A formal presentation of a stratified version of Frege's Basic Laws of Arithmetic

M. Randall Holmes

November 27, 2015

We present a system which the careful reader will realize has the same deductive force as Frege's system in the Basic Laws of Arithmetic, though the formal details will not be exactly the same, with a restriction on the formation of terms which demonstrably prevents the inconsistencies: this will be demonstrated by observing that the system can be interpreted in *NFU* with Rosser's Axiom of Counting. Our notation will be modern, but similar to Frege's in ways which will assist the careful reader in comparing the system with the original.

revision 11/25/2015: further correction to the definition of value-range of a function of two variables: a similar issue with the typing of currying. This is an encapsulated implementation issue: once it is clear that these things can be implemented, one can forget the details.

Some discussion of similarities and differences between Frege's formulation and the logic of Python Marcel will be needed.

11/23/2015: corrected error in the definition of ordered pair (and made corresponding corrections to the definition of projection operators). This was a stupid mechanical mistake. Unfortunately, the corrected definition is even nastier-looking than before. I believe that this correction has no knock-on effects further on (as in the definition of substitution) but I will not be surprised if I find that it does cause a need to correct some details.

It is worth noting that the need for the correction was discovered by entering the original incorrect definitions into the Python version of my

Marcel software: after correcting the definition I was able to prove the basic equations relating the pair and projection operators.

1 The language

There are two sorts of thing in Frege's world, objects and functions. Each object or function term has a natural number type (where by natural number we mean positive integer) or is type-free. All constant expressions are type-free; neither objects nor functions are actually partitioned into sorts by the typing mechanism. The lowest type is type 1, rather than the more usual type 0, because we want the notion of order to have the same indices as the notion of type (objects, the lowest type, are first order, and so we want them to be type 1).

We list the sorts of object term and their typing rules (and rules for free occurrence of bound variables).

variables: There is a countable supply of free object variables, which are type-free. There is a countable supply of bound object variables of each natural number type. There is a countable supply of first argument object variables X_i, one of each type i. There is a type-free first argument proposition variable P, and we could add a type-free first argument number variable M. There is a countable supply of second argument object variables Y_i, one of each type i. There is a second argument proposition variable Q, and we could add a second argument number variable N. A bound variable is the only variable free in itself; nothing is free in any other sort of variable.

constants: There are primitive constants **t** and **f** (the truth-values) and new constants may be introduced by definition, identified with object terms containing no free occurrence of any bound variable (such terms are always type-free). A constant is type-free and no variable is free in it.

unary operator terms: Each unary operator has a type signature, a list of two items, each of which is either `prop` (or `number`) or a natural number. The smallest natural number appearing in a type signature (if one does appear) will be 1. The first item is the input type and the second is the output type. A unary operator term is of the form $?t$,

where ? is a unary operator and t is an object term. If t has natural number type and ? has natural number input and output type, the difference between the type of $?t$ and the type of t will be the difference between the output and input types of ?; otherwise $?t$ will be type-free. The set of variables free in $?t$ is the same as the set of variables free in t.

The usual convention of iteration of a unary operator is understood: $?^0 t$ has the same reference as t, and $?^{n+1} t$ has the same reference as $?^n(?t)$; n here is a natural number of the metatheory.

binary operator terms: Each binary operator has a type signature, a list of three items, each of which is either `prop` (or `number`) or a natural number. The smallest natural number appearing in a type signature (if one does appear) will be 1. The first item is the left input type, the second item is the right input type, and the third item is the output type. A binary operator term is of the form $(t \circ u)$, where t is an object term, u is an object term, and $\circ$ is a binary operator. If t (resp. u) has natural number type and $\circ$ has natural number output and left (resp. right) type, then the difference between the type of $t \circ u$ and the type of t (resp. u) will be the difference between the output type of $\circ$ and the left (resp. right) input type of $\circ$. Otherwise $t \circ u$ is type-free. The set of variables free in $(t \circ u)$ is the union of the set of variables free in t and the set of variables free in u.

In the same category as binary operator terms are two argument terms of the form $f(x, y)$ where f is a free function variable of arity 2 (we provide a countable supply of these variables with each indicated type signature). These are handled as if they were of the shape $x\, f\, y$ with f viewed as the operator; the type signature of each such variable as a binary operator is either $(1, 1, 1)$ or $(1, 1, \texttt{prop})$ (two argument Function or Relation variables).

function variable application terms: These terms are of the form $f(t)$ where f is a function variable and t is an object term. $f(t)$ will be one type lower than f if f has natural number type. $f(t)$ will be the same type as t if t has natural number type. Otherwise $f(t)$ will be type-free. The variables free in $f(t)$ will be the variables free in t plus f itself if it is a bound variable.

