The Higher-Order-Logic Formath

Pieter Audenaert

Abstract

We introduce a new formalization of Higher-Order-Logic (abbreviated Hol), which we baptized Formath, an acronym for FORMAl MATHematics. We discuss the syntax, semantics, deduction-rules, axioms and principles of extension, after which we prove soundness and consistency. The semantics are comparable to other systems for Hol, such as Hol-4 and Hol-Light, but other parts differ from the traditional way of working, for example the deduction-rules and axioms. We discuss these differences in large extent. We also talk about porting theorems to the Formath library, provide examples and discuss the applications.

1 Introduction and Overview

Higher-Order-Logic is an extension of the well-known propositional and predicate logic, which is very suitable to formally prove significant parts of high-level mathematics. Other applications include soft- and hardware specification and verification. It is a general framework in which virtually anything can be specified, after which one can prove properties of the investigated objects.

In our PhD-thesis [6], we used Higher-Order-Logic to prove software correct. Therefore we designed a new functional programming language from scratch, which we called Alfred. Practical restrictions of some systems for Hol made us consider new alternative logics, of which Formath is the result. Of course, Formath and Alfred match each other well, which facilitates the burden of proving software correct. Moreover, we took this opportunity to rework the core of Hol. We introduced a syntactical distinction between free and bound variables, which solves some technical difficulties during instantiation or substitution. Also, a sequent-style using multiple consequent-formulas allows building proofs in an intuitive way. Most important, the

Received by the editors March 2006 - In revised form in January 2007.
Communicated by A. Hoogewijs.

structure behind the deduction-rules and axioms has been changed. An extended rationale for the development of Formath is included at the end of the paper, where we discuss from our point of view the necessity of creating a new version of Higher-Order-Logic.

This text contains a formal introduction to Formath, discussing syntax, semantics, deduction-rules and axioms, principles of extension and proofs of soundness and consistency. Moreover, we compare our rules and axioms in large detail with those of other systems for Hol. Sections on proof-porting, some examples and the main applications of Formath finish the paper.

2 Syntax and Semantics

In our discussion of Formath, based on the Hol-4 documentation [17], we first develop the syntactic elements called type and term, and provide simultaneously a set-theoretic interpretation using the universe. Types will be interpreted as elements in this universe. These elements are type-sets, and terms will be interpreted as elements in these type-sets.

2.1 Universe

We define a "universe" $\mathcal{U}$ to be a certain set of sets. The axiomatizations are different, but the standard-universe which we will immediately describe, corresponds to the universe of other well-known Hol-systems such as Hol-4 [16] and Hol-Light [18], which are themselves modified versions of the original "simple theory of types" [12], which was developed by Church. In writing down the formulas below, we assume the implication $\Rightarrow$ to be right-associative. All symbols have their usual meaning. We discuss the axioms immediately after their listing.

2.1.1 Basic-universe

A "basic-universe" fulfills the following axioms.

- Axiom 1: $\forall X.\ (X \in \mathcal{U}) \Rightarrow \exists Y.\ (Y \in X)$
- Axiom 2: $\forall X.\ (X \in \mathcal{U}) \Rightarrow \forall Y.\ (\emptyset \neq Y \subseteq X) \Rightarrow (Y \in \mathcal{U})$
- Axiom 3: $\forall X.\ (X \in \mathcal{U}) \Rightarrow \forall Y.\ (Y \in \mathcal{U}) \Rightarrow (X \to Y \in \mathcal{U})$
- Axiom 4: $2 \in \mathcal{U} \wedge 2 = \{0, 1\}$

2.1.2 Standard-universe

A "standard-universe" is a basic-universe which also fulfills the following axioms.

- Axiom 5: $Choice \in \prod_{X \in \mathcal{U}} X$
- Axiom 6: $I \in \mathcal{U} \wedge Infinite(I)$

Theorem: There exists a formalization of $\mathcal{U}$ in ZFC [17].

We always assume universes to be standard, except when noted otherwise. The division between the basic-system and the standard-system will be strictly kept up. The first listing of axioms results in a minimal universe for Higher-Order-Logic. The second listing of axioms results in a universe for a logic which is equivalent to the traditional systems such as Hol-4 and Hol-Light. The explicit division results in an elegant formalization of Higher-Order-Logic.

Informally, the basic-axioms express the following.

- Every element in the universe is a non-empty set. Thus, we assume that every type contains at least one element. Empty types don't exist.
- Every non-empty subset of an element in the universe is itself also an element in the universe. Thus, we assume that every non-empty subtype of an existing type is also a type.
- The function-set of two elements in the universe is itself also an element in the universe. The "function-set" of two types X and Y is the set of all total functions from X to Y, written down as $X \to Y$. A total function from X to Y maps every element in X to exactly one element in Y.
- There exists an element in the universe which contains exactly two elements. We traditionally and confusingly write down this set as $2 = \{0, 1\}$. This set will provide an interpretation of the booleans.

Informally, the standard-axioms express the following.

- There is a choice-function available. We write down the product-set of all elements in the universe as $\prod_{X \in \mathcal{U}} X$. An element herein can be seen as a tuple, which contains exactly one element from every type-set. These elements exist, because of the first axiom. Thus, such a tuple can also be seen as a function: for all $X \in \mathcal{U}$ we have that $Choice(X) \in X$ is an element in X.
- There exists an infinite set in the universe, which we will write down as I (i.e. Individuals). An intuitive approach to the concept infinity will suffice for this moment, as we will provide a formal axiomatization in a later section.

2.2 Types

Types are interpreted in Formath as elements in the universe. The syntax of the types is given by so-called type-structures. Thereafter we discuss the semantics.

2.2.1 Type syntax

Traditionally one distinguishes between four different syntactical classes of types; namely type-variables, atomic types, compound types and function-types. But atomic types are specific sets in the universe, and can be defined as compound types having arity 0. Also, the function-operator $\to$ is a specific compound type, having arity 2. Consequently, we include the atomic types as well as the function-types in the compound types and get a simpler definition.

Compound types will be written down using type-operators. These are sometimes called type-constants, to explicit the difference with type-variables. We assume an infinite set of names for type-constants, written as "TyConsts". Also, we assume an infinite set of names for type-variables, written as "TyVars". Assume these two sets to be disjunct.

A "type-structure" Ω is a set of type-constants. A type-constant is specified by a tuple (ν, n), where $\nu \in \text{TyConsts}$ is a constant-name, and $n \in \mathbb{N}$ is a natural number, which denotes the arity of the constant (or operator). Thus $\Omega \subseteq \text{TyConsts} \times \mathbb{N}$. Two different type-constants never get the same name; thus if $(\nu, n_1) \in \Omega$ and $(\nu, n_2) \in \Omega$ then $n_1 = n_2$. The set of "types" corresponding to a certain structure Ω is written as Types_Ω and is defined as the smallest set which fulfills the following conditions.

- If $\sigma \in \text{TyVars}$, then $\sigma \in \text{Types}_\Omega$. Thus, a type can be a "type-variable" α. Type-variables denote any set in the universe. They are written as $\alpha, \beta, \gamma, \ldots$
- If $(\nu, n) \in \Omega$ and $\sigma_i \in \text{Types}_\Omega$ for all $1 \leq i \leq n$, then $(\sigma_1, \ldots, \sigma_n)\nu \in \text{Types}_\Omega$. The special case $(\nu, 0) \in \Omega$ results in $()\nu \in \text{Types}_\Omega$. Thus, a type can be a "compound type" $(\sigma_1, \ldots, \sigma_n)\nu$. Here, the σ_i are argument-types for the type-operator ν, which has arity n. Using type-operators we can construct sets: the type $(\sigma_1, \ldots, \sigma_n)\nu$ denotes the set which results from applying the operator ν to the sets $\sigma_1, \ldots, \sigma_n$. Of course, the arity has to be exact.

To make things clear we provide a few examples of compound types. A compound type of arity 0 is always a certain set in the universe. For example, the type of booleans called "bool" will be interpreted as a set having two elements. The type-operator called "list" is a compound type with arity 1. It constructs ordered sequences of elements from a certain set, which corresponds to our intuition. An example of a type-operator with arity 2 is the compound type "$\rightarrow$". It constructs function-sets, as defined above. This operator is assumed to be infix and right-associative, such that $\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_{n-1} \rightarrow \sigma_n$ is an abbreviation for $\sigma_1 \rightarrow (\sigma_2 \rightarrow \ldots \rightarrow (\sigma_{n-1} \rightarrow \sigma_n) \ldots)$.

We define the set of type-variables which appear in a type σ recursively and we write it down as $tyvars(\sigma)$.

2.2.2 Type semantics

Assume Ω is a type-structure. A "model $\mathcal{M}$ for Ω" is defined by specifying a n-ary function for every type-constant $(\nu, n) \in \Omega$, written down as $\mathcal{M}(\nu) : \mathcal{U}^n \rightarrow \mathcal{U}$. This is the so-called interpretation of ν by $\mathcal{M}$. Thus, when given the sets $X_1, \ldots, X_n$ in the universe, $\mathcal{M}(\nu)(X_1, \ldots, X_n)$ will also be a set in the universe. For a nullary type-operator ν this results in pointing out a certain set $\mathcal{M}(\nu) \in \mathcal{U}$.

Types, built up without using type-variables, are called "monomorphic". In contrast, types containing type-variables are called "polymorphic". Polymorphic types can't be interpreted directly, but only after the appearing type-variables are effectively instantiated with other types. Thus, a polymorphic type does not correspond with exactly one set in the universe, but with a function, taking sets as arguments. Such a function $\mathcal{U}^n \rightarrow \mathcal{U}$ returns a set for every choice of sets for the appearing

type-variables. Of course, n corresponds to the number of different type-variables. An example of a monomorphic resp. polymorphic type is given by *bool* resp. $(\alpha)list$.

Now we will build up a semantics in which a type σ is interpreted, making use of any type-variables α_i, whether or not appearing in σ. This way of working leads to the introduction of so-called types-in-context. A "type-context" αs is a finite, possibly empty list of different type-variables. A "type-in-context" is a pair, written as $\alpha s.\sigma$, with αs a type-context and σ a type, both for a certain type-structure Ω, such that all type-variables appearing in σ also appear somewhere in the list αs. But the list αs may also contain some other type-variables, not appearing in σ. If $\alpha s = \alpha_1, \ldots, \alpha_n$, then we can write down $\alpha s.\sigma$ as $\alpha_1, \ldots, \alpha_n.\sigma$.

For every σ there are of course some minimal type-contexts αs, such that $\alpha s.\sigma$ is a type-in-context, and these minimal type-contexts differ only in the order in which the type-variables appear in the list. We assume a certain total ordering on the set TyVars and define a "canonical type-context" for a type σ to be a minimal type-context αs, such that the type-variables in αs are ordered.

We now define the semantics of types. Assume $\mathcal{M}$ is a model for a type-structure Ω. For every type-in-context $\alpha s.\sigma$ we define a function $[\![\alpha s.\sigma]\!]_{\mathcal{M}} : \mathcal{U}^n \to \mathcal{U}$, where n is the length of the type-context $\alpha s = \alpha_1, \ldots, \alpha_n$, by induction on the structure of σ in the following way.

- If σ is a type-variable, then $\sigma = \alpha_i$ for a certain $i \in \{1, \ldots, n\}$. This i is unique, because all α_i are different by definition of a type-context. Then $[\![\alpha s.\sigma]\!]_{\mathcal{M}}$ is the i^{th} projection, which will map $(X_1, \ldots, X_n) \in \mathcal{U}^n$ to $X_i \in \mathcal{U}$.
- If $\sigma = (\sigma_1, \ldots, \sigma_n)\nu$ is a compound type, then define $S_i = [\![\alpha s.\sigma_i]\!]_{\mathcal{M}}(Xs)$ for $i = 1, \ldots, n$. Then $[\![\alpha s.\sigma]\!]_{\mathcal{M}}$ maps the list Xs to $\mathcal{M}(\nu)(S_1, \ldots, S_n)$.

We now define the "interpretation" of a type σ for a model $\mathcal{M}$ as the function $[\![\sigma]\!]_{\mathcal{M}} : \mathcal{U}^n \to \mathcal{U}$, given by $[\![\alpha s.\sigma]\!]_{\mathcal{M}}$, where αs is the canonical type-context of σ. If σ is monomorphic, then $n = 0$ and $[\![\sigma]\!]_{\mathcal{M}}$ is identified with the element $[\![\sigma]\!]_{\mathcal{M}}()$ in $\mathcal{U}$. We write $[\![\_]\!]_{\mathcal{M}}$ as $[\![\_]\!]$, whenever the model $\mathcal{M}$ is clear out of the context.

To summarize, assume a model $\mathcal{M}$ for a type-structure Ω. Then the interpretation, using the above semantics, will result to certain sets in $\mathcal{U}$ for monomorphic types, and n-ary functions $\mathcal{U}^n \to \mathcal{U}$ for polymorphic types having n different type-variables.

