Computer Implementation of a Philosophy of Mathematics

M. Randall Holmes

Nov 23 2017, 1 pm

1 Introduction

This paper is an account of an attempt to provide a computer implementation of a philosophy of mathematics. The result has been an actual piece of software, the Lestrade Type Inspector, implementing a logical framework which we refer to as Lestrade. We hope that we may be forgiven for the play on literary associations of our name.

Contents

1.1 Philosophical considerations

A number of ideas fit together to form the philosophy which leads to Lestrade: we will focus on the idea, going back to Aristotle, that infinities are potential rather than actual. This is combined with the idea that mathematical proofs

are themselves mathematical objects which need to be taken into account in our metaphysics: this is embodied in a version of the Curry-Howard isomorphism.

This might sound like a story which will end in a constructivist account of mathematics. It does not, or more accurately, doesn't have to. Lestrade supports the usual classical mathematics of Dedekind, Peano, Cantor, Zermelo and so forth. The usual set theory *ZFC* with classical logic is readily implemented under Lestrade. The interpretation of what we are talking about when we reason in classical set theory may be seen to be somewhat different. Of course, constructive theories can also readily be implemented under Lestrade.

We will begin our critique of mathematics based on the idea that infinities should be potential rather than actual by considering *functions*. The usual interpretation of a function in modern classical set theory is as a usually infinite table of values given all at once. We discover what $f(x)$ is by finding the element y such that $\langle x, y\rangle \in f$. This concept competes informally (as, for example, in pedagogy) with an older idea: a function is a rule or method which gives an output when an input (of the right sort) is presented to it. For the student of calculus, the function $(x \mapsto x^2 + 1)$ is embodied in the finite expression $x^2 + 1$ which tells you what to do to the input x (presumably a real number) to get the outcome. Mathematicians have studied this sort of embodiment of functions as expressions with slots in them for the inputs, notably in the λ-calculus of Church, though Frege's treatment of functions (which led to paradox due to inattention to types) and Russell's treatment of propositional functions in *Principia Mathematica* are older, and similar in spirit.

We come up against the actual versus the potential character of infinite totalities in logic when considering the universal quantifier. Is $(\forall x \in D : \phi(x))$ best understood as asserting $\phi(a)$ for each $a \in D$, in effect asserting an infinitary conjunction the size of D? We instead take the view that the meaning of $\phi(x)$ is that we can assert $\phi(a)$ whenever an $a \in D$ is presented to us. This is embodied in a treatment using suitable sorted objects and functions: our evidence for $(\forall x \in D : \phi(x))$ is not understood as a conglomeration of evidence for each $\phi(a)$ (a possibly vast actual infinity of items of evidence) but as a function F which, given any $a \in D$, returns a value $F(a)$ which is evidence for $\phi(a)$, under our modest concept of a function as an entity which returns an output when presented with an input.

We maintain a proper respect for the idea that entities come in various sorts. We also remember that functions (or sets) can cause difficulties if we do not properly pay attention to sorts. The propositional function which takes absolutely any unary predicate $\phi(x)$ (itself a function from the universe to propositions) to $\neg\phi(\phi)$ (which we naively suppose makes sense because we are considering a predicate ϕ applicable to all entities whatsoever) may be called ρ in honor of Russell, and then $\rho(\rho)$ sadly expands to $\neg\rho(\rho)$. Oh dear.

We are also attentive to the possibility of dependent sorting of functions: the example of the universal quantifier supports this. As evidence for the proposition $(\forall x \in D : \phi(x))$ we postulate a function F which takes as input an inhabitant a of D and provides as output evidence for the assertion $\phi(a)$, which we will see we regard as an inhabitant of a sort (`that` $\phi(a)$): this output sort depends on

the value a of the input.

1.2 Compact formal description of the framework (not without philosophical considerations)

This subsection contains an essentially complete description of the framework, though it may be very hard to digest it without looking ahead at actual examples of Lestrade constructions in the following sections.

1.2.1 Sorts

We outline our system of sorts, which implements some of our philosophical prejudices. The things we talk about in Lestrade we call *entities* when we are being entirely noncommittal about their nature. Entities are partitioned into two species, *objects* and *functions*, each of the species being further partitioned into sorts.

We enumerate the object sorts.

We provide a sort `prop` of propositions, and for each p of type `prop` we provide a sort (`that` p) of evidence for p: if pp is of sort `that` p, we know that p is true. It might seem that we should rather call the inhabitants of evidence sorts (`that` p) "proofs" of p, but this involves committing to a particular philosophical view. When we assume p for the sake of argument, we postulate evidence for p (an element of `that` p): to presume that such evidence must actually be a *proof* is to presume a constructivist view.

Some mathematicians want to consider a world of mathematical objects which is unsorted, and some want to consider a world of mathematical objects organized into sorts. We provide support for both views. Lestrade provides a sort `obj` of untyped mathematical objects, a sort `type` of "type labels", and for each type label τ a sort (`in` τ) whose inhabitants we term "objects of type τ". Of course, whenever a new proposition p is constructed, we get a new sort `that` p and whenever a new type label τ is constructed we get a new sort (`in` τ), these sorts being properly called "types" in Lestrade usage.

That we separate propositions and evidence for propositions from types and their inhabitants does not mean that we do not recognize the parallelism between them embodied in the Curry-Howard isomorphism, of which we will have more to say. We are acknowledging pragmatically that sorts inhabited by proofs and sorts inhabited by objects are used differently, and that in a particular theory we may want to treat propositions and their inhabitants uniformly in a way in which we do not want to treat types and their inhabitants. The same approach was taken in Automath, which we regard as the parent of Lestrade.

We also make the side remark that we refer to the objects in the sort `type` as "type labels" rather than "types" because we want to discourage thinking of these objects as collections with their inhabitants as members, and also because the type proper associated with a type label τ is the sort (`in` τ): we systematically avoid any coincidence in notation or terminology for entities and sorts.

We now describe the function sorts. A function sort takes the form

$$[(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (- : \tau)].$$

In this notation the x_i's are variables (bound in the notation) with each x_i of sort τ_i (which may be either an object or a function sort). The function is understood to take arguments t_i of the sorts indicated by τ_i and to present output of sort τ, an object sort. Further, each sort τ_j may depend on variables x_i with $i < j$, and the output sort τ may depend on any or all of the input variables x_i.

At this point we have described the entire sort system of Lestrade. The rest of the story is postulation of new objects and functions of previously given sorts (which may via the constructors `that` and `in` generate new sorts) and determination of the consequences of postulating these constructions.

The very compact description of function sorts embodies a number of philosophical design decisions. In Lestrade, we do not view functions as "first-class entities": that status belongs to objects. The status of functions might be analogous to that of proper classes in familiar set theories which allow such, with the further refinement that a class (non-set) would further be viewed as distinct from a set with the same extension, and obtained from it by an explicit construction: the class would be a function and the set a correlated object. When we declare a function, we are always declaring a function which takes arguments (objects and/or functions) and returns an object. This may be thought of as reversing the "currying" operation on functions as far as possible. When we define a function, it is always via an expression for an object containing object and/or function parameters, in keeping with our rule or expression based metaphorical view of functions. This is not to say that we regard the possible action of functions as limited by our means of forming expressions: primitive declarations of functions simply declare a black box (an oracle as it were) that takes arguments of given dependent sorts and returns an object output of a given dependent sort. Lestrade does not assume (though it may be able to express, and therefore accept as user postulates in theories) validity of arguments about functions based on induction on the structure of the expressions we can currently define: it is the default position of Lestrade that functions yet to be postulated might have unexpected behavior, unless this is specifically excluded by user declared postulates.

It is also worth noting that we have essentially presented dependently typed functions as the *only* built in type construction. We do not provide product or union types, nor do we provide the more general inductive or co-inductive types, though all of these can be implemented in the Lestrade framework with suitable declarations by the user. In this, Lestrade is similar to Automath.

1.2.2 The Lestrade context: the system of "moves"

We want to maintain that we are never considering more than a concrete finite collection of actual entities at any point in a Lestrade construction. We achieve generality not by supposing that we can make propositions about all entities (of

a given sort) at once, but by a device which allows us to introduce "arbitrary" entities of a sort and speak about them, then export what can be shown about "arbitrary" entities to any specific entity of that sort that we may construct. This is handled by the device of "moves" which we now describe.

The declarations at any particular moment in a Lestrade construction are organized into moves $0, 1, \ldots, i, i+1$, where i is a concrete finite number. There are always at least two moves (so all four of the listed moves are present, though they may not all be distinct). Move 0 contains the entities to which the user is permanently committed. Moves $0, 1, \ldots, i$ are inhabited by entities which the user currently takes to be constant. Move i is referred to as the "last move" or "the current move". Move $i + 1$ is referred to as "the next move". About objects at the next move, one knows no more than can be divined from their sorts; they can be thought of as "arbitrary" or "variable". On each move, there is an order, which is the order in which the entities were declared: an entity or its sort can only depend on entities appearing at earlier moves or appearing in the same move earlier in the order on that move.

We can modify the move structure. We can open a new move, incrementing the parameter i, creating a new move $i + 1$ with no declarations in it, and construing the former next move as the new last move. We can close the next move by decrementing i (if i is positive), discarding all declarations in the next move entirely, and reconstruing the former last move as the new next move. We can at any point clear all declarations in the next move (this last is provided as a primitive operation because declarations in move 1 would otherwise be permanent). A refinement that we will not explore now is the ability to save moves by name before closing them, so that they can later be reopened in the same context if desired.

1.2.3 Descriptions of the basic user commands

The way this actually works is best described operationally. The user can execute a number of operations in the basic Lestrade model.

A variable, in all contexts, means an entity which is declared but not defined, and whose declaration appears in the next move.

The user can declare a new object variable in the next move at any point. This new object is placed last in the order on the next move. The sort of this object may depend of course on anything already declared, including variables already declared in the same move.[1]

The user may declare a new function, taking as arguments object and function variables in the next move given in the order in which they were declared (this is a cheap way to enforce dependency conditions) and giving output of a given object type. The argument list must include all variables on which the sort of the output or any of the sorts of the inputs depend.[2] This function is

[1] The same Lestrade command can be used to declare a function variable in the next move, but this is not part of the basic model. We will see that the basic model allows the declaration of function variables with some indirection.

[2] We will very shortly see that some arguments can be left implicit, but this is not part of

added to the last move, at the end of the order on the last move: we postulate the function as a new (relative) primitive constant in world i, not as a variable in move $i+1$ (we will see later how to introduce function variables in the next move). This is how to add axioms and primitive notions to a Lestrade theory.

In the special case where the argument list is null, we are in effect declaring an object constant in the last move. This is the only way to declare an object in move 0 as a primitive.

The user may define a function, giving an argument list of variables from the next move (in the order in which they were declared) and providing an object term depending only on those variables as the output, from which the sort of the output is of course deduced. The argument list must include all variables on which the object term, its sort, or the sorts of any of the inputs depend.[3] This function is added to the last move at the end of the order on the last move.

This is a natural point at which to briefly (as possible) remark on the structure of complex object terms and how to type them (though complex object terms could already have appeared in declarations of new functions or even object variables, as arguments of the `that` and `in` constructors). Terms in the basic Lestrade language are either atomic (and in this case have been declared and have sort which can be looked up) or application terms $f(t_1, \ldots, t_n)$, where f is a function and each t_i is either an object term or an atomic function term. The sort of f being $[(x_1 : \tau_1), \ldots, (x_m : \tau_m) \Rightarrow (-, \tau)]$, it is required for the term to be well-formed that $m = n$, and further the sort of t_1 must be equivalent to τ_1 mod expansion of definitions (if not, the term is ill-formed because ill-sorted) and then the sort of $f(t_1, \ldots, t_n)$ will be $\tau[t_1/x_1]$ if $n = 1$ and otherwise will be the same as the sort of $f^*(t_2, \ldots, t_n)$, where f^* is postulated with sort $[(x_2 : \tau_2[t_1/x_1]), \ldots, (x_n : \tau_n[t_1/x_1]) \Rightarrow (- : \tau[t_1/x_1])]$.

A function defined by a definition $f(x_1, \ldots, x_n) = T$, where each x_i has sort τ_i and T has sort τ, with the usual allowed dependencies of sort, has sort $[(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (-, \tau)]$ and is represented internally as a λ-term

$$[(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (T : \tau)]$$

(where T of course may depend on any or all of the x_i's) and $f(t_1, \ldots, t_n)$ will reduce to $T[t_1/x_1]$ if it sorts correctly and $n = 1$, and otherwise (if it types correctly) reduces to $f^*(t_2, \ldots, t_n)$ (which we then continue to reduce), where f^* represents

$$[(x_2 : \tau_2[t_1/x_1]), \ldots, (x_n : \tau_n[t_1/x_1]) \Rightarrow (T[t_1/x_1] : \tau[t_1/x_1])].$$

In all of these notations, $U[t/x]$ represents the result of substitution of the term t for the variable x in the term U, with the usual issues due to the binding of variables in function sorts and λ-terms. [4]

the basic model.

[3] subject to the same remark about implicit arguments made in the previous paragraph.

[4] Lestrade handles bound variable collision issues by indexing each bound variable appearing in a given function sort term or anonymous function term (λ-term) with a numerical index shared by all variables bound at the top level in that context, and not appearing in any other term in the context.

In the special case where the argument list is null, we are in effect defining an object notation in the last move. To prove a theorem p is to define an object of sort `that` p, under Lestrade, so proof as an activity has been handled, as a species of definition. It is possible to introduce objects by definition in move 0, and in fact theorems would be expected to be objects of evidence types defined at move 0. Objects of evidence types defined at moves of higher index presumably depend on objects which are variable from the move 0 standpoint, representing postulated objects or assumed hypotheses.

We can arrange to have function variables in the next move. This is arranged by opening a new move, declaring variables of the desired input types, declaring a new function of the desired sort, then closing the new move, leaving a non-defined function declared in the next move, usable as a function variable. (Advanced features allow one-line declarations of function variables without opening a new move.)

When a defined constant or function is present in the next move (it was of course necessarily defined at a point when the current next move was the last move) and it is used in a definition or declaration, it is necessary to eliminate the defined notion from any declaration or definition being recorded in the last move (because any information about the meaning of the defined identifier would vanish if the next move were closed, which we are free to do at any time). This is done in the case of constants and in the case of functions appearing in applied position by expanding the definitions. Functions appearing as arguments are replaced by anonymous λ-terms. In the basic Lestrade model, there is no reason for a user even to be able to write an anonymous λ-term: it is always possible using the move system to introduce a function as it were by a definition $f(x) = x^2 + 1$ rather than via a notation like $(x \mapsto x^2 + 1)$, though it is possible (an advanced feature added on top of the basic model) for a user to enter λ-terms as arguments in Lestrade expressions. The strategy would then be to always refer to the function using the name f: when this name passes out of scope due to the move in which it was declared being closed, any occurrences of f as an argument would be replaced by an automatically generated λ-term: these would only appear in recorded sort information. Where a substitution would put an automatically generated λ-term in applied position, a definitional expansion occurs (a β-reduction, as it were); there is no provision for a λ-term to appear anywhere but in argument position either in user-entered text or in Lestrade output.

