Programming and proving with classical types

Cristina Matache, Victor B. F. Gomes, and Dominic P. Mulligan

Computer Laboratory, University of Cambridge

Abstract The propositions-as-types correspondence is ordinarily presented as linking the metatheory of typed λ-calculi and the proof theory of intuitionistic logic. Griffin observed that this correspondence could be extended to classical logic through the use of *control operators*. This observation set off a flurry of further research, leading to the development of Parigot's $\lambda\mu$-calculus. In this work, we use the $\lambda\mu$-calculus as the foundation for a system of *proof terms* for classical first-order logic. In particular, we define an extended call-by-value $\lambda\mu$-calculus with a type system in correspondence with full classical logic. We extend the language with polymorphic types, add a host of data types in 'direct style', and prove several metatheoretical properties. All of our proofs and definitions are mechanised in Isabelle/HOL, and we automatically obtain an interpreter for a system of proof terms *cum* programming language—called μML—using Isabelle's code generation mechanism. Atop our proof terms, we build a prototype LCF-style interactive theorem prover—called μTP—for classical first-order logic, capable of synthesising μML programs from completed tactic-driven proofs. We present example closed μML programs with classical tautologies for types, including some inexpressible as closed programs in the original $\lambda\mu$-calculus, and some example tactic-driven μTP proofs of classical tautologies.

1 Introduction

Propositions are types; λ-terms encode derivations; β-reduction and proof normalisation coincide. These three points are the crux of the propositions-as-types correspondence—a product of mid-20$^{\text{th}}$ century mathematical logic—connecting the proof theory of intuitionistic propositional logic and the metatheory of the simply-typed λ-calculus. As the core ideas underpinning the correspondence gradually crystallised, logicians and computer scientists expanded the correspondence in new directions, providing a series of intellectual bridges between mainstream mathematical logic and theoretical computer science.

Yet, for the longest time, the connection between logic and computation exposed by the propositions-as-types correspondence was thought to be specific to intuitionistic logics. These logics stand in contrast to the classical logic typically used and understood by mainstream mathematicians. Indeed, prior to the early 1990s any extension of the correspondence connecting typed λ-calculi with classical logic was by-and-large considered inconceivable: classical proofs simply did not contain any 'computational content'. Many were therefore surprised when Griffin discovered that *control operators*—of the same family as the `call/cc` made infamous by the Scheme programming language—were the technology required to extend the propositions-as-types correspondence to classical logic [Gri90]. Classical proofs do contain 'computational content' after all.

Griffin's publication set off a flurry of further research into classical computation (see for example [Par92, Par93a, Par93b, BB94, dG94b, dG94a, RS94, dG95, BB95, Par97, BHS97, BBS97, dG98, Bie98, dG01, AH03, AHS04], amongst many others). Soon, refinements of his original idea were developed, most prominently in Parigot's $\lambda\mu$-calculus [Par92], which provided a smoother correspondence between classical logic and computation than Griffin's original presentation and now acts as a nexus for further research into classical computation. Griffin took Felleisen's $\mathcal{C}$ operator [FFKD87] as a primitive and assigned it the classical type $\neg\neg A \to A$, finding it neccessary to impose restrictions on the reduction relation of his calculus. Parigot observed that this latter requirement was a side effect of the single-conclusion Natural Deduction system Griffin used as the basis for his typing relation, and by using a deduction system with multiple conclusions instead, one could engineer a classically-typed calculus that enjoyed the usual confluence and preservation properties whilst imposing fewer *a priori* constraints on the reduction strategy. Expressions can be explicitly named with μ-variables to form *commands*. This, coupled with a new binding form, allows control flow to 'jump' in a similar spirit to exceptions and their handling mechanisms in mainstream programming languages.

In this work, we explore what a theorem-prover based on classical type theory may look like, and propose to using terms of the $\lambda\mu$-calculus as a serialisation, or *proof term*, mechanism for such a system. As a first step, we focus on classical (first-order) logic. In systems based on intuitionistic type-theory, e.g. Coq [HH14] and Matita [ARCT11], one is able to extract a proof term from a completed proof. Aside from their use in facilitating computation within the logic, these proof terms have a number of important purposes:

1. Proof terms are independently verifiable pieces of *evidence* that the respective theorem proving system correctly certified that a goal is indeed a theorem. Proof terms can be checked by an auditing tool, implemented independently of the system kernel, which is able to certify that the proof term in hand indeed represents a valid proof. Accordingly, proof terms in theorem proving systems such as Coq and Matita form an important component of the *trust story* for the respective systems, with this idea of an independent checking tool sometimes referred to as the *De Bruijn criterion* [BW05].
2. Proof terms may act as a bridge between independent theorem proving systems. Early versions of Matita maintained proof-term compatibility with contemporaneous versions of Coq. As a result, the Matita system was able to import theorems proved in Coq, and use and compute with them as ordinarily as if they had been proved within the Matita system itself.
3. Proof terms can facilitate proof transformations, and refactorings. By affecting program transformations at the level of proof terms, one automatically obtains a notion of proof transformation.

4. Proof terms are used to extract the 'computational content' of proofs, which has important applications in computer science (e.g. in extracting verified software from mechanised proofs), and in mathematical logic (e.g. in explorations of the Curry-Howard correspondence, and in realisability theory [Kri16]).

Given the existence of Parigot's $\lambda\mu$-calculus, including typed variants, we may expect to be able to use $\lambda\mu$-terms directly to serialise derivations. However, there are two snags: the issue of data types—which theorem proving systems must support in order to permit the verification of interesting programs—and the necessary use of open terms to encode classical tautologies in Parigot's calculus, a more subtle problem which will be explained below.

Addressing first the issue of data types, Parigot explored data type embeddings in his early papers on the $\lambda\mu$-calculus (see e.g. [Par92, Section 3.5]). In particular, in systems of typed λ-calculi based on intuitionistic logic, one typically observes a uniqueness (or canonicity) result when embedding data types into the language, a property inherited from intuitionistic Natural Deduction where theorems possess a unique cut-free proof. This property fails in the classical setting, as classical Natural Deduction does not possess a corresponding uniqueness property, and 'junk' numbers inhabit the usual type of Church-encoded natural numbers as a result. Instead, one can introduce 'selection functions' that pick out the unique intuitionistic inhabitant of a type from the sea of alternatives—a strategy Parigot followed. After Parigot, several others considered classical calculi extended with data. One popular approach has been to consider CPS translations of types—as in [Mur91, CP11, BU02]—as opposed to introducing data types and their operational semantics into a calculus in 'direct style', or imposing various conditions on the arguments of primitive recursive functions added to the calculus—as in [RS94]. Indeed, it is not until the work of Ong and Stewart working in a call-by-value setting [OS97], and later Geuvers et al. working in a call-by-name setting [GKM13], that calculi with data types and an operational semantics in 'direct style' are presented. In the latter work, the $\lambda\mu$-calculus is augmented with an encoding of the natural numbers, along with a primitive recursor for defining recursive functions over the naturals, in the style of Gödel's System T [Göd58] to obtain the $\lambda\mu^{\mathbf{T}}$-calculus. Doing this whilst maintaining desirable metatheoretical properties—such as preservation and normalisation—is a delicate process, and requires balancing a mostly call-by-name reduction strategy with the necessarily strict reduction needed to obtain the normal form property for natural number data.