A few examples to illustrate this. Assume that Ω contains the type-constants $(bool, 0)$ and $(\to, 2)$. Assume that the model $\mathcal{M}$ maps the first type-constant *bool* to the set of booleans, written as $2 \in \mathcal{U}$. Assume that the interpretation of the second type-constant, the function-operator $\to$, for two sets $X \in \mathcal{U}$ and $Y \in \mathcal{U}$ results in the function-set from X to Y, also written using $\to$. Then the following holds.

- $[\![bool \to bool]\!] = 2 \to 2 \in \mathcal{U}$
- $[\![\alpha.(\alpha \to bool) \to \alpha]\!] : \mathcal{U} \to \mathcal{U}$ is the function which maps $X \in \mathcal{U}$ to $(X \to 2) \to X \in \mathcal{U}$.
- $[\![\alpha, \beta.\alpha \to bool \to bool]\!] : \mathcal{U}^2 \to \mathcal{U}$ is the function which maps $(X, Y) \in \mathcal{U}^2$ to $X \to 2 \to 2 \in \mathcal{U}$.

We conclude this section on type-semantics with discussing "type-instantiation". Assume that σ is a type with type-context αs, where $\alpha s = \alpha_1, \ldots, \alpha_n$. Assume that for all $i = 1, \ldots, n$ the types σ_i are given. We then write $\sigma[\alpha_1, \ldots, \alpha_n/\sigma_1, \ldots, \sigma_n]$ as the result of the simultaneous instantiation of all type-variables α_i by the types σ_i in σ. The result is called an "instantiation" of σ. The following holds by structural induction on σ.

- Given the type-in-context $\alpha s.\sigma$, assume $\sigma' = \sigma[\alpha_1, \ldots, \alpha_n/\sigma_1, \ldots, \sigma_n]$ to be an instantiation of σ. Then $\alpha_1, \ldots, \alpha_n$ and $\sigma_1, \ldots, \sigma_n$ are precisely specified by σ and σ'.
- Given the type-in-context $\alpha s.\sigma$, with n the length of αs, and the types-in-context $\beta s.\sigma_i$, with $i = 1, \ldots, n$ and m the length of βs. Assume $\sigma' = \sigma[\alpha s/\sigma s]$ to be an instantiation of σ. Then $\beta s.\sigma'$ is also a type-in-context and this type is related to $\alpha s.\sigma$ for all $Xs \in \mathcal{U}^m$ as follows: $[\![\beta s.\sigma']\!](Xs) = [\![\alpha s.\sigma]\!]([\![\beta s.\sigma_1]\!](Xs), \ldots, [\![\beta s.\sigma_n]\!](Xs))$.

2.3 Terms

Terms in Formath are interpreted as elements in the type-sets. The syntax of terms is given using so-called term-structures. After that, semantics is discussed.

2.3.1 Term syntax

Traditionally, one distinguishes between four different syntactical classes of terms; namely term-variables, constant terms, function-applications and λ-abstractions. However, remark that the implementation of λ-abstractions is not always that easy, for example when instantiating. In that case, one has to take into account free variables, which can suddenly get bound after the instantiation, which is clearly not wanted. Thus, we propose an alternative syntax, which avoids the problems in se: we will make a syntactical distinction between free and bound variables, using a far analogue of the so-called "de Bruijn-indices". This way of working makes the theoretical discussion a little bit more laborious, and uses the new notion of preterms. An in-depth discussion of this issue is included in a later section.

We assume an infinite set called "TeConsts" of names for preterm-constants. Analogously, assume an infinite set "TeVars" of names for free preterm-variables. Assume these two sets to be disjunct. Let Ω be a type-structure. Then a preterm-constant for Ω is a tuple (c, σ), with $c \in \text{TeConsts}$ and $\sigma \in \text{Types}_\Omega$. A "signature" for Ω is then a set Σ_Ω of these constants. The set of "preterms" for a certain Σ_Ω is now written as $\text{PreTerms}_{\Sigma_\Omega}$ and is defined as the smallest set which fulfills the following conditions.

- If $x \in \text{TeVars}$ and $\sigma \in \text{Types}_\Omega$, then $x_\sigma \in \text{PreTerms}_{\Sigma_\Omega}$. A preterm can thus be a "free preterm-variable" x. A free preterm-variable denotes any element in its type-set. They are written using $x, y, z, \ldots$
- If $n \in \mathbb{N}$ and $\sigma \in \text{Types}_\Omega$, then $@^n_\sigma \in \text{PreTerms}_{\Sigma_\Omega}$. A preterm can thus be a "bound preterm-variable" $@^n$.

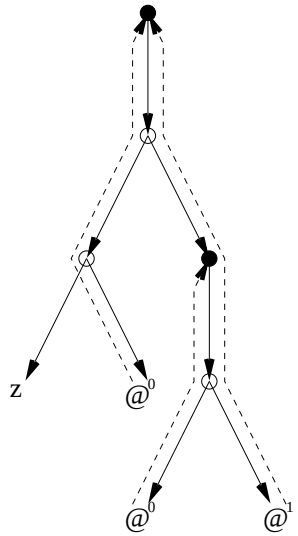
- If $(c, \sigma) \in \Sigma_\Omega$ and $\sigma' \in \text{Types}_\Omega$ is an instantiation of σ, then $c_{\sigma'} \in \text{PreTerms}_{\Sigma_\Omega}$. A preterm can thus be a "preterm-constant" c.
- If $t_{\sigma' \to \sigma} \in \text{PreTerms}_{\Sigma_\Omega}$ and $t'_{\sigma'} \in \text{PreTerms}_{\Sigma_\Omega}$, then $(t_{\sigma' \to \sigma} t'_{\sigma'})_\sigma \in \text{PreTerms}_{\Sigma_\Omega}$. A preterm can thus be a "preterm-function-application" $t\ t'$.
- If $t_\sigma \in \text{PreTerms}_{\Sigma_\Omega}$ and $\sigma' \in \text{Types}_\Omega$, then $(\lambda_{\sigma'}.t_\sigma)_{\sigma' \to \sigma} \in \text{PreTerms}_{\Sigma_\Omega}$. A preterm can thus be a "preterm-λ-abstraction" $\lambda.t$. Remark that we don't write the abstracted preterm-variable behind the λ, but we do keep the type of that variable as a subscript.

Every preterm t_σ is linked to a type σ, out of which type-set the term is assumed to be taken. However, in practice we don't write the type-indication, whenever it is clear out of the context. Then, we write just t instead of the more accurate t_σ. This remark also holds for terms (cf. infra).

Now we define the set of "terms" $\text{Terms}_{\Sigma_\Omega}$ as a certain subset of the preterms. There have to be fulfilled some additional conditions, before a preterm is lifted to be a term. These concern the use of bound variables, which we are going to look at more closely now.

Bound variables have their own syntactical representation, to explicit bindings in a term. Assume a preterm t, having a certain subpreterm $@^n$. Now we look for the binding λ of this bound variable. Therefore, first write down the syntax-tree, associated with the preterm t. Now locate the chosen subpreterm $@^n$. Walk through the syntax-tree upward and choose the $n+1$-th λ above. This one binds the chosen variable. The figure below makes everything clear; it is the syntax-tree of the term $\lambda.z@^0(\lambda.@^0@^1)$. An empty circle $\circ$ represents a function-application, and a filled circle $\bullet$ a λ-abstraction. The dashed arrows show how the binding λ for any $@^n$ can be found.

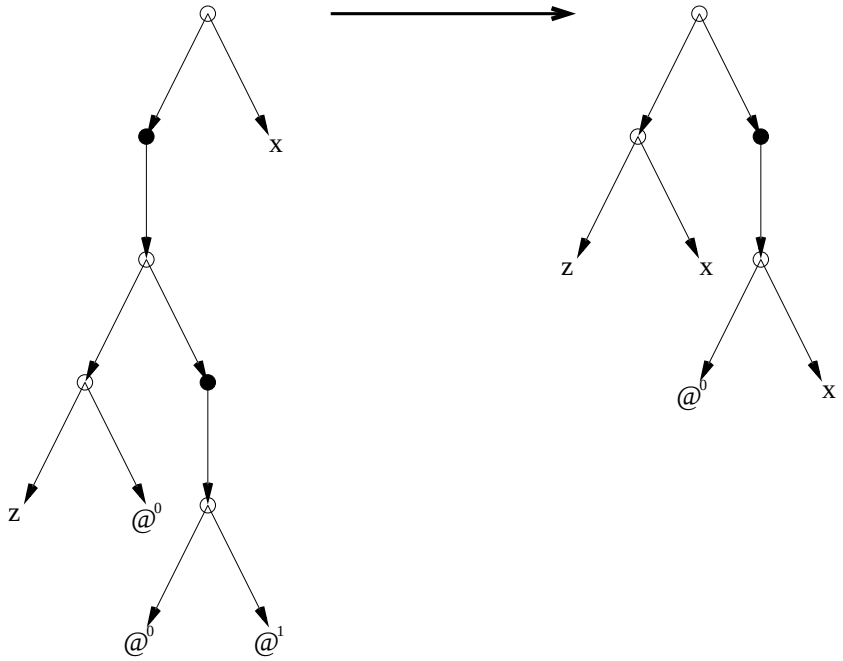
Of course, we assume that for every appearing $@^n$ there indeed are at least $n+1$ upward λ's, otherwise we consider the term not well-formed. This is the first condition to select terms in the set of preterms.



Now consider the set of all subpreterm-variables which are bound by the same upward λ_σ. It is clear that we assume the types of all elements in this set to be equal to σ, which is the second condition to select terms in the set of preterms.

Our analysis of Hol-4, Hol-Light and Formath (cf. infra) contains a discussion and rationale for introducing this syntactical distinction between free and bound variables, so we will suffice here by providing some illustrative examples.

- The expression, traditionally written as $\lambda x.x$, is now written down as $\lambda.@^0$. Indeed, the bound preterm-variable $@^0$ points to the first upward λ in the syntax-tree, which is of course the only possibility.
- The expression, traditionally written as $\lambda x.zx(\lambda y.yx)$, is now written down as $\lambda.z@^0(\lambda.@^0@^1)$. Here, the first $@^0$ points to the outer λ, as is the $@^1$, which is located one level deeper, and consequently gets an incremented index. The second $@^0$ is a bound variable which differs from the first $@^0$. Indeed, the second $@^0$ is bound by the inner λ. By the second condition, the types of the first $@^0$ and the $@^1$ have to be equal.
- Ambiguous expressions such as $\lambda x.z(\lambda x.x)$ (the variable x is bound by the first or the second λ?) do not appear, because we have to make a choice between $\lambda.z(\lambda.@^0)$ and $\lambda.z(\lambda.@^1)$, which is unambiguous.
- The application of a λ-expression on an argument and working out the result is called "β-reducing" the term. We will then have to keep track of the number of encountered λ's while descending the syntax-tree, in order to know which bound variable has to be changed and which not. For example, we have $(\lambda.z@^0(\lambda.@^0@^1))x = zx(\lambda.@^0x)$. The reduction is represented by the thick arrow.



- Traditionally, one encounters problems when instantiating terms. If we want to instantiate x for y in the expression $\lambda x.zxy$, we get the naive and wrong answer $\lambda x.zxx$. Using the new indices we have to instantiate x for y in the expression $\lambda.z@^0y$, which results naturally in $\lambda.z@^0x$. This answer is equivalent to the translation of the traditional $\lambda v.zvx$, in which the name of the bound variable had to be changed in a new, unused variable-name, before the act of instantiation.

We end this section with a few remarks.

- Two free variables with the same name but with different types are, of course, different. However, we avoid the repeated use of names for variables, to prevent confusion. We also often don't write the types, to improve readability, as already remarked.
- Function-applications are assumed to be left-associative. This way, we write down $t_1 t_2 \dots t_{n-1} t_n$ as shorthand for $(\dots(t_1 t_2)\dots t_{n-1})t_n$.
- A term is called monomorphic resp. polymorphic if it does contain no resp. at least one type-variable, analogously to monomorphic resp. polymorphic types (cf. supra). Remark that a term t_σ can be polymorphic, even when σ is monomorphic, for example $(f_{\alpha\to bool} x_\alpha)_{bool}$, where *bool* is the monomorphic type of booleans. We define the set of type-variables which appear in a term t recursively and write it down as $tyvars(t)$.
- We define the set of free term-variables which appear in a term t recursively and write it down as $tevars(t)$. We call a term closed if there are no free term-variables in the term.

2.3.2 Term semantics

Assume Σ_Ω to be a signature for a type-structure Ω. A "model $\mathcal{M}$ for Σ_Ω" is specified by a model for the type-structure Ω, and by choosing, for every term-constant $(c,\sigma) \in \Sigma_\Omega$, an element $\mathcal{M}(c,\sigma) \in \prod_{Xs\in\mathcal{U}^n} [\![\sigma]\!]_\mathcal{M}(Xs)$. Here we consider the Cartesian product of dimension n, where n is the number of different type-variables in σ. This means that we can consider $M(c,\sigma)$ as a function which lets correspond, with every $Xs \in \mathcal{U}^n$, an element in $[\![\sigma]\!]_\mathcal{M}(Xs)$. In the case that $n = 0$ one simply chooses an element in $[\![\sigma]\!]_\mathcal{M}()$.

