Task: **1.3**
Version: **1.0**
Date: **February 1999**

A Type System for *OpenMath*

O. Caprotti and A. M. Cohen

RIACA, The Netherlands

D1.3.2b (Public)

Abstract

This document describes one possible type system for *OpenMath* that is an adaptation of the Extended Calculus of Constructions. Including formally specified type information in *OpenMath* Content Dictionaries allows to assign precise semantical meaning to *OpenMath* objects corresponding to mathematical notions and therefore to perform automatic validation on *OpenMath* objects.

Contents

List of Figures

1 Introduction

In this document we study a possible type system that can be used to formalize the signatures of the *OpenMath* symbols. This is not part of the *OpenMath* Standard since *OpenMath* does not enforce any specific type system for expressing signatures of symbols.

According to the *OpenMath* standard, Content Dictionaries specify the meaning of symbols informally using natural language and formally by assigning type information in the signature. It is not feasible to require that every symbol defined in a Content Dictionary is equipped with a formal signature and, in fact, the standard does not require it. However, when the formal signature is present, it is helpful in determining whether mathematical meaning can be assigned to the object in which the symbol occurs.

There are other advantages to formally specified signatures. For one, they are understood by software applications directly and can be checked automatically. Validation of *OpenMath* objects can be done without any knowledge of Phrasebooks and depends exclusively on the context determined by the Content Dictionaries and on some type information carried by the objects themselves (types of the variables). Additionally, interaction with theorem provers and proof checkers becomes easily achievable. This alone opens the way for a wide range of "intelligent" applications.

A further motivation for adopting a formal type system is that, while in many cases pure type systems can be converted one into the other, the same cannot be said of "less formal" types (e.g., AXIOM types - `anytype` is a problem [1]). Having Content Dictionaries that use a formal type system allows for automatic conversion to a different type system should the need arise.

This chapter describes how types and signatures can be used to identify the mathematical objects represented by *OpenMath* objects.

2 Signatures and types in *OpenMath*

The approach taken here is to define and use a Content Dictionary, called `ecc.omc`, for representing a specific type system, the Extended Calculus of Constructions. Then, since signatures are themselves mathematical objects, they correspond to *OpenMath* objects containing symbols from this Content Dictionary.

This approach does not exclude the possible scenario in which an *OpenMath* application requires Content Dictionaries whose signatures are described using a different type system, namely a different Content Dictionary for the types.

One added benefit of having a Content Dictionary to express a powerful type system is that logical properties of *OpenMath* symbols can be also formally defined as *OpenMath* objects and included as such in symbols' definitions.

As we mentioned, deciding a type for an *OpenMath* object corresponds to assigning mathematical meaning to the object. Because of this, we are interested in a type system with (decidable) type inference. The Extended Calculus of Constructions (ECC) is known to have decidable type inference and thus was chosen as starting point for assigning signatures to *OpenMath* symbols. Moreover, the (Extended) Calculus of Constructions has been implemented in systems like Lego or COQ [5, 6]. These systems, if *OpenMath* compliant, can provide the functionalities for performing type checks on *OpenMath* objects.

We stress that type correctness is not required for *OpenMath* objects to qualify as *OpenMath* objects. In fact, type systems give rigor to *OpenMath* that represent mathematical objects only. The type mechanism is provided as a way of automatically checking "meaningfulness" of *OpenMath* objects that represent mathematical objects. It would not make sense to incorporate the whole *OpenMath* language, including errors. An error occurrence just means that something is wrong anyway.

Applications that receive type-checked objects must agree on the semantics underlying the type signature; in fact these applications understand the Content Dictionary `ecc.omc` used for types. To their advantage, the objects can be converted by the Phrasebook in an automated way.

OpenMath objects can be seen as labelled trees when no other semantical meaning can be assigned using the Content Dictionaries. Applications like editors, that do not use Content Dictionaries, would typically understand *OpenMath* objects in such a way. However, when Content Dictionaries are supported, *OpenMath* objects can be assigned to mathematical objects by use of the signature of the symbols. A fully specified *OpenMath* object, in which variables are typed, can be type-checked and its well-typedness verified in the contexts defined by the relevant Content Dictionaries. When these objects have a type, they are considered meaningful mathematical objects. The type of a symbol or of an *OpenMath* object is an *OpenMath* object corresponding to a type. Mathematical objects and types are described in Section 3.

3 Mathematical objects and types in *OpenMath*

Already in the system AUTOMATH [2], the primitive constructors of the meta-language were application, abstraction and function space type, and they are also the primitives for the much more recent system of Logical Frameworks [3]. Thus, it makes sense to equip *OpenMath* with a Content Dictionary in which symbols for expressing these notions are defined. Moreover, in order to express *OpenMath* signatures, we will need to have symbols for the types of the *OpenMath* basic objects. These and the symbols for pairing, projection and for the cartesian product, are part of the Content Dictionary `ecc` included in Appendix A. This Content Dictionary suffices for expressing objects and types arising in the Extended Calculus of Constructions.

The semantics of the constructors for abstraction, application, projection and pairing is explained formally in terms of reduction and congruence relations.

In what follows, we fix the notation used for mathematical objects and for the types appearing in signatures of Content Dictionaries that use `ecc`. We will mainly use abstract *OpenMath* objects althought we might at times suggest a particular XML encoding. The *OpenMath* objects in the class defined below are called *Strong OpenMath* objects and are built using attribution, application, and symbols from `ecc` or defined in Content Dictionaries that use `ecc`. The mathematical meaning of these objects is closely related to the semantics of the symbols defined using types in the Extended Calculus of Constructions. This meaning can be assigned by using the type-inference rules described later in this chapter.

Basic Objects The *OpenMath* symbols described in the Content Dictionary `ecc`: `integer`, `float`, `string`, `byearray`, `prop`, `symtype`, `omtype` are reserved symbols for *OpenMath* types and they are *Strong OpenMath*. They correspond to the symbolic constant terms integer, float, string, bytearray, prop for logical propositions, symtype for symbolic types, and omtype for the type of symbolic types.