So, rather than the $\lambda\mu$-calculus, we take the $\lambda\mu^{\mathbf{T}}$-calculus as our starting point for a system of proof terms—with a caveat. As Ariola and Herbelin [AH03] noted, and as we alluded to previously, the closed typeable terms of the $\lambda\mu$-calculus (and by extension, the $\lambda\mu^{\mathbf{T}}$-calculus) correspond to a restricted variant of classical logic: 'minimal classical logic'. This logic validates some familiar classical tautologies but not others: the Double Negation Elimination law cannot be captured as a closed, typed program in either calculus, and requires a term with free μ-variables, for example. As working with terms with free variables restricts program transformations and refactoring, this is undesirable from a programming and theorem-proving perspective. We therefore follow Ariola and Herbelin in presenting a calculus with a distinguished μ-constant (called 'top') that can be substituted but not bound by a μ-abstraction. Following our earlier exception analogy, this 'top' element corresponds to an uncatchable exception that bubbles up through a program before eventually *aborting* a program's execution. In this way, familiar classical tautologies and their derivations can be captured as closed programs in our language.

Our system of proof terms is therefore a combination of $\lambda\mu^{\mathbf{T}}$ and the terms of Ariola and Herbelin's calculus. Yet, we must proceed with caution! Adding classical constructs to programming languages and calculi has a fraught history: extending Standard ML with a `call/cc` combinator inadvertently made the language's type system unsound, for example [HL91]. This problem is especially acute as we propose to build a theorem proving system around our terms, and therefore must be sure that our typing system is sound, and reduction well-behaved. We therefore provide a mechanised proof of correctness of the soundness of our proof terms. In particular, our contributions in this work are as follows:

1. We provide an Isabelle/HOL [Gor91] implementation of Parigot's $\lambda\mu$-calculus, along with mechanised proofs of important metatheoretical results. As always, the treatment of name binding is delicate, especially in a calculus with two binding forms with very different runtime behaviour. We use De Bruijn indices [dB72] to handle α-equivalence for λ- and μ-bound variables. This contribution is not discussed in this paper any further, since the next contribution subsumes it. All results can be found in our public repository, mentioned below, and in the Archive of Formal Proofs [MGM17].
2. We extend the calculus above to obtain an explicitly-polymorphic call-by-value[1] variant of the $\lambda\mu$-calculus, *à la* System F [Gir71], mechanised in Isabelle/HOL, along with proofs of desired results. This adds yet another new variety of De Bruijn index for universally quantified type variables. This is presented in Section 2.
3. Extending further, we blend previous work on type systems for full classical logic and work extending Parigot's calculus with data to obtain a typed $\lambda\mu$-calculus with primitive datatypes and a type system corresponding to full first-order classical logic. We provide proofs of progress and type preservation for the reduction relation of this language. This is presented in Section 3.
4. Using our formalisation, we obtain an interpreter for a prototype call-by-value programming language, which we call μ`ML`, using Isabelle/HOL's code generation mechanism and a hand-written parser. We show a closed program whose type is an instance of the Double Negation Elimination law, which is not typeable in the $\lambda\mu$-calculus. The progress and preservation theorems presented in Section 3 ensure that 'well-typed programs do not go wrong'

[1] Strictly speaking, our evaluation strategy is a call-by-weak-head-normal-form. A pure call-by-value $\lambda\mu$-calculus could get stuck when evaluating an application whose argument is a μ-abstraction, which is undesirable. We retain the terminology call-by-value since it gives a better intuition of the desired behaviour.

at runtime, and that our proof terms are therefore well-behaved. This is presented in Section 4.

5. We have built a prototype LCF-style theorem prover called μTP for first-order classical logic around our proof terms. Our theorem prover is able to synthesise μML programs directly from complete tactic-driven proofs in the logic. This theorem prover, as well as example tactic-driven proofs and synthesised programs, is described in Section 5.

All of our proofs are mechanically checked and along with the source code of our LCF kernel are available from a public Bitbucket repository.[2]

2 A polymorphic call-by-value $\lambda\mu$-calculus

Fix three disjoint countably infinite sets of λ-*variables*, μ-*variables* and Λ-*variables* (type variables). We use x, y, z, and so on, to range over λ-variables; α, β, γ, and so on, to range over μ-variables; and a, b, c, and so on, to range over Λ-variables. We then mutually define *terms*, *commands* (or *named terms*) and *types* of the $\lambda\mu$-calculus with the following grammar:

$$\begin{aligned} s,t &::= x \mid \lambda x:\tau.\ t \mid t\ s \mid \Lambda a.\ t \mid t\ \tau \mid \mu\alpha:\tau.\ c \\ c &::= [\alpha]t \\ \sigma,\tau &::= a \mid \forall a.\ \tau \mid \sigma \to \tau \end{aligned}$$

The variables x, α and a are said to be bound in the term $\lambda x:\tau.\ t$, $\mu\alpha:\tau.\ c$ and $\Lambda a.\ t$ respectively. As usual, we work modulo α-equivalence, and write $\mathtt{fv}(t)$, $\mathtt{fcv}(t)$ and $\mathtt{ftv}(t)$ for the set of *free* λ, μ and Λ-*variables* in a term t. These are defined recursively on the structure of t. We call a term t λ-*closed* whenever $\mathtt{fv}(t) = \{\}$, μ-*closed* whenever $\mathtt{fcv}(t) = \{\}$, Λ-*closed* whenever $\mathtt{fcv}(t) = \{\}$ and a term t is simply *closed* whenever it is λ-closed, μ-closed and Λ-closed.

The implementation of terms, commands, and types in Isabelle as HOL data types is straightforward—though terms and commands must be mutually recursively defined. To deal with α-equivalence, we use De Bruijn's nameless representation [dB72] wherein each bound variable is represented by a natural number, its index, that denotes the number of binders that must be traversed to arrive at the one that binds the given variable. Each free variable has an index that points into the top-level context, not enclosed in any abstractions. Under this scheme, if a free variable occurs under n abstractions, its index is at least n. For example, if the index of the free variable x is 3 in the top-level context, the λ-term $\lambda y.\lambda z.((z\ y)\ x)$ is represented in De Bruijn notation as $\lambda.\lambda.((0\ 1)\ 5)$.

In the polymorphic $\lambda\mu$-calculus, there are three distinct binding forms, and therefore we have three disjoint sets of indices. Henceforth, a λ-abstraction is written as $\lambda:\tau.\ t$ where τ is a type annotation and the name of the bound variable is no longer specified. Similarly for μ-abstractions. Universal types and type variable abstractions are simply written as $\forall\tau$ and Λt, respectively.

[2] See: `https://bitbucket.org/Cristina_Matache/prog-classical-types`

$$\frac{\Gamma(x) = \tau}{\Gamma; \Delta \vdash x : \tau} \qquad \frac{\Gamma\langle 0 : \sigma\rangle; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda : \sigma.\ t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash t\ s : \tau}$$

$$\frac{\uparrow^0_\Lambda (\Gamma); \uparrow^0_\Lambda (\Delta) \vdash t : \tau}{\Gamma; \Delta \vdash \Lambda t : \forall \tau} \qquad \frac{\Gamma; \Delta \vdash t : \forall \tau}{\Gamma; \Delta \vdash t\ \sigma : \tau[0 := \sigma]}$$

$$\frac{\Gamma; \Delta\langle 0 : \tau\rangle \vdash_C c}{\Gamma; \Delta \vdash \mu : \tau.\ c : \tau} \qquad \frac{\Gamma; \Delta \vdash t : \tau \qquad \Delta(\alpha) = \tau}{\Gamma; \Delta \vdash_C [\alpha]t}$$

Figure 1. The rules for typing judgements in the polymorphic $\lambda\mu$-calculus.

