1 Introduction

We describe and document a variant of Automath which we have developed in the course of thinking about some ideas in the philosophy of mathematics. The Automath implementation may be viewed as a tool for building implementations of parts of the the mathematical world under the philosophical view. It is an important aspect of the philosophical view that completely general mathematical objects can be accessed by a computer program working in a finitary way (as a computer program must), and that, as we shall see, the mathematical world accessed can be the world of classical, impredicative, nonconstructive mathematics with high orders of infinity. And yet, at the same time, the underlying view is that we only deal with infinities (however large) as potential rather than actual. The moral is that classical mathematics is no less Aristotelean than constructive or impredicative mathematics, properly viewed.

An obvious eccentricity of this implementation is that the user never writes any expression with bound variables. This is not to say (as I have carelessly said at times) that this implementation contains no $\lambda$-abstractions: it does, but they are generated internally (and are displayed to the user). The system could be extended with the ability to directly enter $\lambda$-abstractions in some contexts, but a philosophical point about the nature of abstraction is made by avoiding this.

The description of the system which follows may appear quite mysterious when read by itself: I hope that reading it along with the examples of system output given in the last section will clarify matters.

2 Notation entered by the user

The notation written by the user is quite simple. The user enters (as arguments to commands yet to be discussed) names of various entities and abstractions and various type notations.

Names are either identifiers (strings of letters and digits) or identifiers followed by argument lists of names (an argument list consisting of a sequence of names separated by commas, beginning with an open parenthesis and ending with an open parenthesis). Any well-formed name can be assigned a type in a
way that will be discussed, but the syntax is just this. A future upgrade will
allow infix and prefix operators (but this will be interpretable in the obvious
way in terms of the bare notation).

Types which are entered by the user are \texttt{object, prop, and that \(P\)} where
\(P\) is any term of type \texttt{prop}. There are more types, inhabited by \(\lambda\)-abstractions,
but just as the user never directly enters a \(\lambda\)-abstraction, the user also never
directly enters the type of an abstraction. The system does display abstractions
and their types, however, so the notation for these will be discussed. The user
does type identifiers which refer directly to abstractions.

3 Entities, abstractions, and types

Inhabitants of the user-enterable types \texttt{object, prop, that \(P\)} are termed \textit{en-
tities}. Entities of type \texttt{object} are more specifically mathematical objects of a
general sort, entities of type \texttt{prop} are propositions, and entities of the type \texttt{that \(P\)}
are items of evidence for the truth of proposition \(P\) (one might be tempted
to say proofs, but it is better to say items of evidence or truth-makers for \(P\); it
is not appropriate to assume that elements of \texttt{that \(P\)} are actually proofs in a
particular formal system – though the system does allow the user to introduce
explicit assumptions to this effect if desired – making it very clear that these
are explicit additional assumptions with consequences.) A fully classical view
might be that there are only two propositions, the True (with one inhabitant)
and the False (with no inhabitants), but again this is not a built in assumption,
though it could be explicitly expressed.

The types \texttt{prop} and \texttt{that \(P\)} can also be used to extend the type system if
this is wanted: some “propositions” might be names of additional types, and
the “items of evidence” for these propositions are inhabitants of the additional
types. I could have introduced more type constructors, but for my philosophical
purposes it is better to take a minimal approach.

Functions returning entities are called abstractions. Functions which return
abstractions will not be considered. Classes are viewed as functions which re-
turn propositions (and sets will be entities associated with classes, the latter
being abstractions). The type of a function is determined by the types of the
items in its argument list: individual items in argument lists may be entities
or abstractions. The length of the argument list is fixed by the type of the
function. The type of an item in the list may depend on items appearing earlier
in the list (by means of an item earlier in the list appearing in a type \texttt{that \(P\)}
of an entity later in the list, or in the type of an abstraction appearing later in
the list in more complex ways). The type of an abstraction is represented by
notation for the type of its output followed by an argument list consisting of
pairs of a starred variable and notation for a type; each starred variable may
appear in notations for types later in the list and in the notation for the output
type. The starred variables are of course bound in the type notation. Some or
all of the types in the argument list may be abstraction types; the output type
cannot be an abstraction type.
With the description of types of abstractions, we have completely described the built-in type system. The user can add detail to the type system by specifying constructions in the type $\text{prop}$.