OpenMath symbols defined in Content Dictionaries that use `ecc` are *Strong OpenMath* and

correspond to constants of appropriate type (as defined in their signature). For instance, the symbol `posintT` for the type of positive integers can be defined in a Content Dictionary for integer arithmetic as:

```
<CDDefinition>
        <Name> posintT </Name>
        <Description> The type of positive Integers </Description>
        <Signature> symtype </Signature>
</CDDefinition>
```

and corresponds to a new constant posintT of type symtype.

Basic objects like integers, floating-point numbers, bytearrays, and character strings are *Strong OpenMath* and correspond to constants of type integer, float, bytearray, and string respectively. *OpenMath* variables are *Strong OpenMath* and correspond to variables.

If v is an *OpenMath* variable and t is a *Strong OpenMath* object, then

$$\textbf{attribution}(v, \texttt{type}\ t)$$

is *Strong OpenMath*. It denotes an *OpenMath* variable with type t (typed variable), $v|_t$ and corresponds to the judgement $v : t$. For the moment, we will assume that, if a variable occurs free in a fully typed object then it is equipped with its type (at least once). Types for the other symbols can be derived from the Content Dictionaries. A fully typed *Strong OpenMath* object is, for instance, the object:

$$\textbf{application}(\texttt{sin}, \textbf{attribution}(v, \texttt{type real}))$$

encoded in *xml* as:

```
<OMOBJ><OMA><OMS cd="basic" name="sin"/>
       <OMATTR><OMATP>
              <OMS cd="ecc" name="type"/>
              <OMS cd="ecc" name="real"/>
       </OMATP><OMV name="x"/></OMATTR>
</OMA></OMOBJ>
```

Abstraction If v is an *OpenMath* variable and t, A are *Strong OpenMath* objects, then

$$\textbf{binding}(\texttt{Lambda}, \textbf{attribution}(v, \texttt{type}\ t), A)$$

is *Strong OpenMath* and denotes the function that assigns to the variable v of type t the object A. It corresponds to (its semantics is) the lambda term $\lambda v : \hat{t}.\hat{A}$, where $\hat{t}$ is the term corresponding to the object t and likewise for $\hat{A}$. The variable v is called the λ *bound variable*.

Application If F, A are *Strong OpenMath*, then

$$\textbf{application}(F, A)$$

is *Strong OpenMath* and denotes the application of the object F to object A. It corresponds to the term $(\hat{F}\ \hat{A})$.

The object:

$$\textbf{application}(F, A_1, \ldots, A_n)$$

corresponds to the term:

$$(\ldots((\hat{F}\ \hat{A_1})\ \hat{A_2})\ldots\ \hat{A_n})$$

and is accordingly abbreviated as $(F\ A_1\ \ldots\ A_n)$.

For instance, the *Strong OpenMath* object:

$$\textbf{application}(\textbf{binding}(\texttt{Lambda}, \textbf{attribution}(v, \texttt{type real}), \textbf{application}(\texttt{sin}, v)), \texttt{Pi})$$

corresponds to the term $((\lambda v : real.sin\ v)\ \pi)$. Its XML encoding looks like follows:

```
<OMOBJ><OMA><OMBIND><OMS cd="ecc" name="Lambda" />
       <OMBVAR><OMATTR><OMATP>
         <OMS cd="ecc" name="type"/>
         <OMS cd="ecc" name="real"/>
         </OMATP><OMV name="v"/></OMATTR>
       </OMBVAR>
       <OMA><OMS cd="trig" name="sin" />
            <OMV name="v"/>
       </OMA></OMBIND>
     <OMS cd="trig" name="Pi" />
</OMA></OMOBJ>
```

The mathematical meaning of abstraction and application is given by the β-reduction rule:

$$(\lambda v : t.u)\Omega\ \hookrightarrow_\beta u[v \leftarrow \Omega]$$

where $t[v \leftarrow \Omega]$ denotes the term obtained from u by simultaneously substituting every occurrence of v by Ω. This rule states that abstraction is the inverse operation of application.

Function Space The type constructor for the function space type is `PiType`. If t and u are *Strong OpenMath* objects, and v is an *OpenMath* variable, then

$$\textbf{binding}(\texttt{PiType}, \textbf{attribution}(v, \texttt{type}\ t), u)$$

is *Strong OpenMath*. It represents the type of functions mapping an argument v of type t to a result of type u. It corresponds to the term $\Pi v : \hat{t}.\hat{u}$. If v does not occur in u, then the object denotes $t \rightarrow u$. Pi-types could be used to represent universal quantification as is done in the mathematical property below, but we envision having a Content Dictionary defining logical connectives and quantifiers.

For example, consider in Figure 1 how the `plus` symbol may be defined in a Content Dictionary `integer`. The *Strong OpenMath* object defining the signature corresponds to the function space type described by the term $\Pi x : \mathsf{integer}.(\Pi y : \mathsf{integer}.\mathsf{integer})$ or equivalently $\mathsf{integer} \rightarrow (\mathsf{integer} \rightarrow \mathsf{integer})$. The definition includes a formal mathematical property, identified by the tag `FMP`. It is a *Strong OpenMath* object encoding a formal proposition suitable for automated verification.

Parametric types are also function types. Figure 2 is an example of a possible way of declaring the type $\mathbb{Z}_n$ for integers modulo n, $n \in \mathbb{N}$. The parametric type `intModT` depends

```
<CDDefinition>
        <Name> plus </Name>
        <Description> Integer addition </Description>
        <Signature>
                <OMOBJ><OMBIND><OMS cd="ecc" name="PiType" />
                    <OMBVAR><OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMS cd="ecc" name="integer"/>
                       </OMATP><OMV name="x"/></OMATTR>
                       <OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMS cd="ecc" name="integer"/>
                       </OMATP><OMV name="y"/></OMATTR>
                    </OMBVAR>
                    <OMS cd="ecc" name="integer"/>
                </OMBIND></OMOBJ>
        </Signature>
        <FMP>
             <OMOBJ><OMBIND><OMS cd="ecc" name="ForAll" />
                    <OMBVAR><OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMS cd="ecc" name="integer"/>
                    </OMATP><OMV name="a"/></OMATTR></OMBVAR>
                    <OMA><OMS cd="integer" name="equal" />
                         <OMA><OMS cd="integer" name="plus" />
                         <OMV name="a" />
                         0
                         </OMA>
                    <OMV name="a" />
                    </OMA>
             </OMBIND></OMOBJ>
         </FMP>
</CDDefinition>
```