Capture-avoiding substitution. The polymorphic $\lambda\mu$-calculus has four different substitution actions: a *logical substitution* used to implement ordinary β-reduction, a *structural substitution* used to handle substitution of μ-variables, and two substitutions for types, one into terms, and one into other types.

Write $\uparrow^n_\lambda (t)$ and $\uparrow^n_\mu (t)$ for the De Bruijn *lifting* (or *shifting*) functions for λ- and μ-variables, respectively. These increment the indices of all free λ-variables (respectively μ-variables) in term t that are greater or equal to the parameter n. An analogous pair of operations, the De Bruijn *dropping* (or *down-shifting*), written $\downarrow^n_\lambda (t)$ and $\downarrow^n_\mu (t)$, decrement indices that are strictly greater than n. Using the lifting functions, we define logical substitution recursively on the structure of terms, and write $t[x := s]$ for the term t with all free occurrences of x replaced by s in a capture-avoiding manner. We draw attention to two cases: λ and μ-abstractions. When a substitution is pushed under a λ-abstraction (respectively, a μ-abstraction), the indices of the free λ-variables in s are shifted by 1 so that they keep referring to the same variables as in the previous context:

$$(\lambda : \tau.\ t)[x := s] = \lambda : \tau.\ (t[x + 1 := \uparrow^0_\lambda (s)])$$
$$(\mu : \tau.\ c)[x := s] = \mu : \tau.\ (c[x := \uparrow^0_\mu (s)])$$

Note here that in the first clause above, the λ-variable x is pushed through a λ-abstraction, and must therefore be incremented, whilst in the second clause the λ-variable x is being pushed through a μ-abstraction, and therefore does not need to be incremented as there is no risk of capture.

We can also define de Bruijn lifting functions for free Λ-variables in terms, $\uparrow^n_\Lambda (t)$, and types, $\uparrow^n_\Lambda (\tau)$. Using these functions, define substitution of type τ for all free occurrences of type variable a in term t, $t[a := \tau]$, or in type σ, $\sigma[a := \tau]$. Two interesting cases in these definitions are for Λ-abstractions and $\forall$-types:

$$(\Lambda t)[a := \tau] = \Lambda(t[(a + 1) := \uparrow^0_\Lambda (\tau)])$$
$$(\forall \sigma)[a := \tau] = \forall(\sigma[(a + 1) := \uparrow^0_\Lambda (\tau)])$$

When substituting inside these binders, the index a and the indices of the free type variables in τ must be incremented.

Typing judgement. We implement *typing environments* as (total) functions from natural numbers to types, following the approach of Stefan Berghofer in his formalisation of the simply typed λ-calculus in the Isabelle/HOL library. An empty typing environment may be represented by an arbitrary function of the correct type as it will never be queried when a typing judgement is valid. We split typing environments, dedicating one environment to λ-variables and another to μ-variables, and use Γ and Δ to range over the former and latter, respectively. Consequently, our typing judgement for terms is a four-part relation, $\Gamma; \Delta \vdash t : \sigma$, between two typing contexts, a term, and a type.

We write '$\Gamma; \Delta \vdash t : \sigma$', or say that '$\Gamma; \Delta \vdash t : \sigma$ is *derivable*', to assert that a complete derivation tree rooted at $\Gamma; \Delta \vdash t : \sigma$ and constructed using the rules presented in Figure 1 exists. If $\Gamma; \Delta \vdash t : \sigma$, we say that t is *typeable* in Γ and Δ with type σ. We implement the typing judgement as a pair of mutually recursive inductive predicates in Isabelle—one for terms and one for commands. We write $\uparrow^n_\Lambda (\Gamma)$ and $\uparrow^n_\Lambda (\Delta)$ for the extension of the lifting operations to environments.

Note that care must be taken in the treatment of free variables in the implementation of our typing judgement. In particular, a free variable is represented by its top-level index, and this index must be incremented by 1 for each λ-binder above the variable. For example, consider the judgement $\Gamma, \Delta \vdash \lambda : \tau.\ (3\ 0) : \tau \to \delta$ which states that under the typing environments Γ and Δ the term $\lambda : \tau.\ (3\ 0)$ has type $\tau \to \delta$—the free variable 3 is actually represented by 2 in Γ, that is, $\Gamma\ 2 = \tau \to \delta$. To make sure that the typing environment is kept in a consistent state, the operation to add a new binding to the environment $\Gamma\langle 0 : \tau\rangle$ (respectively, $\Delta\langle 0 : \tau\rangle$) is a *shifting* operation. Here, the value of Γ at 0 is now τ, and all other variables that were previously in the environment are shifted up by one, and if 2 was associated with $\tau \to \delta$, 3 is instead associated with this type after shifting. This shifting operation is defined as:

$$\Gamma\langle i : a\rangle = \lambda j.\ \texttt{if}\ j < i\ \texttt{then}\ \Gamma\ j\ \texttt{else}\ \texttt{if}\ j = i\ \texttt{then}\ a\ \texttt{else}\ \Gamma(j-1).$$

and possesses the useful property $\Gamma\langle n : \tau\rangle\langle 0 : \delta\rangle = \Gamma\langle 0 : \delta\rangle\langle n+1 : \tau\rangle$. This equation is used extensively in our formalisation to rearrange typing contexts.

Important properties of the typing relation may be established by straightforward inductions on derivations in Isabelle. For example:

Theorem 1 (Unicity of typing). *A closed term has at most one type.*

It is also the case that the De Brujin lifting functions preserve a term's typing:

Lemma 1. *If $\Gamma; \Delta \vdash t : \tau$, then*

1. $\Gamma\langle x : \delta\rangle; \Delta \vdash \uparrow^x_\lambda (t) : \tau$
2. $\Gamma; \Delta\langle \alpha : \delta\rangle \vdash \uparrow^\alpha_\mu (t) : \tau$
3. $\uparrow^a_\Lambda (\Gamma); \uparrow^a_\Lambda (\Delta) \vdash \uparrow^a_\Lambda (t) : \uparrow^a_\Lambda (\tau)$

As a corollary, we obtain a proof that logical substitution preserves a term's typing, which is established by induction on the derivation of $\Gamma\langle x : \delta\rangle; \Delta \vdash t : \tau$:

Lemma 2. *If $\Gamma\langle x : \delta\rangle; \Delta \vdash t : \tau$ and $\Gamma; \Delta \vdash s : \delta$ then $\Gamma; \Delta \vdash t[x := s] : \tau$.*

Similarly, type substitution preserves a term's typing:

Lemma 3. *If $\uparrow^a_\Lambda (\Gamma); \uparrow^a_\Lambda (\Delta) \vdash t : \sigma$ then $\Gamma; \Delta \vdash t[a := \tau] : \sigma[a := \tau]$*

Structural substitution. Defining structural substitution is more involved. We follow [GKM13] in defining a generalised version of Parigot's structural substitution with the help of *contexts*, and develop some associated machinery before defining substitution proper. This generalised version of structural substitution is particularly useful when considering extensions of the calculus, which we will discuss later in the paper. First, we define contexts with the following grammar:

$$E ::= \Box \mid E\ t \mid E\ \tau$$

Intuitively, a context is either a 'hole' in a term denoted by $\Box$, which can be filled by an associated *instantiation*, or a context applied to a fixed term—i.e. 'holes' are either at the top-level of a term, or on the left, in application position, applied to a series of fixed arguments. Note that contexts are linear, in the sense that only one hole appears in a context.