At this point we have described all the essential operations in the basic model of Lestrade. We claim that this is a framework in principle adequate for the practice of mathematics in any theory whatsoever. We will support this claim by presenting examples, and we will also try to explain what the possibility of implementing classical theories under Lestrade might tell us about alternative ways of interpreting what these theories are telling us. We should add that we are ourselves quite friendly to a classical Platonist interpretation of mathematics: we are very interested in the fact that theories usually taken as presupposing the Platonist view can apparently be supported on the metaphysically much more parsimonious basis of (our interpretation of) Lestrade.

1.2.4 The matter of terms with bound variables

The basic Lestrade model does not require the user ever to use terms with bound variables. They are supported, however. A user may enter function terms as arguments appearing in Lestrade object terms, and the user may declare a function variable with type a given function sort term. The form of a user-entered function term is $[x_1, \ldots, x_n \Rightarrow T]$, where the x_i's are variables appearing in the next move and T is an object term, and the form of a user-entered function sort term is $[x_1, \ldots, x_n \Rightarrow \tau]$, where the x_i's are variables appearing in the next move and τ is an object sort term. Notice that no form of sort inference or explicit assignment of sorts to variables in a term is supported.

The reference of a function term $[x_1, \ldots, x_n \Rightarrow T]$ is the same as the reference of a new atomic function term with declaration information

$$[(x_1^* : \tau_1^*), \ldots, (x_n^* : \tau_n^*) \Rightarrow (T^*, \tau^*)],$$

where T^* and τ^* are obtained by replacing each x_i with x_i^* in T and τ respectively, and each τ_j^* is obtained by replacing each x_i for which $i < j$ with x_i^* in τ_j (notice that τ_1^* is simply τ_1; we write it with a star just because everything is to be starred). There is no restriction on the order in which the variables x_i appear in the next move: correct dependency structure is enforced by the starring process. Such a term can appear only as an argument: when a substitution would put it in applied position, definitional expansion takes place (the same thing can happen with automatically generated λ-terms created when function identifiers go out of scope). Similarly, the reference of a function term $[x_1, \ldots, x_n \Rightarrow \tau]$ is the same as that of the function sort notation $[(x_1^* : \tau_1^*), \ldots, (x_n^* : \tau_n^*) \Rightarrow (-, \tau^*)]$. The starring process signals the peculiarity of variable binding terms from the standpoint of the basic Lestrade model: variables in the next move are being in effect cloned to serve as variables in a move or moves of higher index which are not officially actually opened.

The role of explicitly bound variables in the move model is a bit unexpected. The difficulty is that these variables, if one constructed the same function terms or function sort terms in the basic model, would actually refer to variables in moves of higher index (one additional move for each nested bracket in such a term). In fact, careful consideration should reveal that this is what is actually happening: one should consider the bound variables to be variables from further moves "cloned" from variables made available in the next move.

It is demonstrable that anything constructible using bound variable terms can be constructed in the basic model, and that anything constructible in the basic model can be constructed using bound variable terms without ever opening a move with higher index than 1.

2 Logic

In this section we discuss the implementation of usual notions of logic and proof strategies under Lestrade. This and subsequent sections include actual dialogue with the Lestrade Type Inspector, which may help to make the very abstract discussion in the first section more concrete.

We begin by providing ourselves with some propositional variables.

```
Lestrade execution:

declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}
```

The `declare` command implements the user operation of declaring object variables in the next move.

```
Lestrade execution:

construct & p q prop

>> &: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


construct -> p q prop

>> ->: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}
```

We declare the operations of conjunction and implication. At this point all we know of them is the types of their input and output. The `construct` command implements the user operation of declaring a new function or object.

```
Lestrade execution:

declare pp that p
```

```
>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}


declare pq that p & q

>> pq: that (p & q) {move 1}


construct Conjunction pp qq that p & q

>> Conjunction: [(.p_1:prop),(pp_1:that .p_1),
>>        (.q_1:prop),(qq_1:that .q_1) => (---:
>>        that (.p_1 & .q_1))]
>>   {move 0}


construct Simplification1 pq that p

>> Simplification1: [(.p_1:prop),(.q_1:prop),
>>        (pq_1:that (.p_1 & .q_1)) => (---:that
>>        .p_1)]
>>   {move 0}


construct Simplification2 pq that q

>> Simplification2: [(.p_1:prop),(.q_1:prop),
>>        (pq_1:that (.p_1 & .q_1)) => (---:that
>>        .q_1)]
>>   {move 0}
```

In these lines, we declare the basic logical operations handling conjunction. This reveals some features in Lestrade which are not described in the basic model. Note that the term language does handle infix notation. Note that `Conjunction` and `Simplification1/2` have additional deduced arguments not shown in the command lines declaring them. From evidence `pp` for `p` and evidence `qq` for `q`, we do get a proof of `p & q`, but Lestrade notices that `Conjunction` is also a function of `p` and `q`: but from the sorts of any arguments `pp` and `qq` supplied, we can deduce the values of the implicit arguments

`p` and `q`.

```
Lestrade execution:

define Conjcomm pq : Conjunction(Simplification2 pq,Simplification1 pq)

>> Conjcomm: [(.p_1:prop),(.q_1:prop),(pq_1:
>>        that (.p_1 & .q_1)) => ((Simplification2(pq_1)
>>        Conjunction Simplification1(pq_1)):that
>>        (.q_1 & .p_1))]
>>   {move 0}
```

Here is our first exercise in proof, presenting commutativity of conjunction as a rule of inference. The `define` command implements the user operation of defining a function or object. It is worth noting in reading the Lestrade response in this dialogue that Lestrade will always write a function with two inputs (the first of which is not a function) in infix notation: `Conjunction` is used as an infix in the reported notation for the type of `Conjcomm`.

```
Lestrade execution:

declare pthenq that p -> q

>> pthenq: that (p -> q) {move 1}


construct Mp pp pthenq that q

>> Mp: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),
>>        (pthenq_1:that (.p_1 -> .q_1)) => (---:
>>        that .q_1)]
>>   {move 0}
```

Here is the logical rule of *modus ponens*, our basic rule for using an implication. The rule for proving an implication, which we give next, is metaphysically more profound.

```
Lestrade execution:

open

     declare pp2 that p

>>       pp2: that p {move 2}


     construct Ded pp2 that q

>>       Ded: [(pp2_1:that p) => (---:that q)]
>>         {move 1}


     close

construct Deduction Ded that p->q

>> Deduction: [(.p_1:prop),(.q_1:prop),(Ded_1:
>>         [(pp2_2:that .p_1) => (---:that .q_1)])
>>         => (---:that (.p_1 -> .q_1))]
>>   {move 0}


declare ded2 [pp=>that q]

>> ded2: [(pp_1:that p) => (---:that q)]
>>   {move 1}


construct Deduction2 ded2 that p->q

>> Deduction2: [(.p_1:prop),(.q_1:prop),(ded2_1:
>>         [(pp_2:that .p_1) => (---:that .q_1)])
>>         => (---:that (.p_1 -> .q_1))]
>>   {move 0}
```

The `open` command carries out the user operation of opening a new next move. `close` closes this move. In between the `open` and `close` command we see the construction of a function from evidence for p to evidence for q. The primitive function `Deduction` takes any such function to evidence for $p \to q$. We do not simply identify proofs of $p \to q$ with functions from proofs of p to proofs of q, as is often done, because our sort system forbids us to identify objects and

functions.

This is our first example of the process of declaring a function variable.

I then demonstrate how advanced features allow demonstration of an equivalent function `Deduction2` without opening a new move (by using one-line declaration of a function variable using a function sort term with variable binding).

```
Lestrade execution:

open

     declare pp2 that p

>>       pp2: that p {move 2}


     define propid pp2 :  pp2

>>       propid: [(pp2_1:that p) => (pp2_1:that
>>              p)]
>>         {move 1}


     close

define Taut1 p : Deduction propid

>> Taut1: [(p_1:prop) => (Deduction([(pp2_2:
>>              that p_1) => (pp2_2:that p_1)])
>>         :that (p_1 -> p_1))]
>>    {move 0}


define Taut2 p : Deduction [pp => pp]

>> Taut2: [(p_1:prop) => (Deduction([(pp_2:that
>>              p_1) => (pp_2:that p_1)])
>>         :that (p_1 -> p_1))]
>>    {move 0}


open

     declare pq1 that p & q

>>       pq1: that (p & q) {move 2}
```

```
     define conjcomm pq1 : Conjcomm pq1

>>       conjcomm: [(pq1_1:that (p & q)) => (Conjcomm(pq1_1):
>>              that (q & p))]
>>         {move 1}


     close

define Conjcomm2 p q :  Deduction conjcomm

>> Conjcomm2: [(p_1:prop),(q_1:prop) => (Deduction([(pq1_2:
>>              that (p_1 & q_1)) => (Conjcomm(pq1_2):
>>              that (q_1 & p_1))])
>>         :that ((p_1 & q_1) -> (q_1 & p_1)))]
>>   {move 0}
```

In this paragraph, we first prove the tautology $P \to P$ in two different ways. The first proof follows the basic model (except for taking advantage of implicit argument inference). The second proof uses a user-entered λ-term. There is a certain philosophical tension in the use of the λ-term: it reads the types of its bound variables from the types of the variables of the same shapes in move 1, but as we can see from the explicit version, it really should be reading them from move 2, which is never opened. The idea is that one should suppose that variables are being cloned from the next move into deeper moves never actually opened in the course of reading a λ-term.

We then prove $(P \wedge Q) \to (Q \wedge P)$. A subtle point is that we could not have issued the command `Deduction Conjcomm` above, even though superficially it might seem that `Conjcomm` does exactly the same thing as `conjcomm`. The issue is that `Conjcomm` also has p and q as hidden arguments, so it is not actually of the right sort to be fed to `Deduction`.

```
Lestrade execution:

clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop
```

```
>> B: prop {move 1}


declare C prop

>> C: prop {move 1}


% prove ((A -> B) & (B -> C)) -> (A -> C)

open

     declare hyp1 that (A -> B) & (B -> C)

>>      hyp1: that ((A -> B) & (B -> C)) {move
>>        2}


     define line1 hyp1 : Simplification1 hyp1

>>      line1: [(hyp1_1:that ((A -> B) & (B
>>             -> C))) => (Simplification1(hyp1_1):
>>             that (A -> B))]
>>        {move 1}


     define line2 hyp1 : Simplification2 hyp1

>>      line2: [(hyp1_1:that ((A -> B) & (B
>>             -> C))) => (Simplification2(hyp1_1):
>>             that (B -> C))]
>>        {move 1}


     % now suppose A to prove A -> C

     open

          declare hyp2 that A

>>           hyp2: that A {move 3}


          define line3 hyp2 : Mp hyp2 (line1 hyp1)

>>           line3: [(hyp2_1:that A) => ((hyp2_1
```

```
>>                  Mp line1(hyp1)):that B)]
>>             {move 2}


          define line4 hyp2 : Mp (line3 hyp2, line2 hyp1)

>>           line4: [(hyp2_1:that A) => ((line3(hyp2_1)
>>                  Mp line2(hyp1)):that C)]
>>             {move 2}


          close

     define line5 hyp1 : Deduction line4

>>      line5: [(hyp1_1:that ((A -> B) & (B
>>             -> C))) => (Deduction([(hyp2_2:
>>                  that A) => (((hyp2_2 Mp line1(hyp1_1))
>>                  Mp line2(hyp1_1)):that C)])
>>             :that (A -> C))]
>>        {move 1}


     close

define Transimp A B C : Deduction line5

>> Transimp: [(A_1:prop),(B_1:prop),(C_1:prop)
>>        => (Deduction([(hyp1_2:that ((A_1 ->
>>             B_1) & (B_1 -> C_1))) => (Deduction([(hyp2_3:
>>                  that A_1) => (((hyp2_3 Mp
>>                  Simplification1(hyp1_2)) Mp
>>                  Simplification2(hyp1_2)):that
>>                  C_1)])
>>             :that (A_1 -> C_1))])
>>        :that (((A_1 -> B_1) & (B_1 -> C_1))
>>        -> (A_1 -> C_1)))]
>>   {move 0}
```

The text above demonstrates how the move structure of Lestrade can be used to implement argument under hypotheses.

Lestrade execution:

```
define <-> A B :  (A -> B) & B -> A

>> <->: [(A_1:prop),(B_1:prop) => (((A_1 ->
>>         B_1) & (B_1 -> A_1)):prop)]
>>   {move 0}
```

We move next to disjunction.

```
Lestrade execution:

clearcurrent

declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


construct V p q prop

>> V: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


declare pp that p

>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}


construct Addition2 p qq that p V q

>> Addition2: [(p_1:prop),(.q_1:prop),(qq_1:
>>         that .q_1) => (---:that (p_1 V .q_1))]
>>   {move 0}
```

```
construct Addition1 q pp that p V q

>> Addition1: [(q_1:prop),(.p_1:prop),(pp_1:
>>        that .p_1) => (---:that (.p_1 V q_1))]
>>   {move 0}


declare pq that p V q

>> pq: that (p V q) {move 1}


declare r prop

>> r: prop {move 1}


declare case1 that p->r

>> case1: that (p -> r) {move 1}


declare case2 that q->r

>> case2: that (q -> r) {move 1}


construct Cases pq case1 case2 that r

>> Cases: [(.p_1:prop),(.q_1:prop),(pq_1:that
>>        (.p_1 V .q_1)),(.r_1:prop),(case1_1:
>>        that (.p_1 -> .r_1)),(case2_1:that (.q_1
>>        -> .r_1)) => (---:that .r_1)]
>>   {move 0}
```

We present constructive rules for negation.

```
Lestrade execution:

construct ?? prop

>> ??: prop {move 0}
```

```
define ~p : p -> ??

>> ~: [(p_1:prop) => ((p_1 -> ??):prop)]
>>   {move 0}


declare absurd that ??

>> absurd: that ?? {move 1}


construct Panic p absurd : that p

>> Panic: [(p_1:prop),(absurd_1:that ??) =>
>>        (---:that p_1)]
>>   {move 0}
```

?? is a canonical false statement, which we will write $\perp$ when typeset. We define $\neg P$ as $P \rightarrow \perp$ and adopt the rule that a false statement implies anything.