Figure 1: Definition of `plus`.

```
<CDDefinition>
        <Name> intModT </Name>
        <Description> The type of integers Mod n <Description>
        <Signature>
                  <OMOBJ><OMBIND><OMS cd="ecc" name="PiType" />
                  <OMBVAR><OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMS cd="ecc" name="posintT"/>
                  </OMATP><OMV name="n"/></OMATTR></OMBVAR>
                  <OMS cd="ecc" name="symtype" />
                  </OMBIND></OMOBJ>
         </Signature>
</CDDefinition>
```

Figure 2: Possible definition of `IntModT`

```
<CDDefinition>
        <Name> Mod </Name>
        <Description> The Mod n function over the Integers </Description>
        <Signature>
               <OMOBJ><OMBIND><OMS cd="ecc" name="PiType" />
                  <OMBVAR><OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMS cd="ecc" name="posintT"/>
                  </OMATP><OMV name="n"/></OMATTR></OMBVAR>
                    <OMBIND><OMS cd="ecc" name="PiType" />
                    <OMBVAR><OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMS cd="ecc" name="integer"/>
                    </OMATP><OMV name="y"/></OMATTR></OMBVAR>
                    <OMS cd="ecc" name="integer" />
                    <OMA><OMS cd="ecc" name="intModT" />
                         <OMV name="n" />
                    </OMA>
                  </OMBIND></OMBIND></OMOBJ>
        </Signature>
</CDDefinition>
```

Figure 3: Possible definition of the function `Mod`

```
<CDDefinition>
        <Name> rationalT </Name>
        <Description> The type of Rationals </Description>
        <Signature><OMOBJ><OMS cd="ecc" name="symtype" /></OMBJ></Signature>
</CDDefinition>

<CDDefinition>
        <Name>Rational</Name>
        <Description> The Rational constructor
               A cartesian product in which the first component identifies
               the numerator and the second component identifies the denominator.
        </Description>
        <Signature>
               <OMOBJ><OMBIND><OMS cd="ecc" name="PiType" />
                    <OMBVAR><OMATTR><OMATP>
                       <OMS cd="ecc" name="type"/>
                       <OMBIND>
                       <OMS cd="ecc" name="SigmaType" />
                       <OMBVAR><OMATTR><OMATP>
                            <OMS cd="ecc" name="type"/>
                            <OMS cd="ecc" name="integer"/>
                       </OMATP><OMV name="x"/></OMATTR></OMBVAR>
                       <OMS cd="ecc" name="integer"/>
                       </OMBIND>
                    </OMATP><OMV name="y"/></OMATTR></OMBVAR>
                    <OMS cd="ecc" name="integer" />
                </OMBIND></OMOBJ>
        </Signature>
</CDDefinition>
```

Figure 4: A possible definition of the constructor `Rational`

on a positive integer n. All instances of this type, like `intModT(3)`,`intModT(5)` and so on, are of type `symtype`. Using this parametric type with a dependent function type, Figure 3 gives a possible definition for the signature of the Mod n operation for any n. The signature is the encoding of the object corresponding to the term $\Pi n : \mathsf{posintT}.(\mathsf{integer} \rightarrow \mathtt{intModT}(n))$.

Cartesian Product The type constructor for the cartesian product type is `SigmaType`. If t and u are *Strong OpenMath* objects, and v is an *OpenMath* variable, then

$$\mathbf{binding}(\mathtt{SigmaType}, \mathbf{attribution}(v, \mathtt{type}\ t), u)$$

is *Strong OpenMath*. It represents the type of pairs consisting of an object v of type t and an object of type u: namely $\Sigma v : \hat{t}.\hat{u}$. If v does not occur in u, then the object denotes $\hat{t} \times \hat{u}$.

As example, consider a possible definition of rational numbers as is done in Figure 4. First, the symbolic type for the type of rationals is introduced and then the constructor `Rational` is defined with signature corresponding to the term $\Pi y : (\Sigma x : \mathsf{integer}.\mathsf{integer}).\mathsf{integer}$:

Pair If A_1, A_2, S are *Strong OpenMath* objects, then

$$\textbf{application}(\texttt{Pair}, A_1, A_2, S)$$

is *Strong OpenMath*. It represents the pair consisting of A_1 and A_2 and having type S. It corresponds to the term $\langle \hat{A_1}, \hat{A_2} \rangle_{\hat{S}}$.

The following abbreviation will be used:

$$\langle A_1, A_2 \ldots, A_{n-1}, A_n \rangle_S$$

for the nested pair

$$\langle \ldots \langle A_1, A_2 \rangle_{S_2} \ldots A_n \rangle_{S_n}$$

with

$$S = \Sigma x_n : (\Sigma x_{n-1} : \ldots (\Sigma x_1 : S_1.S_2). \ldots).S_{n-1}).S_n.$$

Pairs are tipically used for data structures.

Projection If A is a *Strong OpenMath* object then

$$\textbf{application}(\texttt{PairProj1}, A) \quad \textbf{application}(\texttt{PairProj2}, A)$$

are *Strong OpenMath* and they denote the first and second projection. They correspond to the terms $(\pi_1\ \hat{A})$, and $(\pi_2\ \hat{A})$.

The behavior of pairing and projection is given by the σ-reduction rule[1]:

$$(\pi_i\ \langle A_1, A_2 \rangle_S) \hookrightarrow_\sigma A_i \quad (i = 1, 2)$$

This rule states that projections are inverse to new.

New Binders If v is an *OpenMath* variable, B is a *Strong OpenMath* symbol other than `SigmaType`, `PiType`, `Lambda` and t, C are *Strong OpenMath* objects, then

$$\textbf{binding}(B, \textbf{attribution}(\texttt{type}\ t, v), A)$$

is *Strong OpenMath* and denotes the logical function that binds variable v of type t in the object A. It corresponds to the term $(\hat{B}\ \hat{t}\ (\lambda v : \hat{t}.\hat{A}))$, where $\hat{t}$ is the term corresponding to the object t and likewise for $\hat{A}$ and $\hat{B}$.