We write $E[t]$ for the term obtained by instantiating the hole in E with the term t. Naturally, we may wonder when instantiating the hole in a context is type-preserving. To assess this, we define a typing judgement for contexts $\Gamma; \Delta \vdash E : \delta \Leftarrow \tau$ which indicates that the term $E[t]$ has type δ whenever $\Gamma; \Delta \vdash t : \tau$. The definition of this relation is straightforward—and again is easily implemented in Isabelle as an inductive relation—so we elide the definition here. We can characterise the behaviour of this relation with the following lemma, which shows that the relation $\Gamma; \Delta \vdash E : \delta \Leftarrow \tau$ is correct:

Lemma 4. *$\Gamma; \Delta \vdash E[t] : \delta$ if and only if $\Gamma; \Delta \vdash E : \delta \Leftarrow \tau$ and $\Gamma; \Delta \vdash t : \tau$ for some type τ.*

The result follows by induction on the structure of E in one direction, and an induction on the derivation of $\Gamma; \Delta \vdash E : \delta \Leftarrow \tau$ in the other.

De Bruijn shifting operations can be lifted to contexts in the obvious way, commuting with the structure of contexts, lifting individual fixed terms and evaporating on holes. We write $\uparrow^x_\lambda (E)$, $\uparrow^\alpha_\mu (E)$ and $\uparrow^a_\Lambda (E)$ for the extension of the shifting of a λ, a μ and a Λ-variable, respectively, to contexts. These operations preserve a context's typing, in the following sense:

Lemma 5. *1. If $\Gamma; \Delta \vdash E : \sigma \Leftarrow \rho$ then $\Gamma\langle x : \delta\rangle; \Delta \vdash \uparrow^x_\lambda (E) : \sigma \Leftarrow \rho$*
2. If $\Gamma; \Delta \vdash E : \sigma \Leftarrow \rho$ then $\Gamma; \Delta\langle\alpha : \delta\rangle \vdash \uparrow^\alpha_\mu (E) : \sigma \Leftarrow \rho$
3. If $\Gamma; \Delta \vdash E : \sigma \Leftarrow \rho$ then $\uparrow^a_\Lambda (\Gamma); \uparrow^a_\Lambda (\Delta) \vdash \uparrow^a_\Lambda (E) : \uparrow^a_\Lambda (\sigma) \Leftarrow \uparrow^a_\Lambda (\rho)$

The proof is by induction on the derivation of $\Gamma; \Delta \vdash E : \sigma \Leftarrow \rho$, using Lemma 1.

With the extension of shifting to contexts, we may now define a generalised form of structural substitution. We write $t[\alpha := \beta E]$ for the substitution action which recursively replaces commands of the form $[\alpha]s$ in t by $[\beta]E[s[\alpha := \beta E]]$ whenever α is free. Figure 2 provides defining clauses for only the most complex cases. Here, the case split in the last equation is needed to ensure that typing will be preserved under structural substitution.

We now provide an informal explanation of the final clause in Figure 2, but first we note that correctly defining this generalised structural substitution was not *a priori* obvious, and the correct definition only became apparent later in

$$
\begin{aligned}
(\lambda : \tau.\ t)[\alpha \coloneqq \beta E] &= \lambda : \tau.\ (t[\alpha \coloneqq \beta \uparrow^0_\lambda (E))]) \\
(\mu : \tau.\ c)[\alpha \coloneqq \beta E] &= \mu : \tau.\ (c[(\alpha + 1) \coloneqq (\beta + 1) \uparrow^0_\mu (E))]) \\
(\Lambda t)[\alpha \coloneqq \beta E] &= \Lambda(t[\alpha \coloneqq \beta \uparrow^0_\Lambda (E))]) \\
([\gamma]t)[\alpha \coloneqq \beta E] &= \begin{cases} [\beta](E[t[\alpha \coloneqq \beta E]]) & \text{if } \gamma = \alpha, \\ [\gamma - 1](t[\alpha \coloneqq \beta E]) & \text{if } \alpha < \gamma \leq \beta, \\ [\gamma + 1](t[\alpha \coloneqq \beta E]) & \text{if } \beta \leq \gamma < \alpha, \\ [\gamma](t[\alpha \coloneqq \beta E]) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Figure 2. Structural substitution.

the formalisation, after experimentation with a proof of type preservation. As a result, our explanation focusses on the structural substitution applied to a typed term *in context*.

Consider a term t, such that $\Gamma; \Delta \vdash t : \tau$, and the substitution $t[\alpha \coloneqq \beta E]$. After applying the substitution, the free variable α will be replaced in the typing environment by β, so first examine the case $\alpha < \gamma \leq \beta$. If α has been added to the typing environment using the environment update operation, γ *really* represents the variable $\gamma - 1$ shifted up by 1. However, if β is added instead, $\gamma - 1$ is not shifted up, hence the need to decrement γ by 1 when α is replaced by β. The case $\beta \leq \gamma < \alpha$ is similar, following the same logic.

Here, we observe that the generalised structural substitution defined above preserves a term's typing, as the following lemma demonstrates, which follows from an induction on the derivation of $\Gamma; \Delta\langle\alpha : \delta\rangle \vdash t : \tau$, using Lemma 5:

Lemma 6. *If $\Gamma; \Delta\langle\alpha : \delta\rangle \vdash t : \tau$ and $\Gamma; \Delta \vdash E : \sigma \Leftarrow \delta$ then $\Gamma; \Delta\langle\beta : \sigma\rangle \vdash t[\alpha \coloneqq \beta \uparrow^\beta_\mu (E)] : \tau$.*

The reduction relation. The *values* in the $\lambda\mu$-calculus are λ-abstractions and type abstractions, i.e. $v ::= \lambda : \tau.\ t \mid \Lambda\ t$. In Section 3, we will add data to our language, and hence add more values. We use v, v', and so on, to range over values. We say that a term t is in *weak-head-normal form* when one of the following conditions are met: either t is a value, or there exists α and v such that $t = \mu : \tau.[\alpha]v$ with $\alpha \in \texttt{fcv}(v)$ whenever $\alpha = 0$.

Use n, n', and so on, to range over weak-head-normal forms. Define a *call-by-value reduction relation* between terms using the following six core rules:

$$
\begin{aligned}
(\lambda : \tau.\ t)\ n &\longrightarrow t[0 \coloneqq n] \\
(\mu : \tau_1 \to \tau_2.\ n)\ n' &\longrightarrow \mu : \tau_2.\ (n[0 \coloneqq 0(\Box \ \uparrow^0_\mu (n'))) \\
(\mu : \tau.\ [0]v) &\longrightarrow \downarrow^0_\mu (v) \quad \text{provided that } 0 \notin \texttt{fcv}(v) \\
[\alpha](\mu : \tau.\ n) &\longrightarrow \downarrow^\alpha_\mu (n[0 \coloneqq \alpha\Box]) \\
(\Lambda n)\tau &\longrightarrow n[0 \coloneqq \tau] \\
(\mu : \forall\sigma.\ n)\tau &\longrightarrow \mu : (\sigma[0 \coloneqq \tau]).\ (n[0 \coloneqq 0(\Box\tau)])
\end{aligned}
$$

We combine these rules with 5 congruence rules to implement a fully deterministic call-by-value reduction strategy. In Section 3 we will add pairs and primitive recursion combinators to the language, and will maintain the same left-to-right call-by-value strategy.

