Research Article

A “Propositions as Types” Interpretation of Classical Logic

Andrew Powell[1]

1. Imperial College London, United Kingdom

This paper constructs a new "propositions as types" interpretation for first-order classical propositional and first- and higher-order classical predicate logic in terms of computer programs that inhabit types. This new interpretation is simpler than the existing control operator based approach in terms of the requirements on functional programming languages. The paper leverages a polymorphic identity function and proposes a small number of type level operations to interpret principles of classical logic such as double negation elimination. There is a focus on inhabited ($\top$) and uninhabited ($\bot$) types and the non-emptiness of the type $\bot \to \bot$ for example is witnessed by the polymorphic identity function. The type level operations can be viewed as type level combinators: polymorphic identity (for $\bot \to \bot$ and $A \to A$ for non-empty abstract type A), instantiation (for $\bot \to \top$ and $A \to B$ for non-empty abstract types A and B) and destruction (for $\top \to \bot$). The type level operations can be used to produce a generic witness that corresponds to the construction of a witness of a type in intuitionistic type theories and that forms a bridge between classical and intuitionistic type theories. The proofs in the paper use a classical meta-logic, where a type is either inhabited by a term or not, which has the advantage that it enables hypothetical reasoning. All logical connectives are defined in terms of implication and universal quantification. The intent is to witness classical logical truths within a functional programming paradigm, with the caveat that hypothetical reasoning needs to be built into the programming environment, otherwise the program will not be able to determine which disjunct or instance of an existentially quantified proposition is true.

Corresponding author: Andrew Powell, andrew.powell@imperial.co.uk

1. Introduction

In systems of the λ-calculus where terms represent computable functions, it is possible to assign types to terms in order to help ensure that all computations (reduction sequences of the terms applied to an input term) terminate. Typed systems of the λ-calculus typically trade termination of computations with a unique result for lack of computing universality (as there are computable functions which cannot be computed in any given typed system). We will recall this fact at the end of this section and in Section 4. Typed systems of the λ-calculus are due to Alonzo Church (see [1]), who developed the Simple Theory of Types. There have been significant developments by Jean-Yves

 doi.org/10.32388/IY3BKH.2

Girard (see [2][3][4][5]) and Thierry Coquand (see [6]) among others, and the field was later systematised by Henk Barendregt (see [7])).

The "propositions as types" view of logic, due initially to Haskell Curry (see [8][9]) and William Howard (see[10]),[1] states that there is a correspondence (known as the *Curry-Howard Isomorphism* after Curry and Howard) between a type being non-empty and the proposition corresponding to the type being true, and similarly a type being empty and the proposition corresponding to the type being false. Moreover, types are inhabited by (computer) programs, while proofs of propositions correspond to those programs (see [11] for a systematic development). There are many different systems of types used in the (typed) λ-calculus, but they can be very rich systems indeed in terms of expressiveness, the system called the Calculus of Constructions with inductive types (due to [6][12]) being capable of formalising proof in much of mathematics and being the basis of proof assistants like Rocq and Lean.

To give a simple example of the Curry-Howard Isomorphism, consider the type of functions from an abstract type A to another abstract type B, written $A \to B$, then there is a correspondence with $PropCor(A) \implies PropCor(B)$, where $\implies$ is logical implication and $PropCor : Type \to Prop$ is a function from abstract types to abstract propositions. An abstract type is a variable that can stand for any type, and an abstract proposition is a variable that can stand for a function from symbolic expressions (usually called *sentences*) to truth values, true or false. We will use the same variables for types and propositions in this paper where there is no risk of confusion. There are typically types corresponding to logical connectives including existential and universal quantification, written $(\exists x : A)P(x)$ and $(\forall x : A)P(x)$ over some base non-empty abstract type A for first-order quantification, where A is not free in P.

Due mainly to Per Martin-Löf's *intuitionistic type theory* (see [13][14][15]) building on the intuitionistic semantics of Jan van Brouwer, Arend Heyting and Andrei Kolmogorov (see[16] for example), the development of the logic of types has always required explicit construction of a program (a lambda term) or proof that witnesses that a type is not empty, in particular requiring explicit witnesses of existential quantified types. The natural logic of typed systems of the λ-calculus was for a long time taken to be intuitionistic.

This paper presents a new and simple way to understand classical logic in the propositions as types view of logic. The idea is to study inhabited types and uninhabited types in general, which will be represented by $\top$ and $\bot$ respectively, to show that the standard two-element Boolean algebra of truth values applies to them using an interpretation of proofs as programs, and to produce λ-terms that witness the types corresponding to true propositions. This interpretation of classical logic looks like intuitionist logic, but the lack of specific program instantiations (which amount to the use of the identity function) is a characteristic of classical logic. This goal of this paper is to show that propositions as types can be applied in a natural manner to classical logic, and will have been successful if the approach is convincing and that classical logic rules such as double negation elimination are not added to type theory as axioms or definitions but can be taken as constructive logic rules (compare[11] s7.4, s11.8, s11.11). To be clear, the interpretation presented is an alternative to the control operator interpretation of classical logic presented below.

The current literature on propositions as types for classical logic uses an approach pioneered in the late 1980s by Timothy Griffin (see [17]). Griffin produced a way of extending typed systems of the λ-calculus to rules such as double negation elimination in classical logic. What Griffin did was to show that there are functions in some programming languages, known as *call with current continuation*, of the form $f : (A \to R) \to R$ for R an abstract type representing the type of the output of the computable function f on the assumption that any (continuation) function $k : A \to R$ can be referenced and called by f.[2] In terms of double negation elimination, where $R = \bot$, informally we can interpret R as an exception. Then what $(A \to \bot) \to \bot$ says is that if any program $a : A$ leads to an exception, then an exception would result. Hence no $a : A$ can lead to an exception, and thence $a : A$.

In programming terms, following Griffin's method we introduce a program $f : ((A \to \bot) \to \bot) \to A$. f begins with a program $a : A$ (since we are interested in the case where A corresponds to a true proposition and assume that we can find a witness of A),[3] which f saves as a continuation at its start. Now f runs any program $p : (A \to \bot) \to \bot$ on any $g : A \to \bot$. p needs to raise an exception because $p(g) : \bot$, which is impossible. p then checks whether g exists by computing $g(a)$, but since $g(a) : \bot$ we see that g does not exist as $a : A$ exists. So program g raises an exception, and f having run both p and g, now backtracks and continues to run a. $a : A$ follows.

The computing paradigm that Griffin's interpretation relies on requires a powerful programming language which can implement exceptions and handle backtracking when an exception is raised. The programming language used by Griffin in [17] is an idealized version of the Scheme programming language, which is able to catch exceptions and jump to different states of the program's execution. Although there has been a large literature based on Griffin's idea (see [18] [19][20] for example), the idea and the programming language support needed is quite complicated, programs are prone to error and are not type safe (since the state of the program can be altered during run time). More fundamentally, typed systems of the lambda calculus tend not to need exception handling because all computations terminate, so exception handling might not be the right way to interpret typed systems of the lambda calculus that use classical logic. That said, it is true that a witness to $\neg\neg(\exists x : A)P(x)$ for computably decidable P can be taken to be the same as the witness to true $(\exists x : A)P(x)$ on Griffin's interpretation (ignoring the exception handling and backtracking), so any alternative interpretation will need to be tested against this version of Griffin's interpretation.

2. Preliminaries

The approach taken in this paper is to use definitions of logical connectives in terms of "implies", written " $\Longrightarrow$ ", and the corresponding types in terms of the set of functions from one type to another, "$\to$". We assume a standard presentation of type theory as in [11] with $Type$ the kind of types, $Prop$ the kind of propositions and $Bool$ the type comprising $\{\top, \bot\}$, for $\top$ standing for true and $\bot$ for false. We will also use $\top$ and $\bot$ for true and false propositions and for an inhabited and uninhabited type respectively. We will say informally that $A = \top$ means that A is inhabited as a type or true as a proposition, and likewise $A = \bot$ means that A is empty or uninhabited as a type and false as a proposition. We will also use statements of the form $a : \bot$ to mean that a does not exist, since $\bot$ is not inhabited.

In this paper "a inhabits type A" is written $a : A$ and $A \to B$ is type of all functions from A to B. A function $a : A \to B$ is both the name given to a term in the typed λ-calculus and an association of every inhabitant of type A with an inhabitant of type B. The existence of $\top$ and $\bot$ is assumed, and definitions of "for all", written "$\forall$", are given in Appendix A for first, second and higher order logic, and all logical connectives are defined in terms of $\{\top, \bot, \implies, \forall\}$.

