{C}\ (g, {\multimap}\mathsf{C}\ (f,h)) & (\text{for } f : - \mid \Gamma \longrightarrow A, g : - \mid \Gamma' \longrightarrow A', h : S \mid \Delta_0, B, \Delta_1, B', \Delta_2 \longrightarrow C) \\ {\multimap}\mathsf{C}\ (f, {\multimap}\mathsf{C}(g,h)) & \overset{\circ}{=} & {\multimap}\mathsf{C}\ ({\multimap}\mathsf{C}\ (f,g), h) & (\text{for } f : - \mid \Gamma \longrightarrow A, g : - \mid \Delta_0, B, \Delta_1 \longrightarrow A', h : S \mid \Delta_2, B', \Delta_3 \longrightarrow C) \end{array}$$

Thanks to the presence of the primitive rule ${\multimap}\mathsf{C}$, the two cut rules in (6) are admissible in this sequent calculus. In this case, they need to be defined by mutual induction with another cut rule

$$\frac{A' \mid \Gamma \longrightarrow A \quad S \mid \Delta_0, A, \Delta_1 \longrightarrow C}{S \mid \Delta_0, A', \Gamma, \Delta_1 \longrightarrow C}\ \mathsf{ccut}_{\mathsf{Fma}}$$

In the sequent calculus of Section 3.2, the rule $\mathsf{ccut}_{\mathsf{Fma}}$ is definable by first applying pass to the first premise and then using ccut. In the new sequent calculus of the current section, we have to define it simultaneously with scut and ccut because of the added cases for the added primitive rules. It is possible to prove that the embedding ι is a skew multifunctor, in particular it preserves the cut operations.

Notice that the sequent calculus is generally not left-normal. In fact, an attempt to prove the admissibility of the rule act of (10) fails when the premise is of the form $\iota\ f$ for some $f \in \mathbb{M}(-|X,\Phi;Z)$. E.g., we may well have a map in $\mathbb{M}(-|X;Z)$ for some X and Z without there being any map in $\mathbb{M}(X|;Z)$. Therefore the stoup cannot be discarded.

The categorical calculus and the sequent calculus are equivalent. It is possible to construct functions sound and cmplt translating between the two calculi, and show that they form an isomorphism up to the extended equivalence relations $\doteq$ and $\overset{\circ}{=}$.

Focused derivations The focused subcalculus uses that the ax and pass rules of the sequent calculus are admissible from id and loosen (crucially because the presence of $\multimap\mathsf{C}$ makes it possible to commute pass and $\multimap\mathsf{L}$). It has the inference rules from (9) minus the rules pass and ax plus two new rules

$$\frac{\mathbb{M}(T|\Phi;Z)}{T \mid \Phi \longrightarrow_{\mathsf{F}} Z}\ \iota \qquad \frac{- \mid \Gamma \longrightarrow_{\mathsf{I}} A \quad T \mid \Psi,B,\Delta \longrightarrow_{\mathsf{F}} C}{T \mid \Psi,A \multimap B,\Gamma,\Delta \longrightarrow_{\mathsf{F}} C}\ \multimap\mathsf{C}$$

Notice that, in the rule $\multimap\mathsf{C}$, T is restricted to be an optional atom and Ψ a list of atoms. Notice also that the passivation phase is trivial because we have removed the rule pass.

4.2 Starting From a Skew Multigraph

The free skew prounital closed category $\mathbf{FSkPCl}(\mathbb{M})$ over a multicategory $\mathbb{M}$ is special in that we can have a cut-free sequent calculus where the use of the generating multimaps is confined to "direct import" by ι. In fact, no structural rules (neither any cut rules nor pass or ax) are needed beyond the degree that they are readily available to us in the form of composition, loosening and identity in the base skew multicategory $\mathbb{M}$ where they also satisfy the skew multicategory equations. This is possible because the cut rules happen to be admissible from the sound rule $\multimap\mathsf{C}$ that we may choose to take as primitive.

This approach is not robust for extensions with further connectives; we cannot have a similar cut-free sequent calculus for the free skew (unital) closed category $\mathbf{FskCl}(\mathbb{M})$: from $\mathbb{M}(-|\ ;Y)$ and $\mathbb{M}(X|Y;Z)$, we must be able to derive $X \mid \mathsf{I} \longrightarrow Z$, but for this we need ccut as a primitive rule (together with pass) since, differently from $\multimap$, it is unsound to introduce I into the passive context. However, as soon as we introduce primitive cut rules (it suffices to take scut and ccut as primitive), we also need to introduce (i) equations stating that ι preserves compositions as cuts and (ii) also the skew multicategory equations for scut, ccut, ax and pass. The equations (i) can be dispensed with if we start with a *skew multigraph* $(\mathsf{At},\mathsf{DC})$ (of *atoms* and *definite clauses*) rather than a skew multicategory $\mathbb{M}$, so that composition as well as the identities and loosening are only available in terms of scut, ccut, ax and pass. We conjecture that the equations (ii) can then also be avoided in a focused subcalculus with all the inference rules from (9) plus the rule

$$\frac{- \mid \Gamma_1 \longrightarrow_{\mathsf{I}} Y_1 \quad \ldots \quad - \mid \Gamma_n \longrightarrow_{\mathsf{I}} Y_n \quad \mathsf{DC}(T|Y_1,\ldots,Y_n;X) \quad X \mid \Delta \longrightarrow_{\mathsf{F}} C}{T \mid \Gamma_1,\ldots,\Gamma_n,\Delta \longrightarrow_{\mathsf{F}} C}\ \iota'$$

which packages a particular combination of ι and scut and ccut inferences.

5 Conclusions and Future Work

We presented several equivalent presentations of the free skew prounital closed category on a set At. We showed that these correspond to a skew variant of the planar fragment of linear typed lambda-calculus. We constructed two calculi of normal forms: a focused sequent calculus and a normal natural deduction calculus. These solve the coherence problem for skew prounital closed categories by fully characterizing the homsets of $\mathbf{FSkPCl}(\mathsf{At})$. The latter category is left-normal, meaning that its skew aspect is redundant. We restored the skewness by studying deductive systems for the free skew prounital closed category on a skew multicategory and showing that the latter is generally not left-normal.

The development presented in the paper has been fully formalized in the dependently typed programming language Agda. Our Agda formalization also includes a similar proof theoretic analysis of the free *skew closed category* on a set, in which the element set functor J is replaced by a unit object I [30].

The primitive rules of the cut-free sequent calculus of skew closed categories also include left and right introduction rule for the unit I, where again the left rule acts only on the unit in the stoup. Similarly, the natural deduction calculus has introduction and elimination rules for I. Our reduction-free normalization procedures can be adapted to the skew closed case without much difficulty.

In the future, we plan to extend the work of this paper and our previous work on the sequent calculus of the Tamari order [40] and of skew monoidal categories [35, 37] to a proof theoretic investigation of skew monoidal closed categories, i.e. including unit I, tensor $\otimes$ and internal hom $\multimap$ related by an adjunction $- \otimes B \dashv B \multimap -$. We already know from our previous work that the corresponding sequent calculus would not be left-normal. We conjecture that the free skew monoidal closed category on At corresponds to a skew variant of the $(\mathsf{I}, \otimes, \multimap)$ fragment of noncommutative intuitionistic linear logic [2]. It is currently not clear how to extend the normalization procedures of this paper to the skew monoidal closed case, in particular normalization by evaluation, which has not been studied in the planar (or even linear) fragment of lambda-calculus. Inspiration could come from the normalization by hereditary substitution algorithm of Watkins et al. [38] for the propositional fragment of their concurrent logical framework.