And we indicate how to make our logic classical.

```
Lestrade execution:

declare maybe that ~ ~ p

>> maybe: that ~(~(p)) {move 1}


construct Dneg maybe that p

>> Dneg: [(.p_1:prop),(maybe_1:that ~(~(.p_1)))
>>        => (---:that .p_1)]
>>   {move 0}
```

There follows homage to Aristotle.

```
Lestrade execution:

clearcurrent

declare p prop
```

```
>> p: prop {move 1}


% Prove p V ~p

% start by supposing otherwise

open

     declare hyp1 that ~(p V ~p)

>>      hyp1: that ~((p V ~(p))) {move 2}


     % show ~ p

     open

          declare hyp2 that p

>>           hyp2: that p {move 3}


          define line1 hyp2 : Addition1 ~ p hyp2

>>           line1: [(hyp2_1:that p) => ((~(p)
>>                  Addition1 hyp2_1):that (p
>>                  V ~(p)))]
>>             {move 2}


          define line2 hyp2 : Mp (line1 hyp2, hyp1)

>>           line2: [(hyp2_1:that p) => ((line1(hyp2_1)
>>                  Mp hyp1):that ??)]
>>             {move 2}


          close

     define line3 hyp1 :  Deduction line2

>>      line3: [(hyp1_1:that ~((p V ~(p))))
>>             => (Deduction([(hyp2_2:that p)
>>                  => (((~(p) Addition1 hyp2_2)
>>                  Mp hyp1_1):that ??)])
```

```
>>             :that (p -> ??))]
>>        {move 1}


     define line4 hyp1 : Addition2 (p, line3 hyp1)

>>      line4: [(hyp1_1:that ~((p V ~(p))))
>>             => ((p Addition2 line3(hyp1_1)):
>>             that (p V (p -> ??)))]
>>        {move 1}


     define line5 hyp1 : Mp (line4 hyp1, hyp1)

>>      line5: [(hyp1_1:that ~((p V ~(p))))
>>             => ((line4(hyp1_1) Mp hyp1_1):that
>>             ??)]
>>        {move 1}


     close

define excmid0 p : Deduction line5

>> excmid0: [(p_1:prop) => (Deduction([(hyp1_2:
>>             that ~((p_1 V ~(p_1)))) => (((p_1
>>             Addition2 Deduction([(hyp2_3:that
>>                  p_1) => (((~(p_1) Addition1
>>                  hyp2_3) Mp hyp1_2):that ??)]))
>>             Mp hyp1_2):that ??)])
>>        :that (~((p_1 V ~(p_1))) -> ??))]
>>   {move 0}


define Excmid p : Dneg (excmid0 p)

>> Excmid: [(p_1:prop) => (Dneg(excmid0(p_1)):
>>        that (p_1 V ~(p_1)))]
>>   {move 0}
```

The attentive reader should be able to tell from the proof above that Lestrade in many places demonstrates that it knows that $P \rightarrow \bot$ and $\neg P$ are the same thing. We do though want to have the ability to put a result that we prove in the form that we expect.

```
Lestrade execution:

clearcurrent

declare p prop

>> p: prop {move 1}


declare pp that p

>> pp: that p {move 1}


define Fixprop p pp : pp

>> Fixprop: [(p_1:prop),(pp_1:that p_1) => (pp_1:
>>        that p_1)]
>>   {move 0}


declare notso [pp => that ??]

>> notso: [(pp_1:that p) => (---:that ??)]
>>   {move 1}


define Negintro notso : Fixprop (~p,Deduction notso)

>> Negintro: [(.p_1:prop),(notso_1:[(pp_2:that
>>              .p_1) => (---:that ??)])
>>        => ((~(.p_1) Fixprop Deduction(notso_1)):
>>        that ~(.p_1))]
>>   {move 0}
```

The rule `Negintro` is actually a specialization of `Deduction`; `Fixprop` is used to coerce the output into the form $\neg P$ instead of $P \rightarrow \bot$. The idea behind `Fixprop` is that it reads the form of its output from its first argument, while the actual proof presented to it as its second argument may prove something whose type looks different but matches the first argument. We also used a fancy expression for a function sort to avoid opening a move. The expression `[pp => that ??]` represents the type of a function with a single input of the type of `pp` (read from the next move) to an object of type `that ??`.

Now we begin the treatment of quantification. We present the declarations

in two styles, one with use of variable binding terms and one without.

```
Lestrade execution:

clearcurrent

open

     declare x1 obj

>>       x1: obj {move 2}


     construct Pred x1 prop

>>       Pred: [(x1_1:obj) => (---:prop)]
>>         {move 1}


     close

construct Forall Pred prop

>> Forall: [(Pred_1:[(x1_2:obj) => (---:prop)])
>>         => (---:prop)]
>>   {move 0}


open

     declare x1 obj

>>       x1: obj {move 2}


     construct univev1 x1 that Pred x1

>>       univev1: [(x1_1:obj) => (---:that Pred(x1_1))]
>>         {move 1}


     close

construct Ug univev1 that Forall Pred

>> Ug: [(.Pred_1:[(x1_2:obj) => (---:prop)]),
```

```
>>        (univev1_1:[(x1_3:obj) => (---:that
>>             .Pred_1(x1_3))])
>>        => (---:that Forall(.Pred_1))]
>>   {move 0}


declare univev2 that Forall Pred

>> univev2: that Forall(Pred) {move 1}


declare x obj

>> x: obj {move 1}


construct Ui  univev2 x that Pred x

>> Ui: [(.Pred_1:[(x1_2:obj) => (---:prop)]),
>>        (univev2_1:that Forall(.Pred_1)),(x_1:
>>        obj) => (---:that .Pred_1(x_1))]
>>   {move 0}


clearcurrent

declare x1 obj

>> x1: obj {move 1}


declare Pred [x1 => prop]

>> Pred: [(x1_1:obj) => (---:prop)]
>>   {move 1}


construct Forall2 Pred prop

>> Forall2: [(Pred_1:[(x1_2:obj) => (---:prop)])
>>        => (---:prop)]
>>   {move 0}


declare univev1 [x1 => that Pred x1]
```

```
>> univev1: [(x1_1:obj) => (---:that Pred(x1_1))]
>>   {move 1}


construct Ug2 univev1 : that Forall Pred

>> Ug2: [(.Pred_1:[(x1_2:obj) => (---:prop)]),
>>        (univev1_1:[(x1_3:obj) => (---:that
>>             .Pred_1(x1_3))])
>>        => (---:that Forall(.Pred_1))]
>>   {move 0}


declare univev2 that Forall Pred

>> univev2: that Forall(Pred) {move 1}


declare x obj

>> x: obj {move 1}


construct Ui2 univev2, x that Pred x

>> Ui2: [(.Pred_1:[(x1_2:obj) => (---:prop)]),
>>        (univev2_1:that Forall(.Pred_1)),(x_1:
>>        obj) => (---:that .Pred_1(x_1))]
>>   {move 0}
```

We now present the constructive rules for the existential quantifier

```
Lestrade execution:

clearcurrent

declare x1 obj

>> x1: obj {move 1}


declare Pred [x1 => prop]

>> Pred: [(x1_1:obj) => (---:prop)]
```

```
>>   {move 1}


construct Exists Pred prop

>> Exists: [(Pred_1:[(x1_2:obj) => (---:prop)])
>>        => (---:prop)]
>>   {move 0}


declare x obj

>> x: obj {move 1}


declare existsev1 that Pred x

>> existsev1: that Pred(x) {move 1}


construct Ei Pred, x, existsev1 that Exists Pred

>> Ei: [(Pred_1:[(x1_2:obj) => (---:prop)]),
>>        (x_1:obj),(existsev1_1:that Pred_1(x_1))
>>        => (---:that Exists(Pred_1))]
>>   {move 0}


define Eia x, existsev1: Ei Pred,x,existsev1

>> Eia: [(x_1:obj),(.Pred_1:[(x1_2:obj) => (---:
>>             prop)]),
>>        (existsev1_1:that .Pred_1(x_1)) => (Ei(.Pred_1,
>>        x_1,existsev1_1):that Exists(.Pred_1))]
>>   {move 0}


define Eib Pred, existsev1: Ei Pred,x,existsev1

>> Eib: [(Pred_1:[(x1_2:obj) => (---:prop)]),
>>        (.x_1:obj),(existsev1_1:that Pred_1(.x_1))
>>        => (Ei(Pred_1,.x_1,existsev1_1):that
>>        Exists(Pred_1))]
>>   {move 0}
```

```
declare existsev2 that Exists Pred

>> existsev2: that Exists(Pred) {move 1}


declare p prop

>> p: prop {move 1}


declare witnessing [x,existsev1 => that p]

>> witnessing: [(x_1:obj),(existsev1_1:that
>>        Pred(x_1)) => (---:that p)]
>>   {move 1}


construct Eg existsev2, witnessing: that p

>> Eg: [(.Pred_1:[(x1_2:obj) => (---:prop)]),
>>        (existsev2_1:that Exists(.Pred_1)),(.p_1:
>>        prop),(witnessing_1:[(x_3:obj),(existsev1_3:
>>             that .Pred_1(x_3)) => (---:that
>>             .p_1)])
>>        => (---:that .p_1)]
>>   {move 0}
```

An interesting aspect of the existential quantifier is the relationship of existential instantiation to the implicit inference functions. In the universal quantifier rules, reasoning in either direction, one can determine the predicate from the data supplied. But from an expanded statement `Pred x` one cannot determine `Pred` with certainty even given `x`; sometimes one has to supply it because some instances of `x` are accidental occurrences as a constant. Supplying a version with all arguments explicit, then defining versions with different choices of explicit arguments is a good template for what happens with other operations where we can deduce different arguments under different circumstances.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}
```

```
declare y obj

>> y: obj {move 1}


construct = x y prop

>> =: [(x_1:obj),(y_1:obj) => (---:prop)]
>>   {move 0}


construct Reflexive x that x=x

>> Reflexive: [(x_1:obj) => (---:that (x_1 =
>>        x_1))]
>>   {move 0}


define test1 x :  Eia x Reflexive x

>> test1: [(x_1:obj) => ((x_1 Eia Reflexive(x_1)):
>>        that Exists([(x1_3:obj) => ((x1_3 =
>>             x1_3):prop)]))
>>        ]
>>   {move 0}


define test2 x: Eib [y => x=y] Reflexive x

>> test2: [(x_1:obj) => (Eib([(y_2:obj) => ((x_1
>>             = y_2):prop)]
>>        ,Reflexive(x_1)):that Exists([(y_3:obj)
>>             => ((x_1 = y_3):prop)]))
>>        ]
>>   {move 0}


open

     declare y1 obj

>>      y1: obj {move 2}
```

```
     define testpred y1 : x=y1

>>      testpred: [(y1_1:obj) => ((x = y1_1):
>>             prop)]
>>        {move 1}


     close

define test3 x :  Eib testpred, Reflexive x

>> test3: [(x_1:obj) => (Eib([(y1_2:obj) =>
>>             ((x_1 = y1_2):prop)]
>>        ,Reflexive(x_1)):that Exists([(y1_3:
>>             obj) => ((x_1 = y1_3):prop)]))
>>        ]
>>   {move 0}


define test4 : Ug test3

>> test4: [(Ug(test3):that Forall([(x_3:obj)
>>             => (Exists([(y1_4:obj) => ((x_3
>>                  = y1_4):prop)])
>>             :prop)]))
>>        ]
>>   {move 0}
```

In the example above we prove two different existential statements from the same evidence, in one case supplying the witness and letting Lestrade guess the predicate, and in the other case supplying a different predicate.

The proof of `test3` illustrates a style in which the use of a λ-term as an argument is avoided.

The proof `test4` illustrates the addition of a universal quantifier just because we can.

We introduced some basic declarations for equality just for the sake of the example; more declarations follow.

```
Lestrade execution:

% basic additional declarations for equality

clearcurrent
```

```
declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare pred [x=>prop]

>> pred: [(x_1:obj) => (---:prop)]
>>   {move 1}


declare eqev that x=y

>> eqev: that (x = y) {move 1}


declare predev that pred x

>> predev: that pred(x) {move 1}


construct Substitution1 pred, eqev predev that pred y

>> Substitution1: [(pred_1:[(x_2:obj) => (---:
>>             prop)]),
>>        (.x_1:obj),(.y_1:obj),(eqev_1:that (.x_1
>>        = .y_1)),(predev_1:that pred_1(.x_1))
>>        => (---:that pred_1(.y_1))]
>>   {move 0}


construct Substitution2 eqev predev that pred y

>> Substitution2: [(.x_1:obj),(.y_1:obj),(eqev_1:
>>        that (.x_1 = .y_1)),(.pred_1:[(x_2:obj)
>>             => (---:prop)]),
>>        (predev_1:that .pred_1(.x_1)) => (---:
>>        that .pred_1(.y_1))]
>>   {move 0}
```

In addition to reflexivity already given, we need subsitution of equals for equals. Other properties of equality are derived.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare eqev that x=y

>> eqev: that (x = y) {move 1}


define Symmeq eqev:  Substitution1 [y=>y=x] eqev Reflexive x

>> Symmeq: [(.x_1:obj),(.y_1:obj),(eqev_1:that
>>         (.x_1 = .y_1)) => (Substitution1([(y_2:
>>              obj) => ((y_2 = .x_1):prop)]
>>         ,eqev_1,Reflexive(.x_1)):that (.y_1
>>         = .x_1))]
>>   {move 0}
```

The use of variable binding terms makes this proof of symmetry of equality very compact, and perhaps a bit mysterious.

3 Type Theory and the Curry Howard Isomorphism

4 Set Theory

In this section we discuss the implementation of the usual untyped foundations of mathematics using the theory of sets. The Lestrade implementation is not strictly speaking untyped, since while all the sets are of sort `obj`, the machinery of proof involves complex evidence types as usual.

4.1 Russell's Paradox

We begin by implementing set theory carelessly, and coming up against the paradox of Russell.

```
Lestrade execution:

clearcurrent

open

     declare x obj

>>       x: obj {move 2}


     declare pred [x=>prop]

>>       pred: [(x_1:obj) => (---:prop)]
>>         {move 2}


     construct setof pred obj

>>       setof: [(pred_1:[(x_2:obj) => (---:prop)])
>>                => (---:obj)]
>>         {move 1}
```

We declare the set constructor which takes any predicate of objects to the associated set of objects (itself supposed to be an object).

```
Lestrade execution:

     declare y obj

>>       y: obj {move 2}
```

```
     construct E x y prop

>>      E: [(x_1:obj),(y_1:obj) => (---:prop)]
>>        {move 1}
```

We declare the membership relation on objects.

```
Lestrade execution:

     declare inev1 that x E setof pred

>>      inev1: that (x E setof(pred)) {move
>>        2}


     construct Comp1 inev1 that pred x

>>      Comp1: [(.x_1:obj),(.pred_1:[(x_2:obj)
>>                   => (---:prop)]),
>>              (inev1_1:that (.x_1 E setof(.pred_1)))
>>              => (---:that .pred_1(.x_1))]
>>        {move 1}
```

We declare one side of the comprehension scheme: from evidence for $\phi(a)$, get evidence for $a \in \{x \mid \phi(x)\}$.

```
Lestrade execution:

     declare inev2 that pred x

>>      inev2: that pred(x) {move 2}


     construct Comp2 x inev2 that x E setof pred

>>      Comp2: [(x_1:obj),(.pred_1:[(x_2:obj)
>>                   => (---:prop)]),
>>              (inev2_1:that .pred_1(x_1)) =>
>>              (---:that (x_1 E setof(.pred_1)))]
>>        {move 1}
```

We declare the other side of the comprehension scheme: from evidence for $a \in \{x \mid \phi(x)\}$ get evidence for $\phi(a)$.

```
Lestrade execution:

     define Russell :  setof [x=> ~(x E x)]

>>      Russell: [(setof([(x_1:obj) => (~((x_1
>>                  E x_1)):prop)])
>>              :obj)]
>>        {move 1}
```

We define the Russell class, and begin the proof forthwith.

```
Lestrade execution:

     open

          declare rhyp that Russell E Russell

>>           rhyp: that (Russell E Russell)
>>             {move 3}
```

Let R be the Russell class. Our aim is to prove $R \notin R$. Our strategy is to assume $R \in R$ (`rhyp`) and reason to a contradiction.

```
Lestrade execution:

          define line1 rhyp : Comp1 rhyp

>>           line1: [(rhyp_1:that (Russell E
>>                  Russell)) => (Comp1(rhyp_1):
>>                  that ~((Russell E Russell)))]
>>             {move 2}
```