We now define a "context" $\alpha s, xs$ to be a type-context αs, combined with a finite, possibly empty list xs of different free term-variables. A "term-in-context" $\alpha s, xs.t$ is then a context, together with a term t, such that all type-variables in xs or in t appear in the list αs, and all free term-variables in t appear in the list xs. The context $\alpha s, xs$ may contain type-variables and/or free term-variables which do not appear in t. If $xs = x_1, \dots, x_m$ we can write down $\alpha s, xs.t$ as $\alpha s, x_1, \dots, x_m.t$.

We assume a certain total ordering on the set TeVars, and thus there exists, for every term t, a "canonical context", which is minimal for αs and xs, and in which the type- and free term-variables appear ordered.

We now define the semantics of terms. Assume that $\alpha s, xs.t$ is a term-in-context for Σ_Ω. Assume that t has the type τ, and assume that $\alpha s = \alpha_1, \dots, \alpha_n$ and also $xs = x_1, \dots, x_m$, in which every x_i has the type σ_i.

Because $\alpha s, xs.t$ is a term-in-context, both $\alpha s.\tau$ and all $\alpha s.\sigma_i$ are types-in-context. Thus we get the $\mathcal{U}^n \to \mathcal{U}$ functions $[\![\alpha s.\tau]\!]_\mathcal{M}$ and $[\![\alpha s.\sigma_i]\!]_\mathcal{M}$. The interpretation of $\alpha s, xs.t$ for Σ_Ω is then given by an element $[\![\alpha s, xs.t]\!]_\mathcal{M} \in \prod_{Xs\in\mathcal{U}^n}(\prod_{i=1}^m [\![\alpha s.\sigma_i]\!]_\mathcal{M}(Xs)) \to [\![\alpha s.\tau]\!]_\mathcal{M}(Xs)$. Thus, given $Xs = (X_1, \dots, X_n) \in \mathcal{U}^n$ and $ys = (y_1, \dots, y_m) \in [\![\alpha s.\sigma_1]\!]_\mathcal{M}(Xs) \times \dots \times [\![\alpha s.\sigma_m]\!]_\mathcal{M}(Xs)$ we get an element $[\![\alpha s, xs.t]\!]_\mathcal{M}(Xs)(ys)$ in $[\![\alpha s.\tau]\!]_\mathcal{M}(Xs)$.

We now define $[\![\alpha s, xs.t]\!]_\mathcal{M}$ by induction on the structure of t as follows.

- If t is a free variable, then $t = x_j$ for a certain $j \in \{1, \ldots, m\}$. We then define $[\![\alpha s, xs.t]\!](Xs)(ys)$ as y_j.
- If t is a constant $c_{\sigma'}$, with $(c, \sigma) \in \Sigma_\Omega$ and σ' an instantiation of σ, then $\sigma' = \sigma[\beta_1, \ldots, \beta_p/\tau_1, \ldots, \tau_p]$ for certain types $\tau_1, \ldots, \tau_p$, with $\beta_1, \ldots, \beta_p$ the type-variables which appear in σ. Define $[\![\alpha s, xs.t]\!](Xs)(ys)$ as $M(c, \sigma)([\![\alpha s.\tau_1]\!](Xs), \ldots, [\![\alpha s.\tau_p]\!](Xs))$.
- Assume t is a function-application $t_1 t_2$, such that t_1 has the type $\tau' \to \tau$ and t_2 has the type τ'. Then $[\![\alpha s, xs.t_1]\!](Xs)(ys)$ is an element in $[\![\alpha s.\tau' \to \tau]\!](Xs)$, and thus a function which maps elements in the set $[\![\alpha s.\tau']\!](Xs)$ to elements in the set $[\![\alpha s.\tau]\!](Xs)$. We apply this function to the element $[\![\alpha s, xs.t_2]\!](Xs)(ys)$ and define this way $[\![\alpha s, xs.t]\!](Xs)(ys)$ as $([\![\alpha s, xs.t_1]\!](Xs)(ys))([\![\alpha s, xs.t_2]\!](Xs)(ys))$.
- Assume t is a λ-abstraction $\lambda.t'$, such that the abstracted variable has the type τ_1 and t' has the type τ_2. Choose a new free variable-name x, which does not appear in xs. Suppose $\tau = \tau_1 \to \tau_2$, then $[\![\alpha s.\tau]\!](Xs)$ is the set of functions from $[\![\alpha s.\tau_1]\!](Xs)$ to $[\![\alpha s.\tau_2]\!](Xs)$. Now define $[\![\alpha s, xs.t]\!](Xs)(ys)$ as the function in this set which maps $y \in [\![\alpha s.\tau_1]\!](Xs)$ to $[\![\alpha s, xs, x.t'[@^i/x]]\!](Xs)(ys, y)$. Some notational clarification concerning $@^i$ has to be given now. The variables, bound by the outer λ of t, can appear on different levels in the term t', and thus can have different indices. We therefore have to refer to these variables using $@^i$. The expression $t'[@^i/x]$ then denotes the instantiation of these bound variables $@^i$ by x in t'.

Remark that in the above case-analysis only the syntactical classes of terms were considered. We do not have to consider bound variables, as these are not terms, only preterms.

Now define the "interpretation" of a term t_τ in a model $\mathcal{M}$ as the function $[\![t_\tau]\!] \in \prod_{Xs \in \mathcal{U}^n} (\prod_{i=1}^m [\![\alpha s.\sigma_i]\!](Xs)) \to [\![\alpha s.\tau]\!](Xs)$, given by $[\![\alpha s, xs.t_\tau]\!]$, where $\alpha s, xs$ is the canonical context of t_τ. Here, n is the number of different type-variables in t_τ, and αs is the list of these type-variables. Also, m is the number of different free variables in t_τ, given by the list xs, and the σ_i are their respective types. Remark that the list αs, part of the canonical context of t, can be strictly greater than the canonical type-contexts of τ and/or the σ_i. Therefore we can't just write down $[\![\tau]\!]$ or $[\![\sigma_i]\!]$ in the above expression.

An example to illustrate this. To be unambiguous, we will write out all details. Suppose we want to interpret the term $(\lambda.@^0)x$. Then we have to start with $[\![\alpha, x_\alpha.(\lambda_\alpha.@^0_\alpha)_{\alpha\to\alpha} x_\alpha]\!](X)(y)$. We can rewrite this interpretation using the above rules as $([\![\alpha, x_\alpha.(\lambda_\alpha.@^0_\alpha)_{\alpha\to\alpha}]\!](X)(y))([\![\alpha, x_\alpha.x_\alpha]\!](X)(y))$. The λ-abstraction $[\![\alpha, x_\alpha.(\lambda_\alpha.@^0_\alpha)_{\alpha\to\alpha}]\!](X)(y)$ will map its argument $[\![\alpha, x_\alpha.x_\alpha]\!](X)(y)$ to $[\![\alpha, x_\alpha, z_\alpha.z_\alpha]\!](X)(y, [\![\alpha, x_\alpha.x_\alpha]\!](X)(y))$, which is equal to $[\![\alpha, x_\alpha.x_\alpha]\!](X)(y)$ itself. This is the function which maps two arguments $X \in \mathcal{U}$ and $y \in X$ to y. Thus, we have shown that the interpretation of $(\lambda.@^0)x$ is equal to the interpretation of x.

We conclude this section on term-semantics with a discussion of "term-instantiation". Suppose that t is a term with canonical context $\alpha s, xs$, where $\alpha s = \alpha_1, \ldots, \alpha_n$, and $xs = x_1, \ldots, x_m$. Suppose that for all $i = 1, \ldots, n$ the types-in-context $\alpha s'.\tau_i$ are given, and that for all $j = 1, \ldots, m$ the types of the free variables x_j are given by σ_j. If we instantiate, in the list xs, the types τ_i for the type-variables α_i, we get a

new list xs'. The types of the new variables x'_j in this new list, for all $j = 1, \ldots, m$, are given by $\sigma'_j = \sigma_j[\alpha s/\tau s]$. We consider here only instantiations which do not identify different free variables in the list xs after instantiation of the αs by the τs. Such can happen if two free variables have the same name but a different type, but such that these types become identical after instantiation. This condition is not really restrictive, because we can assume that the variables in xs all have a different name. Indeed, it is easy to rename the free variables, without bothering the semantics. The conditions imply that $\alpha s', xs'$ effectively is a context. We then get a new term-in-context $\alpha s', xs'.t'$ by instantiating the types τ_i for the α_i in t, which we write down as $t' = t[\alpha s/\tau s]$.

The interpretations of t and t' are related as follows: $[\![\alpha s', xs'.t']\!](Xs') = [\![t]\!]([\![\alpha s'.\tau_1]\!](Xs'), \ldots, [\![\alpha s'.\tau_n]\!](Xs'))$, where $Xs' \in \mathcal{U}^{n'}$ and the length of $\alpha s'$ is equal to n'. This can be proved by induction on the structure of t.

Now assume that t is a term with canonical context $\alpha s, xs$, where $\alpha s = \alpha_1, \ldots, \alpha_n$, and $xs = x_1, \ldots, x_m$. Suppose that for all $j = 1, \ldots, m$ the types of the free variables x_j are given by σ_j. Suppose that for all $j = 1, \ldots, m$ we have the terms-in-context $\alpha s, xs'.t_j$, with the type of t_j also σ_j. If we instantiate the terms t_j for the free variables x_j in t then we get a new term-in-context $\alpha s, xs'.t'$, which we write down as $t' = t[xs/ts]$.

The interpretations of t and t' are related as follows: $[\![\alpha s, xs'.t']\!](Xs)(ys') = [\![t]\!](Xs)([\![\alpha s, xs'.t_1]\!](Xs)(ys'), \ldots, [\![\alpha s, xs'.t_m]\!](Xs)(ys'))$, where $Xs \in \mathcal{U}^n$ and $ys' \in [\![\alpha s.\sigma'_1]\!] \times \ldots \times [\![\alpha s.\sigma'_{m'}]\!]$, where m' is the length of xs' and σ'_j is the type of x'_j. This can be proved by induction on the structure of t.

2.4 Models

At this point we have to put a certain structure in the model, if we want to use our logic later on. Some basic- and standard-types and -terms are needed therefore. Attention has to be paid to the atypical constants C^{EXT}, also C^{IND} and F^{IND}. The first one will be used to characterize whether or not two functions are equal. The last one is some kind of successor-function, which will have the initial constant C^{IND}. We quickly discuss these constants, as an extended explication for their introduction follows in a later section. They will allow us to define Higher-Order-Logic more elegant. In this section we will write down the equality of two elements $x \in X$ and $x' \in X$ as $x =^X_{\mathcal{U}} x'$, to avoid confusion.

2.4.1 Basic- and standard-type-structures and -models

A type-structure Ω is a "basic-type-structure" if the type of booleans $(bool, 0)$ and the function-type $(\rightarrow, 2)$ are available. If we also have the type of Individuals $(ind, 0)$ we call the structure "standard". A model $\mathcal{M}$ is called a "basic-model" for Ω if $\mathcal{M}(bool) = 2$ and $\mathcal{M}(\rightarrow) = \rightarrow$. We call a model "standard" for Ω if also $\mathcal{M}(ind) = I$. These are the so-called standard-interpretations of $bool$, $\rightarrow$ and ind. We always assume type-structures and their models to be standard, except when noted otherwise.

2.4.2 Basic- and standard-term-signatures and -models

A signature Σ_Ω is a "basic-term-signature" if the constant $=_{\alpha\to\alpha\to bool}$ is available. If we also have the constants $\varepsilon_{(\alpha\to bool)\to\alpha}$, $C^{EXT}_{(\alpha\to\beta)\to(\alpha\to\beta)\to\alpha}$, C^{IND}_{ind} and $F^{IND}_{ind\to ind}$ we call the structure "standard". A model $\mathcal{M}$ is called a "basic-model" for Σ_Ω if the following holds.

- The interpretation of $\mathcal{M}(=, \alpha \to \alpha \to bool) \in \prod_{X\in\mathcal{U}} X \to X \to 2$ is the function which returns for every $X \in \mathcal{U}$ the function $=^X_{\mathcal{U}}$.

We call a model "standard" for Σ_Ω if also the following holds.