Due to the fact that the identity function $i : \bot \to \bot$ is a common witness in classical logic, we use type identity function i to reduce other types to $i : \bot \to \bot$, where $i(A \to \bot) = \bot$, $i((\forall x : A)\bot) = \bot$, $i((\forall P : A \to Bool)\bot) = \bot$ and $i((\forall P : TR(n)(A))\bot) = \bot$, for $TR(1)(A) := (A \to Bool)$ and $TR(n+1)(A) := TR(n)(A) \to Bool$ for $n : \mathbb{N}$ and $n > 0$, if type A is inhabited and any predicate P bound by a universal quantifier acts on $\bot$ under the identity function i. The rationale for using type level functions is set out in Section 5. It should be remembered that computation in any lambda calculus system is a matter of substituting one lambda term into another, so the type reduction must be read syntactically.

The meta-logic of the typed lambda calculus is taken to be classical. That is, it is either the case that a type A has a member, a say, $a : A$, or else A is empty, $A = \bot$. Of course, because A is an abstract type, that is a type variable, a is an abstract term, that is a term variable.[4]

As far as notation is concerned, the emphasis is on human readability. We use brackets rather than dots to indicate priority of function abstraction and application, and will sometimes be explicit about the type of a function in the function body, for example $(\lambda x : \bot)(y : \top) : \bot \to \top$ emphasising that the return type y is such that $y : \top$ rather than being the judgment "y is true". Sometimes the type is added as a subscript, for example $i_{\bot \to \bot}$ meaning that the identity function i has type $\bot \to \bot$ or $i : \bot \to \bot$. When inference requires assumptions (called *context*), we explicitly state the assumptions in natural language, and for clarity also use a deductive notation ("$\vdash$") to represent assumptions that are made in the construction of a witness term. It will be seen that the context provides witnesses for existentially quantified propositions (see Section A.2 *et seq.*). We often mention that one term t (usually a variable) is *substitutable for* another x in a predicate P such that $P(x)$. This means that if, for example, $P(x) = (\forall y : A)Q(x, y)$ and t is substitutable for x in $P(x)$ then y is not a free variable in t (or otherwise that y is *not free* in t). The same definition applies to all well-formed first-order formulas and to all well-formed higher-order formulas that contain higher type variables that appear in Appendix A.

The definitions used in Appendix A are not unnecessarily higher-order, so that for example "A or B" is written $A \vee B := (A \implies \bot) \implies B$ rather than $A \vee B := (\forall C : Prop)((A \implies C) \implies B)$ (used in classical logic) or $A \vee B := (\forall C : Prop)((A \implies C) \implies ((B \implies C) \implies C)))$ (used in second-order or polymorphic intuitionistic type theories such as Girard's System F [4]). The reason for this choice is that negation is treated differently here than in intuitionistic type theories. It is common to define negation intuitionistically as $\neg A := (\forall C : Prop)(A \implies C)$, but we prefer the simpler $\neg A := (A \implies \bot)$ as it says in terms of types that "if $b : A \to \bot$ and $a : A$ then $b(a) : \bot$". This is absurd because $\bot$ is uninhabited; hence A is uninhabited and therefore

doi.org/10.32388/IY3BKH.2

proposition $\neg A$ is true. We go into further detail in Section 5, but in essence if double negation $\neg\neg A = ((A \implies \bot) \implies \bot)$ is true then $A \implies \bot$ is absurd. This means that there is no $d : A \to \bot$, which is possible only if A is inhabited, *i.e.* proposition A is true.

The only exemption to using first-order definitions where possible, other than in the formulation of the second and higher-order predicate calculus, is in the formulation of the identity $i : \bot \to \bot$ in functional programming terms, which could be written $(\bot \implies \bot) := (\forall C : Prop)(C \implies C)$ or $(\bot \to \bot) := (\forall C : Type)(C \to C)$ in terms of types. The reason for using this second-order polymorphic construction for $i : \bot \to \bot$ is so that the identity function can be called by functions $f : \top \to \top$ such as the identity function $i : A \to A$ for inhabited A. When we have $i : \bot \to \bot$ we first find out from the context an appropriate inhabited type, A say, and then use the polymorphism to replace $i : \bot \to \bot$ with $i : A \to A$. For example, given that $(A \to \bot) \to \bot$ reduces to $\bot \to \bot$ at the type level if $a : A$ using $i(A \to \bot) = \bot$, we can replace $\bot \to \bot$ with $A \to A$ if $a : A$, and we could infer that $i : A \to A$ is a witness for $(A \to \bot) \to \bot$. A more complicated example is, given that $(\exists x : A)P(x)$ reduces to type $\bot \to \bot$ if $p(a) : P(a)$ and $a : A$, we can infer $i : P(a) \to P(a)$ and then substitute $i : (\exists x : A)P(x)$ for $i(p(a)) = p(a)$. The reason that this type of inference is valid is that we are only reusing the assumptions in the context. In Appendix A we will indicate the use of this polymorphism rule.

In any system of typed λ-calculus, there are a standard set of term formation rules. The basic abstraction rule can be defined inductively as $(\lambda x : A)t : A \to B$ for variable $x : A$ and term $t : B$, and the basic application rule is $st : B$ for term $s : A \to B$ and term $t : A$. If $s = (\lambda x : A)r : A \to B$, then $st = r[x := t]$, where $r[x := t]$ is the result of substituting t for x in r, which is usually known as β*-reduction*. β-reduction also applies to second and higher order types, so that in this paper we can also form $(\lambda x : TR(n)(A))t : TR(n)(A) \to B$, where t is a term of type B, $TR(1)(A) := (A \to Bool)$ *and* $TR(n+1)(A) := TR(n)(A) \to Bool$ for $n : \mathbb{N}$ (see Section Section A.4 for details). We will respect the role of bound and free variables, and ensure that free variables do not become bound unintentionally by an abstraction (usually called α*-conversion*), and will allow $(\lambda x : A)tx$ to be replaced by term t if variable x is not free in t (called η*-conversion*).

3. Truth Functions

We introduce truth functions as a way to make proving inference rules and axiom schemas easier and less repetitive. Let $\bot$ be any false proposition, called *falsum* traditionally, where in a type context $\bot$ is the empty type, *i.e.* the type with no members. We also introduce a symbol $\top$, *verum* traditionally, for any true proposition and for a non-empty type.

Now note that the type $\bot \to \bot$ is not empty as it contains the identity $i : \bot \to \bot$. Hence $\bot \to \bot = \top$ and $\bot \implies \bot$ is true. Likewise $\bot \to \top$ is not empty because the function $(\lambda x : \bot)(y : \top) : \bot \to \top$ is a function with no input which returns term y, and put slightly differently $(\lambda x : \bot)a : \bot \to A$ when $a : A$, *i.e.* A is not empty for any abstract type A. This construction shows that $\bot \to \top = \top$ and $\bot \implies \top$ is true. Similarly

doi.org/10.32388/IY3BKH.2

$(\lambda x : \top)(y : \top) : \top \to \top$ is a function with input x which returns term y, which represents a standard function that has input term x and returns term y. Hence $\top \to \top = \top$ and $\top \implies \top$ is true. However, there is no function from a non-empty type to an empty type because a function by definition associates every input with an output. Hence $\top \to \bot$ is empty and $\top \to \bot = \bot$, which corresponds to $\top \implies \bot$ is false.

4. Programs that witness Truth Functions

It is possible to provide programs which emulate $i : \bot \to \bot$, and the programs are interpretable as terms in the typed λ-calculus. The idea is that a program that witnesses $(\lambda x : \bot)$ has no input, a program that witnesses $(y : \bot)$ has no output, while a program that witnesses $(\lambda x : \top)$ has some input, and a program that witnesses $(y : \top)$ has some output. This is contrast to the interpretation of $\bot$ as "undefined" or "an exception", which is less applicable to typed systems of λ-calculus than to untyped systems because computations terminate.

Given that $i : \bot \to \bot$ maps no input to no output, i can be regarded as a "no op(eration)", that is as an operation which does nothing. However, since a no op can only be called by a no op, as $(\lambda x : \bot)i(f(x)) : \bot \to \bot$ and $(\lambda x : \bot)g(i(x)) : \bot \to \bot$ only type check if $f(x) : \bot$, $g(x) : \bot$ and $x : \bot$, the "no op" interpretation of $i : \bot \to \bot$ does not fit the functional programming paradigm. It can be seen that $i_{A \to A}(a) = a$ if $a : A$ is a usable "no op" since the identity function does not change the state of a program at all. As noted in Section 2 we justify the use of the context to interpret $i : \bot \to \bot$, but the availability of witnesses depends on the interpretation of $i : \bot \to \bot$ as $i_{A \to A}(a) = a$ if $a : A$.