Acknowledgments. We thank the anonymous referees for extremely valuable comments. T.U. was supported by the Icelandic Research Fund grant no. 196323-052 and the Estonian Ministry of Education and Research institutional research grant no. IUT33-13. N.V. was supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

References

[1] S. Abramsky (2008): *Temperly-Lieb algebra: from knot theory to logic and computation*. In G. Chen, L. Kauffman & S. Lomonaco (eds.), *Mathematics of Quantum Computing and Technology*, *Applied Mathematics and Nonlinear Science Series*, Chapman and Hall/CRC, pp. 415–458, doi: 10.1201/9781584889007. Preprint available at `https://arxiv.org/abs/0910.2737`.

[2] V. M. Abrusci (1990): *Non-commutative intuitionistic linear logic*. *Math. Log. Quart.* 36(4), pp. 297–318, doi: 10.1002/malq.19900360405.

[3] T. Altenkirch, J. Chapman & T. Uustalu (2015): *Monads need not be endofunctors*. *Log. Methods Comput. Sci.* 11(1), article 3, doi: 10.2168/lmcs-11(1:3).

[4] T. Altenkirch, M. Hofmann & T. Streicher (1995): *Categorical reconstruction of a reduction free normalization proof*. In D. H. Pitt, D. E. Rydeheard & P. T. Johnstone (eds.), *Proc. of 6th Int. Conf. on Category Theory and Computer Science, CTCS '95*, *Lect. Notes in Comput. Sci.* 953, Springer, pp. 182–199, doi: 10.1007/3-540-60164-3_27.

[5] J.-M. Andreoli (1992): *Logic programming with focusing proofs in linear logic*. *J. of Log. and Comput.* 2(3), pp. 297–347, doi: 10.1093/logcom/2.3.297.

[6] J. Bénabou (1963), *Catégories avec multiplication*. *C. R. Acad. Sci. Paris* 256, pp. 1887–1890. Available at `http://gallica.bnf.fr/ark:/12148/bpt6k3208j/f1965.image`.

[7] N. Benton, G. Bierman, J. M. E. Hyland & V. C. V. de Paiva (1993): *Linear λ-calculus and categorical models revisited*, E. Börger, G. Jäger, H. Kleine Büning, S. Martini, M. M. Richter (eds.), *Proc. of 6th Wksh. on Computer Science Logic, CSL '92*, *Lect. Notes in Comput. Sci.* 702, Springer, pp. 61–84, doi: 10.1007/3-540-56992-8_6.

[8] U. Berger & H. Schwichtenberg (1991): *An inverse of the evaluation functional for typed lambda-calculus*. In *Proc. of 6th IEEE Ann. Symp. on Logic in Computer Science, LICS'91*, IEEE Comput. Soc., pp. 203–211, doi: 10.1109/lics.1991.151645.

[9] J. Bourke (2017): *Skew structures in 2-category theory and homotopy theory*. J. Homotopy Relat. Str. 12, pp. 31–81, doi: 10.1007/s40062-015-0121-z.

[10] J. Bourke & S. Lack (2018): *Skew monoidal categories and skew multicategories*. J. Alg. 506, pp. 237–266, doi: 10.1016/j.jalgebra.2018.02.039.

[11] S. Eilenberg & G. M. Kelly (1966): *Closed categories*. In S. Eilenberg, D. K. Harrison, S. Mac Lane & H. Röhl (eds.), Proc. of Conf. on Categorical Algebra (La Jolla, 1965), Springer, pp. 421–562, doi: 10.1007/978-3-642-99902-4_22.

[12] R. Houston (2013): *Linear logic without units*. arXiv eprint 1305.2231. Available at `https://arxiv.org/abs/1305.2231`.

[13] C. Keller & T. Altenkirch (2010): *Hereditary substitutions for simple types, formalized*. In V. Capretta & J. Chapman (eds.), Proc. of 3rd ACM SIGPLAN Wksh. on Mathematically Structured Functional Programming, MSFP'10, ACM, pp. 3–10, doi: 10.1145/1863597.1863601.

[14] G. M. Kelly (1964): *On MacLane's conditions for coherence of natural associativities, commutativities, etc.* J. Alg. 1(4), pp. 397–402, doi: 10.1016/0021-8693(64)90018-3.

[15] G. M. Kelly & S. Mac Lane (1971): *Coherence in closed categories*. J. Alg. 1(1), pp. 97–140, doi: 10.1016/0022-4049(71)90013-2. (Erratum (1971): J. Alg. 1(2), p. 219, doi: 10.1016/0022-4049(71)90019-3.)

[16] S. Lack & R. Street (2012): *Skew monoidales, skew warpings and quantum categories*. Theor. Appl. Categ. 26, pp. 385–402. Available at `http://www.tac.mta.ca/tac/volumes/26/15/26-15abs.html`.

[17] J. Lambek (1968): *Deductive systems and categories I: Syntactic calculus and residuated categories*. Math. Syst. Theory 2(4), pp. 287–318, doi: 10.1007/bf01703261

[18] J. Lambek (1969): *Deductive systems and categories II: Standard constructions and closed categories*. In P. Hilton (ed.), Category Theory, Homology Theory and Their Applications I, Lect. Notes in Math. 86, Springer, pp. 76–122, doi: 10.1007/bfb0079385.

[19] J. Lambek (1972): *Deductive systems and categories III: Cartesian closed categories, intuitionist propositional calculus, and combinatory logic*. In F. W. Lawvere (ed.), Toposes Algebraic Geometry and Logic, Lect. Notes in Math. 274, Springer, pp. 57–82. doi: 10.1007/bfb0073965.

[20] F. W. Lawvere (1970): *Equality in hyperdoctrines and comprehension schema as an adjoint functor*. In A. Heller (ed.), Applications of Categorical Algebra, Proc. of Symp. in Pure Math. 17, Amer. Math. Soc., pp. 1–14. doi: 10.1090/pspum/017

[21] M. L. Laplaza (1977): *Coherence in nonmonoidal closed categories*. Trans. Amer. Math. Soc. 230, pp. 293–311, doi: 10.1090/s0002-9947-1977-0444740-9.

[22] S. Mac Lane (1963): *Natural associativity and commutativity*. Rice Univ. Stud. 49(4), pp. 28–46. Available at `http://hdl.handle.net/1911/62865`.

[23] O. Manzyuk (2012): *Closed categories vs. closed multicategories*, Theor. Appl. Categ. 26(5), pp. 132–175. Available at `http://www.tac.mta.ca/tac/volumes/26/5/26-05abs.html`.

[24] C. Mann (1975): *The connection between equivalence of proofs and Cartesian closed categories*, Proc. London Math. Soc. 31(3), pp. 289–310, doi: 10.1112/plms/s3-31.3.289.

[25] C. A. Meredith & A. N. Prior (1963), *Notes on the axiomatics of the propositional calculus*, Notre Dame J. Formal Log., 4, pp. 171–187, doi: 10.1305/ndjfl/1093957574.

[26] G. E. Mints (1977): *Closed categories and the theory of proofs*, Zap. Nauchn. Sem. LOMI 68, pp. 83–114. (In Russian.)
Translated in 1981 in J. Sov. Math. 15, pp. 45–62. doi: 10.1007/bf01404107.
Reprinted in 1992 in G. E. Mints, Selected Papers in Proof Theory, Studies in Proof Theory 3, Bibliopolis/North-Holland, pp. 183–212.