In this way, new quantifiers like the universal or the existential one can be defined in some Content Dictionary by giving an appropriate signature. This is enough to perform type checking. In order to do reasoning however, the new binders have to be defined in terms of primitive ones.

The symbols involved in the notions just described are declared in a new Content Dictionary `ecc` included in Appendix A.

A note on semantics: The reduction relation $\hookrightarrow$ is the compatible closure of the contraction relation $\hookrightarrow_\beta \cup \hookrightarrow_\sigma$. The transitive, reflexive closure of $\hookrightarrow$ is denoted by $\hookrightarrow^*$ and is a reduction relation. When a *Strong OpenMath* object m reduces to an object m', denoted by $m \hookrightarrow^* m'$, and m' does not reduce anymore, then m' is a normal form of m. If a *Strong OpenMath* object m reduces to m_1 and to m_2 then there exists m' such that $m_1 \hookrightarrow^* m'$ and $m_2 \hookrightarrow^* m'$[2]. Thus, if a *Strong OpenMath* object has a normal form, it is unique. The symmetric, transitive closure of $\hookrightarrow^*$ is denoted by $\sim$; it is a congruence (that is, a compatible equivalence relation).

[1] No π-reduction rule because it would invalidate the Church-Rosser property.

[2] The η-rule would invalidate the Church-Rosser Property.

4 Type Checking *OpenMath* Objects

Given a *Strong OpenMath* object, it is possible to take into account the signature information assigned to the symbols by their definition in the Content Dictionaries. This information, when used to perform type inference or type checking on *Strong OpenMath* objects, allows to decide if a *Strong OpenMath* object has a type. If this is the case, then the object is a *meaningful OpenMath object*. Additionally, the inference rules for the type constructors Π and Σ formalize how they are correctly used to build meaningful objects and in this sense they assign semantics to these symbols.

The type of a *Strong OpenMath* object depends on a context. Such context records what the symbols definitions are and what the type of the variables is.

Definition 1 (*OpenMath* Type Context, *FVS*(Δ)) *A list Δ of OpenMath variables with type is a Strong OpenMath type context. The set of free variables FVS(Δ) of the context Δ is defined as the set of names of variables occurring in Δ. The empty context is denoted by* [].

In the rest of this section, Δ denotes a *Strong OpenMath* type context, and $\mathcal{T}$ is the set of symbols $\{\mathsf{prop}, \mathsf{symtype}, \mathsf{omtype}\}$. Elements of $\mathcal{T}$ are denoted by T and are ordered as follows:

$$\mathsf{prop} < \mathsf{symtype} < \mathsf{omtype}.$$

For each symbol s occurring in a *Strong OpenMath* object, the signature information can be inferred from its definition in the relevant Content Dictionary, say c. For instance, if the Content Dictionary c contains a definition of the following form:

```
<CDDefinition>
        <Name>s</Name>
        <Description> ... </Description>
        <Signature> t </Signature>
        [...]
</CDDefinition>
```

then the symbol s defined in c has signature t, and will be denoted in short $s_{c|_t}$. This information is known to the type inference engine so that from the empty context it is possible to derive a new constant $\hat{s}$ of type $\hat{t}$ such that $[] \vdash \hat{s} : \hat{t}$.

1. Four basic objects are typed in a natural way:

   $[] \vdash i : \mathsf{integer}$ if i is an integer.

   The above should read "If i is an integer, then i has type `integer`".

   $[] \vdash f : \mathsf{float}$ if f is a floating point number.

   $[] \vdash s : \mathsf{string}$ if s is a character string.

   $[] \vdash b : \mathsf{bytearray}$ if b is a byte array.

   Special *OpenMath* types are symbolic types:

   $[] \vdash t : \mathsf{symtype}$ if t is in $\{\mathsf{integer}, \mathsf{float}, \mathsf{string}, \mathsf{bytearray}, \mathsf{prop}\}$.

   The type of symtype is omtype.

   $[] \vdash \mathsf{symtype} : \mathsf{omtype}$.

   The symbol omtype has no type.

2. The remaining two basic objects, variables and symbols, are typed according to their definition.

$$\frac{\Delta \vdash \mathrm{t} : \mathsf{T}}{\Delta, \omega|_t \vdash \omega|_t : \mathrm{t}} \text{ if } \omega \notin FVS(\Delta) \qquad \frac{\Delta \vdash \mathrm{t} : \mathsf{T} \quad \Delta \vdash \Omega : u}{\Delta, \omega|_t \vdash \Omega : u} \text{ if } \omega \notin FVS(\Delta)$$

   where ω is either a variable v or a symbol s_c. When T is type prop, the rules above allow to add assumptions to the context, like for instance the mathematical properties assigned to symbols in Content Dictionaries. Another instance of this last rule states that symbolic types introduced in Content Dictionaries have type symtype and can be added to the type context.

3. The rule for abstraction states that abstraction objects have function space types.

$$\frac{\Delta, v|_t \vdash \Omega : u \quad \Delta \vdash (\Pi v : t.u) : \mathsf{T}}{\Delta \vdash \lambda v|_t.\Omega) : (\Pi v : t.u)}$$

   By this rule, in a context in which the variable v is of type `real` and the symbol `sin` is of type `real -> real`, it is possible to derive that the object $\lambda v.(sin\ v)$ is also of type `real -> real`:

$$\frac{\Delta, v|_{\texttt{real}} \vdash (\texttt{sin}\ v) : \texttt{real} \quad \Delta \vdash (\Pi v : \texttt{real.real}) : \mathsf{symtype}}{\Delta \vdash \lambda v : \texttt{real}.(\texttt{sin}\ v) : (\Pi v : \texttt{real.real})}$$

4. Application objects can be typed if the argument has the function domain as type.

$$\frac{\Delta \vdash \mathrm{F} : (\Pi v : t.u) \quad \Delta \vdash \mathrm{A} : t}{\Delta \vdash (F\ A) : u[v \leftarrow A]}$$

   The expression $u[v \leftarrow A]$ denotes the result of substituting free occurrences of the variable v in the term u by the term A.