Since there is always an identity function $i : \top \to \top$ because $\top$ is an inhabited type, we can use the same $i : A \to A$ where $i_{A \to A}(a) = a$ if $a : A$ as was used to interpret $i : \bot \to \bot$. It is possible to consider functions between two inhabited types, A and B say. There is a type *instantiator* function $c_B(a)$ for $a : A$ which constructs a $b : B$. We prefer the latter construction when producing witnesses because $\top$ will be replaced by specific inhabited types in order to produce witnesses, and will primarily use $i : A \to A$ where $i_{A \to A}(a) = a$ if $a : A$ as a way to interpret $i : \bot \to \bot$.

A program that emulates $ExFalso := (\lambda x : \bot)(y : \top) : \bot \to \top$ is a program with no input and some output. Since $(\lambda x : \bot)ExFalso(f(x)) : \bot \to \top$ only type checks if $f(x) : \bot$ and $x : \bot$, this means that f must be $i : \bot \to \bot$. However $(\lambda x : \bot)f(ExFalso(x)) : \bot \to \top$ does type check if $f : \top \to \top$, and so we can say that f can call $Exfalso$. A program like $ExFalso$ is sometimes called a *nullary function* in computer science and can be used to return a *constant* value.

It is possible to emulate a function of type $\top \to \bot$ as a function that has some input and no output, which we write as $F = (\lambda x : \top)(y : \bot) : \top \to \bot$. But we have a problem, for we see that $F(x) : \bot$ by function application, which contradicts the existence of F given that $\bot$ is empty. While it is clearly possible to have a program F that has some input and no output, and such a program has names like *procedure* and *void function*, if all functions and variables are local in scope to a program instantiation of F, F is a useless construct. Moreover, it is impossible to call F with any

function $f : \top \to \top$ since $f(F(x))$ does not type check. In fact the only functions that can call F are the identity $i : \bot \to \bot$ or $ExFalso$. We will assume that F does not exist in any (pure) functional programming language.

5. Operator View of Truth Functions

As we have seen in Section 4, if we view truth functions as operators on programs (and on types), we can view types $\bot \to \bot$ and $\top \to \top$ as the identity operator, i. That is, $(\lambda x : A) i_{A \to A}(x) : A \to A$ whether A is inhabited or not, which can be stated as $i_{A \to A}(A) = A$.

The type $\bot \to \top$ can be viewed as a *type instantiator* operator, c_A, such that $c_A() = a$ if $a : A$. Likewise $\top \to \top$ can be viewed as a *type instantiator* operator function $c_B(a) = b$ for $a : A$ which constructs a $b : B$

We can even give meaning to $\top \to \bot$ at the type level as a *type destructor* operator $D(A) = \bot$ for A non empty. The type destructor $D : Type \to Bool$ is unusual in that it only operates at a type level; there can be no witness function $d : A \to \bot$ as $d(a) : \bot$ if $a : A$, which, as we have seen, is not possible for any function d. Of course $i : \bot \to \bot$ also only operates at the type level, but in that case it is possible to interpret i as a generic identity operator on all types.

What the operator view means is that truth function operators on a type A can be regarded as a combination of identity operators, type instantiators and type destructors. For example, the type instantiator of $\bot \to A$ for inhabited type A corresponds to the type instantiator c_A. The double negation operation $\neg\neg A$ for inhabited type A corresponds to the type $(\top \to \bot) \to \bot$. Thus $(\lambda D : A \to \bot)(D(A)) = (A \to \bot) \to \bot$ if type A is inhabited. We have seen that there is no $d(a) : D(A)$ for $a : A$ because $D(A) = \bot$. Hence the identity function $i_{(A \to \bot) \to \bot} : (A \to \bot) \to \bot$ if $a : A$. Likewise if $g : (A \to \bot) \to \bot$ then if $d : A \to \bot$ we would have $g(d) : \bot$, which is a contradiction unless $d : \bot$ and $g = i_{(\top \to \bot) \to \bot}$. But $d : A \to \bot$ only if there is $a : A$, for then $(A \to \bot) = \bot$. Thus $i_{(A \to \bot) \to \bot} : (A \to \bot) \to \bot$ (if and) only if $a : A$, $d(a) : \bot$ and $i_{(A \to \bot) \to \bot}(d) : \bot$, *i.e.* d does not exist. But $i_{(A \to \bot) \to \bot}$ exists and is the identity function $i : \bot \to \bot$, which can be interpreted as the $i : A \to A$ for inhabited A. If A is not inhabited on the other hand then it is easy to see that $((\bot \to \bot) \to \bot) = \bot$, and we can conclude that $\neg\neg A$ is not inhabited (and *vice versa*, see Section A.1.7).

There are two approaches taken in this paper with regard to how Boolean functions are applied. The first is simply a truth functional computation at type level based on truth value assignments to propositions. We will not go through the mechanical process of verifying the truth of propositional logic as there is nothing novel in doing so. However, the advantage of the proposition as types interpretation of logic is that it is possible to construct the witness function (lambda terms). Truth value assignments are a coarse instrument as they conflate types. This being the case for the witnesses produced in Appendix A, we will only use the following type level reductions, which are the identities $i(A \to \bot) = \bot$, $i((\forall x : A)\bot) = \bot$, $i((\forall P : A \to Bool)\bot) = \bot$ and $i((\forall P : TR(n)(A))\bot) = \bot$ if type A is inhabited and any predicate P bound by a universal quantifier acts on $\bot$ under the identity function i. This construction helps in the justification for the reducibility of $i_{(A \to \bot) \to \bot}$ to $i_{\bot \to \bot}$ (another name for $i : \bot \to \bot$ given above) for example.

6. Logically True Propositions

Boolean logic classifies a proposition as *logically true* if it is true no matter how the truth values of its constituent propositions and Boolean values of predicates (if any) are assigned. For example, to establish the truth of $(A \implies ((A \implies B) \implies B))$, we need to check the truth values of $A \to ((A \to B) \to B)$ for truth values of A and B. If $A = \top$, $B = \top$ then $\top \to ((\top \to \top) \to \top) = \top$. If $A = \top$, $B = \bot$ then $\top \to ((\top \to \bot) \to \bot) = \top$. If $A = \bot$, $B = \top$ then $\bot \to ((\bot \to \top) \to \top) = \top$. Finally, if $A = \bot$, $B = \bot$ then $\bot \to ((\bot \to \bot) \to \bot) = \top$. Thus $A \implies ((A \implies B)) \implies B)$, which is a proposition corresponding to the logical inference rule known as *modus ponens*, is logically true.

In order to produce a witness of type $A \to ((A \to B) \to B)$, if $a : A$, $b : B$, and $f : A \to B$ then $(\lambda x : A)((\lambda y : A \to B)yx) : A \to ((A \to B) \to B)$ and $fa = b$. We will write this more formally as $a : A,\ b : B,\ f : A \to B,\ fa = b \vdash (\lambda x : A)((\lambda y : A \to B)yx) : A \to ((A \to B) \to B)$, from which follows $(\lambda x : A)((\lambda y : A \to B)yx)af = fa = b$. The cases where $A = \bot$ are true by *Ex falso quodlibet* in Section A.1.5. The case where $a : A$ and $B = \bot$ is true as $(\lambda x : A)i_{\bot\to\bot} : A \to (\bot \to \bot)$, since $i(A \to \bot) = \bot$ if type A is inhabited and $(\lambda x : A)i_{\bot\to\bot}a = i_{\bot\to\bot}$. We may in fact replace $i_{\bot\to\bot} : (A \to \bot) \to \bot$ by $i_{A\to A} : (A \to \bot) \to \bot$ if $a : A$ as we have $i_{A\to A}(a) = a$.

A “propositions as types" Boolean logic model for some logically true propositions in first-order classical propositional logic and first-, second- and higher-order classical predicate logic is provided in Appendix A after the discussions and conclusions. The reason for putting the technical exposition in an appendix is to avoid breaking the flow as there is a lot of material even in cut-down form.

7. Discussion

This paper sets out a way to witness the functions of classical logic as computer programs in a functional programming language via the use of hypothetical reasoning. Logical propositions as formulated here are always hypothetical judgments because they are formulated in terms of “ $\implies$ ". Even if the antecedent.of the “ $\implies$ " cannot be proven true (the domain type witnessed), nevertheless the inference stands. Exactly the same is true in purely type-theoretic terms: if $f : A \to B$, then if $a : A$ then $fa : B$. If A is empty, then if B is empty, then identity $i : A \to B$, otherwise instantiator $c_B() : B$. Hypothetical reasoning is built into classical logic and classical type theory. This is why logical truth can be formulated in terms of types which are not in general decidable; a generic witness enables an inference to be seen to be true, while the inference must still follow if there is no witness. Logical truth is a classical notion of what remains true under all truth-values of the constituent propositions. Logically true propositions often come with generic witnesses when viewed as types. By a "generic witness" we mean an identity function or an instantiating function of a type.