[27] D. Prawitz (1965): Natural Deduction: A Proof-Theoretical Study, Stockholm Studies in Philosophy 3, Almqvist & Wiksell.

[28] U. Schreiber, M. Shulman et al. (2009): Closed categories. ncatlab article. (Rev. 49 was by M. Shulman, May 2018. Current version is rev. 62 from July 2020) `https://ncatlab.org/nlab/show/closed+category`

[29] W. J. de Schipper (1975): *Symmetric closed categories*, *Mathematical Centre Tracts* 64, CWI, Amsterdam.

[30] R. Street (2013): *Skew-closed categories*. *J. Pure Appl. Alg.* 217(6), pp. 973–988, doi: 10.1016/j.jpaa.2012.09.020.

[31] M. E. Szabo (1974): *A categorical equivalence of proofs*, *Notre Dame J. Formal Log.* 15(2), pp. 177–191. doi: 10.1305/ndjfl/1093891297.

[32] M. E. Szabo (1978): *Algebra of Proofs*, *Studies in Logic and the Foundations of Mathematics* 88, North-Holland, 1978.

[33] K. Szlachányi (2012): *Skew-monoidal categories and bialgebroids*. *Adv. Math.* 231(3–4), pp. 1694–1730, doi: 10.1016/j.aim.2012.06.027.

[34] A. S. Troelstra (1995): *Natural deduction for intuitionistic linear logic*, *Ann. Pure Appl. Log.* 73(1), pp. 79–108, doi: 10.1016/0168-0072(93)e0078-3.

[35] T. Uustalu, N. Veltri & N. Zeilberger (2018): *The sequent calculus of skew monoidal categories*. *Electron. Notes Theor. Comput. Sci.* 341, pp. 345–370. doi: 10.1016/j.entcs.2018.11.017.
Extended version to appear in C. Casadio & P. Scott (eds.), *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, *Outstanding Contributions to Logic* 20, Springer. Preprint available at `https://arxiv.org/abs/2003.05213`.

[36] T. Uustalu, N. Veltri & N. Zeilberger (2020): *Eilenberg-Kelly reloaded*. *Electron. Notes Theor. Comput. Sci.* 352, pp. 233–256. doi: 10.1016/j.entcs.2020.09.012

[37] T. Uustalu, N. Veltri, N. Zeilberger (to appear): *Proof theory of partially normal skew monoidal categories*. In D. I. Spivak, J. Vicary (eds.), *Proc. of 3rd Applied Category Theory Conf., ACT 2020*, *Electron. Proc. in Theor. Comput. Sci.*, Open Publishing Assoc. Available at `https://cgi.cse.unsw.edu.au/~eptcs/paper.cgi?ACT2020:60`.

[38] K. Watkins, I. Cervesato, F. Pfenning & D. Walker (2004): *A concurrent logical framework: The propositional fragment*. In S. Berardi, M. Coppo, F. Damiani (eds.), *Proc. of Int. Wksh. on Types for Proofs and Programs, TYPES '03*, *Lect. Notes in Comput. Sci.* 3085, Springer, pp. 355–377, doi: 10.1007/978-3-540-24849-1_23.

[39] N. Zeilberger (2018): *A theory of linear typings as flows on 3-valent graphs*. In *Proc. of 33rd Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS '18*, ACM, pp. 919–928, doi: 10.1145/3209108.3209121.

[40] N. Zeilberger (2019): *A sequent calculus for a semi-associative law*. *Log. Methods Comput. Sci.* 15(1), article 9, doi: 10.23638/lmcs-15(1:9)2019.

[41] J. Zucker (1974): *The correspondence between cut-elimination and normalization*, *Ann. Math. Log.* 7(1), pp. 1–112, 1974. doi: 10.1016/0003-4843(74)90010-2

Implementation of Two Layers Type Theory in Dedukti and Application to Cubical Type Theory

Bruno Barras
Inria, Université Paris-Saclay,
ENS Paris-Saclay, CNRS, LSV,
91190, Gif-sur-Yvette, France.

Valentin Maestracci
Université Paris-Saclay,
ENS Paris-Saclay, CNRS, LSV,
91190, Gif-sur-Yvette, France.

In this paper, we make a substantial step towards an encoding of Cubical Type Theory (CTT) in the Dedukti logical framework. Type-checking CTT expressions features a decision procedure in a de Morgan algebra that so far could not be expressed by the rewrite rules of Dedukti. As an alternative, 2 Layer Type Theories are variants of Martin-Löf Type Theory where all or part of the definitional equality can be represented in terms of a so-called external equality. We propose to split the encoding by giving an encoding of 2 Layer Type Theories (2LTT) in Dedukti, and a partial encoding of CTT in 2LTT.

1 Introduction

The goal of this paper is to explore the possibility to express Homotopy Type Theory (HoTT, [9]) in the Dedukti [5] logical framework.

Dedukti is a logical framework which main distinctive feature is the possibility to extend the definitional equality (aka conversion) with a class of rewrite rules. It is intended to be used as a "hub" for proof systems. Many of the logics implemented by those systems can be encoded as *Dedukti theories*, and proofs in those systems can be expressed as Dedukti terms in the corresponding theory. The point is not just to collect the proofs of many logics, but rather to make it easier to translate proofs from one system to the other.

Expressing HoTT as a Dedukti theory is at the same time a problem that challenges the expressiveness of the rewrite rules accepted by Dedukti, but HoTT is an interesting formalism on its own. Moreover, in the long term, it would be interesting to have tools to investigate which proofs can be adapted from more conventional logics (HOL, set theory) to HoTT and conversely.

The paper is organized as follows. We first give an introduction on HoTT, Dedukti and encodings of Type Theory in Dedukti. This will motivate the introduction of Two Level Type Theories (2LTT) as a more flexible way to deal with type theories having a definitional equality two complex to be encoded as rewrite rules. In the second section, we will introduce 2LTTs and their encoding as a Dedukti theory. In the third section, we will give a partial encoding of Cubical as a 2LTT, and the corresponding piece of Dedukti theory.

Both encodings have been implemented and can be found here:

`https://github.com/valent20000/CTTDedukti`.

1.1 Homotopy Type Theories

In the broad sense, HoTT is based on an interpretation of Type Theory (ML, Coq, Agda, NuPRL) where types are topological spaces and proofs of an equality $x = y$ in type A are continuous paths between points x and y of space A.

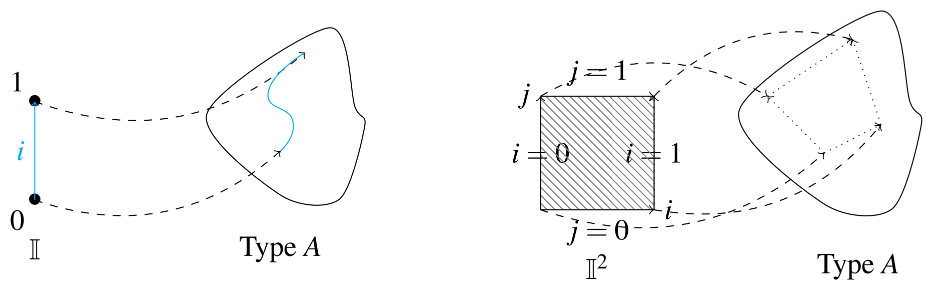

Figure 1: Meaning of judgments $i:\mathbb{I}\vdash t:A$ and $i,j:\mathbb{I}\vdash t:A$

This specific interpretation implies the possibility to extend the theory with new principles. The most famous one is *univalence* expressing that paths between types are exactly the weak equivalences (which is a particular case of isomorphism that deals with higher dimensions). Other extensions include higher inductive types, which are a generalization of inductive definitions. They are used to define spaces such as the circle, the torus, suspensions and many others.