By an application of `Comp1` we reason from $R \in R = \{x \mid x \notin x\}$ to $R \notin R$.

```
Lestrade execution:

          define line2 rhyp : Mp rhyp line1 rhyp
```

```
>>           line2: [(rhyp_1:that (Russell E
>>                  Russell)) => ((rhyp_1 Mp line1(rhyp_1)):
>>                  that ??)]
>>             {move 2}
```

By modus ponens we obtain an absurdity (recalling that negation is defined in terms of implication).

```
Lestrade execution:

          close

     define rfact :  Negintro line2

>>      rfact: [(Negintro([(rhyp_1:that (Russell
>>                  E Russell)) => ((rhyp_1 Mp
>>                  Comp1(rhyp_1)):that ??)])
>>              :that ~((Russell E Russell)))]
>>        {move 1}
```

By the rule of negation introduction developed above (a variant of the deduction theorem) we get a proof of $R \notin R$ from the function `line2` defined in the just-closed block.

```
Lestrade execution:

     define rfact2 : Fixprop(Russell E Russell,Comp2 Russell rfact)

>>      rfact2: [(((Russell E Russell) Fixprop
>>             (Russell Comp2 rfact)):that (Russell
>>             E Russell))]
>>        {move 1}
```

From $R \notin R$ we get by the comprehension axiom `Comp2` evidence for $R \in \{x \mid x \notin x\} = R$. The use of `Fixprop` is cosmetic, so that the result displayed is $R \in R$ rather than $R \in \{x \mid x \notin x\}$.

```
Lestrade execution:

     close
```

```
define Rparadox Comp1, Comp2 : Mp rfact2 rfact

>> Rparadox: [(.E_1:[(x_2:obj),(y_2:obj) =>
>>              (---:prop)]),
>>         (.setof_1:[(pred_3:[(x_4:obj) => (---:
>>                   prop)])
>>              => (---:obj)]),
>>         (Comp1_1:[(.x_5:obj),(.pred_5:[(x_6:
>>                   obj) => (---:prop)]),
>>              (inev1_5:that (.x_5 .E_1 .setof_1(.pred_5)))
>>              => (---:that .pred_5(.x_5))]),
>>         (Comp2_1:[(x_7:obj),(.pred_7:[(x_8:obj)
>>                   => (---:prop)]),
>>              (inev2_7:that .pred_7(x_7)) =>
>>              (---:that (x_7 .E_1 .setof_1(.pred_7)))])
>>         => ((((.setof_1([(x_11:obj) => (~((x_11
>>              .E_1 x_11)):prop)])
>>         .E_1 .setof_1([(x_12:obj) => (~((x_12
>>              .E_1 x_12)):prop)]))
>>         Fixprop Comp2_1(.setof_1([(x_13:obj)
>>              => (~((x_13 .E_1 x_13)):prop)]),
>>         [(x_14:obj) => (~((x_14 .E_1 x_14)):
>>              prop)]
>>         ,Negintro([(rhyp_17:that (.setof_1([(x_18:
>>                   obj) => (~((x_18 .E_1 x_18)):
>>                   prop)])
>>              .E_1 .setof_1([(x_19:obj) => (~((x_19
>>                   .E_1 x_19)):prop)]))
>>              ) => ((rhyp_17 Mp Comp1_1(.setof_1([(x_22:
>>                   obj) => (~((x_22 .E_1 x_22)):
>>                   prop)]),
>>              [(x_23:obj) => (~((x_23 .E_1 x_23)):
>>                   prop)]
>>              ,rhyp_17)):that ??)]))
>>         ) Mp Negintro([(rhyp_26:that (.setof_1([(x_27:
>>                   obj) => (~((x_27 .E_1 x_27)):
>>                   prop)])
>>              .E_1 .setof_1([(x_28:obj) => (~((x_28
>>                   .E_1 x_28)):prop)]))
>>              ) => ((rhyp_26 Mp Comp1_1(.setof_1([(x_31:
>>                   obj) => (~((x_31 .E_1 x_31)):
>>                   prop)]),
>>              [(x_32:obj) => (~((x_32 .E_1 x_32)):
>>                   prop)]
>>              ,rhyp_26)):that ??)]))
>>         :that ??)]
```

```
>>   {move 0}
```

We conducted all our previous reasoning in a block (perhaps we were suspicious), so on emerging from this block we are able to present a function taking any axioms `Comp1` and `Comp2` (from which `E` and `setof` can be determined implicitly) declared as in the block to a proof of the absurd. The display for `Rparadox` is longer than displays above because definitions local to the block are expanded.

We have two observations about this argument. One point is that there is no threat here to the foundations of reason. Our formalization makes it clear that there is a *mistake*, which is in essence the casual assumption that every function from `obj` to `prop` can be identified with an object. We are firm in Lestrade about separating objects from functions, and no such identification can be smuggled in without an explicit postulate, which we show here cannot be safely assumed. At the roots of the paradox lies a confusion not just between different sorts of entity, but between different species of entity, from the Lestrade standpoint.

Our second observation is that not all features of a working set theory are implemented: in this initial development we give only those declarations needed to show the contradiction. There will be more ingredients (such as extensionality principles) in the set theory we implement below.

4.2 Zermelo set theory

In this section we are more confident that we will not get into trouble, so we boldly make our declarations outside a sheltering block. We will roughly speaking follow the original presentation of Zermelo's axioms.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


construct E x y prop

>> E: [(x_1:obj),(y_1:obj) => (---:prop)]
```

```
>>   {move 0}


declare u obj

>> u: obj {move 1}


declare uinx that u E x

>> uinx: that (u E x) {move 1}
```

We declare the membership relation.

```
Lestrade execution:

open

     declare z obj

>>       z: obj {move 2}


     declare zinx that z E x

>>       zinx: that (z E x) {move 2}


     declare ziny that z E y

>>       ziny: that (z E y) {move 2}


     construct ext1 zinx that z E y

>>       ext1: [(.z_1:obj),(zinx_1:that (.z_1
>>              E x)) => (---:that (.z_1 E y))]
>>         {move 1}


     construct ext2 ziny that z E x

>>       ext2: [(.z_1:obj),(ziny_1:that (.z_1
>>              E y)) => (---:that (.z_1 E x))]
```

```
>>        {move 1}


     close

construct Extensionality uinx ext1, ext2 that x=y

>> Extensionality: [(.u_1:obj),(.x_1:obj),(uinx_1:
>>        that (.u_1 E .x_1)),(.y_1:obj),(ext1_1:
>>        [(.z_2:obj),(zinx_2:that (.z_2 E .x_1))
>>             => (---:that (.z_2 E .y_1))]),
>>        (ext2_1:[(.z_3:obj),(ziny_3:that (.z_3
>>             E .y_1)) => (---:that (.z_3 E .x_1))])
>>        => (---:that (.x_1 = .y_1))]
>>   {move 0}
```

These are declarations for weak extensionality: objects with elements that have the same elements are equal.

```
Lestrade execution:

construct Empty obj

>> Empty: obj {move 0}


construct Emptyisempty x : that ~(x E Empty)

>> Emptyisempty: [(x_1:obj) => (---:that ~((x_1
>>        E Empty)))]
>>   {move 0}


define Isset x : (x = Empty) V (Exists [u => u E x])

>> Isset: [(x_1:obj) => (((x_1 = Empty) V Exists([(u_2:
>>             obj) => ((u_2 E x_1):prop)]))
>>        :prop)]
>>   {move 0}
```

We declare the empty set (the first object introduced in Zermelo's axiom of elementary sets) and define the notion of set: an object is a set if it is the empty set or has an element.

Below we prove the theorem that two sets with the same elements are equal (the weak extensionality theorem).

The `typesonly` command suppresses the display of the bodies of definitions: as we will see at the end, the terms representing components of this proof get quite large. The string `...` signals omitted definition bodies. This directive is reversed by `showdefs`.

```
Lestrade execution:

typesonly
open

     declare x1 obj

>>      x1: obj {move 2}


     open

          declare y1 obj

>>           y1: obj {move 3}
```

Choose arbitrary x and y for universal generalization.

```
Lestrade execution:

          open

               declare u1 obj

>>                u1: obj {move 4}


               declare extev that (Isset x1) & (Isset y1) & (Forall[u1 => (u1 E x1) <-> u1 E y1])

>>                extev: that (Isset(x1) & (Isset(y1)
>>                  & Forall([(u1_1:obj) => (((u1_1
>>                       E x1) <-> (u1_1 E y1)):
>>                       prop)]))
>>                  ) {move 4}
```

Assume that x and y are sets and have the same elements.

```
Lestrade execution:
```

```
                define line1 extev : Simplification1 extev

>>                 line1: [(extev_1:that (Isset(x1)
>>                        & (Isset(y1) & Forall([(u1_2:
>>                             obj) => (((u1_2
>>                             E x1) <-> (u1_2
>>                             E y1)):prop)]))
>>                        )) => ( ... :that Isset(x1))]
>>                   {move 3}


                define line2 extev : Simplification1 (Simplification2 extev)

>>                 line2: [(extev_1:that (Isset(x1)
>>                        & (Isset(y1) & Forall([(u1_2:
>>                             obj) => (((u1_2
>>                             E x1) <-> (u1_2
>>                             E y1)):prop)]))
>>                        )) => ( ... :that Isset(y1))]
>>                   {move 3}


                define extstat extev : Simplification2(Simplification2 extev)

>>                 extstat: [(extev_1:that (Isset(x1)
>>                        & (Isset(y1) & Forall([(u1_2:
>>                             obj) => (((u1_2
>>                             E x1) <-> (u1_2
>>                             E y1)):prop)]))
>>                        )) => ( ... :that Forall([(u1_5:
>>                             obj) => (((u1_5
>>                             E x1) <-> (u1_5
>>                             E y1)):prop)]))
>>                        ]
>>                   {move 3}
```

Extract important pieces of the assumption. `line1` is the assumption that x is a set, `line2` is the assumption that y is a set and `extstat` is the assumption that the two sets have the same elements. These will appear with `extev` as an argument.

The strategy of the proof that follows is a proof by cases.

```
Lestrade execution:

                % our goal is to prove x1=y1 by cases

                open

                     declare line3 that x1 = Empty
```

```
>>                      line3: that (x1 = Empty)
>>                        {move 5}


                     % prove x=y
```

Case I: assume that x is the empty set. This falls apart into two cases depending on the nature of y;

```
Lestrade execution:

                     % now prove by cases in Isset y1

                     open

                          declare line5 that y1 = Empty

>>                           line5: that (y1
>>                             = Empty) {move 6}
```

Case IA: y is empty.

```
Lestrade execution:

                          define line9 line5 : Symmeq line5

>>                           line9: [(line5_1:
>>                                  that (y1 =
>>                                  Empty)) =>
>>                                  ( ... :that
>>                                  (Empty = y1))]
>>                             {move 5}


                          define line10 line5 : Substitution2 (line9 line5,line3)

>>                           line10: [(line5_1:
>>                                  that (y1 =
>>                                  Empty)) =>
>>                                  ( ... :that
>>                                  (x1 = y1))]
>>                             {move 5}


                          % prove x=y
```

```
                        close

                    define line11 line3 : Deduction line10

>>                     line11: [(line3_1:that
>>                            (x1 = Empty)) =>
>>                            ( ... :that ((y1
>>                            = Empty) -> (x1
>>                            = y1)))]
>>                       {move 4}
```

The implication expressing the conclusion of Case IA is readily proved using the substitution property of equality.

```
Lestrade execution:

                    open

                        declare u2 obj

>>                         u2: obj {move 6}


                        declare line7 that Exists[u2 =>u2 E y1]

>>                         line7: that Exists([(u2_1:
>>                                obj) => ((u2_1
>>                                E y1):prop)])
>>                           {move 6}
```

This is case IB, where y is inhabited. Actually, the way I prove this is absurd, though it works. I never use the fact that x is empty, which would allow a quick proof by contradiction (I shoudl repair this). On the other hand, this proof makes the point that the weak extensionality theorem actually does not depend on the fact that `Empty` has no elements.

The following block sets things up for a proof by existential generalization.

```
Lestrade execution:

                        open

                            declare w obj

>>                             w: obj {move
>>                               7}
```

```
                              declare wev that w E y1

>>                               wev: that (w
>>                                 E y1) {move
>>                                 7}
```

Provide a witness to the existential statement that y is inhabited.

```
Lestrade execution:

                              define  xxx wev:Ui (extstat extev,w)

>>                               xxx: [(.w_1:
>>                                      obj),(wev_1:
>>                                      that (.w_1
>>                                      E y1))
>>                                      => ( ...
>>                                      :that
>>                                      ((.w_1
>>                                      E x1)
>>                                      <-> (.w_1
>>                                      E y1)))]
>>                                 {move 6}


                              define xxx2 wev :Simplification2 xxx wev

>>                               xxx2: [(.w_1:
>>                                      obj),(wev_1:
>>                                      that (.w_1
>>                                      E y1))
>>                                      => ( ...
>>                                      :that
>>                                      ((.w_1
>>                                      E y1)
>>                                      -> (.w_1
>>                                      E x1)))]
>>                                 {move 6}


                              define xxx3 wev: Mp wev xxx2 wev

>>                               xxx3: [(.w_1:
>>                                      obj),(wev_1:
>>                                      that (.w_1
>>                                      E y1))
>>                                      => ( ...
```

```
>>                                        :that
>>                                        (.w_1
>>                                        E x1))]
>>                                   {move 6}
```

Argue that x is inhabited (which of course we also know is false, but we never use this in the current proof plan!)

We follow with the obvious structured proof that x and y are equal by extensionality, since x is inhabited and x and y have the same elements.