The automatic presumption that abstractions are entities is in our view a fruitful source of paradox. It is quite straightforward to stipulate that there are entities corresponding to a range of abstractions of interest in theories developed under this system, but there is no automatic presumption that abstractions are entities, or indeed that constructions of entities from abstractions are injective (another source of error), though again one can force the latter assumption in a particular context in a theory. It is important to know when one is making a substantial assumption!

4 Contexts, Declarations, Constructions, and Definitions

The activity of the user of this system is to write and enter declarations and definitions in various contexts. At any given moment, one is typing declarations in a working context, which has a parent context (contexts are indexed by the natural numbers; the parent of context $n + 1$ is context $n$; the working context is never context 0). The parent context is inhabited by entities and abstractions to which the user is committed, at least for the sake of argument. In the working context, the entities and abstractions declared can be viewed as hypothetical. One might think of contexts as possible worlds, but it is more accurate to think of contexts as implementing many possible worlds at once (assuming all the hypothetical objects and constructions at once that are present in the working context in a given argument is likely to lead to absurdity).

We discuss basic commands.

Declaring variables: The command

\texttt{declare identifier type}

introduces a hypothetical entity of the given type. The identifier needs to be fresh, of course (not yet declared in the working context or any iterated parent context). The type may depend on entities or abstractions declared earlier (if it is of the form $\text{that } P$.) The type will be $\text{object, prop}$, or some $\text{that } P$, an entity type not an abstraction type. Any identifier appearing in the type needs to have been declared in the working context or some iterated parent of the working context.

This command can be used quite freely: declaring a variable of a particular type does not entail ontological commitments.

Declaring constructions (abstractions or constants): The command \texttt{construct name type}

introduces a new entity or abstraction in the parent context (a new constant or operation to which one is committed). The type is a type of
entities, and any identifier mentioned in it must be declared in the working context or some iterated parent of the working context. The name is either an identifier, not yet declared, in which case an entity is declared in the parent context of that type (to be thought of as a constant whereas the entities declared in the working context are best thought of as variables) or a name with an undeclared leading identifier and an argument list made up entirely of identifiers newly declared in the working context [newly meaning, not declared in any iterated parent context] appearing in the order in which they were declared (this stipulation enforces sensible dependencies without the need for nasty recursive checks). Any non-defined identifier appearing in the type and declared in the working context must appear in the argument list. Definitions of objects new in the working context are expanded. Note that when definitions of abstractions are expanded, one gets $\lambda$-abstractions (notation for these to be discussed below). The output type has to type check successfully, which imposes nontrivial restrictions on the argument list. The leading identifier in the name (the function) is then declared as a new abstraction in the parent context, which can of course then be used to construct new names meaningful in the current context.

This command should not be used quite as blithely as the previous one, as one is committing oneself to something in the parent context. Of course commitments in the parent context are mere hypotheses in its parent context.

The output of the `construct` command includes the type of the entity or abstraction declared (so the user will see the dependent type notation with bound starred variables, though he or she never types such notation).

**Definitions:** The command `define name1 name2`
defines an entity or abstraction to whose existence one is already committed in the parent context (if the definition is well-formed). The conditions on the first name are exactly as in the `construct` command, and one declares an entity or abstraction under the same conditions. One then computes the type of the second name and assigns this type to the new identifier – in the case of an abstraction definition, one has to check that the type of the second name type-checks as a construction from the given argument list (this is the same type checking that occurs in the `construct` command, and in fact the `define` command calls the `construct` command for this purpose: all type checking is done exactly at points where the `construct` command is called). Any non-defined identifier declared in the working context which appears in the second name or anywhere in its (iterated) type declarations must appear in the argument list. Definitions of items new in the working context are expanded.

The output of the `define` command includes the type of the entity or abstraction defined and the definition of the identifier defined, as a $\lambda$-abstract in the case of an abstraction (though there are no $\lambda$s in our
notation, which is described below; it is a variant of de Bruijn’s head binder free notation but more similar to the usual variable binding conventions).

5 Context Management

We discuss higher level commands.

Opening a new context: The user command

\[ \text{Open()} \]

opens a new context, which will contain no declarations, and the erstwhile context becomes the parent context.

Closing a context and returning to the parent context: The user command

\[ \text{Close()} \]

closes the current working context and discards all its declarations; the erstwhile parent context becomes the working context. It is thus important that the construct and declare commands eliminate all information specific to declarations new in the working context (by expanding definitions for example) when they post their declarations to the parent context.