Intuitively, weak-head-normal forms are the subset of terms that cannot be reduced further by our reduction relation defined above, and would ordinarily be considered 'values' in any other context. Indeed, we have the property that, for any term t, if there exists an s such that $t \longrightarrow s$ then t is *not* in normal form. Instead, we reserve the term 'value' for a subset of the normal forms which correspond more closely to what one would ordinarily think of as values, i.e. data elements. In particular, once we add data to our language, values will be gradually expanded to include e.g. the boolean and natural number constants, whilst the definition of weak-head-normal forms will remain static. Note that the structure of normal forms is constrained: they may be values, or they may be a value preceded by a single μ-abstraction and name-part that are irreducible (i.e. they are 'almost' values).

Write $t \longrightarrow u$ to assert that t *reduces in one step* to u, according to the rules above. We write $\longrightarrow^*$ for the *reflexive-transitive closure* of the reduction relation $\longrightarrow$, and write $t \longrightarrow^* u$ to assert that t *reduces* to u.

The first rule—the *logical reduction* rule—is the familiar β-reduction rule of the λ-calculus, and needs no further comment. The second rule—the *structural reduction* rule—pushes a normal form n' under a μ-abstraction. In order to avoid the capture of free μ-variables in n, indices must be appropriately incremented. In the third rule—a form of *extensionality* for μ-abstractions, akin to η-contraction in the λ-calculus—a useless μ-abstraction is garbage collected, and the free μ-variables in the value v are adjusted accordingly when v is no longer under the μ-abstraction. Note that we use a value here, as the term $[0]v$ is a normal form. In the fourth rule—the *renaming* rule—the μ-variables greater than α in n also need to be decremented as the μ-abstration is stripped from the term. The fifth rule is the β-reduction rule for types. The final rule is analogous to the second.

We conclude this section with two important metatheoretical results: type preservation and progress, which together imply that well-typed $\lambda\mu$-terms, interpreted as programs, do not 'go wrong' at runtime. In particular, reduction does not change the type of terms, and well-typed closed terms may either reduce further, or are a normal form. Having proved that logical, structural, and type substitution all preserve typing (in Lemmas 2, 3, and 6 respectively) we establish that reduction in the $\lambda\mu$-calculus has the type preservation property:

Theorem 2 (Preservation). *If $\Gamma; \Delta \vdash t : \tau$ and $t \longrightarrow s$ then $\Gamma; \Delta \vdash s : \tau$.*

The result follows by induction on the derivation of $\Gamma; \Delta \vdash t : \tau$. Finally, we establish the progress property for the reduction relation. Note here that progress holds for λ-closed terms, and there need not be any restriction on the set of free μ-variables in the term being reduced:

Theorem 3 (Progress). *For λ-closed t if $\Gamma; \Delta \vdash t : \tau$ then either t is a normal form or there exists a λ-closed s such that $t \longrightarrow s$.*

3 Some extensions

As mentioned in the introduction, Parigot's $\lambda\mu$-calculus has a number of limitations when considered as a prototype 'classical' programming language.

First, 'real' programming languages contain base data types and type constructors, and the addition of data types to the $\lambda\mu$-calculus has historically been a challenge. Second, closed typed terms do not correspond to 'full' classical logic [AH03], as typed closed terms inhabit the type of Peirce's Law, but not the law of Double Negation Elimination. Parigot overcame this issue by allowing non-closed terms under μ-variables. For instance, a derivation of the Double Negation Elimination law is encoded as the term $\lambda y.\mu\alpha.[\phi](y\ (\lambda x.\mu\delta.[\alpha]x))$ in the $\lambda\mu$-calculus, and we note here that the μ-variable ϕ is free, not appearing bound by any μ-abstraction. From a programming perspective, this is undesirable: two morally equivalent terms inhabiting a type will not necessarily be α-equivalent, and substituting a typed term into an arbitrary context may result in μ-variable capture, restricting or complicating the class of refactorings, optimisations, and other code transformations that one may safely apply to a program.

In this section, we consider a number of extensions to the $\lambda\mu$-calculus mechanisation presented in Section 2. In the first instance, we extend the $\lambda\mu$-calculus to make its type system isomorphic to 'full' classical logic, using a technique developed by Ariola and Herbelin. Then, we follow Geuvers et al. in adding a type of natural numbers in 'direct style' to the expanded calculus. Finally, expanding on this addition, we further add booleans, products, and tagged union types to the calculus, to obtain a language more closely aligned with mainstream functional programming languages.

Full classical types. In order to extend the correspondence between the types of closed $\lambda\mu$-calculus to 'full' classical logic, Ariola and Herbelin [AH03] extended the calculus with a falsity type, $\bot$, and a distinguished μ-constant, $\top$, which behaved as any other μ-variable when interacting with structural substitution but could not be bound in a μ-abstraction. A new typing rule for the type $\bot$ was added to the calculus, to obtain the $\lambda\mu_{\mathtt{top}}$-calculus, which was shown to be isomorphic to classical natural deduction.

Another approach—followed previously by Ong and Stewart [OS97], Bierman [Bie98], and Py [Py98], who all noted the peculiarity of open $\lambda\mu$-terms inhabiting tautologies—was to collapse the syntactic category of commands into that of terms, so that any term could be bound by a μ-abstraction. In this work, we follow Ariola and Herbelin's method, since it appears to be more 'modular': we have an existing mechanisation of the $\lambda\mu$-calculus which can easily be extended with new typing rules and constants. The collapsing method, on the other hand, appears to be more disruptive, requiring large changes to the typing system and grammar of our calculus. Accordingly, we extend $\lambda\mu$ with a top μ-variable and a new type $\bot$. The grammar for commands and types now reads:

$$c ::= \ldots \mid [\top]t \qquad \tau ::= \ldots \mid \bot$$

$$\frac{}{\Gamma; \Delta \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma; \Delta \vdash t : \mathbb{N}}{\Gamma; \Delta \vdash \mathtt{S}\ t : \mathbb{N}}$$

$$\frac{\Gamma; \Delta \vdash r : \rho \qquad \Gamma; \Delta \vdash s : \mathbb{N} \to \rho \to \rho \qquad \Gamma; \Delta \vdash t : \mathbb{N}}{\Gamma; \Delta \vdash \mathtt{nrec} : \rho\ r\ s\ t : \rho}$$

$$\begin{aligned}
\mathtt{S}(\mu : \delta.\ n) &\longrightarrow \mu : \delta.\ n[0 \coloneqq 0\ (\mathtt{S}\Box)] \\
\mathtt{nrec} : \tau\ n\ n'\ 0 &\longrightarrow n \\
\mathtt{nrec} : \tau\ n\ n'\ (\mathtt{S}\ \underline{m}) &\longrightarrow n'\ \underline{n}\ (\mathtt{nrec} : \tau\ n\ n'\ \underline{m}) \\
\mathtt{nrec} : \tau\ n\ n'\ (\mu : \delta.\ n'') &\longrightarrow \mu : \tau.\ n''[0 \coloneqq 0\ (\mathtt{nrec} : \tau\ \uparrow^0_\mu (n)\ \uparrow^0_\mu (n')\ \Box)]
\end{aligned}$$

Figure 3. Typing rules and reduction cases added for natural number data.

Additionally, we extend our definition of normal form to include $\mu : \tau.\ [\top]v$, whenever v is a value. We use Isabelle's option type to extend the domain and range of the structural substitution function to support the $\top$ constant. For example, the renaming rule for $\top$ is $[\top](\mu : \tau.\ c) \longrightarrow \downarrow^0_\mu (c[\mathtt{Some}\ 0 \coloneqq \mathtt{None}\ \Box])$, where `None` denotes the $\top$ constant in the structural substitution.