```
Lestrade execution:

                                   open

                                        declare w1 obj

>>                                         w1: obj
>>                                           {move
>>                                           8}


                                        declare wev1 that w1 E x1

>>                                         wev1:
>>                                           that (w1
>>                                           E x1)
>>                                           {move
>>                                           8}


                                        define line89 wev1 :  Ui (extstat extev,w1)

>>                                         line89:
>>                                           [(.w1_1:
>>                                                obj),
>>                                                (wev1_1:
>>                                                that
>>                                                (.w1_1
>>                                                E
>>                                                x1))
>>                                                =>
>>                                                (
>>                                                ...
>>                                                :
>>                                                that
>>                                                ((.w1_1
>>                                                E
>>                                                x1)
```

```
>>                                        <->
>>                                        (.w1_1
>>                                        E
>>                                        y1)))]
>>                                   {move
>>                                   7}


                                  define line90 wev1 : Simplification1 line89 wev1

>>                                   line90:
>>                                     [(.w1_1:
>>                                        obj),
>>                                        (wev1_1:
>>                                        that
>>                                        (.w1_1
>>                                        E
>>                                        x1))
>>                                        =>
>>                                        (
>>                                        ...
>>                                        :
>>                                        that
>>                                        ((.w1_1
>>                                        E
>>                                        x1)
>>                                        ->
>>                                        (.w1_1
>>                                        E
>>                                        y1)))]
>>                                     {move
>>                                     7}


                                  define line91 wev1 : Mp wev1 line90 wev1

>>                                   line91:
>>                                     [(.w1_1:
>>                                        obj),
>>                                        (wev1_1:
>>                                        that
>>                                        (.w1_1
>>                                        E
>>                                        x1))
>>                                        =>
>>                                        (
>>                                        ...
>>                                        :
>>                                        that
>>                                        (.w1_1
```

```
>>                                        E
>>                                        y1))]
>>                                   {move
>>                                   7}


                                  declare wev2 that w1 E y1

>>                                   wev2:
>>                                     that (w1
>>                                     E y1)
>>                                     {move
>>                                     8}


                                  define lineb89 wev2 : Ui (extstat extev,w1)

>>                                   lineb89:
>>                                     [(.w1_1:
>>                                          obj),
>>                                          (wev2_1:
>>                                          that
>>                                          (.w1_1
>>                                          E
>>                                          y1))
>>                                          =>
>>                                          (
>>                                          ...
>>                                          :
>>                                          that
>>                                          ((.w1_1
>>                                          E
>>                                          x1)
>>                                          <->
>>                                          (.w1_1
>>                                          E
>>                                          y1)))]
>>                                     {move
>>                                     7}


                                  define line92 wev2 : Simplification2 lineb89 wev2

>>                                   line92:
>>                                     [(.w1_1:
>>                                          obj),
>>                                          (wev2_1:
>>                                          that
>>                                          (.w1_1
>>                                          E
```

```
>>                                      y1))
>>                                      =>
>>                                      (
>>                                      ...
>>                                      :
>>                                      that
>>                                      ((.w1_1
>>                                      E
>>                                      y1)
>>                                      ->
>>                                      (.w1_1
>>                                      E
>>                                      x1)))]
>>                                  {move
>>                                  7}


                                define line93 wev2 : Mp wev2 line92 wev2

>>                                 line93:
>>                                   [(.w1_1:
>>                                        obj),
>>                                        (wev2_1:
>>                                        that
>>                                        (.w1_1
>>                                        E
>>                                        y1))
>>                                        =>
>>                                        (
>>                                        ...
>>                                        :
>>                                        that
>>                                        (.w1_1
>>                                        E
>>                                        x1))]
>>                                   {move
>>                                   7}


                                  close

                              define line94 wev:  Extensionality (xxx3 wev, line91, line93)

>>                               line94: [(.w_1:
>>                                      obj),(wev_1:
>>                                      that (.w_1
>>                                      E y1))
>>                                      => ( ...
>>                                      :that
>>                                      (x1 =
```

```
>>                                   y1))]
>>                                {move 6}
```

What appears above is the natural structured argument that because of the hypothesis that x and y have the same elements, we can produce the functions from w and evidence that $w \in x$ to evidence that $w \in y$ and from w and evidence that $w \in y$ to evidence that $w \in x$ which are needed as input to the extensionality axiom. The information that x is inhabited is also needed, and supplied. Again, this whole case could be proved much more compactly!

```
Lestrade execution:

                              close

                         define line95 line7 :  Eg line7 line94

>>                          line95: [(line7_1:
>>                                 that Exists([(u2_2:
>>                                      obj) =>
>>                                      ((u2_2
>>                                      E y1):
>>                                      prop)]))
>>                                 => ( ... :that
>>                                 (x1 = y1))]
>>                               {move 5}


                           close

                      define line96 line3:  Deduction line95

>>                       line96: [(line3_1:that
>>                              (x1 = Empty)) =>
>>                              ( ... :that (Exists([(u2_12:
>>                                   obj) => ((u2_12
>>                                   E y1):prop)])
>>                              -> (x1 = y1)))]
>>                            {move 4}
```

We formally complete Case IB.

```
Lestrade execution:

                      define line97 line3:  Cases (line2 extev ,line11 line3, line96 line3)

>>                       line97: [(line3_1:that
```

```
>>                          (x1 = Empty)) =>
>>                          ( ... :that (x1
>>                          = y1))]
>>                     {move 4}


                  close

               define line98 extev :  Deduction line97

>>                line98: [(extev_1:that (Isset(x1)
>>                       & (Isset(y1) & Forall([(u1_2:
>>                            obj) => (((u1_2
>>                            E x1) <-> (u1_2
>>                            E y1)):prop)]))
>>                       )) => ( ... :that ((x1
>>                       = Empty) -> (x1 = y1)))]
>>                  {move 3}
```

We formally complete the proof of Case I.

```
Lestrade execution:

               open

                  declare u2 obj

>>                   u2: obj {move 5}


                  declare line99 that Exists[u2=>u2 E x1]

>>                   line99: that Exists([(u2_1:
>>                          obj) => ((u2_1 E
>>                          x1):prop)])
>>                     {move 5}
```

This is Case II, x is inhabited, which we prove directly without further case analysis.

```
Lestrade execution:

                  open

                     declare w obj
```

```
>>                          w: obj {move 6}


                        declare wev that w E x1

>>                          wev: that (w E x1)
>>                            {move 6}
```

We introduce a witness to the hypothesis for Case II, setting up for a proof of the goal by existential generalization.

```
Lestrade execution:

                        open

                             declare w1 obj

>>                              w1: obj {move
>>                                7}


                             declare wev1 that w1 E x1

>>                              wev1: that
>>                                (w1 E x1) {move
>>                                7}


                             declare wev2 that w1 E y1

>>                              wev2: that
>>                                (w1 E y1) {move
>>                                7}


                             define zonk w1 :  Ui (extstat extev,w1)

>>                              zonk: [(w1_1:
>>                                      obj) =>
>>                                      ( ...
>>                                      :that
>>                                      ((w1_1
>>                                      E x1)
>>                                      <-> (w1_1
>>                                      E y1)))]
>>                                {move 6}
```

```
                        define zonk1 w1:  Simplification1 (zonk w1)

>>                         zonk1: [(w1_1:
>>                                obj) =>
>>                                ( ...
>>                                :that
>>                                ((w1_1
>>                                E x1)
>>                                -> (w1_1
>>                                E y1)))]
>>                           {move 6}


                        define zonk2 w1:Simplification2 (zonk w1)

>>                         zonk2: [(w1_1:
>>                                obj) =>
>>                                ( ...
>>                                :that
>>                                ((w1_1
>>                                E y1)
>>                                -> (w1_1
>>                                E x1)))]
>>                           {move 6}


                        define zonk3 wev1 : Mp wev1 zonk1 w1

>>                         zonk3: [(.w1_1:
>>                                obj),(wev1_1:
>>                                that (.w1_1
>>                                E x1))
>>                                => ( ...
>>                                :that
>>                                (.w1_1
>>                                E y1))]
>>                           {move 6}


                        define zonk4 wev2 : Mp wev2 zonk2 w1

>>                         zonk4: [(.w1_1:
>>                                obj),(wev2_1:
>>                                that (.w1_1
>>                                E y1))
>>                                => ( ...
>>                                :that
>>                                (.w1_1
>>                                E x1))]
>>                           {move 6}
```

```
                              close

                         define zonk5 wev: Extensionality wev zonk3, zonk4

>>                          zonk5: [(.w_1:obj),
>>                                 (wev_1:that
>>                                 (.w_1 E x1))
>>                                 => ( ... :that
>>                                 (x1 = y1))]
>>                            {move 5}
```

The hypothesis that x and y have the same elements suffice to construct the functions from w and evidence that $w \in x$ to evidence that $w \in y$ and from w and evidence that $w \in y$ to evidence that $w \in x$ which are needed as input to the extensionality axiom: the further input that x is inhabited is provided as the hypothesis of Case II.

Lestrade execution:

```
                         close

                    define zonk6 line99 : Eg line99 zonk5

>>                     zonk6: [(line99_1:that
>>                            Exists([(u2_2:obj)
>>                                 => ((u2_2 E
>>                                 x1):prop)]))
>>                            => ( ... :that (x1
>>                            = y1))]
>>                       {move 4}


                    close

               define zonk7 extev :  Deduction zonk6

>>                zonk7: [(extev_1:that (Isset(x1)
>>                       & (Isset(y1) & Forall([(u1_2:
>>                            obj) => (((u1_2
>>                            E x1) <-> (u1_2
>>                            E y1)):prop)]))
>>                       )) => ( ... :that (Exists([(u2_12:
>>                            obj) => ((u2_12
>>                            E x1):prop)])
>>                       -> (x1 = y1)))]
>>                  {move 3}
```

We formally complete the proof of the implication expressing Case II, first applying existential generalization then deduction.

```
Lestrade execution:

               define zonk8 extev:  Cases(line1 extev, line98 extev, zonk7 extev)

>>                zonk8: [(extev_1:that (Isset(x1)
>>                       & (Isset(y1) & Forall([(u1_2:
>>                            obj) => (((u1_2
>>                            E x1) <-> (u1_2
>>                            E y1)):prop)]))
>>                       )) => ( ... :that (x1
>>                       = y1))]
>>                  {move 3}


               close

          define zonk9 y1:  Deduction zonk8

>>           zonk9: [(y1_1:obj) => ( ... :that
>>                  ((Isset(x1) & (Isset(y1_1)
>>                  & Forall([(u1_42:obj) => (((u1_42
>>                       E x1) <-> (u1_42 E y1_1)):
>>                       prop)]))
>>                  ) -> (x1 = y1_1)))]
>>             {move 2}


          close

     define zonk10 x1: Ug zonk9

>>      zonk10: [(x1_1:obj) => ( ... :that Forall([(y1_46:
>>                  obj) => (((Isset(x1_1) & (Isset(y1_46)
>>                  & Forall([(u1_47:obj) => (((u1_47
>>                       E x1_1) <-> (u1_47 E
>>                       y1_46)):prop)]))
>>                  ) -> (x1_1 = y1_46)):prop)]))
>>              ]
>>        {move 1}


     close

showdefs
define Weakext :  Ug zonk10

>> Weakext: [(Ug([(x1_1:obj) => (Ug([(y1_6:obj)
```

```
>>                  => (Deduction([(extev_8:that
>>                       (Isset(x1_1) & (Isset(y1_6)
>>                       & Forall([(u1_9:obj)
>>                            => (((u1_9 E x1_1)
>>                            <-> (u1_9 E y1_6)):
>>                            prop)]))
>>                       )) => (Cases(Simplification1(extev_8),
>>                       Deduction([(line3_12:
>>                            that (x1_1 = Empty))
>>                            => (Cases(Simplification1(Simplification2(extev_8)),
>>                            Deduction([(line5_16:
>>                                 that (y1_6
>>                                 = Empty)) =>
>>                                 ((Symmeq(line5_16)
>>                                 Substitution2
>>                                 line3_12):that
>>                                 (x1_1 = y1_6))]),
>>                            Deduction([(line7_19:
>>                                 that Exists([(u2_20:
>>                                      obj) =>
>>                                      ((u2_20
>>                                      E y1_6):
>>                                      prop)]))
>>                                 => ((line7_19
>>                                 Eg [(.w_22:
>>                                      obj),(wev_22:
>>                                      that (.w_22
>>                                      E y1_6))
>>                                      => (Extensionality((wev_22
>>                                      Mp Simplification2((Simplification2(Simplification2(extev_
>>                                      Ui .w_22))),
>>                                      [(.w1_26:
>>                                           obj),
>>                                           (wev1_26:
>>                                           that
>>                                           (.w1_26
>>                                           E
>>                                           x1_1))
>>                                           =>
>>                                           ((wev1_26
>>                                           Mp
>>                                           Simplification1((Simplification2(Simplification2(exte
>>                                           Ui
>>                                           .w1_26))):
>>                                           that
>>                                           (.w1_26
>>                                           E
>>                                           y1_6))]
>>                                      ,[(.w1_30:
>>                                           obj),
```

```
>>                                        (wev2_30:
>>                                        that
>>                                        (.w1_30
>>                                        E
>>                                        y1_6))
>>                                        =>
>>                                        ((wev2_30
>>                                        Mp
>>                                        Simplification2((Simplification2(Simplification2(exte
>>                                        Ui
>>                                        .w1_30))):
>>                                        that
>>                                        (.w1_30
>>                                        E
>>                                        x1_1))])
>>                                   :that
>>                                   (x1_1
>>                                   = y1_6))])
>>                              :that (x1_1
>>                              = y1_6))]))
>>                         :that (x1_1 = y1_6))]),
>>                    Deduction([(line99_35:
>>                         that Exists([(u2_36:
>>                              obj) => ((u2_36
>>                              E x1_1):prop)]))
>>                         => ((line99_35 Eg
>>                         [(.w_38:obj),(wev_38:
>>                              that (.w_38
>>                              E x1_1)) =>
>>                              (Extensionality(wev_38,
>>                              [(.w1_39:obj),
>>                                   (wev1_39:
>>                                   that (.w1_39
>>                                   E x1_1))
>>                                   => ((wev1_39
>>                                   Mp Simplification1((Simplification2(Simplification2(extev_
>>                                   Ui .w1_39))):
>>                                   that (.w1_39
>>                                   E y1_6))]
>>                              ,[(.w1_43:obj),
>>                                   (wev2_43:
>>                                   that (.w1_43
>>                                   E y1_6))
>>                                   => ((wev2_43
>>                                   Mp Simplification2((Simplification2(Simplification2(extev_
>>                                   Ui .w1_43))):
>>                                   that (.w1_43
>>                                   E x1_1))])
>>                              :that (x1_1
>>                              = y1_6))])
```

```
>>                          :that (x1_1 = y1_6))]))
>>                     :that (x1_1 = y1_6))])
>>                :that ((Isset(x1_1) & (Isset(y1_6)
>>                & Forall([(u1_47:obj) => (((u1_47
>>                     E x1_1) <-> (u1_47 E
>>                     y1_6)):prop)]))
>>                ) -> (x1_1 = y1_6)))])
>>           :that Forall([(y1_2:obj) => (((Isset(x1_1)
>>                & (Isset(y1_2) & Forall([(u1_3:
>>                     obj) => (((u1_3 E x1_1)
>>                     <-> (u1_3 E y1_2)):prop)]))
>>                ) -> (x1_1 = y1_2)):prop)]))
>>           ])
>>       :that Forall([(x1_48:obj) => (Forall([(y1_49:
>>                obj) => (((Isset(x1_48) &
>>                (Isset(y1_49) & Forall([(u1_50:
>>                     obj) => (((u1_50 E x1_48)
>>                     <-> (u1_50 E y1_49)):
>>                     prop)]))
>>                ) -> (x1_48 = y1_49)):prop)])
>>           :prop)]))
>>       ]
>>   {move 0}
```

The proof of the main result is completed by cases, then by deduction, then by universal generalizations. The reason why we issued the `typesonly` directive before the proof should be evident (we maliciously invoked the `showdefs` directive which reverses `typesonly` before displaying this proof).

The proof is actually quite straightforward and is approached in a natural structured way. The Lestrade output is colossal due to definition expansion; we want to consider whether structuring it differently might reduce the apparent size. This might also be a strong suggestion that there should be a display setting which modifies the display of defined objects to show only their sorts (which are what actually interest us here). It is worth noting that this proof is unlikely to appear in a final version of this paper!

We now proceed to complete Zermelo's axiom of elementary sets (not without further excursions).

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}
```

```
declare y obj

>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


construct pair x y obj

>> pair: [(x_1:obj),(y_1:obj) => (---:obj)]
>>   {move 0}


construct pair1 x y that x E pair x y

>> pair1: [(x_1:obj),(y_1:obj) => (---:that
>>        (x_1 E (x_1 pair y_1)))]
>>   {move 0}


construct pair2 x y that y E pair x y

>> pair2: [(x_1:obj),(y_1:obj) => (---:that
>>        (y_1 E (x_1 pair y_1)))]
>>   {move 0}


declare zev that z E pair x y

>> zev: that (z E (x pair y)) {move 1}


construct pair3 zev that (z = x) V z = y

>> pair3: [(.z_1:obj),(.x_1:obj),(.y_1:obj),
>>        (zev_1:that (.z_1 E (.x_1 pair .y_1)))
>>        => (---:that ((.z_1 = .x_1) V (.z_1
>>        = .y_1)))]
>>   {move 0}
```

These are the basic axioms for the unordered pair.

```
Lestrade execution:

define singleton x: pair x x

>> singleton: [(x_1:obj) => ((x_1 pair x_1):
>>        obj)]
>>   {move 0}


construct the x obj

>> the: [(x_1:obj) => (---:obj)]
>>   {move 0}


construct the1 x that the(singleton x) = x

>> the1: [(x_1:obj) => (---:that (the(singleton(x_1))
>>        = x_1))]
>>   {move 0}


construct the2 x that ~(Exists[y=>x = singleton y]) -> (the x) = Empty

>> the2: [(x_1:obj) => (---:that (~(Exists([(y_2:
>>             obj) => ((x_1 = singleton(y_2)):
>>             prop)]))
>>        -> (the(x_1) = Empty)))]
>>   {move 0}


define Pair x y: singleton x pair x pair y

>> Pair: [(x_1:obj),(y_1:obj) => ((singleton(x_1)
>>        pair (x_1 pair y_1)):obj)]
>>   {move 0}
```

We introduce related notions.