Towards sophisticated declaration management: The user command

\[ \text{Clear()} \]

discards all declarations in the working context (it can get cluttered, especially context 1, from which we have no other way to eliminate anything), but leaves us with the same parent context.

A range of commands with more subtle effects on declarations are worth considering. One could for example save the current set of declarations in the working context as an item in the parent context (note that such a set of declarations might contain saved declarations from deeper contexts). In loading such a set of declarations anew one would have to have some way of avoiding conflicts with new declarations in iterated parent contexts. One might want to be able to remove particular declarations in the working context (but one would then need to check dependencies of other declarations). One might want to be able to load the declarations implicit in an abstraction type. The current set of commands has the merit of being simple and safe; the more complicated ones envisioned would save time and space.

Other commands: We list some incidental commands.

The command

\[ \text{showdec identifier} \]

will show the declaration of the identifier (type information and any definition).
The command

\texttt{showall()}

will show all declarations, labelled with their context levels.

\texttt{Diagnostics()} turns on display of trace information.

\texttt{SetMargin} number sets the line length in the display.

6 Type Checking

Computation of types of terms and the type checking function are defined by a mutual recursion.

The type of an identifier is determined by lookup. The type of a term with an identifier applied to an argument list is determined by taking the output type of the identifier and substituting the items in the argument list for the appropriate starred variables in the corresponding positions in the argument list of the type of the identifier, subject to the condition that the type of each item in the argument list is equal to the corresponding type in the argument list of the type of the identifier after appropriate substitutions of earlier items in the argument list of the term for starred variables taken from the argument list of the type of the identifier.

The equality of types is after expansion of terms for entities or abstractions by definition and/or beta reduction as needed. The expansion is from the top down: if equal terms are obtained in the course of expansion, the process stops, and if it becomes clear in the course of reduction from the top that the terms will not become equal, the process stops.

7 Notation for $\lambda$-abstracts

A $\lambda$-term is obtained from a name, considered as a function of the identifiers newly declared in the working context which appear in it, by enclosing the term in brackets and modifying each identifier of which it is a function by preceding it with a pound sign and following it with a colon followed by its position in the argument list supplied (unstated dependencies of course cause error). In addition, any bound variable appearing in one or more brackets in the term is prefixed with an additional pound sign. Finally, if the term abstracted from is simply a single identifier, the abstraction will be represented by \texttt{Id: number}, where the number is the position of the identifier in the argument list. Bound variables can be recognized by the fact that they start with pound signs. The number of pound signs is the number of brackets one needs to count outward to the term in which they are bound, and the final number is the argument list position from which the bound variable is filled.

This is a de Bruijn level scheme, where the index assigned to a variable (actually the number of pound signs in this case) indicates how many brackets enclose it. This resembles ordinary variable binding but with a mandatory
renaming scheme. Defining beta reduction for this scheme is straightforward (but annoying as always). The user never types $\lambda$-abstractions (which in our notation involve no lambdas) but they are displayed in types and definitions. The reason that they are needed is that the names for abstractions defined in deeper contexts are forgotten when the contexts are closed (which is a good thing – we do not want our namespace cluttered with endless nonce names for functions).

There is a noteworthy general fact about my $\lambda$-abstracts which I think is harmless: they cannot actually be type checked, because arguments not used are not visible in their structure to be typed. I believe that this is harmless (because in fact the type checking is all done at construction time, with types of entities only) but some thought about this point might be useful. My immediate thought is that the thing to observe is that I could add explicit type information about all arguments to the bracketed $\lambda$-abstracts and then think about how this information would be used. It would be used in equality type checking to reject certain equations between abstractions – but would such rejections ever actually occur, and would it even be necessary to reject such identifications of they did occur? What is going on is that there is an implicit sort of subtyping between abstraction types going on here; is it actually harmful? Note though that if it is in the end seen to be harmful it can be fixed: every bracketed object gets the list of types of its arguments as an extra component, and everything that is intended will go through.

My thoughts on this last point, though certainly not final: it appears to me that the types of all abstractions embedded in an abstraction type are deducible. If an abstraction appears in applied position, it can be eliminated by beta reduction: the abstraction function does not carry out beta reductions (now it does, this is fixed). If an abstraction appears as an argument, look at the operator applied to that argument list – this will be (mod elimination of abstractions in applied position as under the previous remark) a type with a component which gives the exact type of the abstraction in question, missing arguments and all (since declared identifiers and starred variables have full type information). So it seems to be impossible for the situation to arise where two types are checked for equality and are concluded to be equal because abstractions of different types are identified by a sort of pun; this will not happen because the position in which the abstraction is embedded in a dependent type actually determines its type completely.