Some members of the HoTT family are just regular Type Theory extended with axioms. The drawback of this approach is that axioms breaks metatheoretical properties such as canonicity. Worse, some useful notions such as simplicial sets seems impossible to express by mere axioms: face map equations require a coherence condition at dimension 2, which in turn requires another coherence condition at higher dimension, etc. This situation is often called “coherence hell”.

In contrast, several members of the HoTT family, called *cubical* ([7],[2]), give a computational interpretation to univalence. In this paper we will focus on the Cubical Type Theory in [7] (called Cubical from now on) that we will briefly introduce in the next section. We believe that adapting this work to the others cubical theories should be easy.

1.2 Cubical Type Theory

The intuition behind Cubical is to follow the definition of paths as continuous functions from interval $[0;1]$ to the points of the topological space.

Cubical is a type theory introducing an interval pretype $\mathbb{I}$.[1]

Having an interval variable i in the context, a judgement $i:\mathbb{I}\vdash t:A$ represents a point t parameterized by the interval, hence a path in A (see Fig. 1, left). From that, one can define a type Path and a path constructor in a way similar to λ-abstraction. It is also possible to apply an expression to path to get back a point of A.

$$\frac{\Gamma\vdash A\ \text{type}}{\Gamma\vdash\langle i\rangle t:\mathsf{Path}\ A\ t(i0)\ t(i1)}\qquad\frac{\Gamma\vdash p:\mathsf{Path}\ A\ x\ y\quad\Gamma\vdash e:\mathbb{I}}{\Gamma\vdash p\,e:A}$$

Formally, the interval $\mathbb{I}$ is defined in a synthetic way: as the free De Morgan algebra on $i,j,k\cdots$. Expanding the definition, this means its terms are elements of the following form : $0\mid 1\mid i\mid 1-r\mid r\wedge s\mid r\vee s$, with $\wedge$ representing the inf and $\vee$ representing the sup of the elements.

Now if we have two interval variables, a judgment $i,j:\mathbb{I}\vdash t:A$ means t is a square in A, as illustrated by Fig. 1. Having n interval variable leads to a n-dimensional cube in A, hence the name of Cubical Type Theory.

[1] $\mathbb{I}$ is only a *pretype*, as it does not enjoy all properties of types: we should not identify 0 and 1 although they are connected by a path.

Some primitives of Cubical refer to expressions that may be defined only on a sub-polyhedra. To do that, we first describe the cube in a synthetic way like we did with the interval. A pretype $\mathbb{F}$ for the faces of the cube are defined with the following grammar:

- 1, the entire cube.
- 0, the empty face.
- $i = 0/i = 1$ the face where $i = 0/i = 1$
- $f_1 \wedge f_2$, the intersection of the faces f_1, f_2
- $f_1 \vee f_2$, the union of the faces f_1, f_2

Contexts may also contain a face to restrict judgments to a sub-polyhedra. For instance, the judgment $i, j : \mathbb{I}; i = 0 \vee j = 1 \vdash t : A$ represents the left and top edges of the square in Fig. 1.

In the general case, type-checking in Cubical features a decision procedure for the inclusion of faces. This is the main challenge in encoding Cubical in Dedukti.

1.3 Encoding Type Theories in Dedukti

Encoding a logic L in Dedukti usually consists in introducing a Dedukti theory (i.e. a set of constants and rewrite rules) $D(L)$, and a mapping $[\![]\!]_L$ from L-formulae to Dedukti types and from L-proofs to terms of type corresponding to the formula they prove.

In the case of Type Theory, one introduces a Dedukti type for “codes of types”, and a decoding function that assigns a Dedukti type to each of these codes. Then, one introduces one constant for each type constructor, and constants for introduction and elimination rules.

The basic property of this encoding is that it must be well-typed, in the sense that a well-formed type must be translated to a well-typed Dedukti term. More specifically this requires that definitionally equal terms must be translated to convertible Dedukti terms. In other terms, we expect that the definitional equality can be expressed as rewrite rules.

As we have explained above, Cubical is a type theory which definitional equality includes the equational theory of a de Morgan algebra. It is far from obvious that it can actually be encoded by the Dedukti rewrite rules.

We prefer to investigate another approach, where part of the conversion is mapped to a kind of propositional equality. Unfortunately, we cannot express conversion as a propositional equality of Cubical (for the same reason that some notions in HoTT cannot be expressed by mere axioms).

Those remarks have led to the introduction of Two Level Type Theories [3].

1.4 Two Level Type Theories

We recall that in Type Theory there are two notions of equality:

- the propositional equality, that represents the intended equality of the logic
- the definitional equality, which in fact is a judgment, which represents objects that should be identified to ensure important properties of the judgments

Two-Layer Type Theories are a class of type theories. Their common point is that they are in fact made of two types theories (both copies of MLTT, with different additional axioms)

- **The internal**: It represents the theory we want to study. It is often equipped with Univalence Axiom, which makes its equality different than the usual equality, and incompatible with axioms like Uniqueness of Identity Proofs (UIP, or K) and functional extensionality (FunExt).
- **The external**: This theory will act as a sort of 'meta-theory' of the internal. We will use its propositional equality as an intermediate equality between the definitional ones (there are two definitional equalities here, the internal and the external), and the propositional equality of the internal theory. It has additional axioms (UIP & FunExt) to make its propositional equality not slightly weaker but as powerful as the usual one.

2 Two Layers Type Theories in Dedukti

2.1 Two Layer Type Theory as a Dedukti theory

In this section, we define Two Layers Type Theories by giving their encoding in Dedukti.

For a more comprehensive definition of 2LTT, we refer to [3], although we had to adapt the definition as they were defined semantically on some specific points.

2LTTs are basically two copies of Martin Löf's Type Theory: one internal layer and an external one. In order to avoid the complexity of having the notion of type and later introduce each universe as a subclass (as is usual in MLTT), we parameterize the notion of type by *levels*, that identify each universe, following Assaf (section 8.3 of [5]). We made the minimal assumptions of those levels, by just assuming a function `lsuc` such that universe l belongs to level `lsuc` l. The Dedukti declarations for that are:

```
Lev : Type.
lsuc : Lev -> Lev.
```

We can now introduce two codes of types: one for the internal layer (`T`) and one for the external one (`xT`), and their corresponding decoding functions `eps` and `xeps`. Let us point out that we have chosen a shallow embedding where 2LTT contexts are identified with Dedukti contexts. So codes of types are not explicitely parameterized by a notion of context.

```
T : Lev -> Type.                           xT : Lev -> Type.
def eps : i : Lev -> T i -> Type.          def xeps : i:Lev -> T i -> Type.
```

In this encoding, an internal type A at level l is a term `A : T l`, an element t of that type A is a term `t : eps l A.T`, and likewise for external types.

The inclusion of a universe l in the bigger universe is taken care by first introducing a code in `t l : T (lsuc l)`, and a rewrite rule to assert that `t l` decodes to `T l`:

```
t : i : Lev -> T (lsuc i).          xt : i : Lev -> xT (lsuc i).
[i] eps (lsuc i) (t i) --> T i.     [i] xeps (lsuc i) (xt i) --> xT i.
```

It remains to lift each type of level l as a type of level `lsuc` l. We omit the external lift which is defined similarly.

```
lUp :  i : Lev -> a : T i -> T (lsuc i).
[i, a] eps (lsuc i) (lUp i a) --> eps i a.
```

The above rewrite rule relate codetypes at level l and their counterpart at level `lsuc` l in a very strong way: they decode to the same type, which means they have the *same* inhabitants.