We define the singleton set, which is, oddly to modern eyes, introduced in an independent clause in Zermelo's original axiom.

We introduce the definite description operator (in a form bounded to sets); this can appear appropriately here as it is a left inverse of the singleton operator.

it would be definable (using primitives introduced later) if we did not allow atoms.

We introduce the Kuratowski pair, an anachronism, because not known to Zermelo, but very useful.

We prove that the Kuratowski ordered pair is indeed an ordered pair.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


declare w obj

>> w: obj {move 1}


typesonly
open

     declare pairsequal that (x Pair y) = (z Pair w)

>>       pairsequal: that ((x Pair y) = (z Pair
>>         w)) {move 2}
```

We prove a frequently useful lemma.

```
Lestrade execution:

     declare p prop
```

```
>>      p: prop {move 2}


     declare porp that p V p

>>      porp: that (p V p) {move 2}


     declare pp that p

>>      pp: that p {move 2}


     define Porp porp :  Cases (porp,Deduction [pp=>pp], Deduction [pp=>pp])

>>      Porp: [(.p_1:prop),(porp_1:that (.p_1
>>             V .p_1)) => ( ... :that .p_1)]
>>        {move 1}
```

We introduce the assumption that $(x, y) = (z, w)$.

```
Lestrade execution:

     define line1 pairsequal :  Substitution2 pairsequal, pair1 (singleton x,x pair y)

>>      line1: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             (singleton(x) E (z Pair w)))]
>>        {move 1}


     define line2 pairsequal:  pair3 line1 pairsequal

>>      line2: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             ((singleton(x) = singleton(z))
>>             V (singleton(x) = (z pair w))))]
>>        {move 1}
```

Since $\{x\} \in \{\{x\}, \{x, y\}\} = \{\{z\}, \{z, w\}\}$, we have $\{x\} = \{z\}$ or $\{x\} = \{z, w\}$. We proceed to reason by cases.

```
Lestrade execution:
```

```
     open

          % case {x}={z} -- goal is x=z

          declare hyp that singleton x = singleton z

>>           hyp: that (singleton(x) = singleton(z))
>>             {move 3}


          define line3 hyp : Substitution2 hyp, x pair1 x

>>           line3: [(hyp_1:that (singleton(x)
>>                  = singleton(z))) => ( ...
>>                  :that (x E singleton(z)))]
>>             {move 2}


          define line4 hyp :  pair3 line3 hyp

>>           line4: [(hyp_1:that (singleton(x)
>>                  = singleton(z))) => ( ...
>>                  :that ((x = z) V (x = z)))]
>>             {move 2}


          define line5 hyp : Porp line4 hyp

>>           line5: [(hyp_1:that (singleton(x)
>>                  = singleton(z))) => ( ...
>>                  :that (x = z))]
>>             {move 2}
```

We have achieved the goal of this case.

```
Lestrade execution:

          close

     define line7 : Deduction line5

>>      line7: [( ... :that ((singleton(x) =
>>             singleton(z)) -> (x = z)))]
>>        {move 1}
```

```
     open

          declare hyp that singleton x = z pair w

>>           hyp: that (singleton(x) = (z pair
>>             w)) {move 3}


          define line8 hyp : Substitution2 (Symmeq hyp,z pair1 w)

>>           line8: [(hyp_1:that (singleton(x)
>>                  = (z pair w))) => ( ... :that
>>                  (z E singleton(x)))]
>>             {move 2}


          define line10 hyp: Porp pair3 line8 hyp

>>           line10: [(hyp_1:that (singleton(x)
>>                  = (z pair w))) => ( ... :that
>>                  (z = x))]
>>             {move 2}


          define line11 hyp:  Symmeq line10 hyp

>>           line11: [(hyp_1:that (singleton(x)
>>                  = (z pair w))) => ( ... :that
>>                  (x = z))]
>>             {move 2}


          close

     define line12: Deduction line11

>>      line12: [( ... :that ((singleton(x)
>>             = (z pair w)) -> (x = z)))]
>>        {move 1}


     define proj1 pairsequal :  Cases(line2 pairsequal, line7, line12)

>>      proj1: [(pairsequal_1:that ((x Pair
```

```
>>             y) = (z Pair w))) => ( ... :that
>>             (x = z))]
>>        {move 1}
```

We have proved $x = z$: it remains to prove $y = w$.

```
Lestrade execution:

     define line13 pairsequal:  Substitution2 (pairsequal, pair2 singleton x, x pair y)

>>      line13: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             ((x pair y) E (z Pair w)))]
>>        {move 1}


     define line14 pairsequal:  pair3 line13 pairsequal

>>      line14: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             (((x pair y) = singleton(z)) V
>>             ((x pair y) = (z pair w))))]
>>        {move 1}
```

Considering $\{x, y\} \in (x, y) = (z, w)$, we draw the conclusion that either $\{x, y\} = \{z\}$ or $\{x, y\} = \{z, w\}$, and proceed to reason by cases.

```
Lestrade execution:

     open

          declare hyp that (x pair y) = singleton z

>>           hyp: that ((x pair y) = singleton(z))
>>             {move 3}


          define line15 hyp : Substitution2 hyp x pair1 y

>>           line15: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (x E singleton(z)))]
>>             {move 2}
```

```
          define line16 hyp : Substitution2 hyp x pair2 y

>>           line16: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (y E singleton(z)))]
>>             {move 2}


          define line17 hyp:  Porp pair3 line15 hyp

>>           line17: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (x = z))]
>>             {move 2}


          define line18 hyp:  Porp pair3 line16 hyp

>>           line18: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (y = z))]
>>             {move 2}


          define line19 hyp:  Substitution2 Symmeq pairsequal, \
                pair2 singleton z, z pair w

>>           line19: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that ((z pair w) E (x Pair
>>                  y)))]
>>             {move 2}


          define line20 hyp : pair3 line19 hyp

>>           line20: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (((z pair w) = singleton(x))
>>                  V ((z pair w) = (x pair y))))]
>>             {move 2}
```

By considering $\{z, w\} \in (z, w) = (x, y)$, we see that $\{z, w\} = \{x\}$ or $\{z, w\} = \{x, y\}$, and yet again we reason by cases.

```
Lestrade execution:

          open

               declare hyp2 that (z pair w) = singleton(x)

>>                hyp2: that ((z pair w) = singleton(x))
>>                  {move 4}


               define line21 hyp2 : Substitution2 (Symmeq line18 hyp,\
                    Substitution2 (line17 hyp,hyp2))

>>                line21: [(hyp2_1:that ((z
>>                       pair w) = singleton(x)))
>>                       => ( ... :that ((y pair
>>                       w) = singleton(y)))]
>>                  {move 3}


               define line22 hyp2:  Substitution2 (line21 hyp2,pair2 y w)

>>                line22: [(hyp2_1:that ((z
>>                       pair w) = singleton(x)))
>>                       => ( ... :that (w E singleton(y)))]
>>                  {move 3}


               define line23 hyp2: Symmeq Porp pair3 line22 hyp2

>>                line23: [(hyp2_1:that ((z
>>                       pair w) = singleton(x)))
>>                       => ( ... :that (y = w))]
>>                  {move 3}


               close

          define line24 hyp: Deduction line23

>>           line24: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (((z pair w) = singleton(x))
```

```
>>                -> (y = w)))]
>>             {move 2}


          open

               declare hyp2 that (z pair w) = x pair y

>>                hyp2: that ((z pair w) = (x
>>                  pair y)) {move 4}


               define line25 hyp2 : Substitution2 (Symmeq line18 hyp,\
                    Substitution2 (line17 hyp,hyp2))

>>                line25: [(hyp2_1:that ((z
>>                       pair w) = (x pair y)))
>>                       => ( ... :that ((y pair
>>                       w) = (y pair y)))]
>>                  {move 3}


               define line26 hyp2:  Substitution2 (line25 hyp2,pair2 y w)

>>                line26: [(hyp2_1:that ((z
>>                       pair w) = (x pair y)))
>>                       => ( ... :that (w E (y
>>                       pair y)))]
>>                  {move 3}


               define line27 hyp2: Symmeq Porp pair3 line26 hyp2

>>                line27: [(hyp2_1:that ((z
>>                       pair w) = (x pair y)))
>>                       => ( ... :that (y = w))]
>>                  {move 3}


               close

          define line28 hyp:  Deduction line27

>>             line28: [(hyp_1:that ((x pair y)
>>                    = singleton(z))) => ( ...
>>                    :that (((z pair w) = (x pair
```

```
>>                 y)) -> (y = w)))]
>>             {move 2}


          define line29 hyp:  Cases (line20 hyp,line24 hyp,line28 hyp)

>>           line29: [(hyp_1:that ((x pair y)
>>                  = singleton(z))) => ( ...
>>                  :that (y = w))]
>>             {move 2}


          close

     define line30 pairsequal : Deduction line29

>>      line30: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             (((x pair y) = singleton(z)) ->
>>             (y = w)))]
>>        {move 1}


     open

          declare hyp that (x pair y) = z pair w

>>           hyp: that ((x pair y) = (z pair
>>             w)) {move 3}


          define line31 hyp: Substitution2 (proj1 pairsequal, hyp)

>>           line31: [(hyp_1:that ((x pair y)
>>                  = (z pair w))) => ( ... :that
>>                  ((z pair y) = (z pair w)))]
>>             {move 2}


          define line32 hyp :  Substitution2(line31 hyp,pair2 z y)

>>           line32: [(hyp_1:that ((x pair y)
>>                  = (z pair w))) => ( ... :that
>>                  (y E (z pair w)))]
>>             {move 2}
```

```
          define line33 hyp:  pair3 line32 hyp

>>           line33: [(hyp_1:that ((x pair y)
>>                  = (z pair w))) => ( ... :that
>>                  ((y = z) V (y = w)))]
>>             {move 2}
```

We need to show that if $y = z$, it follows that $y = w$.

```
Lestrade execution:

          open

               declare hyp2 that y=z

>>                hyp2: that (y = z) {move 4}


               define line34 hyp2:  Substitution2(Symmeq hyp2,\
                    Substitution2(proj1 pairsequal,hyp))

>>                line34: [(hyp2_1:that (y =
>>                       z)) => ( ... :that ((y
>>                       pair y) = (y pair w)))]
>>                  {move 3}


               define line35 hyp2 :  Substitution2(Symmeq line34 hyp2,pair2 y w)

>>                line35: [(hyp2_1:that (y =
>>                       z)) => ( ... :that (w
>>                       E (y pair y)))]
>>                  {move 3}


               define line36 hyp2 :  Symmeq Porp pair3 line35 hyp2

>>                line36: [(hyp2_1:that (y =
>>                       z)) => ( ... :that (y
>>                       = w))]
>>                  {move 3}
```

```
              close

          define line37 hyp: Deduction line36

>>           line37: [(hyp_1:that ((x pair y)
>>                  = (z pair w))) => ( ... :that
>>                  ((y = z) -> (y = w)))]
>>             {move 2}


          declare techfix that y=w

>>           techfix: that (y = w) {move 3}


          define line38 hyp:  Cases (line33 hyp,line37 hyp,\
              Deduction[techfix => techfix])

>>           line38: [(hyp_1:that ((x pair y)
>>                  = (z pair w))) => ( ... :that
>>                  (y = w))]
>>             {move 2}


          close

     define line39 pairsequal :  Deduction line38

>>      line39: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             (((x pair y) = (z pair w)) -> (y
>>             = w)))]
>>        {move 1}


     define proj2 pairsequal :  Cases(line14 pairsequal,\
          line30 pairsequal,line39 pairsequal)

>>      proj2: [(pairsequal_1:that ((x Pair
>>             y) = (z Pair w))) => ( ... :that
>>             (y = w))]
>>        {move 1}


     close
```

```
declare Pairsequal that (x Pair y) = z Pair w

>> Pairsequal: that ((x Pair y) = (z Pair w))
>>   {move 1}


define Proj1 Pairsequal :  proj1 Pairsequal

>> Proj1: [(.x_1:obj),(.y_1:obj),(.z_1:obj),
>>        (.w_1:obj),(Pairsequal_1:that ((.x_1
>>        Pair .y_1) = (.z_1 Pair .w_1))) => (
>>        ... :that (.x_1 = .z_1))]
>>   {move 0}


define Proj2 Pairsequal : proj2 Pairsequal

>> Proj2: [(.x_1:obj),(.y_1:obj),(.z_1:obj),
>>        (.w_1:obj),(Pairsequal_1:that ((.x_1
>>        Pair .y_1) = (.z_1 Pair .w_1))) => (
>>        ... :that (.y_1 = .w_1))]
>>   {move 0}


showdefs
```

We export the projection equality theorems from the block.