It is reasonable to expect that every logical inference rule corresponds to a logically true proposition. The logical inference rule of *modus ponens*, for example, only applies when A is true; yet *modus ponens* is still true as a proposition

when A is false. The view in this article is that logical truths and logically true propositions are interchangeable. It is true that Hilbert-style logical systems focus on a minimal set of inference rules, typically modus ponens, but there is nothing in principle preventing a different choice of inference rule. In natural deduction systems where the aim is to replace axioms with inference rules (introduction and elimination rules), the advantage is that logical complexity can be reduced and then built up again, but there is no reason why one could not start with a logically true proposition and undergo the same process.

There is a view that logical truth could be interpreted as encompassing all truths provable in some typed system of the λ-calculus where a type can be defined and a witness found. For example, in a type theory with the (inductive) type of the natural numbers, $\mathbb{N}$, definable, there will be types defined in terms of $\mathbb{N}$ (such as $(\forall x : \mathbb{N})(\exists y : \mathbb{N})(y = x + x)$ for $x + 0 = x$ and $x + s(y) = s(x + y)$ for $x, y : \mathbb{N}$ and successor function $s : \mathbb{N} \to \mathbb{N}$ where $s(x) \neq 0$) that will be provable in the system insofar as the existence of a witness of the type is concerned. This position appears to be a kind of *logicism*, but the position is only as strong as the cardinality of the language of terms and types that the system uses and only relates to the specific type defined rather than to all well-defined predicates and propositions. Thus the inductive type for $\mathbb{N}$ assumes that $n : \mathbb{N}$ can be constructed for each of countably infinitely many natural number constant terms 0 and inductively $s(n) : \mathbb{N}$ for $n : \mathbb{N}$; and truths about $\mathbb{N}$ are specific to $\mathbb{N}$ and do not apply to types in general. It is best therefore to reserve the term *logical truth* for truths that apply to all well-defined predicates and propositions, and note that *logical inference* is used in reasoning about specific types based on rules for those types. It might be thought that we can extend logic by the use of a richer type theory (like the Calculus of Constructions), but the result is a mathematical system rather than a logical one.

It is possible to regard logically true propositions as a combinatorial game played with non-empty types and the (polymorphic) identity function. This is apparent in the generation of witnesses of non-empty types in Appendix A. While this approach may not be necessary in the case where an axiomatisation of a logic is complete with respect to a semantics (such as first-order predicate logic with the standard semantics, see[21][22] for example), the generative view of logical truth applies to classical logic of any order.

In the proofs in Appendix A, there is the use of a two-tier computation model, one for terms and one for types. This technique enables logical truths to be computationally witnessed (generated) at a type and a term level (in the same way as in Appendix A), but will not work for propositions whose truth values cannot be decided by purely logical principles (such as the truth of a quantified natural number proposition), being instead reliant on additional type-specific inference rules (such as a structural induction principle).

8. Conclusion

In this paper we have provided a simple interpretation of higher order classical logic in terms of (truth) functions witnessing types in the propositions as types view of logic. Functions that inhabit $\bot \to \bot$, $\bot \to \top$ and $\top \to \top$ are the polymorphic identity function, a type instantiator function and a type destructor function, which as programs can

be thought of respectively as a “no op” for the identity, a program to construct a member of a type and a program to delete all members of a type. Double negation elimination, $\neg\neg A$, which is equivalent to $\bot \to \bot$ for an inhabited type, can be thought of as being witnessed by the identity function i acting on a witness $a : A$. We have also seen that type identity functions have been used to produce witness functions for specific types representing logical truths, assuming only type identities of the form $i(A \to \bot) = \bot,\ i((\forall x : A)\bot) = \bot,\ i((\forall P : A \to Bool)\bot) = \bot$ and $i((\forall P : TR(n)(A))\bot) = \bot,$ where $TR(1)(A) := (A \to Bool)$ and $TR(n+1)(A) := TR(n)(A) \to Bool$ for $n : \mathbb{N}$ and $n > 0$, if type A is inhabited and any predicate P bound by a universal quantifier acts on $\bot$ under the identity function i.

Future work could include treating classical logical systems that include equality, comprehension axioms (for second and higher logics) and principles such as the axiom of choice which have a potentially logical status.

Appendix A. Selection of Logical Truths

We now provide a “propositions as types” Boolean logic model for some logically true propositions in first-order classical propositional logic and first-, second- and higher-order classical predicate logic. We will not verify that propositional logic cases can be satisfied by truth values, as that is straightforward, but predicate and high-order logic cases will be checked to show that the substitution rules work correctly. Lambda terms witnessing non-empty types are also produced in interesting cases. The presentation of logical systems here follows[23] in terms of higher-order classical logic, but the presentation includes introduction and elimination rules for natural deduction style logic (see[24][25]).

A.1. First-order Classical Propositional Logic

A.1.1. Implication introduction

To prove $A \implies (B \implies A)$*, in terms of witnesses,* $a : A,\ b : B \vdash (\lambda x : A)((\lambda y : B)x) : A \to (B \to A)$*. Otherwise the cases where* $A = \bot$ *or* $B = \bot$ *are true by* Ex falso quodlibet *in Section A.1.5.*

A.1.2. Curry's Principle

To prove $(A \implies (B \implies C)) \implies ((A \implies B) \implies (A \implies C))$*, in terms of witnesses,* $a : A,\ b : B,\ c : C\ (\lambda x : A \to (B \to C))(\lambda y : A \to B)((\lambda z : A)xz(yz)) : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$*. If* $A = \top, C = \bot$ *and* $B = \top$*, then using* $i(B \to \bot) = \bot, i(A \to \bot) = \bot$ *and* $i((A \to B) \to \bot) = \bot$*, the witness can be written as* $i : \bot \to \bot$ *or* $i_{\bot \to \bot}$*.*

A.1.3. Definition of negation

We define $\neg A := (A \implies \bot)$*.* ***Negation introduction*** $(A \implies \bot) \implies \neg A$ *and* ***negation elimination*** $\neg A \implies (A \implies \bot)$ *follow.*

doi.org/10.32388/IY3BKH.2

A.1.4. Falsum introduction

To prove $A \implies (\neg A \implies \bot)$, we need to check the truth values of $(A \to ((A \to \bot) \to \bot))$ for truth values of A. If $A = \top$, $(\top \to ((\top \to \bot) \to \bot) = \top)$, and if $A = \bot$, $\bot \to \bot = (\bot \to ((\bot \to \bot) \to \bot) = \top)$. In terms of witnesses, if $a : A$ then using $i(A \to \bot) = \bot$ and $i_{\bot\to\bot} : (A \to \bot) \to \bot$, we have $a : A \vdash (\lambda x : A)i_{\bot\to\bot} : A \to ((A \to \bot) \to \bot)$ as a witness, which has the property that $(\lambda x : A)i_{\bot\to\bot}a = i_{\bot\to\bot}$. We can also use the assumption $a : A$ to replace $i : \bot \to \bot$ with $i : A \to A$.

A.1.5. Falsum elimination (ex falso quodlibet sequitur)

To prove $\bot \implies A$ we saw in Section 5 that the witness if $A = \top$ is the type instantiator $c_A : \bot \to A$, otherwise it is $i_{\bot\to\bot}$. We can then note that $(\bot \implies A) \implies A$ is true only if A is true, which translates into $a : A \vdash (\lambda x : \bot \to A)c_A() : (\bot \to A) \to A$, where $(\lambda x : \bot \to A)(c_A())c_A = a$.

A.1.6. Contraposition (modus tollens)

To prove $(A \implies B) \implies ((B \implies \bot) \implies (A \implies \bot))$ if $a : A$ and $b : B$ then using $i(B \to \bot) = \bot$, $i(A \to \bot) = \bot$ and $i_{\bot\to\bot} : (B \to \bot) \to (A \to \bot)$, we can produce the witness term $a : A, b : B \vdash (\lambda x : A \to B)i_{\bot\to\bot} : (A \to B) \to ((B \to \bot) \to (A \to \bot))$. In this case, $(\lambda x : A \to B)i_{\bot\to\bot}c_B(a) = i_{\bot\to\bot}$, where type instantiator $c_B(a) : B$. We could replace $i_{\bot\to\bot} : (B \to \bot) \to (A \to \bot)$ with $(\lambda g : B \to \bot)gc_B : (B \to \bot) \to (A \to \bot)$, but as g does not exist given that $c_B(a) : B$, we will use $(\lambda g : B \to A)gc_B : (B \to A) \to (A \to A)$ instead.