In a well-typed term, the command $[\top]t$ is well-typed whenever t is of type $\bot$. We therefore extend our typing system with the following additional rule:

$$\frac{\Gamma; \Delta \vdash t : \bot}{\Gamma; \Delta \vdash_C [\top]t}$$

Note that $\top$ need not be added to the μ-context Δ in the rule's premiss.

We may now write μ-closed terms that are not typeable in the $\lambda\mu$-calculus. For example, a proof of the Double Negation Elimination law is encoded as the term $\lambda.\mu.[\top](0\ (\lambda.\mu.[1]0))$ (with type annotations omitted).

Adding the natural numbers. We follow [GKM13] and extend our calculus with the natural numbers in 'direct style'. The grammar of terms, types, values and contexts is extended with the following new syntactic categories:

$$\begin{aligned}
r, s, t &::= \ldots \mid 0 \mid \mathtt{S}\ t \mid \mathtt{nrec} : \tau\ r\ s\ t \\
\tau &::= \ldots \mid \mathbb{N} \\
v &::= \ldots \mid 0 \mid \mathtt{S}\ v \\
E &::= \ldots \mid \mathtt{S}\ E \mid \mathtt{nrec} : \tau\ r\ s\ E
\end{aligned}$$

Modulo changes to the definition of values, the definition of normal forms does not change. Here, the terms 0 and `S` denote zero and the successor functions, respectively, used to embed the unary natural numbers into the calculus, *à la* Gödel's System T. The term $\mathtt{nrec} : \tau\ r\ s\ t$ is a primitive recursion combinator, and reduces to r if t is zero, and to $s\ t\ (\mathtt{nrec} : \tau\ r\ s\ \underline{n})$ if t is $\mathtt{S}\ \underline{n}$.

The typing rules and reduction relation are extended to handle these new cases, and the extensions are presented in Figure 3. Here, we write $\underline{m}$ for m-fold applications of $\mathtt{S}$ to 0, and note that in the last reduction rule for $\mathtt{nrec}$, the index of μ-variables are incremented by 1 to avoid capture of free variables. Evaluation proceeds in a left-to-right direction when evaluating the $\mathtt{nrec}$ combinator, in line with our previously established convention.

Adding booleans, products, and tagged unions. We extend further, adding booleans, products and tagged unions. The grammar of terms, types and values are extended as follows, with contexts also extended in a similar fashion:

$$\begin{aligned}
t, r, s ::= \ldots \ &| \ \mathtt{true} \ | \ \mathtt{false} \ | \ \mathtt{if} : \tau \ t \ \mathtt{then} \ r \ \mathtt{else} \ s \ | \langle t, s\rangle : \tau \ | \ \pi_1 t \ | \ \pi_2 t \ | \\
& \ \mathtt{inl} : \tau \ t \ | \ \mathtt{inr} : \tau \ t \ | \ \mathtt{case} : \tau \ t \ \mathtt{of} \ \mathtt{inl} \ x \Rightarrow s \ | \ \mathtt{inr} \ y \Rightarrow r \\
\sigma, \tau ::= \ldots \ &| \ \mathtt{Bool} \ | \ \sigma \times \tau \ | \ \sigma + \tau \\
v, w ::= \ldots \ &| \ \mathtt{true} \ | \ \mathtt{false} \ | \ \langle v, w\rangle : \tau \ | \ \mathtt{inl} : \tau \ v \ | \ \mathtt{inr} : \tau \ v
\end{aligned}$$

Modulo changes to the grammar of values, normal forms remain unchanged. Here, $\langle t, s\rangle : \tau$ denotes a *pairing* construction, and $\pi_1 t$ and $\pi_2 t$ the first and second *projection* functions of a pair, respectively. Note that type annotations are used liberally, for example in the $\mathtt{if}$ construct the whole expression has type τ. This is somewhat unusual as such a simple type system would never ordinarily require such heavy type annotation. However, the reduction behaviour of the calculus makes the annotations necessary, as they provide the type for the continuation variable when reducing a μ-abstraction, for example, in the rule:

$$\mathtt{if} : \tau \ (\mu : \sigma.n) \ \mathtt{then} \ s \ \mathtt{else} \ r \longrightarrow \mu : \tau.(n[0 := 0 \ (\mathtt{if} : \tau \ \square \ \mathtt{then} \uparrow^0_\mu (s) \ \mathtt{else} \uparrow^0_\mu (r))]$$

It is straightforward to extend the De Bruijn shifting and substitution functions to cope with the new datatypes and type constructors. We note that when handling $\mathtt{case} : \rho \ (\mathtt{inl} : \sigma + \tau \ t) \ \mathtt{of} \ \mathtt{inl} \ x \Rightarrow s \ | \ \mathtt{inr} \ y \Rightarrow r$, the indices of free λ-variables in s and r need to be incremented, since we implicitly represent the bound variables x and y by the De Brujin index 0 inside s and r, respectively.

We include the additional typing and reduction rules for booleans, products and tagged unions. For every type constructor, we have introduction and elimination rules, along with a congruence rule for μ-abstractions. In this extended system, it is straightforward to prove type preservation and progress, with both results following by straightforward inductions.

Theorem 4 (Preservation). *If $\Gamma; \Delta \vdash t : \tau$ and $t \longrightarrow s$ then $\Gamma; \Delta \vdash s : \tau$.*

Theorem 5 (Progress). *For λ-closed t if $\Gamma; \Delta \vdash t : \tau$ then either t is a normal form or there exists a λ-closed s such that $t \longrightarrow s$.*

Further, we can characterise the syntactic form of closed values, based on their type, producing a form of inversion result:

Lemma 7. *If t is a λ-closed value and $\Gamma; \Delta \vdash t : \sigma$ then:*

1. *If $\sigma = \mathbb{N}$ then either $t = \mathtt{0}$ or there exists a value $\underline{n}$ such that $t = \mathtt{S} \ \underline{n}$,*

2. *If* $\sigma = \texttt{Bool}$ *then either* $t = \texttt{true}$ *or* $t = \texttt{false}$*,*
3. *If* $\sigma = \tau_1 + \tau_2$ *then either* $t = \texttt{inl} : \tau_1\ s$ *or* $t = \texttt{inr} : \tau_2\ u$ *for values* s *and* u*,*
4. *If* $\sigma = \tau_1 \times \tau_2$ *then* $t = \langle s, u \rangle : \tau_1 \times \tau_2$ *for values* s *and* u*,*
5. *If* $\sigma = \tau_1 \rightarrow \tau_2$ *then* $t = \lambda : \tau_1.s$ *for some term* s*.*
6. *If* $\sigma = \forall \tau$ *then* $t = \Lambda s$ *for some term* s*.*

The result follows by a straightforward induction. At first glance, Lemma 7 may appear useless, as our progress and preservation theorems merely guarantee that evaluation of a λ-closed program either diverges or evaluation progresses to a normal form. Note however that normal forms are constrained to either be a value, or some irreducible μ-abstraction and name-part combination wrapping a value, i.e. 'almost a value'. In either case, Lemma 7 can be used to inspect the syntactic structure of the value based on its type.