```
Lestrade execution:

declare xzev that singleton x = singleton z

>> xzev: that (singleton(x) = singleton(z))
>>   {move 1}


define Equalsingletons x z : line7

>> Equalsingletons: [(x_1:obj),(z_1:obj) =>
>>        (Deduction([(hyp_2:that (singleton(x_1)
>>             = singleton(z_1))) => (Cases(pair3((hyp_2
>>             Substitution2 (x_1 pair1 x_1))),
>>             Deduction([(pp_4:that (x_1 = z_1))
>>                  => (pp_4:that (x_1 = z_1))]),
>>             Deduction([(pp_5:that (x_1 = z_1))
>>                  => (pp_5:that (x_1 = z_1))]))
```

```
>>              :that (x_1 = z_1))])
>>         :that ((singleton(x_1) = singleton(z_1))
>>         -> (x_1 = z_1)))]
>>   {move 0}
```

We export the lemma `line7` as an independent theorem.

We now present the Axiom of Separation, the presumably safe version of the contradictory axiom of comprehension from which Russell's paradox can be deduced. The idea is that a predicate determines a set, when restricted to a previously given set.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare pred [x => prop]

>> pred: [(x_1:obj) => (---:prop)]
>>   {move 1}


construct setof x pred obj

>> setof: [(x_1:obj),(pred_1:[(x_2:obj) => (---:
>>              prop)])
>>         => (---:obj)]
>>   {move 0}


construct Sep0 x pred that Isset setof x pred

>> Sep0: [(x_1:obj),(pred_1:[(x_2:obj) => (---:
>>              prop)])
>>         => (---:that Isset((x_1 setof pred_1)))]
```

```
>>   {move 0}


declare inev1 that y E x

>> inev1: that (y E x) {move 1}


declare inev2 that pred y

>> inev2: that pred(y) {move 1}


construct Sep01 pred, inev1 inev2 that y E setof x pred

>> Sep01: [(pred_1:[(x_2:obj) => (---:prop)]),
>>        (.y_1:obj),(.x_1:obj),(inev1_1:that
>>        (.y_1 E .x_1)),(inev2_1:that pred_1(.y_1))
>>        => (---:that (.y_1 E (.x_1 setof pred_1)))]
>>   {move 0}


define Sep1 inev1 inev2:  Sep01 pred, inev1 inev2

>> Sep1: [(.y_1:obj),(.x_1:obj),(inev1_1:that
>>        (.y_1 E .x_1)),(.pred_1:[(x_2:obj) =>
>>             (---:prop)]),
>>        (inev2_1:that .pred_1(.y_1)) => (Sep01(.pred_1,
>>        inev1_1,inev2_1):that (.y_1 E (.x_1
>>        setof .pred_1)))]
>>   {move 0}


declare inev3 that y E setof x pred

>> inev3: that (y E (x setof pred)) {move 1}


construct Sep2 inev3 that y E x

>> Sep2: [(.y_1:obj),(.x_1:obj),(.pred_1:[(x_2:
>>             obj) => (---:prop)]),
>>        (inev3_1:that (.y_1 E (.x_1 setof .pred_1)))
>>        => (---:that (.y_1 E .x_1))]
>>   {move 0}
```

```
construct Sep3 inev3 that pred y

>> Sep3: [(.y_1:obj),(.x_1:obj),(.pred_1:[(x_2:
>>             obj) => (---:prop)]),
>>        (inev3_1:that (.y_1 E (.x_1 setof .pred_1)))
>>        => (---:that .pred_1(.y_1))]
>>   {move 0}
```

We prove that for every set x, there is y such that $y \notin x$.

```
Lestrade execution:

define Notin x : setof x [y=> ~(y E y)]

>> Notin: [(x_1:obj) => ((x_1 setof [(y_2:obj)
>>             => (~((y_2 E y_2)):prop)])
>>        :obj)]
>>   {move 0}


open

     declare hyp that Notin x E x

>>      hyp: that (Notin(x) E x) {move 2}


     open

          declare hyp2 that Notin x E Notin x

>>           hyp2: that (Notin(x) E Notin(x))
>>             {move 3}


          define line1 hyp2 : Sep3 hyp2

>>           line1: [(hyp2_1:that (Notin(x)
>>                  E Notin(x))) => (Sep3(hyp2_1):
>>                  that ~((Notin(x) E Notin(x))))]
>>             {move 2}
```

```
          define line2 hyp2 : Mp hyp2 line1 hyp2

>>           line2: [(hyp2_1:that (Notin(x)
>>                  E Notin(x))) => ((hyp2_1 Mp
>>                  line1(hyp2_1)):that ??)]
>>             {move 2}


          close

     define line3 :  Negintro line2

>>      line3: [(Negintro([(hyp2_1:that (Notin(x)
>>                  E Notin(x))) => ((hyp2_1 Mp
>>                  Sep3(hyp2_1)):that ??)])
>>             :that ~((Notin(x) E Notin(x))))]
>>        {move 1}


     define line4 hyp:  Fixprop (Notin x E Notin x,Sep1 hyp line3)

>>      line4: [(hyp_1:that (Notin(x) E x))
>>             => (((Notin(x) E Notin(x)) Fixprop
>>             (hyp_1 Sep1 line3)):that (Notin(x)
>>             E Notin(x)))]
>>        {move 1}


     define line5 hyp:  Mp line4 hyp line3

>>      line5: [(hyp_1:that (Notin(x) E x))
>>             => ((line4(hyp_1) Mp line3):that
>>             ??)]
>>        {move 1}


     close

define Reallynotin x : Negintro line5

>> Reallynotin: [(x_1:obj) => (Negintro([(hyp_2:
>>             that (Notin(x_1) E x_1)) => ((((Notin(x_1)
>>             E Notin(x_1)) Fixprop (hyp_2 Sep1
>>             Negintro([(hyp2_4:that (Notin(x_1)
>>                  E Notin(x_1))) => ((hyp2_4
>>                  Mp Sep3(hyp2_4)):that ??)]))
```

```
>>             ) Mp Negintro([(hyp2_6:that (Notin(x_1)
>>                  E Notin(x_1))) => ((hyp2_6
>>                  Mp Sep3(hyp2_6)):that ??)]))
>>             :that ??)])
>>        :that ~((Notin(x_1) E x_1)))]
>>   {move 0}
```

The theorem above is the tame local version of Russell's paradox: for every set x we can produce a non-member of x (quite explicitly!)

We briefly address the question as to whether the axiom implemented in `Sep1-3` is an axiom strictly speaking or a scheme. Our position is that its status is intermediate. As long as we do not define second order quantification (which is possible in Lestrade with suitable declarations), the logical power of our axioms is no greater than that of the usual first order scheme (basically, as long as the only classes (functions from `obj` to `prop`) we can construct are the ones which are defined using the constructions of first order logic). But if we add more primitives allowing construction of classes in more complex ways, we automatically get more power. In Automath, where quantification over any sort is unavoidably implemented, definitions along these lines would unmistakably give second order Zermelo. We do know how to restrict the power of separation so that adding new constructions will not give it any more power than the first order scheme, but use of such axioms would be laborious.

We now tackle the implementation of the axiom of power set.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


define C x y : (Isset x) & Forall [z=> (z E x) -> z E y]
```

```
>> C: [(x_1:obj),(y_1:obj) => ((Isset(x_1) &
>>        Forall([(z_2:obj) => (((z_2 E x_1) ->
>>             (z_2 E y_1)):prop)]))
>>        :prop)]
>>   {move 0}


construct Pow x obj

>> Pow: [(x_1:obj) => (---:obj)]
>>   {move 0}


declare inev that z E y

>> inev: that (z E y) {move 1}


declare inpow that y E Pow x

>> inpow: that (y E Pow(x)) {move 1}


construct Pow1 inev inpow that z E x

>> Pow1: [(.z_1:obj),(.y_1:obj),(inev_1:that
>>        (.z_1 E .y_1)),(.x_1:obj),(inpow_1:that
>>        (.y_1 E Pow(.x_1))) => (---:that (.z_1
>>        E .x_1))]
>>   {move 0}


construct Powb1 inpow that Isset y

>> Powb1: [(.y_1:obj),(.x_1:obj),(inpow_1:that
>>        (.y_1 E Pow(.x_1))) => (---:that Isset(.y_1))]
>>   {move 0}


declare sety that Isset y

>> sety: that Isset(y) {move 1}


declare infun [z,inev => that z E x]
```

```
>> infun: [(z_1:obj),(inev_1:that (z_1 E y))
>>        => (---:that (z_1 E x))]
>>   {move 1}


construct Pow2 sety infun that y E Pow x

>> Pow2: [(.y_1:obj),(sety_1:that Isset(.y_1)),
>>        (.x_1:obj),(infun_1:[(z_2:obj),(inev_2:
>>             that (z_2 E .y_1)) => (---:that
>>             (z_2 E .x_1))])
>>        => (---:that (.y_1 E Pow(.x_1)))]
>>   {move 0}
```

We define the subset relation and introduce basic declarations for the power set construction. We actually have not mentioned the subset relation in the formulation of our primitives supporting the power set: we demonstrate appropriate equivalences. We require that subsets not be atoms, but we do allow atoms to have the empty set as a subset; elements of power sets are stipulated to be sets.

```
Lestrade execution:

open

     declare powev1 that y E Pow x

>>      powev1: that (y E Pow(x)) {move 2}


     declare powev2 that y C x

>>      powev2: that (y C x) {move 2}


     open

          declare z1 obj

>>           z1: obj {move 3}


          open
```

```
              declare ev1 that z1 E y

>>               ev1: that (z1 E y) {move 4}


              define line1 ev1 : Pow1 ev1 powev1

>>               line1: [(ev1_1:that (z1 E
>>                      y)) => ((ev1_1 Pow1 powev1):
>>                      that (z1 E x))]
>>                 {move 3}


              close

          define line2 z1 :  Deduction line1

>>           line2: [(z1_1:obj) => (Deduction([(ev1_2:
>>                       that (z1_1 E y)) => ((ev1_2
>>                       Pow1 powev1):that (z1_1
>>                       E x))])
>>                  :that ((z1_1 E y) -> (z1_1
>>                  E x)))]
>>             {move 2}


          close

     define subpow1 powev1 : Fixprop (y C x,Conjunction(Powb1 powev1,Ug line2))

>>      subpow1: [(powev1_1:that (y E Pow(x)))
>>             => (((y C x) Fixprop (Powb1(powev1_1)
>>             Conjunction Ug([(z1_4:obj) => (Deduction([(ev1_5:
>>                       that (z1_4 E y)) => ((ev1_5
>>                       Pow1 powev1_1):that (z1_4
>>                       E x))])
>>                  :that ((z1_4 E y) -> (z1_4
>>                  E x)))])))
>>             ):that (y C x))]
>>        {move 1}


     open

          declare z1 obj
```

```
>>          z1: obj {move 3}


          declare ev1 that z1 E y

>>          ev1: that (z1 E y) {move 3}


          define line1 z1:  Ui Simplification2 powev2 z1

>>          line1: [(z1_1:obj) => ((Simplification2(powev2)
>>                 Ui z1_1):that ((z1_1 E y)
>>                 -> (z1_1 E x)))]
>>            {move 2}


          define line3 z1 ev1 :  Mp ev1 line1 z1

>>          line3: [(z1_1:obj),(ev1_1:that
>>                 (z1_1 E y)) => ((ev1_1 Mp
>>                 line1(z1_1)):that (z1_1 E
>>                 x))]
>>            {move 2}


          close

     define subpow2 powev2 : Pow2 Simplification1 powev2 line3

>>      subpow2: [(powev2_1:that (y C x)) =>
>>             ((Simplification1(powev2_1) Pow2
>>             [(z1_3:obj),(ev1_3:that (z1_3 E
>>                  y)) => ((ev1_3 Mp (Simplification2(powev2_1)
>>                  Ui z1_3)):that (z1_3 E x))])
>>             :that (y E Pow(x)))]
>>        {move 1}


     close

declare Powev1 that y E Pow x

>> Powev1: that (y E Pow(x)) {move 1}
```

```
declare Powev2 that y C x

>> Powev2: that (y C x) {move 1}


define Subpow1 Powev1:subpow1 Powev1

>> Subpow1: [(.y_1:obj),(.x_1:obj),(Powev1_1:
>>        that (.y_1 E Pow(.x_1))) => (((.y_1
>>        C .x_1) Fixprop (Powb1(Powev1_1) Conjunction
>>        Ug([(z1_4:obj) => (Deduction([(ev1_5:
>>                  that (z1_4 E .y_1)) => ((ev1_5
>>                  Pow1 Powev1_1):that (z1_4
>>                  E .x_1))])
>>             :that ((z1_4 E .y_1) -> (z1_4 E
>>             .x_1)))])))
>>        ):that (.y_1 C .x_1))]
>>   {move 0}


define Subpow2 Powev2:subpow2 Powev2

>> Subpow2: [(.y_1:obj),(.x_1:obj),(Powev2_1:
>>        that (.y_1 C .x_1)) => ((Simplification1(Powev2_1)
>>        Pow2 [(z1_3:obj),(ev1_3:that (z1_3 E
>>             .y_1)) => ((ev1_3 Mp (Simplification2(Powev2_1)
>>             Ui z1_3)):that (z1_3 E .x_1))])
>>        :that (.y_1 E Pow(.x_1)))]
>>   {move 0}
```

We have presented a proof of the equivalence of membership in the power set as defined with our given axioms with the subset relation as usually formalized. We could have given these rules as the basic axioms for the power set construction as well: we think that the ones we have given are somehow more basic. In any event the approaches are demonstrably equivalent.

We now develop the axiom of union. In developing the primitives for union, we were not as successful at avoiding explicit use of logical constructions as we were with power set.

```
Lestrade execution:

clearcurrent

declare x obj
```

```
>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


construct Union x obj

>> Union: [(x_1:obj) => (---:obj)]
>>   {move 0}


declare inev1 that y E z

>> inev1: that (y E z) {move 1}


declare inev2 that z E x

>> inev2: that (z E x) {move 1}


construct Union1 inev1 inev2 that y E Union x

>> Union1: [(.y_1:obj),(.z_1:obj),(inev1_1:that
>>        (.y_1 E .z_1)),(.x_1:obj),(inev2_1:that
>>        (.z_1 E .x_1)) => (---:that (.y_1 E
>>        Union(.x_1)))]
>>   {move 0}


declare inev3 that y E Union x

>> inev3: that (y E Union(x)) {move 1}


construct Union2 inev3 that Exists [z=>(y E z) & z E x]
```

```
>> Union2: [(.y_1:obj),(.x_1:obj),(inev3_1:that
>>        (.y_1 E Union(.x_1))) => (---:that Exists([(z_2:
>>             obj) => (((.y_1 E z_2) & (z_2 E
>>             .x_1)):prop)]))
>>        ]
>>   {move 0}


construct Union3 x that Isset Union x

>> Union3: [(x_1:obj) => (---:that Isset(Union(x_1)))]
>>   {move 0}
```

We define some basic operations of set theory (from the standpoint of undergraduate students: they are actually seen to be derivative notions here).

```
Lestrade execution:

declare u obj

>> u: obj {move 1}


declare v obj

>> v: obj {move 1}


define U x y :  Union (x pair y)

>> U: [(x_1:obj),(y_1:obj) => (Union((x_1 pair
>>        y_1)):obj)]
>>   {move 0}


define cap x y :  setof x [z => z E y]

>> cap: [(x_1:obj),(y_1:obj) => ((x_1 setof
>>        [(z_2:obj) => ((z_2 E y_1):prop)])
>>        :obj)]
>>   {move 0}


define -- x y :  setof x [z => ~(z E y)]
```

```
>> --: [(x_1:obj),(y_1:obj) => ((x_1 setof [(z_2:
>>             obj) => (~((z_2 E y_1)):prop)])
>>        :obj)]
>>   {move 0}


define X x y :  setof ((Pow(Pow(x U y))), \
     [z=>Exists[u=> Exists[v=>(u E x) & (v E y) & (z = (u Pair v))]]])

>> X: [(x_1:obj),(y_1:obj) => ((Pow(Pow((x_1
>>        U y_1))) setof [(z_2:obj) => (Exists([(u_3:
>>                  obj) => (Exists([(v_4:obj)
>>                       => (((u_3 E x_1) & ((v_4
>>                       E y_1) & (z_2 = (u_3
>>                       Pair v_4)))):prop)])
>>                  :prop)])
>>             :prop)])
>>        :obj)]
>>   {move 0}
```

The union, intersection, relative difference and Cartesian product operations of course cry out for proofs of relevant theorems.

We develop the axiom of infinity (using Zermelo's original implementation of the natural numbers).