A.1.7. Double negation elimination

To prove $\neg\neg A \implies A$ we cast $\neg\neg A$ as $(A \implies \bot) \implies \bot$. A witness to the non-emptiness of $(A \to \bot) \to \bot$ is the identity function, $i_{\bot\to\bot}$ if A is inhabited; while if $A = \bot$, $((\bot \to \bot) \to \bot) = \bot$, hence if A is empty then so is $(A \to \bot) \to \bot$. Formally, using $i(A \to \bot) = \bot$ and $i_{\bot\to\bot} : \neg\neg A$, $a : A \vdash (\lambda x : \bot \to \bot)a : \neg\neg A \to A$, where $(\lambda x : \bot \to \bot)ai_{\bot\to\bot} = a$. We may replace $i_{\bot\to\bot} : \neg\neg A$ with $i_{A\to A} : \neg\neg A$ given $a : A$ as $i_{A\to A}(a) = a$.

A.1.8. Definition of disjunction

We define $A \vee B := (A \implies \bot) \implies B$ or in type terms $A \vee B := (A \to \bot) \to B$.

A.1.9. Disjunction introduction

To prove the inference $A \implies (A \vee B)$, if $a : A$ and $b : B$, then using $i(A \to \bot) = \bot$, a witness is $a : A$, $c_B() : B \vdash (\lambda x : A)c_B : A \to (\bot \to B)$, while if $a : A$ and B is uninhabited a witness is $a : A \vdash (\lambda x : A)i_{\bot\to\bot} : A \to (\bot \to \bot)$, and if A is uninhabited and $b : B$ a witness is $b : B \vdash (\lambda x : \bot \to \bot)b$, where $(\lambda x : \bot \to \bot)bi_{\bot\to\bot} = b$. We may replace $i_{\bot\to\bot}$ by $i_{A\to A}$ in the former case, and by $i_{B\to B}$ in the latter case.

doi.org/10.32388/IY3BKH.2

A.1.10. Disjunction elimination

To prove $(A \implies C) \implies ((B \implies C) \implies ((A \vee B) \implies C))$, if $a : A$, $b : B$ and $c : C$ then using $i(A \to \bot) = \bot$ a witness is $a : A,\ b : B,\ c : C \vdash (\lambda x : A \to C)((\lambda y : B \to C)(\lambda z : \bot \to B)(yz()) : (A \to C) \to ((B \to C) \to ((\bot \to B) \to C))$, and $(\lambda x : A \to C)((\lambda y : B \to C)(\lambda z : \bot \to B)(yz()) f g c_B = g c_B() = c$, where $f : A \to C$, $g : B \to C$, $c_B : \bot \to B$, $fa = c, c_B() = b$ and $gb = c$. If A, B and C are all empty, the witness is $i_{(\bot\to\bot)\to((\bot\to\bot)\to(\bot\to\bot))}$.

A.1.11. Excluded middle (tertium non datur)

To prove $A \vee \neg A$ we use the definition $A \vee B := (A \implies \bot) \implies B$ in A.1.8 to obtain $(A \implies \bot) \implies (A \implies \bot)$ or in type terms $(A \to \bot) \to (A \to \bot)$. Using $i(A \to \bot) = \bot$ we can see that the identity $i_{\bot\to\bot} : A \vee \neg A$ is a witness if A is not empty, while if A is empty $i_{(\bot\to\bot)\to(\bot\to\bot)} : A \vee \neg A$ is a witness. In fact, if $a : A$ we may replace $i_{\bot\to\bot} : A \vee \neg A$ with $i_{A\to A} : A \vee \neg A$.

A.1.12. Definition of conjunction

We define $A \wedge B := (A \implies (B \implies \bot)) \implies \bot$ or in type terms $A \wedge B := (A \to (B \to \bot)) \to \bot$.

A.1.13. Conjunction introduction

To prove $(A \implies (B \implies (A \wedge B))$, if $a : A$ and $b : B$ then using $i(A \to \bot) = \bot, i(B \to \bot) = \bot$ and $i_{\bot\to\bot} : A \wedge B$, a witness is $a : A,\ b : B \vdash (\lambda x : A)((\lambda y : B) i_{\bot\to\bot}) : A \to (B \to A \wedge B)$, where $(\lambda x : A)((\lambda y : B) i_{\bot\to\bot}) ab = i_{\bot\to\bot}$. We may replace $i_{\bot\to\bot} : A \wedge B$ with type instantiator $c_B : A \wedge B$ for $a : A$ and $c_B(a) = b$.

A.1.14. Conjunction elimination

To prove $(A \wedge B) \implies A$, if $a : A$ and $b : B$ then using $i(A \to \bot) = \bot, i(B \to \bot) = \bot$ and $i_{\bot\to\bot} : A \wedge B$, a witness is $a : A,\ b : B \vdash (\lambda x : \bot \to \bot) a : A \wedge B \to A$, since $(\lambda x : \bot \to \bot) a i_{\bot\to\bot} = a$. Likewise $a : A,\ b : B \vdash (\lambda x : \bot \to \bot) b : A \wedge B \to B$, since $(\lambda x : \bot \to \bot) b i_{\bot\to\bot} = b$. We may replace $i_{\bot\to\bot} : A \wedge B$ with type instantiator $c_A : A \wedge B$ for $b : B$ and $c_A(b) : A$ in the case of type $A \wedge B \to A$ and with type instantiator $c_B : A \wedge B$ for $a : A$ and $c_B(a) : B$ in the case of type $A \wedge B \to B$.

A.2. First-order Predicate Logic

A.2.1. Definition of first-order universal quantification

We define $(\forall x : A) P(x)$ to be the type which has a member of the form $(\lambda x : A) p(x)$ for $p(x) : P(x)$ if $(\forall x : A) P(x)$ is non-empty. In order for $P(x)$ to depend on $x : A$ we write $p : (x : A) \to P(x)$, where $x : (x : A)$, so $p(x) : P(x)$ as before.

doi.org/10.32388/IY3BKH.2

A.2.2. First-order universal introduction

To prove $(P \implies (x : A \implies Q(x))) \implies (P \implies (\forall x : A)Q(x))$, where variable $x : A$ is not free in P, if $t : P \to ((x : A) \to Q(x))$, then $(tp)x : Q(x)$ for $p : P$ and $x : A$ if such a term p exists or $P = \bot$ otherwise. Then $(\lambda p : P)(\lambda x : A)(tp)x : P \to (\forall x : A)Q(x)$ because p does not depend on $x : A$ and so P does not depend on A. If x were free in P, then we only be able to conclude that $(\lambda p : P)(\lambda x : A)(tp)x : (\forall x : A)(P \to Q(x))$. If $P = \bot$ or $(P \implies (x : A \implies Q(x))) = \bot$ then $P \implies (\forall x : A)Q(x)$ by ex falso quodlibet (see Section A.1.5). A witness of the type $(P \to (x : A \to Q(x))) \to (P \to (\forall x : A)Q(x))$ is:

$$d : A,\ e : P,\ f : P \to ((x : A) \to Q(x)) \vdash (\lambda y : P \to ((x : A) \to Q(x))((\lambda p : P)(\lambda x : A)(yp)x)$$

given that $(\lambda y : P \to ((x : A) \to Q(x)((\lambda p : P)(\lambda x : A)(yp)x)fe : (\forall x : A)Q(x)$ and $(\lambda y : P \to ((x : A) \to Q(x))((\lambda p : P)(\lambda x : A)(yp)x)fed : Q(d)$.

A.2.3. First-order universal elimination

To prove $(\forall x : A)(P(x)) \implies P(t)$, for t any term of type A, if $p : (\forall x : A)P(x)$ and $t : A$ then $pt : P(t)$. If $(\forall x : A)P(x) = \bot$, then $\bot \implies P(y)$ by ex falso quodlibet (see Section A.1.5). A witness of the type $(\forall x : A)(P(x)) \to P(t)$ is $t : A,\ p(t) : P(t) \vdash (\lambda p : (\forall x : A)(P(x)))(p(t))$.

A.2.4. Definition of first-order existential quantification

We define $(\exists x : A)P(x)$ to be the type of $(\forall x : A)(P(x) \to \bot) \to \bot$. In order to handle quantification, we introduce a new type identity rule, which is $i((\forall x : A)\bot) = \bot$ if type A is inhabited and $(\forall x : A)$ acts on $\bot$. If $p(a) : P(a)$ and $a : A$ then $(\exists x : A)P(x) = \bot \to \bot$, or otherwise put $i_{\bot\to\bot} : (\exists x : A)P(x)$. We now appeal to the polymorphic definition of $i_{\bot\to\bot}$ applied to the context $p(a) : P(a)$ and $a : A$ to replace $i_{\bot\to\bot}$ with $a : A,\ p(a) : P(a) \vdash i_{P(a)\to P(a)} : (\exists x : A)P(x)$.