- The interpretation of $\mathcal{M}(\varepsilon, (\alpha \to bool) \to \alpha) \in \prod_{X\in\mathcal{U}}(X \to 2) \to X$ is the function which returns for every $X \in \mathcal{U}$ the function $\varepsilon^X_{\mathcal{U}}$. The function $\varepsilon^X_{\mathcal{U}}$ takes a $f \in (X \to 2)$ as argument and is defined as follows.

$$\varepsilon^X_{\mathcal{U}} f = \begin{cases} Choice(X) & \text{if } f^{-1}\{1\} = \emptyset \\ Choice(f^{-1}\{1\}) & \text{if } f^{-1}\{1\} \neq \emptyset \end{cases}$$

  Here we defined $f^{-1}\{1\} = \{x \in X : fx =^2_{\mathcal{U}} 1\}$. Thus, if there exists an x which fulfills f, then the result is an x' which fulfills f. In the other case the result can be anything.
- We interpret $\mathcal{M}(C^{EXT}, (\alpha \to \beta) \to (\alpha \to \beta) \to \alpha) \in \prod_{X_1,X_2\in\mathcal{U}}(X_1 \to X_2) \to (X_1 \to X_2) \to X_1$ to be the function which, for every $X_1 \in \mathcal{U}$ and every $X_2 \in \mathcal{U}$, returns the function $ext^{X_1,X_2}_{\mathcal{U}}$. The function $ext^{X_1,X_2}_{\mathcal{U}}$ takes a $f \in (X_1 \to X_2)$ and a $g \in (X_1 \to X_2)$ as arguments and is defined as follows.

$$ext^{X_1,X_2}_{\mathcal{U}} fg = \begin{cases} Choice(X_1) & \text{if } h^{-1}\{0\} = \emptyset \\ Choice(h^{-1}\{0\}) & \text{if } h^{-1}\{0\} \neq \emptyset \end{cases}$$

  Here we defined $h^{-1}\{0\} = \{x \in X_1 : fx \neq^{X_2}_{\mathcal{U}} gx\}$. Thus, if there exists an x such that fx is not equal to gx, then the result is an x' such that fx' is not equal to gx'. In the other case the result can be anything.
- The interpretation of $\mathcal{M}(C^{IND}, ind) \in I$ is any element in $I \in \mathcal{U}$.
- Construct an infinite, ordered sequence of elements in I which starts with $\mathcal{M}(C^{IND}, ind)$, and in which all elements are different. This is possible, because I is assumed to be infinite, and thus we can always choose a new element, different from all previous ones. The interpretation of $\mathcal{M}(F^{IND}, ind \to ind) \in I \to I$ is the function which gives, when iterated on the argument $\mathcal{M}(C^{IND}, ind)$, exactly the above constructed infinite, ordered sequence of elements in I. If we abbreviate $\mathcal{M}(C^{IND}, ind)$ resp. $\mathcal{M}(F^{IND}, ind \to ind)$ by C resp. F, we thus write down the infinite, ordered sequence of elements in I, starting with C and not containing any element more than once, as $C, F(C), F(F(C)), F(F(F(C))), \ldots$

These are the so-called standard-interpretations of $=$, ε, C^{EXT}, C^{IND} and F^{IND}. We always assume term-signatures and their models to be standard, except when noted otherwise. From now on we simplify the notation $=^X_{\mathcal{U}}$ back to $=$.

3 Logic

This section describes the deduction-rules, axioms and extensional principles of Formath, which are at times quite different from the ones traditionally used. We build a formal logic, and show that the deductive system is sound for the set-theoretic semantics given in the previous section. We show that Formath is consistent.

3.1 Sequents

Given a signature Σ_Ω, we define a "formula" as a term of type *bool*. A "sequent" is then defined as a pair (Γ, Γ'), with Γ and Γ' two finite sets of formulas over Σ_Ω. We call Γ the "antecedent", and Γ' the "consequent". Traditionally one assumes that Γ' contains exactly one element. We deviate from this line of working, as we can define our deduction-rules and axioms more elegant this way.

Assume a model $\mathcal{M}$ for Σ_Ω, assume that $\Gamma = \{t_1, \ldots, t_p\}$ and that $\Gamma' = \{t'_1, \ldots, t'_{p'}\}$. Suppose that $\alpha s, xs$ is a context which contains all type-variables and all free term-variables appearing in $t_1, \ldots, t_p, t'_1, \ldots, t'_{p'}$. Assume that $\alpha s = \alpha_1, \ldots, \alpha_n$, and $xs = x_1, \ldots, x_m$. Also assume that the type of x_j is given by σ_j, for $j = 1, \ldots, m$. We say that $\mathcal{M}$ is a "model for (Γ, Γ')" and write down $\Gamma \models_\mathcal{M} \Gamma'$ if for every $Xs \in \mathcal{U}^n$ and every $ys \in \prod_{j=1}^m [\![\alpha s.\sigma_j]\!]_\mathcal{M}(Xs)$ the following holds. If $[\![\alpha s, xs.t_k]\!]_\mathcal{M}(Xs)(ys) = 1$ for all $k = 1, \ldots, p$ then there exists at least one $l \in \{1, \ldots, p'\}$ such that $[\![\alpha s, xs.t'_l]\!]_\mathcal{M}(Xs)(ys) = 1$. In the case that $p = 0$ we write $\models_\mathcal{M} \Gamma'$, which means that there exists a t'_l with $l \in \{1, \ldots, p'\}$ such that the function $[\![t'_l]\!]_\mathcal{M} \in \prod_{Xs \in \mathcal{U}^n} (\prod_{j=1}^m [\![\alpha s.\sigma_j]\!]_\mathcal{M}(Xs)) \to 2$ is constant with value $1 \in 2$.

3.2 Deduction-rules

A "deductive system" $\mathcal{D}$ is a set of pairs $(L, (\Gamma, \Gamma'))$ where L is a list of sequents, possibly empty, and (Γ, Γ') is also a sequent. A sequent (Γ, Γ') can be deduced from a set of sequents Δ using a deductive system $\mathcal{D}$ if and only if there exist sequents $(\Gamma_1, \Gamma'_1), \ldots, (\Gamma_n, \Gamma'_n)$ such that the following two conditions hold.

- $(\Gamma, \Gamma') = (\Gamma_n, \Gamma'_n)$
- For all i such that $1 \leq i \leq n$ either $(\Gamma_i, \Gamma'_i) \in \Delta$ or $(L_i, (\Gamma_i, \Gamma'_i)) \in \mathcal{D}$ holds for a certain set L_i of elements in $\Delta \cup \{(\Gamma_1, \Gamma'_1), \ldots, (\Gamma_{i-1}, \Gamma'_{i-1})\}$.

The sequence $(\Gamma_1, \Gamma'_1), \ldots, (\Gamma_n, \Gamma'_n)$ is a "proof" of (Γ, Γ') from Δ using $\mathcal{D}$. We write this down as $\Gamma \vdash_{\mathcal{D},\Delta} \Gamma'$, but often drop the subscripts $\mathcal{D}$ and Δ.

In practice a deductive system is often specified by giving a few schematic inference-rules, in the following form.

$$\frac{\Gamma_1 \vdash \Gamma'_1 \quad \ldots \quad \Gamma_n \vdash \Gamma'_n}{\Gamma \vdash \Gamma'}$$

The sequents above the line are called the "hypotheses" of the rule and the sequent below the line the "conclusion". In this notation meta-variables can appear, which stand for any term of the correct type. If we instantiate these meta-variables

by concrete terms we get a specific deduction-rule of our deductive system. Sometimes the use of such an inference-rule is constrained by expressing conditions on the possible meta-instantiations.

We now introduce the five deduction-rules of Formath. The set of rules we use is different from the sets traditionally used. We use the notation $\Gamma - \Gamma'$ for the set-theoretic difference of Γ and Γ'.

- Equality Introduction: $$\frac{\Gamma_1 \vdash \Gamma_2 \qquad \Gamma_3 \vdash \Gamma_4}{\Gamma_1 - \{p\}, \Gamma_3 - \{q\} \vdash p = q, \Gamma_2 - \{q\}, \Gamma_4 - \{p\}}$$
- Equality Elimination: $$\frac{\Gamma_1 \vdash \Gamma_2 \qquad \Gamma_3 \vdash \Gamma_4}{\Gamma_1, \Gamma_3 \vdash q, \Gamma_2 - \{p\}, \Gamma_4 - \{p = q\}}$$
- Type Instantiation: $$\frac{\Gamma_1 \vdash \Gamma_2}{\Gamma_1[\alpha/\sigma] \vdash \Gamma_2[\alpha/\sigma]}$$
- Term Instantiation: $$\frac{\Gamma_1 \vdash \Gamma_2}{\Gamma_1[x/t] \vdash \Gamma_2[x/t]}$$
- Beta-Reduction: $$\vdash (\lambda.t_1)t_2 = t_1[@^i/t_2]$$

Concerning the β-reduction, we remind about the remark we made about the notation $@^i$. If we apply the function $\lambda.t_1$ to the argument t_2, we have to instantiate some bound variables in t_1 by t_2. These bound variables can appear on different levels in the term t_1, and thus can have different indices. Again, we write very generally $@^i$ to refer to these bound variables. It is then clear that the above rule indeed specifies β-reduction.

3.3 Axioms

In this section we provide a short introduction to the axioms of Formath, as a large discussion is included in a later section. We distinguish between basic-axioms and standard-axioms. The first ones are the axioms, needed to define Higher-Order-Logic in se. The last ones are the axioms, needed to get a standard Hol-system.

3.3.1 Basic-axioms

The "basic-axioms" are the following.

- Assumption: $x \vdash x$
- Reflexivity: $\vdash x = x$
- Combination: $f = g, x = y \vdash fx = gy$

Remark that x, y, f and g are normal free variables, and not meta-variables, in contrast to, for example, p and q in the deduction-rules. In the Assumption-axiom, the type of x is *bool*, in the Reflexivity-axiom the type of x is α, and in the Combination-axiom the types of f and g resp. x and y are $\alpha \to \beta$ resp. α.

3.3.2 Standard-axioms

The "standard-axioms" are the following.

- Choice: $Px \vdash P(\varepsilon P)$
- Extensionality: $f(C^{EXT}fg) = g(C^{EXT}fg) \vdash f = g$
- Individuals$_1$: $F^{IND}x = F^{IND}y \vdash x = y$
- Individuals$_2$: $C^{IND} = F^{IND}x \vdash y$

In the Choice-axiom, we can be more accurate by writing $P_{\alpha\to bool}$, x_α and $\varepsilon_{(\alpha\to bool)\to\alpha}$. The Choice-axiom expresses the following. For all predicates P the term εP will fulfill P whenever there exists a term x which fulfills P.

In the Extensionality-axiom the free variables f and g are both of the type $\alpha \to \beta$. The constant C^{EXT} has the type $(\alpha \to \beta) \to (\alpha \to \beta) \to \alpha$. Applying C^{EXT} to f and g results in a term $C^{EXT}fg$. If there exists a term t such that ft is different from gt, then also $f(C^{EXT}fg)$ will be different from $g(C^{EXT}fg)$, because of the standard-interpretation of C^{EXT}. The Extensionality-axiom expresses, paradoxically, the following: two functions are equal if they have the same image for an argument for which they have a different image, if such an argument exists.

In the Individuals-axioms the constants F^{IND} and C^{IND} are of the respective types $ind \to ind$ and ind. The free variable x always has the type ind, but the free variable y in the first Individuals-axiom has the type ind, while in the second Individuals-axiom it has the type $bool$. The first Individuals-axiom expresses the following: two terms x and y are equal if their images after applying F^{IND} are equal. The second Individuals-axiom expresses the following: any formula y holds if there exists a term x, such that applying F^{IND} to it gives C^{IND}.

The standard-model is thus axiomatized rather exceptionally. A rationale for this set of axioms is given in a section below.

3.4 Theories

A "theory" $\mathcal{T}$ is a quadruple $\mathcal{T} = (\text{Struc}_\mathcal{T}, \text{Sig}_\mathcal{T}, \text{Axioms}_\mathcal{T}, \text{Theorems}_\mathcal{T})$. Here $\text{Struc}_\mathcal{T}$ is a type-structure, which we call the type-structure of $\mathcal{T}$, and $\text{Sig}_\mathcal{T}$ is a term-signature, which we call the term-signature of $\mathcal{T}$. The set of sequents $\text{Axioms}_\mathcal{T}$ is called the axioms of $\mathcal{T}$, and the set of sequents $\text{Theorems}_\mathcal{T}$ is called the theorems of $\mathcal{T}$. All elements in the set of theorems can be deduced from the set of axioms by the deductive system, or by theory-extensions (cf. infra).

We define $\text{Types}_\mathcal{T} = \text{Types}_{\text{Struc}_\mathcal{T}}$ and $\text{Terms}_\mathcal{T} = \text{Terms}_{\text{Sig}_\mathcal{T}}$ to be the set of constructible types using the type-structure of $\mathcal{T}$, and the set of constructible terms using the term-signature of $\mathcal{T}$.

A "model of a theory" is specified by a model for the underlying type-structure, term-signature, axioms and theorems.

3.4.1 Basic-theory

The "basic-theory" is the quadruple $(\{(bool, 0), (\to, 2)\}, \{(=, \alpha \to \alpha \to bool)\}, \{Assumption, Reflexivity, Combination\}, \{\})$.

3.4.2 Standard-theory

The "standard-theory" is the quadruple $(\{(bool, 0), (\rightarrow, 2), (ind, 0)\}, \{(=, \alpha \rightarrow \alpha \rightarrow bool), (\varepsilon, (\alpha \rightarrow bool) \rightarrow \alpha), (C^{EXT}, (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha), (F^{IND}, ind \rightarrow ind), (C^{IND}, ind)\}, \{Assumption, Reflexivity, Combination, Choice, Extensionality, Individuals_1, Individuals_2\}, \{\})$.

We always assume theories to be standard, except when noted otherwise.

3.5 Extensions

A theory $\mathcal{T}'$ is called a "theory-extension" of a theory $\mathcal{T}$ if the following conditions hold.