   As example, take the usual $(\texttt{sin}\ x)$. If from the context it is possible to derive the premises of the rule below, then the consequence allows to derive a type for the object, hence a "mathematical meaning" for it.

$$\frac{\Delta \vdash \texttt{sin} : (\Pi v : \texttt{real.real}) \quad \Delta \vdash x : \texttt{real}}{\Delta \vdash (\texttt{sin}\ x) : \texttt{real}}$$

   The rule also implies that symbols whose `FunctorClass` is `function` should have signature corresponding to a function space type. More generally, also symbols whose `FunctorClass` is `constructor` have function space type as signature.

5. The rule scheme for function type has been adapted from the Extended Calculus of Constructions to the finite number of type universes of *OpenMath*.

$$\frac{\Delta \vdash t : \mathsf{T}_1 \qquad \Delta, v|_t \vdash u : \mathsf{T}_2}{\Delta \vdash (\Pi v : t.u) : \mathsf{T}_3} \quad \text{if } (\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3) \in \Pi_\mathsf{R}$$

where the set Π_R of rules for the Π constructor is the following:

$$\Pi_\mathsf{R} = \bigcup_{\substack{\mathsf{t}_1 \in \mathsf{T} \\ \mathsf{t}_2 \in \{\mathsf{symtype}, \mathsf{omtype}\}}} \{(\mathsf{t}_1, \mathsf{prop}, \mathsf{prop}), (\mathsf{t}_1, \mathsf{t}_2, \max(\mathsf{t}_1, \mathsf{t}_2))\}$$

In the case in which T_2 is prop, Π represents the universal quantifier.

6. The type inference rule for pairs makes sure that pair objects are typed by cartesian products.

$$\frac{\Delta \vdash (\Sigma v : t.u) : \mathsf{T} \qquad \Delta \vdash A_1 : t \qquad \Delta \vdash A_2 : u[v \leftarrow A_1]}{\Delta \vdash \langle A_1, A_2 \rangle_{\Sigma v:t.u} : (\Sigma v : t.u)}$$

7. Type inference rules for projections guarantees that projection functions are applied only to objects whose types are cartesian products.

$$\frac{\Delta \vdash \Omega : (\Sigma v : t.u)}{\Delta \vdash (\pi_1\ \Omega) : t} \qquad \frac{\Delta \vdash \Omega : (\Sigma v : t.u)}{\Delta \vdash (\pi_2\ \Omega) : u[v \leftarrow (\pi_1\ \Omega)]}$$

8. The rule scheme for cartesian product is a general rule like in [4]. In particular, it allows the type of the second component of a pair to have type prop. This is especially useful to assign a type to algebraic structures, consisting of a set, and an operation together with some axioms.

$$\frac{\Delta \vdash t : \mathsf{T}_1 \qquad \Delta, v|_t \vdash u : \mathsf{T}_2}{\Delta \vdash (\Sigma v : t.u) : \mathsf{T}_3} \quad \text{if } (\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3) \in \Sigma_\mathsf{R}$$

where the set Σ_R of rules for the Σ constructor is the following:

$$\Sigma_\mathsf{R} = \bigcup_{\mathsf{t}_1, \mathsf{t}_2 \in \mathcal{T}} \{(\mathsf{t}_1, \mathsf{t}_2, \max(\max(\mathsf{t}_1, \mathsf{t}_2), \mathsf{symtype})\}$$

9. The next rule depends on the equivalence relation $\sim$ defining the intended meaning of the primitive constructors for abstraction, application, projection and pairing. Here is the last rule:

$$\frac{\Delta \vdash \Omega : t \qquad \Delta \vdash t' : \mathsf{T}}{\Delta \vdash \Omega : t'} \quad \text{if } t \sim t'$$

Decidability of $\sim$ can be shown by adapting the proof given by Luo in [4]

OM2ECC $(m; t)$

Input: m a *Strong OpenMath* object.

Output: t an ECC term corresponding to m.

Global: `ecc` the *OpenMath* Content Dictionary for ECC
$C_1, \ldots, C_n$ a set of *Strong OpenMath* Content Dictionaries

(0) [Initialize.]
Check *IsStrongOM*$(m; b)$. If b =FALSE then return.

(1) [Empty Context and Basic Object.]
If m is a basic object, then it is one of the following:

- (a) [m is a reserved constant.]
  if $m \in \{\mathsf{integer}, \mathsf{float}, \mathsf{string}, \mathsf{bytearray}, \mathsf{prop}, \mathsf{symtype}\}$,
  then t:=m.
- (b) [m is an integer.] t:=m.
- (c) [m is a floating point number.] t:=m.
- (d) [m is a byte array.] t:=m.
- (e) [m is a character string.] t:=m.
- (f) [m is a symbol.] t:=m.
- (g) [m is a variable with name v.] t:=v.

(2) [Attribution, $m = \mathbf{attribution}(A_0, S_1\ A_1,\ \ldots\ , S_n\ A_n)$.]
Compute *OM2ECC*$(A_0; a)$. If $S_i = \texttt{type}$ for some $1 \leq i \leq n$, then compute *OM2ECC*$(A_i; a_i)$ and set $t := a : a_1$. Otherwise $t := a$.

(3) [Application $m = \mathbf{application}(A_1, \ldots, A_n)$]
Compute *OM2ECC*$(A_i; a_i)$ for all $1 \leq i \leq n$ (since m is *Strong OpenMath* $n \leq 2$).
If $A_1 = \texttt{Pair}$, then set t:=$\langle a_2, a_3 \rangle_{a_4}$.
Otherwise set t:=$(a_1\ a_2\ \ldots\ a_n)$.

(4) [Binding, $m = \mathbf{binding}(B, \mathbf{attribution}(V, \texttt{type}\ U), C)$]
Compute *OM2ECC*$(U; u)$, *OM2ECC*$(V; v)$ and *OM2ECC*$(C; c)$.
If $B = \texttt{SigmaType}$, set t:=$\Sigma\ v : u.c$.
If $B = \texttt{PiType}$, set t:=$\Pi\ v : u.c$.
If $B = \texttt{Lambda}$, set t:=$\lambda\ v : u.c$.
Otherwise, $m = \mathbf{application}(B, U, \mathbf{binding}(\texttt{Lambda}, \mathbf{attribution}(V, \texttt{type}\ U), C))$ and compute *OM2ECC*$(m; m')$. Set t:=m'.