A.2.5. First-order existential introduction

To prove $P(t) \implies (\exists x : A)P(x)$ for any well-formed term t of type A, then we need to show that $P(t) \to (\exists x : A)P(x) = \bot$ leads to a contradiction. $P(t) \to (\exists x : A)P(x) = \bot$ leads to $P(t) = \top$ and $(\exists x : A)P(x) = \bot$ and by definition of first-order existential quantification $((\forall x : A)(P(x) \to \bot) \to \bot) = \bot$. It follows that $(\forall x : A)(P(x) \to \bot) = \top$ and by definition of first order universal elimination $P(t) \to \bot = \top$ for term t. Hence, substituting $P(t) = \top$ we have $(\top \to \bot) = \bot = \top$, contradiction. If $p(t) : P(t)$ and $y : A$ then using $i(P(t) \to \bot) = \bot, i((\forall x : A)\bot) = \bot$ and $i_{\bot\to\bot} : (\exists x : A)P(x)$, a witness is

$$t : A,\ p(t) : P(t) \vdash (\lambda z : P(y))i_{\bot\to\bot} : P(t) \to (\exists x : A)P(x)$$

where $(\lambda z : P(t))i_{\bot\to\bot}p(t) = i_{\bot\to\bot}$. In this case $i_{P(t)\to P(t)} : (\exists x : A)P(x)$ where $p(t) : P(t)$ is a witness of $i_{\bot\to\bot}$ using the argument from Section A.2.4.

A.2.6. First-order existential elimination

To prove $(\exists x:A)P(x) \implies ((\forall x:A)(P(x) \implies Q) \implies Q)$, where x is not free in Q, we note that $(\exists x:A)P(x) \to ((\forall x:A)(P(x) \to Q) \to Q) = \bot$ only if $(\exists x:A)P(x) = \top$ and $((\forall x:A)(P(x) \to Q) \to Q) = \bot$, that is $Q = \bot$ and $(\forall x:A)(P(x) \to Q) = \top$. It follows that $(\forall x:A)(P(x) \to \bot) = \top$. But then by definition of $(\exists x:A)P(x)$, we have $((\forall x:A)(P(x) \to \bot) \to \bot) = \top$ and therefore by substitution $(\top \to \bot) = \bot = \top$, contradiction. Therefore $(\exists x:A)P(x) \to ((\forall x:A)(P(x) \to Q) \to Q) = \top$ and $(\exists x:A)P(x) \implies ((\forall x:A)(P(x) \implies Q) \implies Q)$ follows. The reason x is not free in Q is that if x were free in Q the valid inference would be $(\exists x:A)P(x) \implies ((\forall x:A)(P(x) \implies Q(x)) \implies (\exists x:A)Q(x))$, which does not eliminate the existential quantifier. If $p(w):P(w)$, $w:A$ and $r:(\forall x:A)(P(x)$ for variable w substitutable for x, then, using $i(P(x) \to \bot) = \bot$ and $i((\forall x:A)\bot) = \bot$ and $i_{\bot\to\bot}:(\exists x:A)P(x)$, a witness is of the form

$$w:A,\ p(w):P(w),\ r:(\forall x:A)(P(x) \to Q) \vdash$$
$$(\lambda y:(\exists x:A)P(x))((\lambda z:(\forall x:A)(P(x) \to Q))(z(w))y(w))) :$$
$$(\exists x:A)P(x) \to ((\forall x:A)(P(x) \to Q) \to Q)$$

where $(\lambda y:(\exists x:A)P(x))((\lambda z:(\forall x:A)(P(x) \to Q))(z(w))y(w))pr:Q$, which follows from $w:A,\ p(w):P(w) \vdash i_{P(w)\to P(w)}:(\exists x:A)P(x)$ from Section A.2.4, noting that $y(w) = i_{P(w)\to P(w)}(w):P(w)$.

A.3. Second-order Predicate Logic

A.3.1. Definition of second-order universal quantification

We define $(\forall P:A \to Bool)S(P)$ to be the type which has a member of the form $(\lambda P:A \to Bool)s(P)$ for $s(P):S(P)$ if $(\forall P:A \to Bool)S(P)$ is not empty . Here $Bool$ is the type containing true or false, often written as $2 = \{0,1\}$, coding false $\bot$ as 0 and true $\top$ as 1 say. In order for $s(P)$ to depend on $P:A \to Bool$ we write $s:(P:A \to Bool) \to S(P)$, where $P:(P:A \to Bool)$, so $s(P):S(P)$ as before.

A.3.2. Second-order universal introduction

To prove $(R \implies ((P:A \to Bool) \implies S(P))) \implies (R \implies (\forall P:A \to Bool)S(P))$, where variable $P:A \to Bool$ is not free in R, $S(P):Bool$, if $t:R \to ((P:A \to Bool) \to S(P)))$, then $(tr)P:S(P)$ for $r:R$ if such a term r exists or $R = \bot$ otherwise. Then $(\lambda r:R)(\lambda P:A \to Bool)(tr)P:R \to (\forall P:A \to Bool)S(P)$ because r does not depend on $P:A \to Bool$ and so R does not depend on $A \to Bool$. If $R = \bot$ or $(R \implies (P:A \to Bool) \implies S(P))) = \bot$ then $R \implies (\forall P:A \to Bool)S(P))$ by ex falso quodlibet (see Section A.1.5). A witness of the type $(R \to (P:A \to Bool) \to S(P))) \to (R \to (\forall P:A \to Bool)S(P))$ is:

$$d:A \to Bool,\ e:R,\ f:R \to ((P:A \to Bool) \to S(P)) \vdash$$
$$(\lambda y:R \to ((P:A \to Bool) \to S(P)))((\lambda r:R)(\lambda P:A \to Bool)(yr)P)$$

given that $(\lambda y:R \to ((P:A \to Bool) \to S(P)))((\lambda r:R)(\lambda P:A \to Bool)(yr)P)fe:(\forall P:A \to Bool)S(P)$ and $(\lambda y:R \to ((P:A \to Bool) \to S(P)))((\lambda r:R)(\lambda P:A \to Bool)(yr)P)fed:S(d)$.

A.3.3. Second-order universal elimination

To prove $(\forall P : A \to Bool)(S(P)) \implies S(t)$*, where* t *is a term of type* $A \to Bool$*, if* $p : (\forall P : A \to Bool)S(P)$ *then* $pt : S(t)$ *if* $t : A \to Bool$*. If* $(\forall P : A \to Bool)(S(P)) = \bot$*, then* $\bot \implies S(t)$ *by ex falso quodlibet (see Section A.1.5). A witness of the type* $(\forall P : A \to Bool)S(P)) \to S(t)$ *is* $t : A \to Bool,\ s(t) : S(t) \vdash (\lambda s : (\forall P : A \to Bool)(S(P)))(s(t))$.

A.3.4. Definition of second-order existential quantification

We define $(\exists P : A \to Bool)S(P)$ *to be the type of* $(\forall P : A \to Bool)(S(P) \to \bot) \to \bot$*. In order to handle quantification, we introduce a new type reduction rule, which is* $i(\forall P : A \to Bool)\bot) = \bot$ if type A is inhabited *and any predicate* $P : A \to Bool$ *bound by a universal quantifier acts on* $\bot$ *under the identity function* i*. If* $s(P) : S(P)$ *then* $(\exists P : A \to Bool)S(P) = \bot \to \bot$*, or otherwise put* $i_{\bot\to\bot} : (\exists P : A \to Bool)S(P)$*. We now appeal to the polymorphic definition of* $i_{\bot\to\bot}$ *applied to the context* $s(Q) : S(Q)$ *for* $Q : A \to Bool$ *substitutable for* P *to replace* $i_{\bot\to\bot}$ *with* $Q : A \to Bool,\ s(Q) : S(Q) \vdash i_{S(Q)\to S(Q)} : (\exists P : A \to Bool)S(P)$.