We then implemented the usual primitive types of MLTT to populate the universes. As an example, internal dependent pairs are declared by introducing a constant `Sig` for the typecode, `pair` is the introduction rules, and `p1` and `p2` are the projections:

```
Sig : i : Lev -> A : T i -> (eps i A -> T i) -> T i.
def tSig := (i : Lev => A : T i => B : (eps i A -> T i)
  => eps i (Sig i A B)).

def pair : i : Lev -> A : T i -> B : (eps i A -> T i) ->
    a : eps i A -> b : eps i (B a) -> tSig i A B.
def p1 : i : Lev -> A : T i -> B : (eps i A -> T i) ->
    p : tSig i A B -> eps i A.
def p2 : i : Lev -> A : T i -> B : (eps i A -> T i) ->
    p : tSig i A B -> eps i (B (p1 i A B p)).

[i,A,B,a,b] p1 i A B (pair i A B a b) --> a.
[i,A,B,a,b] p2 i A B (pair i A B a b) --> b.
```

Actually, we define the following types, both at the internal and external layer, and at each level:

Internal	External
`False i : T i`	`xFalse i : xT i`
`True i : T i`	`xTrue i : xT i`
`Nat i : T i`	`xNat i : xT i`
`Pi i (A:T i)(B:x:eps i A->T i) : T i`	`xPi i (A:xT i)(B:x:xeps i A->xT i) : xT i`
`Sig i (A:T i)(B:x:eps i A->T i):T i`	`xSig i (A:xT i)(B:x:xeps i A->xT i):xT i`
`Sum i (A:T i) (B:T i) : T i`	`xSum i (A:xT i) (B:xT i) : xT i`
`Eq i (A:T i)(x:eps i A)(y:eps i A):T i`	`xEq i (A:xT i)(x:xeps i A)(y:xeps i A):xT i`

So far, 2LTTs feature two copies of MLTT, each one totally independent from the other. In order to include the internal layer into the external one, 2LTTs feature a coercion function c that assigns an external to each internal type.

```
def c  :  i : Lev -> T i -> xT i.
```

In [3], the coercion was defined in semantical terms. Here, coercion is such each internal type A is *isomorphic* to $\texttt{c}(A)$. By lifting internal types into the external world, the coercion allows us to see the internal world as a sort of sub-world of the external world. This allows to express properties of the internal world using the external equality.

This isomorphism can be encoded in different ways, from the most general to the most specific:

- Assuming the existence of functions between $\texttt{eps}(A)$ and $\texttt{xeps}(\texttt{c}(A))$, which are inverse one of each other, propositionally.
- Assuming the existence of functions between $\texttt{eps}(A)$ and $\texttt{xeps}(\texttt{c}(A))$, which are inverse one of each other, definitionally.
- Assuming that both (code)types decode to the same type.

The third option is similar to the one chosen for the universe inclusion, but the goal of 2LTTs is to be as general as possible, so we would better avoid it. However, the first one would probably need a coherence condition, and would make the system harder to use. For these reasons we have chosen the second option:

```
def isoUp :   i : Lev -> A : T i -> eps i A -> tc i A.
def isoDown : i : Lev -> A : T i -> tc i A  -> eps i A.
[i, A, a] isoDown i A (isoUp i A a) --> a.
[i, A, a] isoUp i A (isoDown i A a) --> a.
```

As stated before, 2LTTs are a class of type theories. They are a sort of 'à la carte' type theory where one can add additional axioms concerning coercion to tune it the way one wants it to be.

Like the definition of coercion, most of these axioms had a semantic definition. Here is a list of the ones (as introduced in [3]) we were able to express in a syntactic manner:

- Coercion can be required to be injective:

  ```
  (; < T1 > ;)
  def T1 : l : Lev -> A : T l -> B : T l ->
           p : xtTEq l (c l A) (c l B) -> tTEq l A B.
  ```

  where `xtTEq` and `tTEq` are shorthands for the types of elements of respectively external and internal equality

- The repletion axiom: an external type isomorphic to a $c(A)$ also has an antecedent by c.

  ```
  (; < T3 > ;)
  repletion : l : Lev -> A : xT l -> B : T l ->
              p : xtTEq l A (c l B) -> T l.
  [l, A, B, e] c l (repletion l A B e) --> A.
  ```

- Integers are a fairly simple type with simple rules. One would expect $c(\mathbb{N})$ and x$\mathbb{N}$ to be isomorphic, but it is actually not the case for technical reasons. There is only a morphism x$\mathbb{N} \to c(\mathbb{N})$.

  A possible additional axiom is to make this morphism an isomorphism definitionally.

  The union, eq and false types can have similar additional axioms, while in the case of the product, sum and 1 types, the isomorphism is already there.

- One can also make these types isomorphic through that isomorphism not definitionally (ie by rewriting), but only up to external equality.

- One can make all these isomorphisms (including the ones for pi, sig, and 1, with example code below) equalities instead of just isomorphisms (cf option 3 for the coercion).

  ```
  (; < T2 >  Primitive Isomorphisms c A ~ xA become equality ;)
  [l] c l True --> xTrue l.
  [l, A, B] c l (Pi l A B) --> xPi l (c l A) (clift l A B).
  [l, A, B] c l (Sig l A B) --> xSig l (c l A) (clift l A B).
  ```

We implemented all the above axioms in Dedukti.

Interestingly, there was in the list of axioms that couldn't be implemented an axiom called $(A5)$, which requires that external equality validates the reflection rule (that is, externally equal types are considered definitionally equal). This could not be implemented in Dedukti. In it was possible, then we would be able to encode Cubical in Dedukti without resorting to 2LTTs.

We also tested the usability of this encoding by formulating the axiom of univalence. Here, for the sake of brevity, we only give the weaker form which states that the type $A = B$ (where A and B are types of level l) is weakly equivalent to $A \approx B$, the type of weak equivalence between A and B:

```
WeakUnivalence : l : Lev -> A : T l -> B : T l ->
  eps (lsuc l) (Equiv (lsuc l) (TEq l A B) (lUp l (Equiv l A B))).
```

Note that $A = B$ actually belongs to level `lsuc` l, hence the need to lift $A \approx B$ one universe up. Also, the notion of weak equivalence occurs twice but at different levels. This remark is the reason that made us opt for a presentation of type theories with a hierarchy of universes

2.2 Translating Two Layer Type Theories

We define a straightforward translation, with every type/term associated to the one with the same name in Dedukti, and the convention that variables share the same name in 2LTT and Dedukti:

- $[\![x]\!]_l =$ `x`
- $[\![\sum_{x:A}(B)]\!]_l =$ `Sig l` $[\![A]\!]_l$ `(x: eps l` $[\![A]\!]_l$ `=>` $[\![B(x)]\!]_l$`)`
- $[\![\mathrm{pair}_{[x:A]B}(a,b)]\!]_l =$ `pair l` $[\![A]\!]_l$ `(x: eps l` $[\![A]\!]_l$ `=>` $[\![B(x)]\!]_l$`)` $[\![a]\!]_l$ $[\![b]\!]_l$
- ...