4 μML

μML is a prototype implementation of a strict 'classical' programming language derived from the calculus presented in Section 3. The core of the interpreter is derived from our Isabelle mechanisation via code generation: the interpreter's type-checker and reduction mechanism is extracted and paired with a handwritten parser using the Menhir parser generator [PRG17] and a thin transformation of the Abstract Syntax Tree into a nameless form. An operational semantics is provided by a small-step evaluation function that is proved sound and complete with respect to our evaluation relation. We therefore have the following theorem:

Theorem 6 (Determinism). *The reduction relation of the polymorphic $\lambda\mu$-calculus extended with datatypes is deterministic.*

Finally, our progress and preservation theorems ensure that μML programs do not 'go wrong' at runtime.

Example: Double Negation Elimination. We present a closed μML program inhabiting a type corresponding to the law of Double Negation Elimination:

```
tabs(A) ->
  fun (x : (A -> bot) -> bot) ->
   bind (a : A) -> [abort]. (x (fun (y : A) ->
     bind (b : bot) -> [a]. y
   end end)) end end end
```

Here, the keywords `bind` and `abort` are μML's rendering of the μ-abstraction and the distinguished μ-constant, $\top$, respectively, of our underlying calculus, whilst `[a].t` introduces a command (with name `a`). The keywords `tabs` and `forall` introduce a Λ-abstraction and a universal type respectively. When passed the program above, μML type-checks it, and presents the type back to the user:

```
... : forall(A)(((A -> bot) -> bot) -> A)
```

That is, $\neg\neg A \rightarrow A$, as expected. We obtain the value `fun (x : bot) -> x` after supplying `fun (f : (bot -> bot) -> bot) -> f (fun (x : bot) -> x)` to this function, along with a type parameter, and evaluating, as expected.

```
conjecture (mk_all_t (mk_arrow_t (mk_arrow_t (mk_arrow_t
    (mk_var_t 0) mk_bot_t) mk_bot_t) (mk_var_t 0)));
  apply 0 all_intro_tac;          apply 0 imp_intro_tac;
  apply 0 mu_top_intro_tac;
  apply 0 (imp_elim_tac (mk_arrow_t (mk_var_t 0) mk_bot_t));
  apply 0 (assm_tac 0);           apply 0 imp_intro_tac;
  apply 0 (mu_label_intro_tac 1); apply 0 (assm_tac 0);
qed ();
```

Figure 4. A μTP tactic-driven proof of the conjecture $\forall A.\ \neg\neg A \longrightarrow A$.

Example: implication and product. We present a closed μML program inhabiting a type corresponding to an instance of the classical tautology $\neg(A \to \neg B) \to A \wedge B$:

```
tabs(A) -> tabs(B) ->
  fun (x : (A -> B -> bot) -> bot) ->
    bind (a : A * B) -> [abort]. (x (fun (y : A) ->
      fun (z : B) -> bind (b : bot) ->
        [a]. {y, z} : A * B
    end end end)) end end end end
```

Here, $\{y, z\} : \tau$ is μML's concrete syntax for explicitly-typed pairs. When passed the program above, μML type-checks it, and presents the type back to the user:

```
... : forall(A)(forall(B)(((A -> B -> bot) -> bot) -> A * B))
```

That is, $\neg(A \to \neg B) \to A \times B$, as expected.

5 Synthesis via theorem-proving: μTP

We now present a small, prototype interactive theorem prover based on classical first-order logic, called μTP, and built around our μML proof terms. In particular, this system is able to synthesise μML programs from proofs.

With μTP, we follow the LCF-approach [Mil79] and provide a compact system kernel, written in OCaml, which provides an abstract type of theorems, with smart constructors being the only means of constructing a valid inhabitant of this type. Each smart constructor implements a particular proof rule from our typing relation, mapping valid theorems to new valid theorems. As well as incrementally constructing a formula (i.e. a μML type), each forward proof step also incrementally builds a μML term. Outwith the kernel, we provide a mechanism for backwards-proof, via a system of *tactics*, and a notion of a *proof state*. We note that we need not construct a μML term at all during backwards proof, and therefore there is no need to introduce metavariables into our programs to denote missing pieces of program deriving from incomplete derivations. Rather, the only step that synthesises a μML program is the last collapsing of a completed backwards proof, upon calling `qed`, via a series of *valuation* functions that 'undo' each backwards proof step, wherein a complete μML program is produced.

```
conjecture (mk_all_t (mk_all_t (mk_arrow_t (mk_arrow_t
    (mk_neg_t (mk_var_t 0)) (mk_var_t 1)) (mk_sum_t
      (mk_var_t 0) (mk_var_t 1)))));
  apply 0 all_intro_tac;       apply 0 all_intro_tac;
  apply 0 imp_intro_tac;       apply 0 mu_top_intro_tac;
  apply 0 (imp_elim_tac (mk_neg_t (mk_var_t 0)));
  apply 1 imp_intro_tac;       apply 1 (mu_label_intro_tac 1);
  apply 1 disj_left_intro_tac; apply 1 (assm_tac 0);
  apply 0 imp_intro_tac;       apply 0 (mu_label_intro_tac 1);
  apply 0 disj_right_intro_tac;
  apply 0 (imp_elim_tac (mk_neg_t (mk_var_t 0)));
  apply 1 (assm_tac 0);        apply 0 (assm_tac 1);
qed ();
```

Figure 5. A μTP tactic-driven proof of the conjecture $\forall B.\forall A.(\neg B \longrightarrow A) \longrightarrow B \vee A$.

We have used μTP to prove several theorems in classical first-order logic, and have successfully extracted μML programs from their proofs, including both programs presented in Section 4. As didactic examples we provide μTP proof scripts for the classical theorems $\forall A.\ \neg\neg A \longrightarrow A$ and $\forall B.\ \forall A.\ (\neg B \longrightarrow A) \longrightarrow B \vee A$ in Figure 4 and Figure 5, respectively.[3] The Law of Excluded Middle can be easily derived as well, and indeed follows almost immediately from the second theorem. Neither of these two theorems are intuitionistically derivable.

Proof construction with μTP is interactive. The function `conjecture` takes a closed formula (type) as conjecture and sets up an initial proof state, which can be pretty-printed for inspection by the user. The function `apply` takes a goal number and a tactic to apply and either progresses the proof state using that tactic or fails. Finally, `qed` closes a proof, producing an element of type `thm`, raising an error if the proof is not yet complete. This `qed` step also typechecks the μML proof term by the proof to ensure that the correct program was synthesised.

The basic tactics provided by the μTP system invert each of the typing rules of the μML language. For example, `all_intro_tac` works backwards from a universally quantified goal, whilst `imp_intro_tac` works backwards from an implicational goal, introducing a new assumption. Two tactics—`mu_top_intro_tac` and `mu_label_intro_tac`—are the primitive means of affecting classical reasoning in μTP, corresponding to the two ways to introduce a μ-binder and command combination in the underlying μML proof term. Note that this direct reasoning with μ-binders is low-level, and only intended to 'bootstrap' the theorem proving system: once familiar classical reasoning principles such as the Law of Excluded Middle are established, the user need never have to resort to using either of these two tactics directly.

A μML proof term serialising a μTP proof can be obtained programmatically by the user after finishing a proof. For example the program

[3] Note that formulae are currently manually constructed, due to the lack of a parser.

```
  tabs (B) -> tabs (A) -> (fun (x : (B -> bot) -> A) ->
    (bind (a : B + A) -> ([abort]. ((fun (y : B -> bot) ->
    (bind (b : bot) -> ([a]. ((inr (x y) : B + A)))))
    (fun (z : B) -> (bind (c : bot) ->
      ([a]. ((inl z : B + A))))))))))
```

is automatically extracted from the proof presented in Figure 5. The fact that this program inhabits the correct type is easily established.