- $\text{Struc}_{\mathcal{T}} \subseteq \text{Struc}_{\mathcal{T}'}$
- $\text{Sig}_{\mathcal{T}} \subseteq \text{Sig}_{\mathcal{T}'}$
- $\text{Axioms}_{\mathcal{T}} = \text{Axioms}_{\mathcal{T}'}$
- $\text{Theorems}_{\mathcal{T}} \subseteq \text{Theorems}_{\mathcal{T}'}$

The extension of the type-structure and term-signature is discussed extensively later on. We only accept definitional extensions, which means that new types and terms can be introduced only by defining them using already constructed types and terms. The proof of consistency becomes very clear this way, without losing expressivity in the logic. Extending the set of axioms is not allowed, if we want to guarantee consistency. The only way the set of theorems can be extended is by deducing new sequents using the deduction-rules or by type-/term-extensions.

3.5.1 Type-structure-extension

It is useful to characterize new types using other already constructed types. A new type is defined as a subset of an already existing type, and that subset is specified by a predicate. To guarantee that the new type contains at least one element, we have to provide a so-called witness, which fulfills the predicate. Under certain conditions we can allow type-variables during this process.

We discuss the "type-extension" in greater detail. Assume a theory $\mathcal{T} = (\text{Struc}_{\mathcal{T}}, \text{Sig}_{\mathcal{T}}, \text{Axioms}_{\mathcal{T}}, \text{Theorems}_{\mathcal{T}})$, and assume a type σ, a new type-name ν and a list of type-variables $\alpha s = \alpha_1, \ldots, \alpha_n$. Assume a term-constant $P_{\sigma \rightarrow bool}$ and a term t_σ. Assume that the following conditions are fulfilled.

- $tyvars(\sigma) \subseteq \alpha s$
- $\vdash Pt \in \text{Theorems}_{\mathcal{T}}$

We now define two new term-constants which we will call $TO^{\nu}_{\sigma \rightarrow (\alpha_1, \ldots, \alpha_n)\nu}$ and $FROM^{\nu}_{(\alpha_1, \ldots, \alpha_n)\nu \rightarrow \sigma}$. These map elements from one type to elements in the other type. Consider the following two new sequents.

- Type Extension$^{\nu}_1$: $\vdash Px = (FROM^{\nu}(TO^{\nu}x) = x)$

- Type Extension$_2^\nu$: $\quad \vdash TO^\nu(FROM^\nu x) = x$

These sequents relate σ and $(\alpha_1, \ldots, \alpha_n)\nu$. The first theorem expresses that we can map elements from the old type to elements in the new type, and afterwards return to the old type. We will get the original element back, if and only if this element fulfills the predicate. The second theorem expresses that we can map an element in the new type to an element in the old type, and afterwards map this result back to the new type, always getting the original element back. The new theory can be written as $\mathcal{T}' = (\text{Struc}_\mathcal{T} \cup \{(\nu, n)\}, \text{Sig}_\mathcal{T} \cup \{(TO^\nu, \sigma \rightarrow (\alpha_1, \ldots, \alpha_n)\nu), (FROM^\nu, (\alpha_1, \ldots, \alpha_n)\nu \rightarrow \sigma)\}, \text{Axioms}_\mathcal{T}, \text{Theorems}_\mathcal{T} \cup \{TypeExtension_1^\nu, TypeExtension_2^\nu\})$.

3.5.2 Term-signature-extension

It is useful to characterize new constant terms using other already constructed terms. A new constant term is defined to be equal to another term. Under certain conditions we can allow type-variables during this process.

We discuss the "term-extension" in greater detail. Assume a theory $\mathcal{T} = (\text{Struc}_\mathcal{T}, \text{Sig}_\mathcal{T}, \text{Axioms}_\mathcal{T}, \text{Theorems}_\mathcal{T})$, and assume a type σ, a new term-name c and a term t_σ. Assume that the following conditions are fulfilled.

- $tyvars(t_\sigma) = tyvars(\sigma)$
- $tevars(t_\sigma) = \emptyset$

We then define the new term-constant c_σ. Consider the following new sequent.

- Term Extension^c: $\quad \vdash c = t$

This sequent relates c and t. The new theory is $\mathcal{T}' = (\text{Struc}_\mathcal{T}, \text{Sig}_\mathcal{T} \cup \{(c, \sigma)\}, \text{Axioms}_\mathcal{T}, \text{Theorems}_\mathcal{T} \cup \{TermExtension^c\})$.

3.6 Soundness and Consistency

A theory is called "consistent" if not every sequent can be deduced. The existence of a model of the theory is sufficient to get consistency. We thus start with the standard-model and the standard-theory, and extend these by deducing sequents and defining types/terms. We now have to show that these operations are "sound", which means that there will exist a model of the new theory, under condition that there existed a model of the old theory. It is of prime importance that we consider a standard-model, where type- and term-constants are interpreted standardly, as already discussed.

3.6.1 Deduction-rules

We start by showing that the deduction-rules are sound.

Theorem: The deduction-rules are sound.
Proof:

- Equality Introduction. We have to show that from $\Gamma_1 \models \Gamma_2$ and $\Gamma_3 \models \Gamma_4$ follows that $\Gamma_1 - \{p\}, \Gamma_3 - \{q\} \models p = q, \Gamma_2 - \{q\}, \Gamma_4 - \{p\}$. Take some model $\mathcal{M}$ such that for all $t \in (\Gamma_1 - \{p\}) \cup (\Gamma_3 - \{q\})$ holds that $[\![\alpha s, xs.t]\!](Xs)(ys) = 1$. We distinguish four cases.
  - It holds that $[\![\alpha s, xs.p]\!](Xs)(ys) = [\![\alpha s, xs.q]\!](Xs)(ys) = 1$. Then also $[\![\alpha s, xs.p = q]\!](Xs)(ys) = ([\![\alpha s, xs. =]\!](Xs)(ys))([\![\alpha s, xs.p]\!](Xs)(ys))([\![\alpha s, xs.q]\!](Xs)(ys)) = 1$, because the equality is interpreted standardly.
  - It holds that $[\![\alpha s, xs.p]\!](Xs)(ys) = [\![\alpha s, xs.q]\!](Xs)(ys) = 0$. Then also $[\![\alpha s, xs.p = q]\!](Xs)(ys) = ([\![\alpha s, xs. =]\!](Xs)(ys))([\![\alpha s, xs.p]\!](Xs)(ys))([\![\alpha s, xs.q]\!](Xs)(ys)) = 1$.
  - It holds that $[\![\alpha s, xs.p]\!](Xs)(ys) = 1$ and $[\![\alpha s, xs.q]\!](Xs)(ys) = 0$. Then for all $t \in \Gamma_1$ it also holds that $[\![\alpha s, xs.t]\!](Xs)(ys) = 1$. Thus there will exist a $t' \in \Gamma_2$ such that $[\![\alpha s, xs.t']\!](Xs)(ys) = 1$. Moreover, from $[\![\alpha s, xs.q]\!](Xs)(ys) = 0$ it follows that t' will be different from q and thus $t' \in \Gamma_2 - \{q\}$.
  - It holds that $[\![\alpha s, xs.p]\!](Xs)(ys) = 0$ and $[\![\alpha s, xs.q]\!](Xs)(ys) = 1$. Then for all $t \in \Gamma_3$ it also holds that $[\![\alpha s, xs.t]\!](Xs)(ys) = 1$. Thus there will exist a $t' \in \Gamma_4$ such that $[\![\alpha s, xs.t']\!](Xs)(ys) = 1$. Moreover, from $[\![\alpha s, xs.p]\!](Xs)(ys) = 0$ it follows that t' will be different from p and thus $t' \in \Gamma_4 - \{p\}$.

- Equality Elimination. We have to show that from $\Gamma_1 \models \Gamma_2$ and $\Gamma_3 \models \Gamma_4$ follows that $\Gamma_1, \Gamma_3 \models q, \Gamma_2 - \{p\}, \Gamma_4 - \{p = q\}$. Take some model $\mathcal{M}$ such that for all $t \in \Gamma_1 \cup \Gamma_3$ it holds that $[\![\alpha s, xs.t]\!](Xs)(ys) = 1$. Then there will exist a $t' \in \Gamma_2$ and a $t'' \in \Gamma_4$ such that $[\![\alpha s, xs.t']\!](Xs)(ys) = [\![\alpha s, xs.t'']\!](Xs)(ys) = 1$. If there does not exist a $t''' \in (\Gamma_2 - \{p\}) \cup (\Gamma_4 - \{p = q\})$ such that $[\![\alpha s, xs.t''']\!](Xs)(ys) = 1$ then t' has to be equal to p and t'' has to be equal to $p = q$, and thus $[\![\alpha s, xs.p]\!](Xs)(ys) = [\![\alpha s, xs.p = q]\!](Xs)(ys) = 1$. Thus follows $[\![\alpha s, xs.q]\!](Xs)(ys) = 1$.
- Type Instantiation. We prove the more general result that from $\Gamma_1 \models \Gamma_2$ it follows that $\Gamma_1[\alpha s/\tau s] \models \Gamma_2[\alpha s/\tau s]$. Take some model $\mathcal{M}$ such that for all $t' \in \Gamma_1[\alpha s/\tau s]$ it holds that $[\![\alpha s', xs'.t']\!](Xs')(ys') = 1$. These t' are related to the corresponding $t \in \Gamma_1$ as follows: $[\![\alpha s', xs'.t']\!](Xs')(ys') = [\![t]\!]([\![\alpha s'.\tau_1]\!](Xs'), \ldots, [\![\alpha s'.\tau_n]\!](Xs'))(ys') = 1$. Thus there exists a $t'' \in \Gamma_2$ such that $[\![t'']\!]([\![\alpha s'.\tau_1]\!](Xs'), \ldots, [\![\alpha s'.\tau_n]\!](Xs'))(ys') = [\![\alpha s', xs'.t''']\!](Xs')(ys') = 1$ for the corresponding $t''' \in \Gamma_2[\alpha s/\tau s]$.
- Term Instantiation. We prove the more general result that from $\Gamma_1 \models \Gamma_2$ it follows that $\Gamma_1[xs/ts] \models \Gamma_2[xs/ts]$. Take some model $\mathcal{M}$ such that for all $t' \in \Gamma_1[xs/ts]$ it holds that $[\![\alpha s, xs'.t']\!](Xs)(ys') = 1$. These formulas t' are related to the corresponding formulas $t \in \Gamma_1$ in the following way:

$[\![\alpha s, xs'.t']\!](Xs)(ys') = [\![t]\!](Xs)([\![\alpha s, xs'.t_1]\!](Xs)(ys'), \ldots, [\![\alpha s, xs'.t_m]\!](Xs)(ys')) = 1$. Thus there exists a $t'' \in \Gamma_2$ such that $[\![t'']\!](Xs)([\![\alpha s, xs'.t_1]\!](Xs)(ys'), \ldots, [\![\alpha s, xs'.t_m]\!](Xs)(ys')) = [\![\alpha s, xs'.t'''']\!](Xs)(ys') = 1$ for the corresponding $t''' \in \Gamma_2[xs/ts]$.

- Beta-Reduction. We show that $\models (\lambda.t_1)t_2 = t_1[@^i/t_2]$. Take some model $\mathcal{M}$, then there holds $[\![\alpha s, xs.(\lambda.t_1)t_2]\!](Xs)(ys) = ([\![\alpha s, xs.(\lambda.t_1)]\!](Xs)(ys))([\![\alpha s, xs.t_2]\!](Xs)(ys))$. Because λ-abstractions are interpreted standardly we can rewrite this as $[\![\alpha s, xs, x.t_1[@^i/x]]\!](Xs)(ys, [\![\alpha s, xs.t_2]\!](Xs)(ys)) = [\![\alpha s, xs.t_1[@^i/t_2]]\!](Xs)(ys)$. Thus $[\![\alpha s, xs.(\lambda.t_1)t_2 = t_1[@^i/t_2]]\!](Xs)(ys) = 1$.

3.6.2 Axioms

We also have to show that the basic-axioms are fulfilled in the basic-model.

Theorem: The basic-axioms are sound.
Proof:

- Assumption. We have to show that $x \models x$. Take some model $\mathcal{M}$ such that $[\![\alpha s, xs.x]\!](Xs)(ys) = 1$, then $[\![\alpha s, xs.x]\!](Xs)(ys) = 1$ trivially holds.
- Reflexivity. We have to show that $\models x = x$. Take some model $\mathcal{M}$, then it trivially holds that $[\![\alpha s, xs.x = x]\!](Xs)(ys) = 1$.
- Combination. We have to show that $f = g, x = y \models fx = gy$. Take some model $\mathcal{M}$ such that $[\![\alpha s, xs.f = g]\!](Xs)(ys) = [\![\alpha s, xs.x = y]\!](Xs)(ys) = 1$. Then also $[\![\alpha s, xs.f]\!](Xs)(ys) = [\![\alpha s, xs.g]\!](Xs)(ys)$ and $[\![\alpha s, xs.x]\!](Xs)(ys) = [\![\alpha s, xs.y]\!](Xs)(ys)$. Thus $([\![\alpha s, xs.f]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys)) = ([\![\alpha s, xs.g]\!](Xs)(ys))([\![\alpha s, xs.y]\!](Xs)(ys))$. Thus $[\![\alpha s, xs.fx]\!](Xs)(ys) = [\![\alpha s, xs.gy]\!](Xs)(ys)$. Thus $[\![\alpha s, xs.fx = gy]\!](Xs)(ys) = 1$.