(5) [Return.]
Return t.

Figure 5: The algorithm *OM2ECC*

For the Extended Calculus of Constructions, and hence for the system of rules above, type checking and type inference are decidable.

Decidability of type inference for *Strong OpenMath* objects is shown by giving an algorithm which is a modification of the algorithm for ECC type inference. This algorithm, called *TypeInfer*, defined in Figure 6 and Figure 7, takes as input an *Strong OpenMath* type context Γ, and an *OpenMath* object m. If the object m is *Strong OpenMath*, then the algorithm decides if there is a type t such that $\Gamma \vdash m : t$ holds. If the object m is not *Strong OpenMath*, then the algorithm cannot say anything about the object.

Observe that a *Strong OpenMath* type context is given as input to the algorithm. This context is built by signatures of symbols in Content Dictionaries and by the types of the variables in the objects. For *Strong OpenMath* objects, the symbols' type can be found in signature available in the Content Dictionaries, whereas the type of the variables must be explicitely given in the object. In this version of the inference rules, all variables are equipped with type. This means that an *OpenMath* object built using *Strong OpenMath* symbols, but not containing the type for each variable, is not *Strong OpenMath*. It is of course desirable to be able to skip types for variables, but then we must use a semi-decidable procedure to guess types for the variables and we are not, for the moment, concerned with this. The reason why type inference of variables is not simple is that there might be several correct types that can be assigned to a variable. For instance, the expression $(f\ 2)$ where f is an higher-order variable can be given any type consistent with Π_{R}. Nevertheless, it is not hard to imagine a semi-decision procedure that would guess types for the variables in order to turn an *OpenMath* object into a *Strong OpenMath* object that is well-typed.

Proposition 1 *For Strong OpenMath objects, type inference is decidable.*

PROOF. To show this, it is enough to show correctness and termination of algorithm *TypeInfer*.

By inspecting the algorithm, it is easy to see that it is conformant to the *OpenMath* type inference rules. This ensures correctness.

For showing termination, first remark that the recursive calls have smaller arguments with respect to the length of the context and the object. Hence, no infinite recursion can occur. The only possible source for non-termination are the reduction steps. However, since reduction is only performed on well-typed objects, it is terminating and hence, the overall algorithm is also terminating.

Having decidable type inference, also implies that type checking is decidable.

Proposition 2 *For Strong OpenMath objects, type checking is decidable.*

PROOF. Let Ω be an *OpenMath* term and let t be a type and Δ be an *OpenMath* type context, we have to show that $\Delta \vdash \Omega : t$ is decidable. Using the type inference algorithm, we know either that Ω has no type, or that Ω has type t', namely $\Delta \vdash \Omega : t'$. If t and t' are the same then

TypeInfer $(\Gamma, m; b, t)$

Input: Γ an *Strong OpenMath* type context.
m an *OpenMath* object.

Output: b a boolean constant.
t a *Strong OpenMath* object such that if m is *Strong OpenMath*and $b = true$, then $\Gamma \vdash m : t$, otherwise m has no type in Γ. When m is not *Strong OpenMath*, nothing can be said.

Global: ecc the *OpenMath* Content Dictionary for ECC
$C_1, \ldots, C_n$ a set of *Strong OpenMath* Content Dictionaries

(0) [Initialize.]
Check *IsStrongOM*$(m; b)$. If b =FALSE then return.
Else m is a *Strong OpenMath* object. Set b:= TRUE.

(1) [Empty Context and Atomic Object.]
If $\Gamma = []$ and m is an atomic object then it is one of the following:

(a) [Reserved Constants.]
$m \in \{\mathsf{integer}, \mathsf{float}, \mathsf{string}, \mathsf{bytearray}, \mathsf{prop}\}$,
then t:=symtype,
else if m is symtype, then t:=omtype.

(b) [m is an integer.] t:=integer.

(c) [m is a floating point number.] t:=float.

(d) [m is a byte array.] t:=bytearray.

(e) [m is a character string.] t:=string.

(f) [m is another atomic object.] b:= FALSE and t:=m.

(2) [Non-empty Context and Basic Object.]
Let $\Gamma = \{\Gamma', \omega|_u\}$. If $\omega \notin FVS(\Gamma')$, check *TypeInfer*$(\Gamma', u; b', t')$. If $b' = true$ and $t' \hookrightarrow^* \mathsf{T}$ where $\mathsf{T} \in \mathcal{T}$,
then if $m = \omega$ set t:=u,
else $m \neq \omega$. In this case, check *TypeInfer*$(\Gamma', m; b, t)$.
If $\omega \in FVS(\Gamma')$ or $b' = false$ or $t' \not\hookrightarrow^* \mathsf{T}$ where $\mathsf{T} \in \mathcal{T}$, then set b:= FALSE and t:=m.

(3) [Composite Object.]
Call composite type inference *CTypeInfer*$(\Gamma, m; b, t)$.

(4) [Return.]
Return b and t.

Figure 6: The algorithm *TypeInfer*

CTypeInfer $(\Gamma, m; b, t)$

Input: Γ an *OpenMath* type context.
m a composite *Strong OpenMath* object.

Output: b a boolean constant.
t a *Strong OpenMath* object such that if $b = true$ then $\Gamma \vdash m : t$, otherwise m has no type in Γ.

(0) [Initialize.]
Set b:= FALSE and t:=m.

(1) [Structural Induction on m.]
Depending on the structure of m, choose one of the following:

(a) [Abstraction, $m = \lambda v : m_1.m_2$]
Check *TypeInfer*$(\Gamma, m_1; b_1, t_1)$ and *TypeInfer*$(\Gamma \cup \{v|_{m_1}\}, m_2; b_2, t_2)$.
If b_1 and b_2 are TRUE, and $t_1 \hookrightarrow^* \mathsf{T}$ where $\mathsf{T} \in \mathcal{T}$, and $t_2 \neq \mathsf{omtype}$,
then b:= TRUE and t:=$\Pi v : m_1.m_2$.