We also define the translation of context :

$$[\![x_1 :_{l_1} A_1, \cdots, x_n :_{l_n} A_n]\!] = \texttt{x1: eps } [\![A_1]\!]_{l_1}, \ldots, \texttt{xn: eps } [\![A_n]\!]_{l_n}.$$

As stated before when we first talked about how 2LTTs were implemented, encoding has the particularity that it encodes types, not into Dedukti types, but into type codes, that is elements of type `T` l for internal types, and `xT` l for external types. It encodes terms into terms of type the 'realization' of $[\![A]\!]_l$, that is `eps` l $[\![A]\!]_l$, and similarly for the external layer.

2.3 Soundness of the encoding

There are two important properties that the translation is expected to have:

Soundness: this means that the translation defined above preserves typability, and hence provability too. It also means that our encoding is powerful enough to prove everything that could be proven in the theory of 2LTTs.

Conservativity: this would mean that any property provable in the encoding can be proved in the thoery of 2LTTs, in other words that our encoding is not too powerful. From this we would be able to use Dedukti as a 2LTT type-checker.

While soundness is quite straightforward to prove (since all of the definitional equality of 2LTTs could be encoded by rewrite rules), conservativity is quite hard to prove.

The soundness property consists of 9 statements, one for each judgment kind:

THEOREM —

- If Γ 2LTT context, then $[\![\Gamma]\!]_l$ Dedukti context.
- If $\Gamma \vdash_{2L} A$ type l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l :$ `T l`
- If $\Gamma \vdash_{2L} A = A'$ type l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l \leftrightarrow^* [\![A']\!]_l :$ `T l`
- If $\Gamma \vdash_{2L} A$ type l, $\Gamma \vdash_{2L} t : A$, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l :$ `eps l` $[\![A]\!]_l$
- If $\Gamma \vdash_{2L} A$ type l, $\Gamma \vdash_{2L} t = t' : A$ then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l \leftrightarrow^* [\![t']\!]_l :$ `eps l` $[\![A]\!]_l$
- If $\Gamma \vdash_{2L} A$ xtype l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l :$ `xT l`
- If $\Gamma \vdash_{2L} A = A'$ xtype l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l \leftrightarrow^* [\![A']\!]_l :$ `xT l`
- If $\Gamma \vdash_{2L} A$ xtype l, $\Gamma \vdash_{2L} t : A$, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l$ `: xeps l` $[\![A]\!]_l$
- If $\Gamma \vdash_{2L} A$ xtype l,$\Gamma \vdash_{2L} t = t' : A$ then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l \leftrightarrow^* [\![t']\!]_l$ `: xeps l` $[\![A]\!]_l$

The proof is by mutual induction on the judgments.

Conservativity is a much harder problem, and we have not proven it yet. However, we make the following conjecture a conservativity result. Every Dedukti proof which context and type are in the image of the translation correspond to a 2LTT derivation:

CONJECTURE —
Given a Dedukti judgment

$$x_1 : T_1, \ldots, x_n : T_n \vdash_{Dk} t : A$$

where all T_is are of the form $\texttt{eps}\, l_i\, [\![U_i]\!]_{l_i}$ or $\texttt{xeps}\, l_i\, [\![U_i]\!]_{l_i}$ and A is convertible to a term of the form $\texttt{eps}\, l\, [\![B]\!]_l$ or $\texttt{xeps}\, l\, [\![B]\!]_l$, then there exists a 2LTT terms u such that

$$x_1 : U_1, \ldots, x_n : U_n \vdash_{2L} u : B$$

This obviously cannot hold if the logic of Dedukti is inconsistent (unless 2LTTs are themselves inconsistent). The idea of the proof is to translate only proofs in normal form, and assume strong normalization of the reduction rules of Dedukti. Another source of inspiration is [4], where conservativity is proven for an encoding of Pure Type Systems.

3 Cubical Type Theory in Dedukti

This section introduces a partial encoding of Cubical as an extension of the 2LTT Dedukti theory. We focused of the main primitive of Cubical: composition. As we have already mentionned, the typing rule of composition is probably beyond the capabilities of Dedukti's rewrite rules, and we expect 2LTTs to be a trade-off where expressivity is recovered at the cost of building parts of definitional equality derivations by hand. We try to express the largest fragment by rewrite rules, and the rest will be encoded in the external layer.

We do not consider the primitives related to glueing, which is the main feature that makes univalence provable in Cubical.

When viewing Cubical as an instance of a 2LTT, the leading idea is that the internal layer contains the object theory (Cubical) while the external layer is that of the meta-theory. More concretely, the internal layer contains the types of Cubical, while the external layer contains the pretypes ($\mathbb{I}$, and $\mathbb{F}$) and the judgments of Cubical.

We first introduce a level `cL` for the primitive pretypes of Cubical: $\mathbb{I}$ and $\mathbb{F}$ which are declared as external types.

```
cL : Lev.                  def T := xT cL.          def ceps := xeps cL.
def cEq := xEq cL. (; and all types at level cL with prefix c ;)
```

Symbol `cEq` is used to express convertibility of preobjects. For the sake of conciseness we also define a symbol to represent convertibility of objects, using the coercion to lift internal objects to the external layer:

```
def CubicalEq (l : Lev) (A : T l) (a : eps l A) (b : eps l A) :=
    xEq l (c l A) (isoUp l A a) (isoUp l A b).
```

In order to interpret the conversion rule, we need more interaction between both layers, by allowing elimination of an external equality at level `cL` towards an internal type:

```
def CubicalJ :
  l:Lev -> A:cT -> x:ceps A -> P: (y:ceps A->cEq A x y->T l) ->
  eps l (P x (crefl A x)) ->
  y:ceps A -> e:cEq A x y -> eps l (P y e).
```

3.1 The interval pretype $\mathbb{I}$

Implementing the grammar of intervals and faces needs care, because the type-checking of Dedukti requires confluence of the set of rewrite rules. An algebra $(A, \vee, \wedge, 0, 1, \neg)$ is a De Morgan algebra if $\vee$ and $\wedge$ are associative and commutative and

$$x \wedge x = x \qquad x \wedge 0 = 0 \qquad x \wedge 1 = x$$
$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \qquad \neg(x \wedge y) = \neg x \vee \neg y \qquad \neg\neg x = x$$

The dual laws are derivable from these. In the currently distributed version of Dedukti, commutativity cannot be added as a rewrite rule, and idempotence being non-linear may break confluence. Zero, neutral, involution and De Morgan laws are straightforward. Associativity can be oriented in an arbitrary direction. Regarding distributivity, we cannot have a law and the dual one (neither the left nor the right one) or normalization is lost. Having the left and right laws at the same time creates a critical pair that cannot be closed without commutativity. So, we can have at most one of the four. Considering that distributivity is probably used only in very few cases, we decided to implement none as a rewrite rule. The rules that cannot be expressed as rewrite rules are thus stated as external equations.

```
I : cT.           0 : ceps I.         1 : ceps I.
def Imin : ceps I -> ceps I -> ceps I.
def Imax : ceps I -> ceps I -> ceps I.
def sym : ceps I -> ceps I.
(; rewrite rules, completed by symmetry ;)
[i] Imin 0 i --> 0  [i] Imin i 0 --> 0.
[i] Imin 1 i --> i [i] Imin i 1 --> i.
[i] Imax 0 i --> i  [i] Imax i 1 --> 0.
[i] Imax 1 i --> 1 [i] Imax i 1 --> 1.
[] sym 0 --> 1  [] sym 1 --> 0.
[i,j] sym (Imin i j) --> Imax (sym i) (sym j).
[i,j] sym (Imax i j) --> Imin (sym i) (sym j).
[i] sym (sym i) --> i.
[i,j,k] Imin (Imin i j) k --> Imin i (Imin j k).
[i,j,k] Imax (Imax i j) k --> Imax i (Imax j k).
(; properties expressed as external equations;
   Imin laws derived by duality ;)
Imax_idem : i:ceps I -> cEq (Imax i i) i.
Imax_comm : i:ceps I -> j:ceps I -> cEq I (Imax i j) (Imax j i).
Imax_dist : i : ceps I -> j : ceps I -> k : ceps I ->
   cEq (Imax (Imin i j) k) (Imin (Imax i k) (Imax j k).
```