We also have to show that the standard-axioms are fulfilled in the standard-model.

Theorem: The standard-axioms are sound.
Proof:

- Choice. We have to show that $Px \models P(\varepsilon P)$. Take some model $\mathcal{M}$ such that $[\![\alpha s, xs.Px]\!](Xs)(ys) = 1$. This means that there exists an element $[\![\alpha s, xs.x]\!](Xs)(ys)$ that fulfills the predicate $[\![\alpha s, xs.P]\!](Xs)(ys)$. Thus $[\![\alpha s, xs.\varepsilon P]\!](Xs)(ys)$ is an element that fulfills the predicate $[\![\alpha s, xs.P]\!](Xs)(ys)$, because ε is interpreted standardly. It follows that $[\![\alpha s, xs.P(\varepsilon P)]\!](Xs)(ys) = 1$.
- Extensionality. We have to show that $f(C^{EXT}fg) = g(C^{EXT}fg) \models f = g$. Take some model $\mathcal{M}$ for the antecedent $[\![\alpha s, xs.f(C^{EXT}fg) = g(C^{EXT}fg)]\!](Xs)(ys) = 1$. Assume there exists a $[\![\alpha s, xs.x]\!](Xs)(ys)$ for which it holds that $([\![\alpha s, xs.f]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys)) \neq ([\![\alpha s, xs.g]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys))$. Then $[\![\alpha s, xs.C^{EXT}fg]\!](Xs)(ys)$ also has to be such an element, because C^{EXT} is interpreted standardly. Thus follows that $([\![\alpha s, xs.f]\!](Xs)(ys))([\![\alpha s, xs.C^{EXT}fg]\!](Xs)(ys)) \neq ([\![\alpha s, xs.g]\!](Xs)(ys))([\![\alpha s, xs.C^{EXT}fg]\!](Xs)(ys))$,

and thus also $[\![\alpha s, xs.f(C^{EXT}fg) = g(C^{EXT}fg)]\!](Xs)(ys) = 0$, which is contradictory. Thus there does not exist a $[\![\alpha s, xs.x]\!](Xs)(ys)$ for which $([\![\alpha s, xs.f]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys)) \neq ([\![\alpha s, xs.g]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys))$, thus $[\![\alpha s, xs.f]\!](Xs)(ys) = [\![\alpha s, xs.g]\!](Xs)(ys)$. It follows that $[\![\alpha s, xs.f = g]\!](Xs)(ys) = 1$.

- Individuals$_1$. We have to show that $F^{IND}x = F^{IND}y \models x = y$. Take some model $\mathcal{M}$ such that $[\![\alpha s, xs.F^{IND}x = F^{IND}y]\!](Xs)(ys) = 1$. Then holds that $[\![\alpha s, xs.F^{IND}x]\!](Xs)(ys) = [\![\alpha s, xs.F^{IND}y]\!](Xs)(ys)$, and thus $([\![\alpha s, xs.F^{IND}]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys)) = ([\![\alpha s, xs.F^{IND}]\!](Xs)(ys))([\![\alpha s, xs.y]\!](Xs)(ys))$, thus also $[\![\alpha s, xs.x]\!](Xs)(ys) = [\![\alpha s, xs.y]\!](Xs)(ys)$, because F^{IND} is interpreted standardly. It follows that $[\![\alpha s, xs.x = y]\!](Xs)(ys) = 1$.
- Individuals$_2$. We have to show that $C^{IND} = F^{IND}x \models y$. Take some model $\mathcal{M}$ such that $[\![\alpha s, xs.C^{IND} = F^{IND}x]\!](Xs)(ys) = 1$. Thus follows $[\![\alpha s, xs.C^{IND}]\!](Xs)(ys) = ([\![\alpha s, xs.F^{IND}]\!](Xs)(ys))([\![\alpha s, xs.x]\!](Xs)(ys))$, which is contradictory, because C^{IND} is interpreted standardly. Thus there does not exist a model $\mathcal{M}$ for which $[\![\alpha s, xs.C^{IND} = F^{IND}x]\!](Xs)(ys) = 1$. For all these non-existent models of course also $[\![\alpha s, xs.y]\!](Xs)(ys) = 1$.

3.6.3 Extensions

We also have to show that the type- and term-extensions are sound.

Theorem: The extensions are sound.
Proof:

- Type-extension. We have to show that the theory $\mathcal{T}'$ is consistent. Take some model $\mathcal{M}$ of the theory $\mathcal{T}$. We construct a model $\mathcal{M}'$ of the new theory by interpreting the new type- and term-constants in the universe.

  We have $\vdash Pt \in \text{Theorems}_{\mathcal{T}}$. Thus the subset of $[\![\alpha s.\sigma]\!](Xs)$ which is defined by $[\![\alpha s.P]\!](Xs)$ is not empty, and thus it exists in the universe $\mathcal{U}$. We interpret $[\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$ as this subset. This is possible because $tyvars(\sigma) \subseteq \alpha s$.

  Consider the set of functions which map elements in $[\![\alpha s.\sigma]\!](Xs)$ to elements in $[\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$. Consider the functions in this set which map all elements in the subset of $[\![\alpha s.\sigma]\!](Xs)$ defined by $[\![\alpha s.P]\!](Xs)$ canonically to $[\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$. We interpret TO^{ν} as any such function. We interpret $FROM^{\nu}$ as the function which maps all elements in the set $[\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$ canonically to $[\![\alpha s.\sigma]\!](Xs)$.

  We now prove the two additional theorems about the constants TO^{ν} and $FROM^{\nu}$. Take some element $[\![\alpha s, xs.x]\!](Xs)(ys) \in [\![\alpha s.\sigma]\!](Xs)$. The function $[\![\alpha s.TO^{\nu}]\!](Xs)$ will map this element canonically to the element $[\![\alpha s, xs.TO^{\nu}x]\!](Xs)(ys) \in [\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$ if and only if the element fulfills the predicate $[\![\alpha s.P]\!](Xs)$, i.e. $[\![\alpha s, xs.Px]\!](Xs)(ys) = 1$. The function $[\![\alpha s.FROM^{\nu}]\!](Xs)$ maps $[\![\alpha s, xs.TO^{\nu}x]\!](Xs)(ys)$ canonically to the element $[\![\alpha s, xs.FROM^{\nu}(TO^{\nu}x)]\!](Xs)(ys) \in [\![\alpha s.\sigma]\!](Xs)$. Thus follows that $[\![\alpha s, xs.FROM^{\nu}(TO^{\nu}x)]\!](Xs)(ys) = [\![\alpha s, xs.x]\!](Xs)(ys)$ if and only if it holds that

$[\![\alpha s, xs.Px]\!](Xs)(ys) = 1$. Thus it holds that $[\![\alpha s, xs.Px = (FROM^\nu(TO^\nu x) = x)]\!](Xs)(ys) = 1$.

Now take some element $[\![\alpha s, xs.x]\!](Xs)(ys) \in [\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$. The function $[\![\alpha s.FROM^\nu]\!](Xs)$ maps this element canonically to the element $[\![\alpha s, xs.FROM^\nu x]\!](Xs)(ys) \in [\![\alpha s.\sigma]\!](Xs)$. This element of course fulfills the predicate $[\![\alpha s.P]\!](Xs)$, i.e. $[\![\alpha s, xs.P(FROM^\nu x)]\!](Xs)(ys) = 1$. Thus the function $[\![\alpha s.TO^\nu]\!](Xs)$ maps this element canonically to the element $[\![\alpha s, xs.TO^\nu(FROM^\nu x)]\!](Xs)(ys) \in [\![\alpha s.(\alpha_1, \ldots, \alpha_n)\nu]\!](Xs)$.Thus it holds that $[\![\alpha s, xs.TO^\nu(FROM^\nu x)]\!](Xs)(ys) = [\![\alpha s, xs.x]\!](Xs)(ys)$. Thus follows that $[\![\alpha s, xs.TO^\nu(FROM^\nu x) = x]\!](Xs)(ys) = 1$.

- Term-extension. We have to show that the theory $\mathcal{T}'$ is consistent. Take some model $\mathcal{M}$ of the theory $\mathcal{T}$. We construct a model $\mathcal{M}'$ of the new theory by interpreting the new term-constant c as $\mathcal{M}'(c, \sigma) = [\![t]\!]$. This is possible because $tyvars(t_\sigma) = tyvars(\sigma)$ and $tevars(t_\sigma) = \emptyset$. Thus $[\![\alpha s, xs.c = t]\!](Xs)(ys) = 1$.

We thus get that the Formath-logic is sound, and the deduced theories are consistent.

4 Discussion

In this section we discuss a few traditional systems for Higher-Order-Logic. We compare these with Formath, at the same time highlighting both positive and negative aspects of the new rules and axioms. In general, we can say that Formath was not meant to be a competitor to other systems, but rather an alternative formalization of Higher-Order-Logic: we tried to rework Hol to obtain a clean and beautiful definition which is appealing to the mathematician. As such, we hope that the nice structure of our formalization will attract the reader's attention. As will become clear below, main principles while designing Formath were orthogonality and symmetry of rules and axioms, cleanliness of code, a very small core, and equiconsistency with the other main-stream Hol distributions.

To ensure readability and a consistent notation for all systems, we make use of the traditional way of writing terms, i.e. without the syntactical distinction between free and bound variables.

4.1 Hol-4 and Hol-Light

We inspect the systems Hol-4 and Hol-Light in depth, to get insight in the structure of Higher-Order-Logic. Hol-4 is the best-known Hol-implementation. Hol-Light is a reworked version of Hol-4.

4.1.1 Definition of Hol-4

The tool Hol-4 [16] is the latest version in a series of automatized proofsystems for Higher-Order-Logic. In this environment we can deduce theorems and declare proof-strategies. Lots of built-in procedures can be used to prove automatically simple theorems. The tool uses traditional λ-calculus: there is no syntactical distinction between free and bound variables. Consequents contain exactly one formula. The environment is developed in Moscow-ML [25].

Deduction-rules
Hol-4 uses the following deduction-rules.

- Disch: $$\frac{\Gamma \vdash q}{\Gamma - \{p\} \vdash p \Rightarrow q}$$
- Mp: $$\frac{\Gamma_1 \vdash p \qquad \Gamma_2 \vdash p \Rightarrow q}{\Gamma_1, \Gamma_2 \vdash q}$$
- Inst_Type: $$\frac{\Gamma \vdash p}{\Gamma \vdash p[\alpha s / \sigma s]}$$
  None of the $\alpha_i \in \alpha s$ may occur in Γ, and we possibly have to rename some variables in p, to avoid identifying distinct variables after instantiation.
- Subst: $$\frac{\Gamma_1 \vdash t_1 = t'_1 \quad \ldots \quad \Gamma_n \vdash t_n = t'_n \qquad \Gamma \vdash p[t_1, \ldots, t_n]}{\Gamma_1, \ldots, \Gamma_n, \Gamma \vdash p[t'_1, \ldots, t'_n]}$$
  The term $p[t_1, \ldots, t_n]$ is a formula p in which we choose a few occurrences of $t_1, \ldots, t_n$. The term $p[t'_1, \ldots, t'_n]$ is the formula p in which the chosen occurrences of $t_1, \ldots, t_n$ are replaced by $t'_1, \ldots, t'_n$. We possibly have to rename some bound variables in p, to avoid that some free variables in $t'_1, \ldots, t'_n$ suddenly get bound after substitution.
- Beta_Conv: $$\vdash (\lambda x.t_1)t_2 = t_1[x/t_2]$$
  We possibly have to rename some bound variables in t_1, to avoid that some free variables in t_2 suddenly get bound after substitution.
- Assume: $$p \vdash p$$
- Refl: $$\vdash t = t$$
- Abs: $$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$$
  Here the variable x may not occur free in Γ.

The actual implementation of the system includes other deduction-rules as well.

Axioms
Hol-4 uses the following axioms.

- Select_Ax: $\vdash \forall Px.Px \Rightarrow P(\varepsilon P)$
- Eta_Ax: $\vdash \forall f.(\lambda x.fx) = f$
- Infinity_Ax: $\vdash \exists f.One\_One f \wedge \neg Onto f$
- Imp_Antisym_Ax: $\vdash \forall pq.(p \Rightarrow q) \Rightarrow (q \Rightarrow p) \Rightarrow (p = q)$
- Bool_Cases_Ax: $\vdash \forall p.(p = T) \vee (p = F)$

One has, since introducing this logic, shown that the axiom Imp_Antisym_Ax is dependent from the other parts of the system, and in fact this axiom is now proved in the Hol-4 system. The logical operators have their usual interpretation, and *One_One* resp. *Onto* are predicates expressing injectivity resp. surjectivity of functions.

Extensions
Hol-4 permits theory-extensions by type-definition, term-definition and term-specification. The first two are analogous to the permitted extensions in Formath. Term-specification permits the introduction of new term-constants by specifying properties of these new constants.