(b) [Pairing, $m = \langle m_1, m_2 \rangle_u$]
If u is of the form $\Sigma v : u_1.u_2$,
check *TypeInfer*$(\Gamma, u; b', t')$, *TypeInfer*$(\Gamma, m_1; b_1, t_1)$, *TypeInfer*$(\Gamma, m_2; b_2, t_2)$.
If b_1, b_2, and b' are TRUE, and $t_1 \sim u_1$, $t_2 \sim u_2[v \leftarrow m_1]$,
then b:= TRUE and t:=u.

(c1) [Projection1 $m = (\pi_1 m')$]
Check *TypeInfer*$(\Gamma, m'; b', t')$. If b' is TRUE and $t' \hookrightarrow^* \Sigma v : u_1.u_2$,
then b:= TRUE and t:=u_1.

(c2) [Projection2 $m = (\pi_2 m')$]
Check *TypeInfer*$(\Gamma, m'; b', t')$. If b' is TRUE and $t' \hookrightarrow^* \Sigma v : u_1.u_2$,
then b:= TRUE and t:=$u_2[v \leftarrow (\pi_1 m')]$.

(d) [Application, $m = (m_1 m_2)$]
Check *TypeInfer*$(\Gamma, m_1; b_1, t_1)$ and *TypeInfer*$(\Gamma, m_2; b_2, t_2)$.
If b_1 and b_2 are TRUE, and $t_1 \hookrightarrow^* \Pi v : u_1.u_2$, $t_2 \sim u_1$,
then b:= TRUE and t:=$u_2[v \leftarrow m_2]$.

(e) [Function Space $m = \Pi v : m_1.m_2$.]
Check *TypeInfer*$(\Gamma, m_1; b_1, t_1)$, *TypeInfer*$(\Gamma \cup \{v|_{m_1}\}, m_2; b_2, t_2)$.
If b_1 and b_2 are TRUE, $t_1 \hookrightarrow^* \mathsf{T_1}$ and $t_2 \hookrightarrow^* \mathsf{T_2}$ where $\mathsf{T_1}, \mathsf{T_2} \in \mathcal{T}$,
find $\mathsf{T_3}$ such that $(\mathsf{T_1}, \mathsf{T_2}, \mathsf{T_3}) \in \Pi_{\mathsf{R}}$ and set b:= TRUE, t:=$\mathsf{T_3}$.

(f) [Cartesian Product $m = \Sigma v : m_1.m_2$.]
Check *TypeInfer*$(\Gamma, m_1; b_1, t_1)$, *TypeInfer*$(\Gamma \cup \{v|_{m_1}\}, m_2; b_2, t_2)$.
If b_1 and b_2 are TRUE, $t_1 \hookrightarrow^* \mathsf{T_1}$ and $t_2 \hookrightarrow^* \mathsf{T_2}$ where $\mathsf{T_1}, \mathsf{T_2} \in \mathcal{T}$,
find $\mathsf{T_3}$ such that $(\mathsf{T_1}, \mathsf{T_2}, \mathsf{T_3}) \in \Sigma_{\mathsf{R}}$ and set b:= TRUE, t:=$\mathsf{T_3}$.

(3) [Return.]
Return b and t.

Figure 7: The algorithm *CTypeInfer*

X1

IsStrongOM$(m; b)$

Input: m an *OpenMath* object.

Output: b a boolean constant such that
$b =$TRUE if m is *Strong OpenMath* and
$b =$FALSE otherwise.

(0) [Initialize]
Set b:= FALSE.
If $m = \mathbf{error}(S, A_1, \ldots, A_n)$, return.

(1) [Structural Induction on m.]
Depending on the structure of m (m is not an error object), choose one of the following:

(a) [Basic]
If m is a basic *Strong OpenMath* symbol, set b:= TRUE.

(b) [Attribution, $m = \mathbf{attribution}(A_0, S_1\ A_1,\ \ldots\ , S_n\ A_n)$]
Check *IsStrongOM*$(A_0; b_0)$. Set b:=b_0. If $b =$FALSE then return.
Otherwise, if there is S_i = `type` for $1 \leq i \leq n$, check *IsStrongOM*$(A_i; b_i)$ and set b:=b_i.
Return.

(c) [Application $m = \mathbf{application}(A_1, \ldots, A_n)$]
Set b':=TRUE, $i := 1$.
While b' and $i \leq n$ do $\{IsStrongOM(A_i; b'); i^{++}\}$.
Set b:=b'.

(d) [Binding, $m = \mathbf{binding}(A_0, v_1, \ldots, v_{n-1}, A_n)$]
Check *IsStrongOM*$(A_0; b_0)$, *IsStrongOM*$(v_0; b_1)$, ..., *IsStrongOM*$(v_{n-1}; b_{n-1})$ and *IsStrongOM*$(A_n; b_n)$.
If all $b_0 = b_1 = \ldots = b_n =$TRUE, set b:= TRUE.

(3) [Return.]
Return b.

Figure 8: The algorithm *IsStrongOM*

$\Delta \vdash \Omega : t$. Otherwise, using the type inference algorithm on the given type t, it is possible to find out if there is a type universe T such that $\Delta \vdash t : \mathsf{T}$ which asserts that t is a correct type. In this case, it is enough to verify that $t \sim t'$ holds. This is easy since the relation $\sim$ is decidable for correct *OpenMath* objects. In fact, the reduction relation is confluent and terminating follows from the same result for the Extended Calculus of Constructions. ■

Type checking and type inference algorithms are available in proof checker software like Lego or COQ [5, 6]. Should these systems become *OpenMath* compliant, they could easily provide the functionalities needed to test meaningfulness of *OpenMath* objects.

To summarize:

- *OpenMath* objects do not need to pass type checks to qualify as *OpenMath* objects;
- error and attribution objects are not tested for type correctness;
- type checking can be carried out in *OpenMath* at the cost of ignoring error and attribution objects, and making use of additional Content Dictionaries;
- no necessity for average *OpenMath* client to use it. It is just and additional feature for rigour and future automated understanding and proof checking of math;
- *Strong OpenMath* objects enable the exchange of formal mathematics between human beings, proof checkers, and computer algebra systems.