8 Outline of rewriting feature to be added

Add a new basic command

```
rewrite name1 name 2
```

This will introduce an object of a new kind (a rewrite rule) which will be typed as a function sending entities of type $P(name1)$ to $P(name2)$, but will have a special use. When a rewrite rule appears as an argument to a defined object, the rewriting it allows will be used in computing equality
of types. This command is analogous to \texttt{construct}; there should also be a command which allows definition of a rewrite rule from a given function sending entities of type \texttt{that }$P($name1$)$ to \texttt{that }$P($name2$)$.

Notice that since there is no provision for rewrite objects to appear in syntax, the argument places inhabited by rewrite objects will not be manifest in \texttt{\lambda}-abstractions (see the remark above about type checking and missing arguments in abstraction terms). Nonetheless, the rewrite objects are doing something important!

The rewriting feature introduces computation as a basic component of the system, and might with care serve as a foundation for making it a programming language as well as a proof checker. An \texttt{expand} command is provided which will expand definitions and carry out beta-reductions in a term; where definitions involve rewrite rules, the rewriting should be applied as well.

It occurred to me that an advanced feature which might be wanted eventually is provision for arguments which are large packages of rewrite rules.

\section{Sample output}

The program should eventually, but does not yet, take input in exactly the form of the commands it echoes here. The input is a bit more cluttered with quotation marks. But this should give an idea of what is going on.

\begin{verbatim}
declare p: prop
declare q: prop
construct And(p,q): prop
   And: construct
      prop( (*p:prop), (*q:prop) )
construct If(p,q): prop
   If: construct
      prop( (*p:prop), (*q:prop) )
construct False: prop
   False: variable
   prop

Declare the basic propositional operators.

declare pp: that p
declare qq: that q
construct AndProof(p,q,pp,qq): that And(p,q)
   AndProof: construct
      that And( (*p,*q) ((*p:prop), (*q:prop), (*pp:
      that *p), (*qq:that *q) )
\end{verbatim}
declare rr: that And(p,q)
construct AndElim1(p,q,rr): that p
  AndElim1: construct
  that *p((*p:prop),(*q:prop),(*rr:that And(*p, *q)))
construct AndElim2(p,q,rr): that q
  AndElim2: construct
  that *q((*p:prop),(*q:prop),(*rr:that And(*p, *q)))

This definition of the introduction and elimination rules for conjunction should be exactly as expected.

open 2
declare pp2: that p
construct Ded(pp2): that q
  Ded: construct
  that q((*pp2:that p))
define sameproof(pp2) as pp2: that p
  sameproof: construct
  that p((*pp2:that p))
  Id:0
close 2
construct IfProof(p,q,Ded): that If(p,q)
  IfProof: construct
  that If(*p,*q)((*p:prop),(*q:prop),(*Ded: that *p))

IfProof implements implication introduction (the deduction theorem) and takes a function (an abstraction) as an argument, as one would expect. With attention one can see here how this is done using contexts without any use of explicit terms for abstractions. Notice that declarations in deeper contexts are indented.

The declaration of sameproof is used in the next commented section.

define TautProof(p) as IfProof(p,p,sameproof):
  that If(p,p)
  TautProof: construct
  that If(*p,*p)((*p:prop))
  [IfProof(#p:0,#p:0,Id:0)]
Using the function \texttt{sameproof} defined above, the proposition $p \rightarrow p$ is proved. That it is proved is witnessed by the type of \texttt{TautProof}; notice the proof object recorded on the last line.

```
declare rr2: that If(p,q)
construct MP(p,q,pp,rr2): that q
MP: construct
  that \*q((\*p:prop),(*q:prop),(*pp:that \*p),
  (*rr2:that If(*p,*q)))
```

The rule of modus ponens. It is worth noting that there is no particular reason to think that $\text{MP}(p,q,pp,\text{IfProof}(D))$ will be $D(pp)$: one could of course introduce an assumption to this effect, if one wishes to compute with truthmakers.

```
declare absurd: that False
construct Panic(p,absurd): that p
  Panic: construct
  that \*p((\*p:prop),(*absurd:that False))
define Not(p) as If(p,False): prop
  Not: construct
  prop((\*p:prop))
  [If(#p:0,False)]
declare maybe: that Not(Not(p))
construct DNeg(p,maybe): that p
  DNeg: construct
  that \*p((\*p:prop),(*maybe:that Not(Not(*p))))
```