4.1.2 Definition of Hol-Light

The tool Hol-Light [18] is an entirely new reimplementation of Higher-Order-Logic. In comparison with the other systems for Hol we can mention that Hol-Light's core is cleaner, but has equal power. The tool also uses traditional λ-calculus and consequents also contain exactly one formula. The version 1.0 of Hol-Light, which we used, is developed in Caml-Light [11]. Currently, a new and improved version 2.20 is available.

Deduction-rules
Hol-Light uses the following deduction-rules.

- Deduct_Antisym_Rule: $\dfrac{\Gamma_1 \vdash p \qquad \Gamma_2 \vdash q}{\Gamma_1 - \{q\}, \Gamma_2 - \{p\} \vdash p = q}$
- Eq_Mp: $\dfrac{\Gamma_1 \vdash p \qquad \Gamma_2 \vdash p = q}{\Gamma_1, \Gamma_2 \vdash q}$
- Inst_Type: $\dfrac{\Gamma \vdash p}{\Gamma[\alpha s/\sigma s] \vdash p[\alpha s/\sigma s]}$
- Inst: $\dfrac{\Gamma \vdash p}{\Gamma[xs/ts] \vdash p[xs/ts]}$
- Beta: $\vdash (\lambda x.t)x = t$
- Assume: $p \vdash p$
- Refl: $\vdash t = t$
- Mk_Comb: $\dfrac{\Gamma_1 \vdash f = g \qquad \Gamma_2 \vdash x = y}{\Gamma_1, \Gamma_2 \vdash fx = gy}$
- Abs: $\dfrac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$
  Here the variable x may not occur free in Γ.
- Trans: $\dfrac{\Gamma_1 \vdash t_1 = t_2 \qquad \Gamma_2 \vdash t_2 = t_3}{\Gamma_1, \Gamma_2 \vdash t_1 = t_3}$

It is known that the rule Trans is dependent from the other parts of the system [14], but the system slows down by $\pm 10\%$ without it, which explains its inclusion.

Axioms
Hol-Light uses the following axioms.

- Select_Ax: $\vdash \forall Px.Px \Rightarrow P(\varepsilon P)$
- Eta_Ax: $\vdash \forall f.(\lambda x.fx) = f$
- Infinity_Ax: $\vdash \exists f.One\_One f \wedge \neg Onto f$

Extensions
Hol-Light permits theory-extensions by type- and term-definition. These are analogous to the permitted extensions in Formath.

4.1.3 Analysis of Hol-4, Hol-Light and Formath

After having described the different systems, we now examine them in more detail. We developed the insights below by experimenting with lots of different sets of rules and axioms, to learn which are dependent from others and to get an idea of their expressivity. Some ideas developed from corresponding with J. Harrison [14], the designer of Hol-Light, and L. Théry [28].

- The usual systems for Higher-Order-Logic do not make a syntactical distinction between free and bound term-variables. Consequently, one has to check whether free variables do not suddenly get bound during instantiation or substitution. Moreover, in any implementation these checks have to be built in the core of the logic system, which is to be avoided. In Formath, we make a syntactical distinction between free and bound variables. Let us have a closer look at this decision.

  The problem of free-variable-capture is well-known, although it has taken a long time before it was understood how it could be tackled. Most formal languages do not make a syntactical distinction between free and bound variables, which sometimes leads to awkward situations and misleading formulas. "Famous logicians have made embarrassing errors here" [1]. Some people suggested using "de Bruijn-indices", which are in this context an implementation issue: they are covered by code. Working internally with pointers or object-references can be more efficient than performing the necessary syntactical checks. However, this solution is not entirely satisfactory because pure "de Bruijn-indices" often make life difficult while implementing them.

  To get rid of the problem of free-variable-capture, without having to implement traditional "de Bruijn-indices", we decided to make a syntactical distinction between free and bound variables using explicit indices. We consider this a handy way of avoiding any problems during instantiation or substitution. For example, to type-check a term using a recursive descent it is sufficient to maintain a simple list (containing the types of the bound variables), and to β-reduce a term all one needs is a simple counter (to count the number of λ-abstractions). In our opinion, the implementation was quite straightforward; at least easier than implementing pure "de Bruijn-indices", or the necessary syntactical checks during instantiation or substitution using the conventional notation.

Debates about the (dis-)advantages of the respective approaches often rely on subjective arguments, as the recent comparison by S. Berghofer and C. Urban [10] shows. They conclude that the merits of the different approaches depend on which goal one pursues. Another study was done by F. Kamareddine and A. Rios [23].

Concluding, we can say that both approaches have arguments pro and contra. Moreover, the choice made does not influence the power of the logic in se at all. In our opinion, it is a matter of personal taste. In fact, the first versions of Formath used the conventional style, but at some point we decided to take the current approach.

- There exist lots of different calculi for logic systems, such as natural-deduction-style, sequent-style and tableaux-style. In Formath, we opt for a sequent-style using several consequent-formulas. This is an extension of the traditional way of working, which is quite intuitive compared to permitting only a single consequent-formula. It integrates easily in a prover for building proofs backwards, and also provides a nice symmetry between antecedent and consequent. Our deduction-rules are based on traditional rules, but tuned to the new sequent-style (e.g. Equality Introduction or Equality Elimination).
- In Hol-4, the link between antecedent- and consequent-formulas follows from Disch and Mp, using implication. This term-constant is added to the system as a basic constant. The corresponding rules in Hol-Light resp. Formath, namely Deduct_Antisym_Rule and Eq_Mp resp. Equality Introduction and Equality Elimination use equality instead of implication, nicely achieving the same expressivity. Note that Imp_Antisym_Ax from Hol-4, Deduct_Antisym_Rule from Hol-Light and Equality Introduction from Formath express the same idea.
- Type- and term-instantiations using Inst_Type and Inst allow to instantiate several different variables simultaneously, which might complicate the code for these rules. We allow only single instantiations in the Formath-rules Type Instantiation and Term Instantiation. This slows down the process of checking theorems, but speed is not of prime importance to us if we can achieve simpler code.
- The rule Subst in Hol-4 is quite complicated, and is, in our opinion, too complex for being a basic rule. Both Hol-Light and Formath use instantiation instead of substitution, which appears cleaner because term-variables are really schematic. Implementing instantiation is easy. This approach is analogous to type-instantiation, serving the symmetry of the system.
- However, due to exchanging Subst by Term Instantiation we have to add the extra axiom Combination to Formath. Hol-Light uses the rule Mk_Comb to the same purpose. In Hol-4 the link between equal functions and equal arguments after function-application is trivial by Subst, but both Hol-Light and Formath have to express that the results are equal too. This is a basic statement following from the semantics, and thus we chose to axiomatize this fact, instead of deducing it from a rule.
- In the systems Hol-4 and Hol-Light the rules Assume, Refl, Mk_Comb,... are implemented using ML-functions. If you call the function "Assume" with some formula p as argument, it returns the sequent $p \vdash p$. In some sense

these functions generate theorems. The number of functions which are able to generate theorems is to be minimized, in order to obtain maximal safety for any implementation of our logic. Formath uses an axiom $x \vdash x$, and every time we want to deduce a sequent $p \vdash p$, we just instantiate the term p for the variable x in the axiom. We can do this using the rule for term-instantiation, which is to be kept in the system anyway. Instead of a piece of code which is executed time after time with different arguments and which dynamically generates lots of theorems, we keep one single theorem, which can be inspected statically for errors. This way we try to limit the total number of lines of code which can be executed, but have to accept a slow-down of the system.

- We chose for a straightforward introduction of the set of Individuals instead of using the Dedekind-definition of an infinite set, axiomatized by Infinity_Ax, as done in Hol-4 and Hol-Light. Using the constants C^{IND} and F^{IND} we have the two axioms Individuals_1 and Individuals_2 defining together a denumerable/countable infinite subset of the set Individuals which itself might be non-denumerable/not countable.
- Both Hol-4 and Hol-Light introduce a few elementary and general definitions before writing down the axioms which use these constants, for example Select_Ax. This approach is certainly correct, but did not appear flexible to us; what if we want to use an alternative definition for some basic connectives? In Formath we start with a basic-model, from which the definitions for the standard-model are strictly separated. To express the Formath-axioms we only have to specify the relevant constants without additional general definitions. Introducing the constants is always strictly linked to introducing the respective axioms, and as such we are able to choose very accurately which constants and properties we assume and which we do not. This way of working needed a reformulation of the axioms, and resulted in a separation between the basic- and standard-system throughout the whole design of Formath. One can contrast this approach to the one of Hol-4, where, for example, one defines the existential quantor $\vdash \exists = (\lambda P.P(\varepsilon P))$, mixing a traditional constant and the axiom of choice.
- Of course, the entire Higher-Order-Logic can be built up using only combinatoric terms, without λ-expressions, which are in this sense "syntactic sugar": they are useful only because they improve readability. Traditional systems use the rule Abs, apart from the rule Beta_Conv in Hol-4 and the Beta-rule in Hol-Light, which can be seen as definitions for λ-expressions. By reformulating the principle of extensionality via the axiom Extensionality, it becomes possible to deduce Abs itself. Thus, in axiomatizing Formath we only need a single λ-expression in the rule Beta-Reduction. One can remove this rule, if one also adjusts the theorem Term Extension accordingly.
- The axiom Extensionality needs some additional explanation. The principle of extensionality says that two functions are equal to each other if the result of applying one of these functions to some argument is always equal to the result of applying the other function to the same argument. Traditionally this is expressed by Eta_Ax using η-reductions. In Formath we want to avoid the use of λ, aiming at one single λ-expression in the entire logic (namely in

Beta-Reduction). Using Eta_Ax it is also not possible to express extensionality between any two functions: we have to link a λ-function to its defining term. We now explain how we developed our axiom Extensionality, and also discuss some alternatives.

- We can introduce extensionality using a rule, instead of the axioms Eta_Ax or Extensionality. We write down this rule for example as follows.

$$\frac{\Gamma \vdash fx = gx}{\Gamma \vdash f = g}$$

  Here x may not occur in Γ, f or g. The symbols Γ, f, g and x are of course meta-variables, together with the restriction that x has to be instantiated by a variable. We do not consider this way of working because we want to try to minimize the amount of executable code, instead of adding another deduction-rule.
- We can introduce extensionality using an axiom, instead of a rule. We also don't want to use any λ, but on the opposite side we want to express extensionality between any two functions f and g. We can think of some new, unspecified constant C^{EXT} which is used in the new axiom $fC^{EXT} = gC^{EXT} \vdash f = g$. Indeed, we can't deduce anything about C^{EXT} because we don't know this constant, and thus if $fC^{EXT} = gC^{EXT}$ it also has to hold that $f = g$. However, this axiom, taken together with the other rules and axioms, results in an inconsistent system. Indeed, suppose that we instantiate f resp. g by the $bool \to bool$-functions $\lambda x.T$ (here T means the value $True$) resp. $\lambda x.(x = C^{EXT})$. We then get $(\lambda x.T)C^{EXT} = (\lambda x.(x = C^{EXT}))C^{EXT} \vdash (\lambda x.T) = (\lambda x.(x = C^{EXT}))$. Thus follows $T = (C^{EXT} = C^{EXT}) \vdash (\lambda x.T) = (\lambda x.(x = C^{EXT}))$ and thus $\vdash (\lambda x.T) = (\lambda x.(x = C^{EXT}))$. The constant C^{EXT} is interpreted as T or F (here F means $False$), because our semantics interprets the set of booleans as a set containing only two elements. A case-analysis gives on one side $\vdash (\lambda x.T)F = (\lambda x.(x = T))F$ and analogously on the other side $\vdash (\lambda x.T)T = (\lambda x.(x = F))T$ by function-application using the argument F resp. T. Thus follows in the first case $\vdash T = (F = T)$ and in the second case $\vdash T = (T = F)$, which is inconsistent twice.
- A consistent solution is found by parameterizing C^{EXT} by f and g themselves, which invalidates the scheme above and results in the axiom Extensionality which was already introduced.

- In Hol-Light one can deduce that the logic is classical by means of the axiom of choice Select_Ax. To that aim, one has to use the infinitary version of the disjunction $\vdash \vee = (\lambda pq.\forall r.(p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r)$ to define the existential quantor $\vdash \exists = (\lambda P.\forall q.(\forall x.(Px \Rightarrow q)) \Rightarrow q)$. In Hol-4, an additional axiom Bool_Cases_Ax is introduced. In Formath, classicality of the logic automatically follows from using sets of consequent-formulas. We deduce this in a semi-formal way, as the entire proof is rather long.

1	x	$\vdash$	x	$Assumption$	$-$
2	F	$\vdash$	F	$TermInstantiation$	1
3		$\vdash$	$x = F, x, F$	$EqualityIntroduction$	$1, 2$
4	x	$\vdash$	x, F	$EqualityElimination$	$1, 3$
5		$\vdash$	$x, x \Rightarrow F$	$Discharge$	4
6		$\vdash$	$x, \neg x$	$\neg - Definition$	5

This means, of course, that Formath does not let you choose between working classically or constructively.