3.2 Paths

We then define the type of paths together with its introduction and elimination rules. Since paths are types in Cubical, they are encoded in the internal layer.

```
def Path :   A : cT -> u : ceps A -> v : ceps A -> cT.
def lam : A : cT -> p : (ceps I -> ceps A) ->
     ceps (Path A (p 0) (p 1)).
def app : A : cT -> u : ceps A -> v : ceps A ->
     ceps (Path A u v) -> ceps I -> ceps A.
(; Computational rules ;)
[A,u,v,p] app A u v (lam A f) e --> f e.   (; beta ;)
[A,u,v,p] app A u v p 0 --> u  [A,u,v,p] app A u v p 1.
```

In the definition of Cubical, the last two definitional equalities above implement the rules

$$\frac{\Gamma \vdash p : \mathsf{Path}\, A\, u\, v}{\Gamma \vdash p\, 0 = u : A \qquad \Gamma \vdash p\, 1 = v : A}$$

which requires typing information about path p. This could be a problem since the conversion (and rewrite rules) of Dedukti is applied on terms without any typing information. Fortunately, in our encoding application is annotated with all of the information needed.

We also note that at this point paths are not related to the internal equality `Eq`.

It remains to express the key primitive of Cubical: composition. The typing rule involves the notions of interval variable (which are just external variables of type $\mathbb{I}$) and face ($\mathbb{F}$).

3.3 Faces

Let us first define faces, following the explanations in the introduction.

```
F : cT. (; Type of Faces;)
0f : ceps F. (; Empty face ;)
1f : ceps F. (; Whole cube ;)
def eq0 : ceps I -> ceps F. (; eq0 i is the face i = 0 ;)
def eq1 : ceps I -> ceps F. (; eq1 i is the face i = 1 ;)
def Fmin: ceps F -> ceps F -> ceps F. (; Intersection of faces ;)
def Fmax: ceps F -> ceps F -> ceps F. (; Union of faces ;)
(; Rewrite rules and equations (problem similar to the interval);)
[f] Fmin 0f f --> 0f.
...
Fdiscr : i : ceps I -> cEq F (Fmin (eq0 i) (eq1 i)) 0f.
```

We do not give details of how the algebraic properties of faces are turned into either rewrite rules or an equation. We only give the crucial property `Fdiscr` that there is no intersection between the opposite faces of a cube.

An important remark is that this does not exactly follow the syntax of the faces of Cubical, since the face $(i = 1)$ requires i to be a *variable*, which cannot be enforced in our shallow embedding. In Cubical when an interval variable i is substituted by an interval expression e, in a face $(i = 1)$, it is replaced following the rules (and similarly for $i = 0$):

$$(i=1)[i/0] = 0 \qquad (i=1)[i/1] = 1 \qquad (i=1)[i/1-e] = (i=0)[i/e]$$
$$(i=1)[i/e_1 \vee e_2] = (i=1)[i/e_1] \vee (i=1)[i/e_2]$$
$$(i=1)[i/e_1 \wedge e_2] = (i=1)[i/e_1] \wedge (i=1)[i/e_2]$$

This is quite naturally expressed by rewrite rules (straightfowardly adapted to `eq0`):

```
[] eq1 0 --> 0f       [] eq1 1 --> 1f    [e] eq1 (sym e) --> eq0 e.
[i,j] eq1 (Imax i j) --> Fmax (eq1 i) (eq1 j).
[i,j] eq1 (Imin i j) --> Fmin (eq1 i) (eq1 j).
```

Given a context Γ and a face ϕ of Γ, the context Γ,ϕ is a restriction of context Γ where the interval variables must belong to ϕ. Since we use a shallow embedding of context, we need to represent a face as an external type (actually a proposition) of witnesses that the interval variable belong to ϕ. So, the Dedukti type `F` above is a code of types with a decoding function `faceType`.

The definition one would like to make would thus be something along the lines of:

```
(; First attempt ;)
def faceType : ceps F -> cT.
[]      faceType 0f          --> cFalse.
[]      faceType 1f          --> cTrue.
[a]     faceType (eq1 a)     --> cEq I 1 a.
[a]     faceType (eq0 a)     --> cEq I 0 a.
[a, b]  faceType (Fmax a b)  --> cSum (faceType a) (faceType b).
[a, b]  faceType (Fmin a b)  --> cSig (faceType a) (_=>faceType b).
```

where we see that intersection (resp. union) is represented by the cartesian product (resp. sum), and the base constraint $(i=0)$ by an external equality.

But this definition breaks confluence. Here is an example of critical pair:

```
faceType (Fmin 1f 1f) --> cSig cTrue (_=> cTrue)
faceType (Fmin 1f 1f) --> faceType 1f --> cTrue
```

If we try to recover confluence by closing this critical pair, the types `cTrue` and `cSig cTrue (_=> cTrue)` become convertible and hence allow to apply a projection to an inhabitant of `cTrue`. This destroys many good properties (e.g. canonicity) of the external layer, and this may lead to have erroneous Cubical proofs accepted by the encoding.

As a workaround, we decided to not have rewrite rules associated to `faceType`, but rather have an *isomorphism* between `faceType` ϕ and its intended type. We will also need that those types are actually propositions (i.e. all inhabitants must be equal). This is the case for faces 0, 1 and $\phi_1 \wedge \phi_2$. This holds also for $(i=1)$ and $(i=1)$ if the external layer enjoys Uniqueness of Identity Proofs (or axiom K). But having $\phi_1 \vee \phi_2$ isomorphic to a disjoint sum is a problem because the latter type may not be a proposition. We need a new external type, for instance a truncated sum. This type as the same introduction rules similar to disjoint sum, but the elimination rule requires a coherence condition (that will be given in the definition `TermSys` below).

3.4 Systems

We now focus on the Cubical notion of *system*, which allows to define functions whose value depends on where we are on the cube. A system is an expression of the form $[\phi_1 \to a \mid \phi_2 \to b]$, which evaluates to a on ϕ_1, and b on ϕ_2. This expression is defined on $\phi_1 \vee \phi_2$ which is called the extent of that system. Obviously, this makes sense only when a and b coincide on $\phi_1 \wedge \phi_2$ w.r.t. definitional equality. In Cubical, systems are valid terms only if their extent is equal to $1f$.

In our encoding, a system of type A and extent ϕ, is a term of type `ceps(faceType phi) -> eps l A`. The embedding into terms of type A is done by applying a proof that the current context is on face ϕ. So, a (binary) partial system is build with the following symbol:

```
def TermSys : l : Lev -> f1 : ceps F -> f2 : ceps F ->
```

```
   tau : T l ->
   A1 : (ceps(faceType f1) -> eps l tau) ->
   A2 : (ceps(faceType f2) -> eps l tau) ->
   coh : (e : ceps (faceType (Fmin f1 f2)) ->
          tCubicalEq l tau (A1 (fp1 f1 f2 e)) (A2 (fp2 f1 f2 e))) ->
   ceps (faceType (Fmax f1 f2)) ->  eps l tau.
```

which implements the rule

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, \phi_1 \vdash a_1 : A \qquad \Gamma, \phi_2 \vdash a_2 : A \qquad \Gamma, \phi_1 \wedge \phi_2 \vdash a_1 = a_2 : A}{\Gamma, \phi_1 \vee \phi_2 \vdash [\phi_1 \to a_1 \mid \phi_2 \to a_2] : A}$$

Actually, we implemented a more general rule where the first premisse is $\Gamma, \phi_1 \vee \phi_2 \vdash A$ type, but this adds a lot of technicalities on the handling of face witnesses. We shall not give all the details here, but making those witnesses irrelevent (either definitionaly or propositionaly) is necessary.