6 Conclusions

Proof terms for theorem-proving systems based on intuitionistic type-theory—such as Coq and Matita—serve both as a means of communication between systems, and as a means of independently auditing the reasoning steps taken during a mechanised proof. In this latter respect, proof terms form a crucial component of the *trust story* of a theorem proving system. In this work, we explore the use of proof term technology in theorem-proving systems based on classical, rather than intuitionistic, logic.

In particular, we have used the $\lambda\mu$-calculus as the foundation for a system of proof terms for classical first-order logic. For this to be effective, two extensions were considered: adding data types in 'direct style', and extending the calculus to a full correspondence with classical logic so that all classical tautologies can be serialised by closed proof terms. Accordingly, we designed μML—either a prototype classical programming language or a system of proof terms for classical logic—based on the call-by-value $\lambda\mu$-calculus extended with data types and a distinguished μ-constant, $\top$. All of our proofs have been mechanised in Isabelle/HOL to guard against any unsoundness in the μML type system, and an interpreter for μML was extracted from our Isabelle/HOL definitions.

Atop our system of proof terms, we have built a small prototype LCF-style interactive theorem proving system for classical logic, called μTP. The μTP user may automatically synthesise μML programs from completed tactic-driven μTP proofs. We have presented a number of example tactic-driven proofs of classically (but not intuitionistically) derivable theorems that we have proved in μTP as evidence of the utility of our approach.

Future work The logic considered in this work—first-order logic—is expressive, but strictly less expressive than the higher-order logics typically used in established interactive theorem proving systems. We therefore aim to extend μML further, first to a classical variant of F_ω, and then to consider a classical Calculus of Constructions. Lastly, to make μTP usable as a 'realistic' theorem proving system suitable for formalising real mathematics, one would need features such as conversions, tacticals, and a global definition database. We leave these for future work.

Related work There has been extensive prior work by the developers of Isabelle to retrofit the theorem prover with a system of proof terms, primarily by Berghofer [BN00, Ber03]. Following the standard argument in favour of the LCF design philosophy, one only needs to trust the Isabelle kernel implementation in order to trust any proof carried out in the system, and proof terms *as a source of trust* are strictly not needed. However, this argument breaks down when the system kernel is complex and hard to manually audit—as in the case of Isabelle. As we summarised in the Introduction, proof terms convey several other advantages, such as permitting communication between systems and facilitating proof transformation, making their retrofit an attractive prospect for users and developers of existing systems. We note here that there is a difference in philosophy between our work and that of Berghofer: we take as our starting point a system of proof terms and build a theorem prover around them; Berghofer takes an existing theorem prover and extracts a system of proof terms tailored to it.

Tangentially, the Open Theory format [KH12] is intended to facilitate communication of higher-order logic theorems and definitions between theorem proving systems in the wider HOL family (HOL4, HOL Light, ProofPower, and Isabelle). Here, the unit of communication is a sequence of primitive HOL inferences, rather than a proof term.

Acknowledgments. Gomes and Mulligan acknowledge funding from EPSRC grant EP/K008528 ('REMS: Rigorous Engineering for Mainstream Systems'). We thank the anonymous referees and Peter Sewell for their helpful comments.

References

AH03. Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP*, 2003.

AHS04. Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In *ICFP*, 2004.

ARCT11. A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The Matita interactive theorem prover. In *CADE*, 2011.

BB94. F. Barbanera and S. Berardi. A symmetric lambda calculus for "classical" program extraction. In *TACS*, 1994.

BB95. F. Barbanera and S. Berardi. A strong normalization result for classical logic. *Annals of Pure and Applied Logic*, 76(2), 1995.

BBS97. F. Barbanera, S. Berardi, and M. Schivalocchi. "Classical" programming-with-proofs in $\lambda^{\mathtt{PA}}_{\mathtt{Sym}}$: An analysis of non-confluence. In *TACS*, 1997.

Ber03. S. Berghofer. *Proofs, programs and executable specifications in higher order logic*. PhD thesis, Technical University Munich, Germany, 2003.

BHS97. G. Barthe, J. Hatcliff, and M. H. Sørensen. A notion of classical pure type system. *Electronic Notes in Theoretical Computer Science*, 6, 1997.

Bie98. G. M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In *MFCS*, 1998.

BN00. S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In *TPHOLS*, pages 38–52, 2000.

BU02. G. Barthe and T. Uustalu. CPS translating inductive and coinductive types. In *PEPM*, 2002.

BW05. H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philosophical Transactions A*, 363(1835), 2005.

CP11. T. Crolard and E. Polonowski. A program logic for higher-order procedural variables and non-local jumps. *CoRR*, abs/1112.1554, 2011.

dB72. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5), 1972.

dG94a. P. de Groote. A CPS-translation of the lambda-μ-calculus. In *CAAP*, 1994.

dG94b. P. de Groote. On the relation between the lambda-mu-calculus and the syntactic theory of sequential control. In *LPAR*, 1994.

dG95. P. de Groote. A simple calculus of exception handling. In *TLCA*, 1995.

dG98. P. de Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structures in Computer Science*, 8(6), 1998.

dG01. P. de Groote. Strong normalization of classical natural deduction with disjunction. In *TLCA*, 2001.

FFKD87. M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. F. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52, 1987.

Gir71. J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, 1971.

GKM13. H. Geuvers, R. Krebbers, and J. McKinna. The $\lambda\mu^{\mathtt{T}}$-calculus. *Annals of Pure and Applied Logic*, 164(6), 2013.

Göd58. K. Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 12(3-4), 1958.

Gor91. M. J. C. Gordon. Introduction to the HOL system. In *HOL*, 1991.

Gri90. T. Griffin. A formulae-as-types notion of control. In *POPL*, 1990.

HH14. G. P. Huet and H. Herbelin. 30 years of research and development around Coq. In *POPL*, 2014.

HL91. B. Harper and M. Lillibridge. ML with callcc is unsound: `https://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html`, 1991.

KH12. R. Kumar and J. Hurd. Standalone tactics using OpenTheory. In *ITP*, pages 405–411, 2012.

Kri16. J-L. Krivine. Bar recursion in classical realisability: Dependent choice and continuum hypothesis. In *CSL*, pages 25:1–25:11, 2016.

MGM17. C. Matache, V. B. F. Gomes, and D. P. Mulligan. The $\lambda\mu$-calculus. *Archive of Formal Proofs*, 2017.

Mil79. R. Milner. *LCF: A way of doing proofs with a machine*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979.

Mur91. C. R. Murthy. An evaluation semantics for classical proofs. In *LICS*, 1991.

OS97. C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *POPL*. ACM Press, 1997.

Par92. M. Parigot. Lambda-Mu-Calculus: An algorithmic interpretation of Classical Natural Deduction. In *LPAR*, 1992.

Par93a. M. Parigot. Classical proofs as programs. In *KGC*, 1993.

Par93b. M. Parigot. Strong normalization for second order classical natural deduction. In *LICS*, 1993.

Par97. M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4), 1997.

PRG17. F. Pottier and Y. Régis-Gianas. The Menhir parser generator, 2017.

Py98. W. Py. *Confluence en $\lambda\mu$-calcul.* PhD thesis, 1998.

RS94. J. Rehof and M. H. Sørensen. The λ_Δ-calculus. In *TACS*, 1994.