As is clear from the discussion above, we tried to stick to our main principles as much as possible while designing Formath. We believe that we achieved our goals, although by times we had to make some hard decisions while extracting the best ideas from both Hol-4 and Hol-Light.

4.2 Porting theorems

The usability of a logic not only depends on the syntax, semantics, deduction-rules, axioms or principles of extension. The availability of a large library of many useful theorems is at least as important. Therefore we ported the entire Hol-Light library to Formath. Indeed, this library is structured very well and contains lots of interesting types (booleans, pairs, lists, natural numbers,...) and deductions (amongst which the Fundamental Theorem of Calculus). Checking these 2178 theorems takes exactly 3126636 inferences using the Hol-Light-logic, in contrast to 8399559 using the Formath-logic, resulting in a factor ± 2.7.

Maybe it is worthwhile to explain how we ported the Hol-Light library to Formath. Above the core of Hol-Light we defined functions which implement the deduction-rules from Formath, and we proved the Formath-axioms, using the deduction-rules and axioms from Hol-Light. Next, we reimplemented the deduction-rules from Hol-Light, and proved the Hol-Light axioms, using only the already implemented Formath-rules and Formath-axioms. After this, we could run all the proofs from the Hol-Light library, through both the Hol-Light layer and the Formath layer. In the Formath layer we could write out all inferences, to check them in a separate/stand-alone implementation of Formath. Moreover, this way of working immediately gives a proof of the equiconsistency of Hol-Light and Formath. Indeed, these systems are equivalent.

Porting entire libraries of theorems is a rare thing. Only a few prooftools are able to import deductions from other systems. That is a pity, because it saves a lot of work, and can provide a nice addition to a system. The latest version of Hol-Light facilitates the process of porting theorems, using "proofrecording". At the time we performed above experiments this package was not included in the distribution yet, so we had to write our own code to extract the theorems. Some other more recent work on porting Hol-theorems was done by S. McLaughlin [24] and S. Obua [26].

4.3 Examples

We will now give a few example definitions and derivations in Formath. First we provide the definitions of a few typical logical constants. We assume an infix-notation for the equality, and we have omitted the types to ensure readability, although these can sometimes be quite complicated, e.g. the definition of T actually is

$$\vdash T_{bool} =_{bool\to bool\to bool} (=_{bool\to bool\to bool}=_{(bool\to bool\to bool)\to(bool\to bool\to bool)\to bool}=_{bool\to bool\to bool})$$

instead of the short form below.

- $\vdash T = (===)$
- $\vdash \wedge = (\lambda pq.(\lambda f.fpq) = (\lambda f.fTT))$
- $\vdash \Rightarrow = (\lambda pq.p \wedge q = p)$
- $\vdash \forall = (\lambda P.(P = \lambda x.T))$
- $\vdash \vee = (\lambda pq.\forall r.(p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r)$
- $\vdash F = (\forall p.p)$
- $\vdash \neg = (\lambda p.(p \Rightarrow F))$
- $\vdash \exists = (\lambda P.\neg\forall x.\neg Px)$

Of course, other definitions are possible as well.

As a first example derivation we provide the proof for the following rule which allows to add any formula to the consequent of a sequent.

$$\frac{\Gamma_1 \vdash \Gamma_2}{\Gamma_1 \vdash \Gamma_2, p}$$

Of course, this does not make sense in the case that already $p \in \Gamma_2$, so in the following proof we assume that $\Gamma_2 - \{p\} = \Gamma_2$.

1	Γ_1	$\vdash$	Γ_2	*Hypothesis*	–
2		$\vdash$	$x = x$	*Reflexivity*	–
3		$\vdash$	$p = p$	*TermInstantiation*	2
4	Γ_1	$\vdash$	Γ_2, p	*EqualityElimination*	1, 3

The next example proves the symmetry of equality, namely $x = y \vdash y = x$. From this it is easy to prove that $\vdash \forall xy.(x = y) = (y = x)$. The deduction is presented in a semi-formal way, as the entire proof is rather long (e.g. we dropped applications of the rule Type Instantiation). Moreover, to avoid ambiguity we do not use the infix-notation for the equality.

1	$= fg, = xy$	$\vdash$	$= (fx)(gy)$	*Combination*	–
2	$===, = xy$	$\vdash$	$= (= x)(= y)$	*TermInstantiation*	1
3		$\vdash$	$= xx$	*Reflexivity*	–
4		$\vdash$	$===$	*TermInstantiation*	3
5	$= xy$	$\vdash$	$= (===)(= (= x)(= y))$	*EqualityIntroduction*	2, 4
6	$= xy$	$\vdash$	$= (= x)(= y)$	*EqualityElimination*	4, 5
7	$= (= x)(= y), = xx$	$\vdash$	$= (= xx)(= yx)$	*TermInstantiation*	1
8	$= (= x)(= y)$	$\vdash$	$= (= xx)(= (= xx)(= yx))$	*EqualityIntroduction*	3, 7
9	$= (= x)(= y)$	$\vdash$	$= (= xx)(= yx)$	*EqualityElimination*	3, 8
10	$= (= x)(= y)$	$\vdash$	$= yx$	*EqualityElimination*	3, 9
11	$= xy$	$\vdash$	$= (= (= x)(= y))(= yx)$	*EqualityIntroduction*	6, 10
12	$= xy$	$\vdash$	$= yx$	*EqualityElimination*	6, 11

As a last example, we carve out the naturals from the Individuals using the inductive definition $\lambda x.\forall P.((PC \wedge \forall y.(Py \Rightarrow P(Fy))) \Rightarrow Px)$, which is nothing but a characteristic predicate with witness C. More information about inductive type- and term-definitions in Hol can be found in the paper by J. Harrison [13]. This way we developed all usual types and constants in Formath.

4.4 Applications and Rationale for Formath

We will now proceed by discussing the main application of our new logic. Formath was developed for our PhD-thesis [6], which was about proving software correct.

We wanted to develop a new environment for building software which can't fail. Therefore, we designed a new functional programming language from scratch, which we baptized Alfred, an acronym for Another Language for Lazy Functional REDuction. This language has some unique features, e.g. it has only one built-in function, which is used to steer the order of evaluation. All data-types (booleans, pairs, lists, natural numbers,...) and functions operating on these data (for evaluating boolean expressions, comparing pairs, appending lists, adding numbers,...) are built up using user-defined combinators, much in the style of Combinatory Logic [9]. This way, all data and functions are constructed in layers, one above the other. A complete hierarchy was built, and at this moment Alfred is a fully-fledged general-purpose programming language.

We started proving that our Alfred-library contains only correct code, by specifying and verifying all layers using the prooftool PVS [27]. Therefore we designed an Alfred-compiler, which on one side can translate Alfred-sources to executable binaries, and on the other side translate the same sources to PVS-specifications. For simple combinators the compiler also provides correctness proofs, i.e. PVS-proofscripts. Needless to say that it is not possible to compile every Alfred-function to a decent logical specification of that function, let alone generate the accompanying proof of correctness...

So although this way of working was more or less fine, there were some defects to our approach, e.g. we did not have access to the full PVS-sources, which obstructed developing a fine-tuned environment. Even more, PVS allows on-the-fly axioms, which implies that not all types and terms are effectively constructed. The system seemed big and inflexible to us, so eventually we decided to migrate to other prooftools. After looking at Hol-4 and Hol-Light, which are open-source and more safe, we thought about developing our own proofchecker. Indeed, Hol-4 and Hol-Light are very decent systems, but instead of relying on Moscow-ML or Caml-Light, why not implementing a verifier for Higher-Order-Logic in our programming language Alfred, and maybe verify the verifier itself [15]? We started coding this proofchecker, but after some time it became clear that some parts of the Hol-logic could be reworked, at the same time heading towards some of our principles of design which we mentioned above. We ended by redesigning the total core, using lots of ideas from both PVS, Hol-4 and Hol-Light.

It is a good question to ask whether it was worthwhile and necessary to spend time to rework the core of Hol, considering the excellent systems Hol-4 and Hol-Light. We hope that the above discussion of the structure underlying Formath has convinced the mathematical reader of our system's own merits.

At this moment we work on an integrated environment, which allows writing Alfred-code, together with Formath-annotations, and immediately proving the necessary theorems to establish the correctness of the code, much in the spirit of ACL2 [2]. Although the system is still in an experimental phase, the results seem to be promising. Indeed, both Alfred and Formath are built up using layers, which of course are tuned to each other. The tools match each other well, which facilitates the burden of proving software correct. Starting from a very small core in both Alfred and Formath, the code-library and proof-library are built up next to each other, providing a tight coupling between formal specifications and actual implementations in every layer. This way we hope to develop a nice environment to design and verify source-code for large software-projects.

5 Conclusion

We introduced a new formalization of Hol, called Formath, and discussed the syntax, semantics, deduction-rules, axioms and principles of extension, after which we proved soundness and consistency. We discussed the main similarities and differences between the systems Hol-4, Hol-Light and Formath, provided example proofs, and talked about porting theorems to the Formath library and its applications to verifying software.

Main principles while designing Formath were orthogonality and symmetry of rules and axioms, cleanliness of code, a very small core, and equiconsistency with the other main-stream Hol distributions. We believe that we achieved these goals, and hope to attract the mathematician's interest by having designed an appealing definition of Higher-Order-Logic.

Currently we are developing an integrated environment which allows construction of correct software in a straightforward manner using Formath. Building up both formal specifications and actual implementations, layer above layer, makes it possible to develop an entirely checked library of source-code, which will allow us to design and verify source-code for large software-projects.

Acknowledgment

The author wants to thank Ghent University for funding his work, and H. Blontrock, A. Hoogewijs and the anonymous referees for their suggestions.

References

[1] H. Abelson, G. Sussman, J. Sussman, Structure and Implementation of Computer Programs, Second Edition, The MIT Press, 1996.

[2] ACL2, http://www.cs.utexas.edu/users/moore/acl2

[3] P. Audenaert, Formele Specificaties in PVS, Master-Thesis, Ghent University, 2000.

[4] P. Audenaert, Higher-Order Partial Predicates in PVS, Proceedings of the Sixth Dutch Prooftool Day, Utrecht, 2002.

[5] P. Audenaert, Memory Modeling in PVS, Invited Contribution, Proceedings of the Fifth Dutch Prooftool Day, Ghent, 2000.

[6] P. Audenaert, Specificatie en Verificatie van Functionele Programmatuur in Hogere-Orde-Logica, PhD-Thesis, Ghent University, 2004.

[7] P. Audenaert, Toward a Generic Verified Prooftool, Supplemental Proceedings of the Fourteenth International Conference on Theorem Proving in Higher Order Logics, Edinburgh, 2001.

[8] P. Audenaert, Verifying Alfred-Code in PVS, Proceedings of the Seventh Dutch Prooftool Day, Amsterdam, 2003.

[9] H. Barendregt, The Lambda Calculus, its Syntax and Semantics, Studies in Logic, Volume 103, North-Holland Publishing Company, 1981.

[10] S. Berghofer, C. Urban, A Head-to-Head Comparison of de Bruijn Indices and Names, Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Seattle, 2006.

[11] Caml-Light, http://caml.inria.fr

[12] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic, Volume 5, 1940.

[13] J. Harrison, Inductive Definitions: Automation and Application, Proceedings of the Eighth International Workshop on Higher Order Logic Theorem Proving and its Applications, Utah, 1995.

[14] J. Harrison, Private Communication, 2003.

[15] J. Harrison, Towards Self-Verification of HOL Light, Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, 2006.

[16] Hol-4, http://hol.sourceforge.net

[17] Hol-4 System Description, http://hol.sourceforge.net

[18] Hol-Light, http://www.cl.cam.ac.uk/users/jrh/hol-light

[19] A. Hoogewijs, P. Audenaert, A PVS-Proof for a Memory Modeling Problem is a Proof!, The Bulletin of Symbolic Logic, Volume 8, Number 1, 2002.

[20] A. Hoogewijs, P. Audenaert, Combinatory Logic, a Bridge to Verified Programs, The Bulletin of Symbolic Logic, Volume 10, Number 2, 2004.

[21] A. Hoogewijs, P. Audenaert, Formath: Higher-Order Logic Revised, The Bulletin of Symbolic Logic, Volume 11, Number 2, 2005.

[22] A. Hoogewijs, P. Audenaert, Implementing Undefinedness in a Two-Valued Prooftool through a Four-Valued Kleene Logic, The Bulletin of Symbolic Logic, Volume 9, Number 1, 2003.

[23] F. Kamareddine, A. Rios, Pure Type Systems with de Bruijn Indices, The Computer Journal, Volume 45, Number 2, 2002.

[24] S. McLaughlin, An Interpretation of Isabelle/HOL in HOL Light, Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, 2006.

[25] Moscow-ML, http://www.dina.dk/~sestoft/mosml.html

[26] S. Obua, S. Skalberg, Importing HOL into Isabelle/HOL, Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, 2006.

[27] PVS, http://pvs.csl.sri.com

[28] L. Théry, Private Communication, 2003.

Department of Pure Mathematics and Computer Algebra
Ghent University
Galglaan 2, S22
9000 Ghent
Belgium
email : paudenae@cage.ugent.be