A.3.5. Second-order existential introduction

To prove $S(t) \implies (\exists P : A \to Bool)S(P)$ *for* t *a well-formed term of type* $A \to Bool$*, then we need to show that* $S(t) \to (\exists P : A \to Bool)S(P) = \bot$ *leads to a contradiction.* $S(t) \to (\exists P : A \to Bool)S(P) = \bot$ *leads to* $S(t) = \top$ and $(\exists P : A \to Bool)S(P) = \bot$ and by definition of second-order existential quantification $(((\forall P : A \to Bool)(S(P) \to \bot) \to \bot) = \bot$. It follows that $(\forall P : A \to Bool)(S(P) \to \bot) = \top$ and by definition of second-order universal elimination $(S(t) \to \bot) = \top$*. Hence substituting* $S(t) = \top$ *we have* $(\top \to \bot) = \bot = \top$*, contradiction. If* $t : A \to Bool$*,* $s(t) : S(t)$ *then using* $i(S(P) \to \bot) = \bot$*,* $i((\forall P : A \to Bool)(\bot) = \bot$ *and* $i_{\bot\to\bot} : (\exists P : A \to Bool)S(P)$*, a witness of type* $S(t) \to (\exists P : A \to Bool)S(P)$ *is*

$$t : A \to Bool,\ s(t) : S(t) \vdash (\lambda x : S(t))i_{\bot\to\bot}$$

where $(\lambda x : S(Q))i_{\bot\to\bot}s(t) = i_{\bot\to\bot}$*. In this case* $i_{S(t)\to S(t)} : (\exists P : A \to Bool)S(P)$ *where* $s(t) : S(t)$ *is a witness of* $i_{\bot\to\bot}$ *using the argument from Section A.3.4.*

A.3.6. Second-order existential elimination

To prove $(\exists P : A \to Bool)S(P) \implies ((\forall P : A \to Bool)(S(P) \implies R) \implies R)$*, where* P *is not free in* R*, we note that* $((\exists P : A \to Bool)S(P) \to ((\forall P : A \to Bool)(S(P) \to R) \to R) = \bot$ *only if* $(\exists P : A \to Bool)S(P) = \top$ *and* $((\forall P : A \to Bool)(S(P) \to R) \to R) = \bot$*, that is* $R = \bot$ *and* $(\forall P : A \to Bool)(S(P) \to R) = \top$*. It follows that* $(\forall P : A \to Bool)(S(P) \to \bot) = \top$*. But then by definition of* $(\exists P : A \to Bool)S(P)$*, we have* $(((\forall P : A \to Bool)(S(P) \to \bot) \to \bot) = \top$ *and therefore by substitution* $(\top \to \bot) = \bot = \top$*, contradiction. Therefore* $((\exists P : A \to Bool)S(P) \to ((\forall P : A \to Bool)(S(P) \to R) \to R) = \top$ *and* $(\exists P : A \to Bool)S(P) \implies ((\forall P : A \to Bool)(S(P) \implies R) \implies R)$ *follows. The reason* P *is not free in* R *is that if*

doi.org/10.32388/IY3BKH.2

P were free in R the valid inference would be $(\exists P : A \to Bool)S(P) \implies ((\forall P : A \to Bool)(S(P) \implies R(P)) \implies (\exists P : A \to Bool)R(P))$, which does not eliminate the existential quantifier. If $Q : A \to Bool$, $s(Q) : S(Q)$ and term $t : (\forall P : A \to Bool)(S(P) \to R)$ for predicate variable Q substitutable for P, then, using $i(S(P) \to \bot) = \bot$, $i((\forall P : A \to Bool)\bot) = \bot$ and $i_{\bot\to\bot} : (\exists P : A \to Bool)S(P)$, a witness is of the form:

$$Q : A \to Bool,\ s(Q) : S(Q), t : (\forall P : A \to Bool)(S(P) \to R) \vdash$$
$$(\lambda x : (\exists P : A \to Bool)S(P))(\lambda z : (\forall P : A \to Bool)(S(P) \to R))((z(Q)x(Q)) :$$
$$(\exists P : A \to Bool)S(P) \to ((\forall P : A \to Bool)(S(P) \to R) \to R)$$

where $(\lambda x : (\exists P : A \to Bool)S(P))(\lambda z : (\forall P : A \to Bool)(S(P) \to R))(z(Q)x(Q))st : R$, which follows from $Q : A \to Bool,\ s(Q) : S(Q) \vdash i_{S(Q)\to S(Q)} : (\exists P : A \to Bool)S(P)$ from Section A.3.4, noting that $x(Q) = i_{S(Q)\to S(Q)}(Q) : S(Q)$.

A.4. Higher-order Predicate Logic

A.4.1. Definition of higher-order universal quantification

We define types above A by induction $TR(1)(A) := (A \to Bool)$ and $TR(n+1)(A) := TR(n)(A) \to Bool$ for $n : \mathbb{N}$ and $n > 0$. There is no reason in principle why we cannot use transfinite induction by introducing $TR(\omega)(A) := (\lambda n : \mathbb{N})TR(n)(A)$ for example, but here we will follow tradition and treat higher order logic as being of finite type above type A. All higher order types here are Boolean-valued, as we want to form terms about properties, properties of properties, etc.. Then we define $(\forall P : TR(n)(A))S(P)$ to be the type which has a member of the form $(\lambda P : TR(n)(A))s(P)$ for $s(P) : S(P)$ if $(\forall P : TR(n)(A))S(P)$ is not empty.

A.4.2. Higher-order universal introduction

To prove $(R \implies ((P : TR(n)(A)) \implies S(P))) \implies (R \implies (\forall P : TR(n)(A))S(P))$, where variable $P : TR(n)(A)$ is not free in R, $S(P) : Bool$, if $t : R \to (P : TR(n)(A) \to S(P)))$, then $(tr)P : S(P)$ for $r : R$ if such a term r exists or $R = \bot$ otherwise. Then $(\lambda r : R)(\lambda P : TR(n)(A))trP : R \to (\forall P : TR(n)(A))S(P)$ because r does not depend on $P : TR(n)(A)$ and so R does not depend on $TR(n)(A)$. If $R = \bot$ or $(R \implies (P : TR(n)(A)) \implies S(P))) = \bot$ then $R \implies (\forall P : TR(n)(A))S(P))$ by ex falso quodlibet (see Section A.1.5). A witness of the type $(R \to (P : TR(n)(A)) \to S(P))) \to (R \to (\forall P : TR(n)(A))S(P))$ is:

$$d : TR(n)(A),\ e : R,\ f : R \to ((P : TR(n)(A)) \to S(P)) \vdash$$
$$(\lambda y : R \to ((P : TR(n)(A)) \to S(P)))((\lambda r : R)(\lambda P : TR(n)(A))yrP)$$

given that $(\lambda y : R \to ((P : TR(n)(A)) \to S(P)))((\lambda r : R)(\lambda P : TR(n)(A))yrP)fe : (\forall P : TR(n)(A))S(P)$ and $(\lambda y : R \to ((P : TR(n)(A)) \to S(P)))((\lambda r : R)(\lambda P : TR(n)(A))yrP)fed : S(d)$.

A.4.3. Higher-order universal elimination

To prove $(\forall P : TR(n)(A))(S(P)) \implies S(t)$*, where* t *is a term of type* $TR(n)(A)$*, if* $p : (\forall P : TR(n)(A))S(P)$ *then* $pt : S(t)$ *if* $t : TR(n)(A)$*. If* $(\forall P : TR(n)(A))(S(P)) = \bot$*, then* $\bot \implies S(t)$ *by* ex falso quodlibet *(see Section A.1.5). A witness of the type* $(\forall P : TR(n)(A))S(P)) \to S(t)$ *is* $t : TR(n)(A),\ s(t) : S(t) \vdash (\lambda s : (\forall P : TR(n)(A))(S(P)))(s(t))$.

A.4.4. Definition of higher-order existential quantification

We define $(\exists P : TR(n)(A))S(P)$ *to be the type of* $(\forall P : TR(n)(A))(S(P) \to \bot) \to \bot$*. In order to handle quantification, we introduce a new type reduction rule, which is* $i(\forall P : TR(n)(A))\bot) = \bot$ if type A is inhabited *and any predicate* $P : TR(n)(A)$ *bound by a universal quantifier acts on* $\bot$ *under the identity function* i*. If* $s(P) : S(P)$ *then* $(\exists P : TR(n)(A))S(P) = \bot \to \bot$*, or otherwise put* $i_{\bot\to\bot} : (\exists P : TR(n)(A))S(P)$*. We now appeal to the polymorphic definition of* $i_{\bot\to\bot}$ *applied to the context* $s(Q) : S(Q)$ *for* $Q : TR(n)(A)$ *substitutable for* P *to replace* $i_{\bot\to\bot}$ *with* $Q : TR(n)(A),\ s(Q) : S(Q) \vdash i_{S(Q)\to S(Q)} : (\exists P : TR(n)(A))S(P)$.