5 Conclusion

The goal of this document is to provide *OpenMath* with a level of semantical validation. This is achieved by using formal signatures in the *OpenMath* Content Dictionaries. Such signatures express types from the Extended Calculus of Constructions.

A ecc Content Dictionary

```
<CD>

<!--

        ##############################
        #                            #
        #   ECC Types for OM         #
        #                            #
        ##############################

    The ecc CD declares the symbols used in specifying the
    signature of other OM symbols according to the OpenMath
    ESPRIT deliverable 1.3.2b  "A type system for OpenMath"
    by O. Caprotti, and  A. Cohen.

    An OpenMath object which uses any of the symbols here defined,
    has to use them in accordance with these definitions.

    Initial version: O. Caprotti and H. Elbers (July 2, 1998)
    Updated: O. Caprotti (March 5, 1999)
    Official version 1.0



 -->

    <CDName> ecc </CDName>
    <CDURL> http://www.riaca.win.tue.nl/projects/OpenMath/ftp/ecc.ocd </CDURL>
    <CDExpire>  </CDExpire>
    <CDStatus> private </CDStatus>

    <Description> Declaration of symbols for types and objects
    </Description>


    <!--

        ##############################
        #                            #
        #  Definition of constants   #
        #                            #
        ##############################

    -->


    <CDDefinition>
        <Name> omtype </Name>
        <Description> The type of symbolic type symtype
        </Description>
    </CDDefinition>
```

```
<CDDefinition>
    <Name> symtype </Name>
    <Description> The type of symbolic types introduced in other CDs
    </Description>
    <Signature> omtype </Signature>
</CDDefinition>

<CDDefinition>
    <Name> integer </Name>
    <Description> The type of integers
    </Description>
    <Signature> symtype </Signature>
    <CMP> the object
          <OBJ>
          <OMATTR>
          <OMATP>
          <OMS name="type"/>
          <OMS cd="ecc" name="integer" />
          <OMATP>
          </OMATTR>
          <OMI> 0 </OMI>
          </OBJ>
          is well-typed.
    </CMP>
</CDDefinition>

<CDDefinition>
    <Name> float </Name>
    <Description> The type of floating point numbers
    </Description>
    <Signature> symtype </Signature>
    <CMP> the object
          <OBJ>
          <OMATTR>
          <OMATP>
          <OMS name="type"/>
          <OMS cd="ecc" name="float" />
          <OMATP>
          </OMATTR>
          <OMF> 1.0 </OMF>
          </OBJ>
          is well-typed.
    </CMP>
</CDDefinition>

<CDDefinition>
    <Name> string </Name>
    <Description> The type of character strings
    </Description>
    <Signature> symtype </Signature>
    <CMP> the object
          <OBJ>
          <OMATTR>
```

```
            <OMATP>
            <OMS name="type"/>
            <OMS cd="ecc" name="string" />
            <OMATP>
            </OMATTR>
            <OMSTR> hello world </OMSTR>
            </OBJ>
            is well-typed.
    </CMP>
</CDDefinition>

<CDDefinition>
    <Name> bytearray </Name>
    <Description> The type of byte arrays
    </Description>
    <Signature> symtype </Signature>
    <CMP> the object
            <OBJ>
            <OMATTR>
            <OMATP>
            <OMS name="type"/>
            <OMS cd="ecc" name="bytearray" />
            </OMATP>
            </OMATTR>
            <OMB>Hk=w2Hs3Kd9kjasd</OMB>
            </OBJ>
            is well-typed.
    </CMP>
</CDDefinition>

<CDDefinition>
    <Name> prop </Name>
    <Description> The type of propositions
    </Description>
    <Signature> symtype </Signature>
    <CMP> the object
            <OBJ>
            <OMATTR>
                   <OMATP>
                   <OMS name="type"/>
                   <OMS cd="ecc" name="prop" />
                   </OMATP>
                   <OMS cd="basic" name="true" />
            </OMATTR>
            </OBJ>
            is well-typed.
    </CMP>
</CDDefinition>


<!--

    ####################################
    #                                  #
```

```
        #  Definition of type constructors  #
        #                                   #
        #####################################

    -->


 <CDDefinition>
        <Name> PiType </Name>
        <Description> The type constructor of function space.
               It takes a list type-attributed variables and
               an OpenMath object.
        </Description>
    </CDDefinition>


 <CDDefinition>
        <Name> SigmaType </Name>
        <Description> The type constructor of cartesian products.
               It takes a list of type-attributed variables and
               an OpenMath object.
         </Description>
    </CDDefinition>


    <!--

        ######################################
        #                                    #
        #  Definition of object constructors  #
        #                                    #
        ######################################

    -->



 <CDDefinition>
        <Name> Lambda </Name>
        <Description> The abstraction constructor. It is
               followed by a list of type-attributed variables
               and an OpenMath object.
        </Description>
    </CDDefinition>


<CDDefinition>
        <Name> Pair </Name>
        <Description> The pairing constructor. It takes two
               OpenMath objects as first element and second
               element of the pair, and a third optional
               OpenMath object that represents the type of
               this pair.
        </Description>
```

```
    </CDDefinition>

<CDDefinition>
        <Name> PairProj1 </Name>
        <Description> The first projection function. It
               satisfies sigma-reduction.
        </Description>
    </CDDefinition>

<CDDefinition>
        <Name> PairProj2 </Name>
        <Description> The second projection function. It
               satisfies sigma-reduction.
        </Description>
    </CDDefinition>


</CD>
```

References

[1] Robert Constable. Personal Communication, July 1998.

[2] N.G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In M. Laudet, D. Lacombe, and L. Nolin, editors, *Symposium on Automatic Demonstration: held at Versailles/France, December 1968*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer Verlag, 1970.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[4] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford Science Publications, 1994.

[5] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual*. Department of Compter Science, University of Edinburgh, 1992.

[6] Projet Coq. *The Coq Proof Assistant: The standard library*, version 6.1 edition. Available at `http://www.ens-lyon.fr/LIP/groupes/coq`.