```
Lestrade execution:

clearcurrent

declare m obj

>> m: obj {move 1}


declare n obj

>> n: obj {move 1}


construct Nat obj

>> Nat: obj {move 0}
```

```
construct Zeroax that Empty E Nat

>> Zeroax: that (Empty E Nat) {move 0}


declare natev that n E Nat

>> natev: that (n E Nat) {move 1}


construct Succax natev that singleton n E Nat

>> Succax: [(.n_1:obj),(natev_1:that (.n_1 E
>>        Nat)) => (---:that (singleton(.n_1)
>>        E Nat))]
>>   {move 0}


declare I obj

>> I: obj {move 1}


declare izeroev that Empty E I

>> izeroev: that (Empty E I) {move 1}


declare  iniev that n E I

>> iniev: that (n E I) {move 1}


declare inductionev [n,iniev => that singleton n E I]

>> inductionev: [(n_1:obj),(iniev_1:that (n_1
>>        E I)) => (---:that (singleton(n_1) E
>>        I))]
>>   {move 1}


declare  iniev2 that n E I

>> iniev2: that (n E I) {move 1}
```

```
construct Induction izeroev inductionev, iniev2 that n E Nat

>> Induction: [(.I_1:obj),(izeroev_1:that (Empty
>>        E .I_1)),(inductionev_1:[(n_2:obj),(iniev_2:
>>             that (n_2 E .I_1)) => (---:that
>>             (singleton(n_2) E .I_1))]),
>>        (.n_1:obj),(iniev2_1:that (.n_1 E .I_1))
>>        => (---:that (.n_1 E Nat))]
>>   {move 0}
```

In defining the primitives for the natural numbers, we continue our pattern of avoiding logical constructions in the declarations of our primitives. A more usual form of mathematical induction using quantifiers can certainly be developed, and proofs of the axioms of Peano arithmetic would be in order here. Development of the iteration theorem would also be useful.

We now develop the axiom of choice. There is a quick and brutal way, and the usual way.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


construct Choose x obj

>> Choose: [(x_1:obj) => (---:obj)]
>>   {move 0}


declare inev that y E x

>> inev: that (y E x) {move 1}
```

```
construct Choose1 inev that Choose x E x

>> Choose1: [(.y_1:obj),(.x_1:obj),(inev_1:that
>>        (.y_1 E .x_1)) => (---:that (Choose(.x_1)
>>        E .x_1))]
>>   {move 0}
```

It's that simple: provide a way to choose an element from each nonempty set. And results about the conservative nature of global choice as an extension of ordinary set theory suggest that this is essentially harmless. However, we present a more cautious approach as well.

```
Lestrade execution:

clearcurrent

declare P obj

>> P: obj {move 1}


declare A obj

>> A: obj {move 1}


declare B obj

>> B: obj {move 1}


declare x obj

>> x: obj {move 1}


declare ainp that A E P

>> ainp: that (A E P) {move 1}


declare binp that B E P

>> binp: that (B E P) {move 1}
```

```
declare xina that x E A

>> xina: that (x E A) {move 1}


declare xinb that x E B

>> xinb: that (x E B) {move 1}


declare inhabited  [A,ainp => that Exists[x=>x E A]]

>> inhabited: [(A_1:obj),(ainp_1:that (A_1 E
>>        P)) => (---:that Exists([(x_2:obj) =>
>>             ((x_2 E A_1):prop)]))
>>        ]
>>   {move 1}


declare pairwise  [A,B,x,ainp,binp,xina,xinb => that A=B]

>> pairwise: [(A_1:obj),(B_1:obj),(x_1:obj),
>>        (ainp_1:that (A_1 E P)),(binp_1:that
>>        (B_1 E P)),(xina_1:that (x_1 E A_1)),
>>        (xinb_1:that (x_1 E B_1)) => (---:that
>>        (A_1 = B_1))]
>>   {move 1}


declare C1 obj

>> C1: obj {move 1}


declare c obj

>> c: obj {move 1}


define choiceset P C1 : Forall [c=> (c E C1) -> \
     Exists[A => (A E P) & singleton c = A cap C1]] \
          & Forall[A=>(A E P)-> Exists[c => (c E C1) & c E A]]

>> choiceset: [(P_1:obj),(C1_1:obj) => ((Forall([(c_2:
```

```
>>             obj) => (((c_2 E C1_1) -> Exists([(A_3:
>>                  obj) => (((A_3 E P_1) & (singleton(c_2)
>>                  = (A_3 cap C1_1))):prop)]))
>>             :prop)])
>>        & Forall([(A_4:obj) => (((A_4 E P_1)
>>             -> Exists([(c_5:obj) => (((c_5
>>                  E C1_1) & (c_5 E A_4)):prop)]))
>>             :prop)]))
>>        :prop)]
>>   {move 0}


construct Choice P, inhabited, pairwise  that Exists[C1 => choiceset P C1]

>> Choice: [(P_1:obj),(inhabited_1:[(A_2:obj),
>>             (ainp_2:that (A_2 E P_1)) => (---:
>>             that Exists([(x_3:obj) => ((x_3
>>                  E A_2):prop)]))
>>             ]),
>>        (pairwise_1:[(A_4:obj),(B_4:obj),(x_4:
>>             obj),(ainp_4:that (A_4 E P_1)),
>>             (binp_4:that (B_4 E P_1)),(xina_4:
>>             that (x_4 E A_4)),(xinb_4:that
>>             (x_4 E B_4)) => (---:that (A_4
>>             = B_4))])
>>        => (---:that Exists([(C1_5:obj) => ((P_1
>>             choiceset C1_5):prop)]))
>>        ]
>>   {move 0}
```

We present the usual formulation of the axiom of choice, asserting that every partition has a choice set.

4.3 Toward a proof of the Well-Ordering Theorem

Our aim in this section is to prove the Well-Ordering Theorem. Supporting the proof of this theorem seems to have been the immediate motivation for Zermelo's presentation of his axioms in 1908. We will emulate Zermelo in attempting a proof of the theorem which makes no use of the ordered pair: Zermelo did not know that ordered pairs were representable in set theory; this was revealed by Norbert Wiener in 1914.

Since we are not going to represent relations as sets of ordered pairs, we have to explain how we are going to represent them.

Lestrade execution:

```
clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


declare R [x,y => prop]

>> R: [(x_1:obj),(y_1:obj) => (---:prop)]
>>   {move 1}


define transitive R : Forall [x=>Forall [y=>Forall [z=>((x R y) & (y R z))->x R z]]]

>> transitive: [(R_1:[(x_2:obj),(y_2:obj) =>
>>             (---:prop)])
>>        => (Forall([(x_3:obj) => (Forall([(y_4:
>>                  obj) => (Forall([(z_5:obj)
>>                       => ((((x_3 R_1 y_4) &
>>                       (y_4 R_1 z_5)) -> (x_3
>>                       R_1 z_5)):prop)])
>>                  :prop)])
>>             :prop)])
>>        :prop)]
>>   {move 0}


declare A obj

>> A: obj {move 1}


define field R, A: Forall[x=>Forall[y=> (x R y) -> (x E A) & y E A]]
```

```
>> field: [(R_1:[(x_2:obj),(y_2:obj) => (---:
>>             prop)]),
>>        (A_1:obj) => (Forall([(x_3:obj) => (Forall([(y_4:
>>                  obj) => (((x_3 R_1 y_4) ->
>>                  ((x_3 E A_1) & (y_4 E A_1))):
>>                  prop)])
>>             :prop)])
>>        :prop)]
>>   {move 0}


declare fldev that field R,A

>> fldev: that field(R,A) {move 1}


define reflexive fldev : Forall[x=>(x E A) -> x R x]

>> reflexive: [(.R_1:[(x_2:obj),(y_2:obj) =>
>>             (---:prop)]),
>>        (.A_1:obj),(fldev_1:that field(.R_1,
>>        .A_1)) => (Forall([(x_3:obj) => (((x_3
>>             E .A_1) -> (x_3 .R_1 x_3)):prop)])
>>        :prop)]
>>   {move 0}


define relrep x y A: (y E Union A) & Forall[z=> ((z E A) & (y E z))-> x E z]

>> relrep: [(x_1:obj),(y_1:obj),(A_1:obj) =>
>>        (((y_1 E Union(A_1)) & Forall([(z_2:
>>             obj) => ((((z_2 E A_1) & (y_1 E
>>             z_2)) -> (x_1 E z_2)):prop)]))
>>        :prop)]
>>   {move 0}


declare u obj

>> u: obj {move 1}


define seg fldev u :  setof A [y=>y R u]

>> seg: [(.R_1:[(x_2:obj),(y_2:obj) => (---:
>>             prop)]),
```

```
>>        (.A_1:obj),(fldev_1:that field(.R_1,
>>        .A_1)),(u_1:obj) => ((.A_1 setof [(y_3:
>>             obj) => ((y_3 .R_1 u_1):prop)])
>>        :obj)]
>>   {move 0}


define segset fldev u : Sep0 A [y=>y R u]

>> segset: [(.R_1:[(x_2:obj),(y_2:obj) => (---:
>>             prop)]),
>>        (.A_1:obj),(fldev_1:that field(.R_1,
>>        .A_1)),(u_1:obj) => ((.A_1 Sep0 [(y_3:
>>             obj) => ((y_3 .R_1 u_1):prop)])
>>        :that Isset((.A_1 setof [(y_4:obj) =>
>>             ((y_4 .R_1 u_1):prop)]))
>>        )]
>>   {move 0}


define Setrel fldev : setof (Pow A, [z=>Exists [x=>z=seg fldev x]])

>> Setrel: [(.R_1:[(x_2:obj),(y_2:obj) => (---:
>>             prop)]),
>>        (.A_1:obj),(fldev_1:that field(.R_1,
>>        .A_1)) => ((Pow(.A_1) setof [(z_3:obj)
>>             => (Exists([(x_4:obj) => ((z_3
>>                  = (fldev_1 seg x_4)):prop)])
>>             :prop)])
>>        :obj)]
>>   {move 0}
```

We present the basic definitions supporting an implementation of quasi-orders (reflexive, transitive relations) as sets, making no use of ordered pairs. A quasi-order R on a set A is implemented as the set of segments

$$\mathtt{seg}_R(x) = \{y \in A : yRx\}$$

in the relation on the set. If the relation R is represented by the set r, we have $xRy \leftrightarrow y \in \bigcup r \wedge (\forall z \in r : y \in z \to x \in z)$. Our first aim is to prove that all quasi-orders are represented by sets in this way.

```
Lestrade execution:

clearcurrent
```

```
open

     declare x obj

>>       x: obj {move 2}


     declare y obj

>>       y: obj {move 2}


     declare z obj

>>       z: obj {move 2}


     declare R [x,y=> prop]

>>       R: [(x_1:obj),(y_1:obj) => (---:prop)]
>>         {move 2}


     declare transev that transitive R

>>       transev: that transitive(R) {move 2}


     declare A obj

>>       A: obj {move 2}


     declare fldev that field R, A

>>       fldev: that field(R,A) {move 2}


     declare reflev that reflexive fldev

>>       reflev: that reflexive(fldev) {move
>>         2}


     open
```

```
          declare u obj

>>           u: obj {move 3}


          declare relev1 that x R y

>>           relev1: that (x R y) {move 3}


          declare relev2 that x relrep y, Setrel fldev

>>           relev2: that relrep(x,y,Setrel(fldev))
>>             {move 3}
```

In text, we will refer to `Setrel fldev` as r, the set representing the reflexive, transitive relation R with field A. Our first aim is to show that if xRy (`relev1`) then $y \in \bigcup r \wedge (\forall z \in r : y \in z \rightarrow x \in z)$. We begin by observing that $y \in \bigcup r$ because $y \in \mathtt{seg}_R(y) \in r$.

```
Lestrade execution:

          define line1 :  Ui (Ui fldev x, y)

>>           line1: [(((fldev Ui x) Ui y):that
>>                  ((x R y) -> ((x E A) & (y
>>                  E A))))]
>>             {move 2}


          define line2 relev1 :  Mp relev1 line1

>>           line2: [(relev1_1:that (x R y))
>>                  => ((relev1_1 Mp line1):that
>>                  ((x E A) & (y E A)))]
>>             {move 2}


          define line3 relev1 : Simplification1 line2 relev1

>>           line3: [(relev1_1:that (x R y))
>>                  => (Simplification1(line2(relev1_1)):
>>                  that (x E A))]
```

```
>>             {move 2}


          define line4 relev1 : Simplification2 line2 relev1

>>           line4: [(relev1_1:that (x R y))
>>                  => (Simplification2(line2(relev1_1)):
>>                  that (y E A))]
>>             {move 2}


          define line5 : Ui reflev y

>>           line5: [((reflev Ui y):that ((y
>>                  E A) -> (y R y)))]
>>             {move 2}


          define line7 relev1 :  Mp line4 relev1 line5

>>           line7: [(relev1_1:that (x R y))
>>                  => ((line4(relev1_1) Mp line5):
>>                  that (y R y))]
>>             {move 2}


          define line8 relev1 : Fixprop(y E seg fldev y,\
               Sep01 ([u=>u R y], line4 relev1,  line7 relev1))

>>           line8: [(relev1_1:that (x R y))
>>                  => (((y E (fldev seg y)) Fixprop
>>                  Sep01([(u_2:obj) => ((u_2
>>                       R y):prop)]
>>                  ,line4(relev1_1),line7(relev1_1))):
>>                  that (y E (fldev seg y)))]
>>             {move 2}
```

We need to show that $\mathtt{seg}_R(y) \in \mathcal{P}(A)$.

```
Lestrade execution:

          define line9 :  segset fldev y

>>           line9: [((fldev segset y):that
```

```
>>                  Isset((A setof [(y_1:obj)
>>                       => ((y_1 R y):prop)]))
>>                  )]
>>              {move 2}


          declare v obj

>>           v: obj {move 3}


          declare  insegev that v E seg fldev y

>>           insegev: that (v E (fldev seg y))
>>             {move 3}


          define line10 : Fixprop((seg fldev y) E Pow A ,\
       Pow2 line9 [v,insegev => Simplification1\
            Mp(Sep3 insegev, Ui(Ui fldev v,y))])

>>           line10: [((((fldev seg y) E Pow(A))
>>                  Fixprop (line9 Pow2 [(v_2:
>>                       obj),(insegev_2:that
>>                       (v_2 E (fldev seg y)))
>>                       => (Simplification1((Sep3(insegev_2)
>>                       Mp ((fldev Ui v_2) Ui
>>                       y))):that (v_2 E A))]))
>>                  :that ((fldev seg y) E Pow(A)))]
>>             {move 2}


          define line11 : Eib [u=> (fldev seg y) = fldev seg u] Reflexive (fldev seg y)

>>           line11: [(Eib([(u_1:obj) => (((fldev
>>                       seg y) = (fldev seg u_1)):
>>                       prop)]
>>                  ,Reflexive((fldev seg y))):
>>                  that Exists([(u_2:obj) =>
>>                       (((fldev seg y) = (fldev
>>                       seg u_2)):prop)]))
>>                  ]
>>             {move 2}


          define line12 :  Fixprop((fldev seg y) E Setrel fldev, Sep1 (line10,line11))
```

```
>>           line12: [((((fldev seg y) E Setrel(fldev))
>>                  Fixprop (line10 Sep1 line11)):
>>                  that ((fldev seg y) E Setrel(fldev)))]
>>             {move 2}


          define line13 relev1 :  Union1 line8 relev1 line12

>>           line13: [(relev1_1:that (x R y))
>>                  => ((line8(relev1_1) Union1
>>                  line12):that (y E Union(Setrel(fldev))))]
>>             {move 2}
```