Basic declarations for negation. I need to add example proofs! The logic implemented here is classical; this is not an essential consequence of our framework, which is adaptable to constructive logic as well (in our opinion). In fact, we need to discuss why $\text{DNeg}$ is an innocent construction, as part of unfolding our philosophical program.

```
open 2
declare test: that And(p,q)
define line1(test) as AndElim1(p,q,test):
  that p
  line1: construct
  that p((\*test:that And(p,q)))
  [AndElim1(p,q,#test:0)]
```
define line2(test) as AndElim2(p,q,test):
    that q
    line2: construct
    that q((*test:that And(p,q)))
    [AndElim2(p,q,#test:0)]
define line3(test) as AndProof(q,p,line2(test),
    line1(test)): that And(q,p)
    line3: construct
    that And(q,p)((*test:that And(p,q)))
    [AndProof(q,p,line2(#test:0),line1(#test:0))]
close 2
define TestProof(p,q) as IfProof(And(p,q),
    And(p,q),line3): that If(And(p,q),And(q,
    p))
    TestProof: construct
    that If(And(*p,*q),And(*q,*p))((*p:prop),
    (*q:prop))
    [IfProof(And(#p:0,#q:1),And(#q:1,#p:0),
    [AndProof(#q:1,#p:0,AndElim2(#p:0,#q:1,
    ##test:0),AndElim1(#p:0,#q:1,##test:0))]]

The full proof of $p \land q \rightarrow q \land p$. Again, as expected, the type of TestProof
 tells us what has been proved, and the proof object is given as an elaborate
 abstraction term.

open 2
declare x1: object
construct P(x1): prop
    P: construct
    prop((*x1:object))
close 2
construct Forall(P): prop
    Forall: construct
    prop((*P:prop((*x1:object))))
declare x: object
declare uu: that Forall(P)
construct UI(P,x,uu): that P(x)
    UI: construct
    that *P(*x)((*P:prop((*x1:object))),(*x: object),(*uu:that Forall(*P)))
open 2
declare x1: object
construct Gen(x1): that P(x1)
Gen: construct
that P(*x1)((*x1:object))

close 2

construct UG(P,Gen): that Forall(P)
UG: construct
that Forall(*P)((*P:prop((*x1:object))),
(*Gen:that *P(*x1)((*x1:object))))

Declarations for the universal quantifier and its proof methods. Notice the use of contexts to avoid actually typing any abstraction terms.

construct Exists(P): prop
Exists: construct
prop((*P:prop((*x1:object))))
declare xxx: that P(x)
construct EI(P,x,xxx): that Exists(P)
EI: construct
that Exists(*P)((*P:prop((*x1:object))),
(*x:object),(*xxx:that *P(*x)))
declare r: prop
declare using: that Exists(P)
open 2
declare x1: object
declare xx1: that P(x1)
construct use(x1,xx1): that r
use: construct
that r((*x1:object),(*xx1:that P(*x1)))

close 2
construct EG(P,r,using,use): that r
EG: construct
that *r((*P:prop((*x1:object))),(*r:prop),
(*using:that Exists(*P)),(*use:that *r((*x1: object),(*xx1:that *P(*x1)))))

Basic declarations for the existential quantifier and its proof methods.

open 2
declare x1: object
declare x2: object
construct Rel(x1,x2): prop
Rel: construct
prop((*x1:object),(*x2:object))
define Rel2(x2) as Rel(x,x2): prop
  Rel2: construct
  prop((*x2:object))
  [Rel(x,#x2:0)]
define Rel1(x2) as Rel(x2,x): prop
  Rel1: construct
  prop((*x2:object))
  [Rel(#x2:0,x)]
close 2
define Exists2(x,Rel) as Exists(Rel2): prop
  Exists2: construct
  prop((*x:object),(*Rel:prop((*x1:object), (*x2:object))))
  [Exists([#Rel:1(#x:0,##x2:0)])]
define Forall2(x,Rel) as Forall(Rel1): prop
  Forall2: construct
  prop((*x:object),(*Rel:prop((*x1:object), (*x2:object))))
  [Forall([#Rel:1(##x2:0,#x:0)])]
open 2
declare z: object
define Exists21(z) as Exists2(z,Rel): prop
  Exists21: construct
  prop((*z:object))
  [Exists2(#z:0,Rel)]
define Forall21(z) as Forall2(z,Rel): prop
  Forall21: construct
  prop((*z:object))
  [Forall2(#z:0,Rel)]
close 2
define ForallExists(Rel) as Forall(Exists21): prop
  ForallExists: construct
  prop((*Rel:prop((*x1:object),(*x2:object))))
  [Forall([Exists2(##z:0,#Rel:0)])]
define ExistsForall(Rel) as Exists(Forall21): prop
  ExistsForall: construct
  prop((*Rel:prop((*x1:object),(*x2:object))))
  [Exists([Forall2(##z:0,#Rel:0)])]
Definitions of nested quantifiers. These require attention, to see how such things are managed in this system.