This condition and the fact that the theory uses dependent types makes the use of systems in practice really complex, especially when more than two branches are involved since one has to check multiple side conditions.

3.5 Composition and the example of filling

The formal definition of composition is

$$\frac{\Gamma \vdash \phi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \text{ type} \qquad \Gamma, \phi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A(i0)[\phi \to u(i0)]}{\Gamma \vdash \text{comp}^i\ A\ [\phi \to u]\ a_0\ :\ A(i1)[\phi \to u(i1)]}$$

where the notation $\Gamma \vdash t : A[\phi \to u]$ is a shorthand for $\Gamma \vdash t : A$ and $\Gamma, \phi \vdash u = u : A$, in other words t as type A and is definitionally equal to u on face ϕ. This cannot be expressed via rewriting. So we used external equality both for the premisse and the conclusion, which is split into two symbols:

```
def primCompTerm : l : Lev -> phi : ceps F -> A : (ceps I -> T l)
    -> u : (ceps (faceType phi) -> i : ceps I -> eps l (A i))
    -> a0 : eps l (A 0)
    -> (e : ceps (faceType phi)-> tCubicalEq l (A 0) a0 (u e 0))
    -> eps l (A 1).

def primCompEq : l : Lev -> phi : ceps F -> A : (ceps I -> T l)
     -> u : (ceps (faceDataType phi) -> i : ceps I -> eps l (A i))
    -> a0 : eps l (A 0)
    -> coh :(e:ceps(faceType phi -> tCubicalEq l (A 0) a0 (u e 0))
    -> e : ceps (faceType phi)
    -> tCubicalEq l (A 1) (primCompTerm l phi A u a0 coh) (u e 1).
```

Combining composition with systems, one can prove many important theorems such as transitivity of paths. To make an example use of our encoding as well as to test how usable it was in practice, we implemented the example of filling, which draws the line between a point a_0 and the composition $\text{comp}^i\ A\ [\phi \to u]\ a_0$.:

$$\text{fill}^i\ A\ [\phi \to u]\ a_0\ j\ :\ A(j)$$

Because of the verbosity of our encoding, making the proof by hand (*i.e.* creating the fill term) was quite complicated in Dedukti. The situation would be better with the development version of Dedukti which features an interactive proof construction engine.

3.6 Translation of Cubical expressions

The soundness and conservativity of the encoding (using an translation function) is clearly a quite hard problem. Here we will only sketch how those results should be obtained.

The first step is to translate Cubical terms into a Dedukti well-typed terms in the theory of 2LTTs extended with the Cubical-specific piece of theory described above. This is quite difficult since parts of the definitional equality of Cubical is translated to external equalities of 2LTTs. Given two Cubical-convertible types A and B, a term M of types A is also of type B, while in the 2LTT encoding, $[\![M]\!]$ has type $[\![A]\!]$, but getting a term of type $[\![B]\!]$ requires the transport principle `CubicalJ`. Formally, one would say that the translation domain is Cubical derivations rather than mere terms.

Proving the soundness of this translation raises the difficulty that the same term may be typed by different derivations (using conversion at different places), resulting in translated terms that may not be convertible. One important lemma is to show they are actually equal w.r.t. external equality. This is where it is important that the external equality satisfies axiom UIP and functional extensionality. We note that is problem is equivalent to the one of encoding extensional type theory into intensional type theory extended with the above two axioms, see [10]

The conservativity proof follows the same idea as the one of 2LTTs.

4 Conclusion

We gave an encoding of 2LTTs as a Dedukti theory, and specialized it to an encoding of a subsystem of Cubical Type Theory (excluding glueing).

The 2LTT encoding was rather straightforward, once the typing rules are expressed syntactically. Definitional equality could be encoded as rewrite rules, which made the soundness proof rather easy. However, we consider 2LTTs as an important logical framework to express theories which definitional equality is very rich. Other theories could be expressed in our encoding. Beyond the HoTT family of formalisms, we may cite CoqMTU [6], an extension of type theory with a decidable first-order theory.

The Cubical encoding required much more care. There are two main reasons for this. Firstly, Cubical is based on an algebraic structure (De Morgan algebra) which properties cannot be encoded easily as rewrite rules (commutativity, distributivity, idempotence). The answer to this problem may be to extend the expressivity of the rewrite rules of Dedukti: rewriting modulo AC is being considered, but it is not clear if this work is useful for theories with more properties, like having a unit element. The second reason is that the definitional equality of Cubical includes a decision procedure. Rewrite rules cannot inspect the context, so it does seem possible to encode it without giving up the shallow embedding for faces.

Future work

There are several directions that this work may take. Of course, one is to establish the soundness and conservativity results that we expect hold for our encodings.

We also plan to study how glueing can be added to the encoding. At first glance, the reduction rule associated to glue types cannot be expressed as rewrite rules, so there are probably several interesting problems to study there.

A more concrete concern would be to define the translation function from Cubical to our encoding. In this paper, we have done it by hand on several examples (like filling). It appeared to become very complex when systems are involved. Writing an elaboration function that fills the gaps would be the next step before instrumenting Cubical proof systems to produce Dedukti terms that could be rechecked.

References

[1] Thorsten Altenkirch, Paolo Capriotti & Nicolai Kraus (2016): *Extending Homotopy Type Theory with Strict Equality*. doi:`10.4230/LIPIcs.CSL.2016.21`.

[2] Carlo Angiuli, Kuen-Bang Hou (Favonia) & Robert Harper (2018): *Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities*. In Dan R. Ghica & Achim Jung, editors: *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, *LIPIcs* 119, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 6:1–6:17, doi:`10.4230/LIPIcs.CSL.2018.6`.

[3] Danil Annenkov, Paolo Capriotti, Nicolai Kraus & Christian Sattler (2017): *Two-Level Type Theory and Applications*.

[4] Ali Assaf (2015): *Conservativity of Embeddings in the lambda Pi Calculus Modulo Rewriting*. In Thorsten Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, *LIPIcs* 38, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 31–44, doi:`10.4230/LIPIcs.TLCA.2015.31`.

[5] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory*. *Unpublished manuscript*. Available at `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

[6] Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub & Qian Wang (2011): *CoQMTU: A Higher-Order Type Theory with a Predicative Hierarchy of Universes Parametrized by a Decidable First-Order Theory*. In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, IEEE Computer Society, pp. 143–151, doi:`10.1109/LICS.2011.37`.

[7] Cyril Cohen, Thierry Coquand, Simon Huber & Anders Mörtberg (2016): *Cubical Type Theory: a constructive interpretation of the univalence axiom*.

[8] Martin Hofmann (1997): *Syntax and semantics of dependent types*.

[9] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study.

[10] Théo Winterhalter, Matthieu Sozeau & Nicolas Tabareau (2019): *Eliminating reflection from type theory*. In Assia Mahboubi & Magnus O. Myreen, editors: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, ACM, pp. 91–103, doi:`10.1145/3293880.3294095`.