A.4.5. Higher-order existential introduction

To prove $S(t) \implies (\exists P : TR(n)(A))S(P)$ *for* t *a well-formed term of type* $TR(n)(A)$*, then we need to show that* $S(t) \to (\exists P : TR(n)(A))S(P) = \bot$ *leads to a contradiction.* $S(t) \to (\exists P : TR(n)(A))S(P) = \bot$ *leads to* $S(t) = \top$ and $(\exists P : TR(n)(A))S(P) = \bot$ and by definition of second-order existential quantification $(((\forall P : TR(n)(A))(S(P) \to \bot) \to \bot) = \bot$. It follows that $(\forall P : TR(n)(A))(S(P) \to \bot) = \top$ and by definition of second-order universal elimination $S(t) \to \bot = \top$*. Hence substituting* $S(t) = \top$ *we have* $(\top \to \bot) = \bot = \top$*, contradiction. If* $t : TR(n)(A)$*,* $s(t) : S(t)$ *then using* $i(S(P) \to \bot) = \bot$*,* $i((\forall P : TR(n)(A))(\bot) = \bot$ *and* $i_{\bot\to\bot} : (\exists P : TR(n)(A))S(P)$*, a witness of type* $S(t) \to (\exists P : TR(n)(A))S(P)$ *is*

$$t : TR(n)(A),\ s(t) : S(t) \vdash (\lambda x : S(t))i_{\bot\to\bot}$$

where $(\lambda x : S(Q))i_{\bot\to\bot}s(t) = i_{\bot\to\bot}$*. In this case* $i_{S(t)\to S(t)} : (\exists P : TR(n)(A))S(P)$ *where* $s(t) : S(t)$ *is a witness of* $i_{\bot\to\bot}$ *using the argument from Section A.4.4.*

A.4.6. Higher-order existential elimination

To prove $(\exists P : TR(n)(A))S(P) \implies ((\forall P : TR(n)(A))(S(P) \implies R) \implies R)$*, where* P *is not free in* R*, we note that* $((\exists P : TR(n)(A))S(P) \to ((\forall P : TR(n)(A))(S(P) \to R) \to R) = \bot$ *only if* $(\exists P : TR(n)(A))S(P) = \top$ *and* $((\forall P : TR(n)(A))(S(P) \to R) \to R) = \bot$*, that is* $R = \bot$ *and* $(\forall P : TR(n)(A))(S(P) \to R) = \top$*. It follows that* $(\forall P : TR(n)(A))(S(P) \to \bot) = \top$*. But then by definition of* $(\exists P : TR(n)(A))S(P)$*, we have* $(((\forall P : TR(n)(A))(S(P) \to \bot) \to \bot) = \top$ *and therefore by substitution* $(\top \to \bot) = \bot = \top$*, contradiction. Therefore* $((\exists P : TR(n)(A))S(P) \to ((\forall P : TR(n)(A))(S(P) \to R) \to R) = \top$ *and* $(\exists P : TR(n)(A))S(P) \implies ((\forall P : TR(n)(A))(S(P) \implies R) \implies R)$ *follows. The reason* P *is not free in* R *is that if*

doi.org/10.32388/IY3BKH.2

P were free in R the valid inference would be $(\exists P : TR(n)(A))S(P) \implies ((\forall P : TR(n)(A))(S(P) \implies R(P)) \implies (\exists P : TR(n)(A))R(P))$, which does not eliminate the existential quantifier. If $Q : TR(n)(A)$, $s(Q) : S(Q)$ and term $t : (\forall P : TR(n)(A))(S(P) \to R)$ for predicate variable Q substitutable for P, then, using $i(S(P) \to \bot) = \bot$, $i((\forall P : TR(n)(A))\bot) = \bot$ and $i_{\bot \to \bot} : (\exists P : TR(n)(A))S(P)$, a witness is of the form:

$$Q : TR(n)(A),\ s(Q) : S(Q), t : (\forall P : TR(n)(A))(S(P) \to R) \vdash$$
$$(\lambda x : (\exists P : TR(n)(A))S(P))(\lambda z : (\forall P : TR(n)(A))(S(P) \to R))((z(Q))x(Q)) :$$
$$(\exists P : TR(n)(A))S(P) \to ((\forall P : TR(n)(A))(S(P) \to R) \to R)$$

where $(\lambda x : (\exists P : TR(n)(A))S(P))(\lambda z : (\forall P : TR(n)(A))(S(P) \to R))(z(Q)x(Q))st : R$, which follows from $Q : TR(n)(A),\ s(Q) : S(Q) \vdash i_{S(Q) \to S(Q)} : (\exists P : TR(n)(A))S(P)$ from Section A.4.4, noting that $x(Q) = i_{S(Q) \to S(Q)}(Q) : S(Q)$.

Acknowledgements

I would like to acknowledge the comments of all 3 Qeios reviewers, Miklós Erdélyi-Szabó, Ferruccio Guidi and Rodrigo Geraldo Ribeiro, which were incorporated in this version of the paper. In particular, the abstract has been rewritten using words suggested by Rodrigo Geraldo Ribeiro.

Footnotes

[1] See the history in[26] for example.

[2] We can write $k(a) = (\lambda x : A)k(x)(a)$ for any term a substitutable for variable x. Then we see that $f(k) = k(a)$ and f extends the witness $a : A$ through k so that $f = (\lambda k : A \to R)(k(a))$.

[3] The assumption of a witness to A is sufficient from a logical truth perspective, as the assumption is hypothetical. If the witness is hypothetical we expect the witness to be generic, such as the identity function.

[4] The results of this paper may seem less surprising if one assumes a classical meta-logic, but classical meta-logic enables hypothetical reasoning: if a type A has a member then conclude assertion X, else if A is empty conclude assertion Y, therefore conclude assertion X or Y.

References

1. ^*Church A (1940). "A Formulation of the Simple Theory of Types." J Symbol Logic. 5(2):56–68.*
2. ^*Girard J-Y (1972). "Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur" [Functional Interpretation and Cut Elimination of Higher-Order Arithmetic]. Université de Paris VII.*
3. ^*Girard J-Y (1973). "Quelques Résultats sur les Interpretations Fonctionnelles" [Some Results on Functional Interpretations]. In: Mathias ARD, Rogers H, editors. Cambridge Summer School in Mathematical Logic 1971. Heidelberg: Springer. pp. 232–252.*

4. a, bGirard J-Y (1989). Proofs and Types. Cambridge, UK: Cambridge University Press.
5. ^Girard J-Y (1986). "The System F of Variable Types, Fifteen Years Later." Theor Comput Sci. **45**:159–192.
6. a, bCoquand T, Huet G (1988). "The Calculus of Constructions." Inf Comput. **76**:95–120.
7. ^Barendregt H (1991). "Introduction to Generalized Type Systems." J Funct Program. **1**(2):125–154.
8. ^Curry HB (1934). "Functionality in Combinatory Logic." Proc Natl Acad Sci USA. **20**(11):584–90.
9. ^Curry HB, Feys R (1958). Combinatory Logic. Amsterdam: North-Holland.
10. ^Howard WA (1980). "The Formulae-as-Types Notion of Construction." In: Seldon JP, Hindley JR, editors. To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. New York: Academic Press. pp. 479–490.
11. a, b, cGeuvers H, Nederpelt R (2014). Type Theory and Formal Proof. Cambridge, UK: Cambridge University Press.
12. ^Paulin-Mohring C (2015). "Introduction to the Calculus of Inductive Constructions." In: Woltzenlogel Paleo B, Delahayee D, editors. All About Proofs, Proofs for All. Rickmansworth, UK: College Publications.
13. ^Martin-Löf P (1982). "Constructive Mathematics and Computer Programming." In: Cohen LJ, editor. Logic, Methodology and Philosophy of Science VI, 1979. Amsterdam: North-Holland. pp. 153–175.
14. ^Martin-Löf P (1984). Intuitionistic Type Theory. Naples: Bibliopolis.
15. ^Dybjer P, Palmgren E (2024). "Intuitionistic Type Theory." In: Zalta EN, Nodelman U, editors. The Stanford Encyclopedia of Philosophy. Stanford, CA: Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2024/entries/type-theory-intuitionistic/.
16. ^Troelstra AS (2011). "History of Constructivism in the 20th Century." In: Kennedy J, Kossake R, editors. Set Theory, Arithmetic, and Foundations of Mathematics: Theorems, Philosophies. Cambridge, UK: Cambridge University Press. pp. 150–179.
17. a, bGriffin TG (1989). "A Formulae-as-Type Notion of Control." In: POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 47–58.
18. ^Murthy C (1991). "An Evaluation Semantics for Classical Proofs." In: Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science. Los Alamitos, CA: IEEE; IEEE Press. pp. 96–107.
19. ^Parigot M (1992). "$\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction." In: Voronkov A, editor. Lecture Notes in Computer Science. pp. 190–201.
20. ^van Bakel S (2023). "Adding Negation to Lambda Mu." Log Methods Comput Sci. **19**(2):12:1–12:41.
21. ^Gödel K (1930). "Die Vollständigkeit der Axiome des Logischen Funktionenkalküls" [The Completeness of the Axioms of the Logical Function Calculus]. Monatsh Math. **37**(1):349–360.
22. ^Boolos G, Jeffrey R (1989). Computability and Logic. 3rd ed. New York: Cambridge University Press.
23. ^Hilbert D, Ackermann W (1950). Principles of Mathematical Logic. New York: AMS Chelsea.
24. ^Gentzen G (1969). "Investigations into Logical Deduction." In: Szabo ME, editor. The Collected Papers of Gerhard Gentzen. Amsterdam: North-Holland. pp. 68–131.
25. ^Prawitz D (2006). Natural Deduction: A Proof-Theoretical Study. New York: Dover.

26. ^Wadler P (2015). "Propositions As Types." Commun ACM. **58**(12):75–84.

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.