open 2
  declare hyp: that ExistsForall(Rel)
open 3
  declare x1: object
  declare hyp2: that Forall21(x1)
open 4
  declare x3: object
  define Rel1a(x3) as Rel(x3,x1): prop
    Rel1a: construct
    prop((x3:object))
    [Rel(#x3:0,x1)]
  define test(x3) as UI(Rel1a,x3,hyp2): that Rel1a(x3)
    test: construct
    that Rel1a(*x3)((x3:object))
    [UI(Rel1a,#x3:0,hyp2)]
  declare x4: object
open 5
  declare x5: object
  define Rel2a(x5) as Rel(x4,x5): prop
    Rel2a: construct
    prop((x5:object))
    [Rel(x4,#x5:0)]
  close 5
define test2(x4) as EI(Rel2a,x1,test(x4)): that Exists(Rel2a)
  test2: construct
  that Exists([Rel(*x4,#x5:0)])(*x4:object))
  [EI([Rel(#x4:0,#x5:0)],x1,test(#x4:0))]
define Prop1(x4) as Exists(Rel2a): prop
  Prop1: construct
  prop((x4:object))
  [Exists([Rel(#x4:0,#x5:0)])]
  close 4
define test3(x1,hyp2) as UG(Prop1,test2): that Forall(Prop1)
  test3: construct
  that Forall([Exists([Rel(#x4:0,#x5:0)])])
\[(\forall x_1: \text{object}) (\exists x_2: \text{object}) \] - EG(Forall121, ForallExists(\text{Rel}), hyp, test3): that ForallExists(\text{Rel})

\[\text{test4: construct that ForallExists(\text{Rel})(\forall x_1: \text{object}))}\]

The proof that \((\exists y. \forall x. P(x, y))\) implies \((\forall x. (\exists y. P(x, y)))\).

```plaintext
declare x4: object
declare x5: object
construct EqualObjects(x4, x5): prop
    EqualObjects: construct prop((\forall x_1: \text{object}), (\forall x_2: \text{object}))
open 3
declare w: object
construct Q(w): prop
    Q: construct prop((\forall w: \text{object}))
close 3
declare eq0: that EqualObjects(x4, x5)
declare YesQ: that Q(x4)
construct Subs(x4, x5, Q, YesQ): that Q(x5)
    Subs: construct that *Q(x5)((\forall x_1: \text{object}), (\forall x_2: \text{object}),
        (*Q: prop((\forall w: \text{object})), (*YesQ: that *Q(x4))))
open 3
open 4
declare e: object
construct R(e): prop
    R: construct prop((\forall e: \text{object}))
close 4
declare aa: that R(x4)
```
construct bb(R,aa): that R(x5)
    bb: construct
    that *R(x5)((*R:prop(*e:object)),(*aa:that *R(x4)))
close 3
construct Identity(x4,x5,bb): that EqualObjects(x4, x5)
    Identity: construct
    that EqualObjects(*x4,*x5)((*x4:object), (*x5:object),(*bb:that *R(*x5)((*R:prop(*e: object))),(*aa:that *R(*x4))))
open 3
open 4
declare e: object
construct R(e): prop
    R: construct
    prop(*e:object))
close 4
declare aa: that R(x4)
define eqtest(R,aa) as aa: that R(x4)
    eqtest: construct
    that *R(x4)((*R:prop(*e:object)),(*aa:that *R(x4)))
Id:1
close 3
define EqReflex(x4) as Identity(x4,x4,eqtest):
    that EqualObjects(x4,x4)
    EqReflex: construct
    that EqualObjects(*x4,*x4)((*x4:object))
[Identity(#x4:0,#x4:0,Id:1)]

Basic declarations for equality. The proof that equality is reflexive.