If predicate variables were to be added, and if f were a predicate variable, $f(t)$ would be type-free in all cases (and a syntactical proposition), and if f and t both have natural number type, the type of f must exceed the type of t by one.

bracketed application term: Where t is an object term and u and v are object terms, $[t](u)$ and $[t](u, v)$ will be terms under certain conditions. $[t]$ must be a well-formed bracket abstraction term (see below). In a term $[t](u)$, t may not contain any second argument variable except in a proper bracketed subterm. The set of variables free in $[t](u)$ or $[t](u, v)$ will be the set of variables free in u in the first case, and the union of the set of variables free in u and the set of variables free in v in the second case. The type of $[t](u)$ is typed from the type signature of $[t]$ exactly as if the term were $[t]u$ with $[t]$ a unary operator. The type of $[t](u, v)$ is computed from the type signature of $[t]$ exactly as if the shape of the term were $u[t]v$ with $[t]$ a binary operator.

Function term: A Function (course-of-values) term will be of the form $(\lambda x.t)$ where t is an object term and x is a bound object variable. The type of $(\lambda x.t)$ will be one higher than the type of x if there is any variable other than x free in t and t is not type-free; the term will otherwise be type-free. If t has a type it will be the same as the type of x. The variables free in $(\lambda x.t)$ will be those free in t, other than x.

quantification terms: These terms are of the form $(\forall u.t)$ where u is a bound object or function variable and t is an object term. This term is type-free. The variables free in $(\forall u.t)$ are those free in t, other than u.

We list the sorts of function term and their typing rules (and rules for free variables). An object term is a syntactical proposition if it is either a truth value (**t** or **f**), a quantification term or an operator term with the operator having output type `prop`. A syntactical number is an operator term with the operator having output type `number`.

function variables: There is a countable supply of free function variables (which are type-free), a countable supply of bound function variables of each type, a first argument function variable F_i of each type i, and a second argument function variable G_i of each type i. It would be reasonable to add predicate variables but they are not formally necessary.

A bound function variable is the only variable free in itself; nothing is free in any other sort of function variable.

bracketed abstraction terms : A bracketed abstraction term will be of the form $[t]$ where t is an object term. The term t may contain no more than one first argument variable and no more than one second argument variable outside of proper bracketed subterms. No bound variable may occur free in t. We define the type signature of a term $[t]$ as a pair or triple of numbers, the first being the type of the first argument variable (or `prop` if it is P, or `number` if it is M, or 1 if no such variable is present), the second (if there are three items) being the type of the second argument variable (or `prop` if it is Q, or `number` if it is N, or 1 if no such variable is present), and the last being the type of t (this being `prop` if t is a syntactical proposition or `number` if t is a syntactical number, or 1 if it is type-free and cannot be otherwise typed). The minimum natural number in the type signature must be 1 (if the signature cannot satisfy this condition, the term is ill-formed, but this can always be fixed by a uniform displacement of types of argument variables). New unary and binary operators can be defined by equating them with bracket abstraction terms, giving them the same type signature. No variable is free in a bracketed abstraction term.

It is important to note that the notation $(\lambda x.[P](x))$ or $(\lambda x.[P](y,x))$ is unambiguous: the term P cannot contain any occurrence of x, and the same is true for quantification terms.

The bracketed abstraction terms correspond to Frege's function constants, which contain one or two free variables just as here, and just as here he uses special shapes for these variables. The device of brackets allows us to include these notations as constants in terms, in the very specific context of bracketed function application terms, which gives us a notation for substitution.

We list the primitive operators.

truth values: **t** and **f** are primitive object constants.

assertion: – is a unary operator with signature $(\texttt{prop}, \texttt{prop})$.

negation: $\neg$ is a unary operator with signature $(\texttt{prop}, \texttt{prop})$.

implication: $\rightarrow$ is a binary operator with signature $(\texttt{prop}, \texttt{prop}, \texttt{prop})$.

equality: $=$ is a binary operator with signature $(1, 1, \texttt{prop})$.

description: θ is a unary operator with signature $(2, 1)$.

We list some important defined operators. The equations given with these operators are not theorems or axioms at this point: they will be proved using the axioms and rules given below. We do not supply type indices on argument variables: these are always deducible.

application: The binary operator ‘ is defined as

$$[\theta(\lambda z.(\exists f.Y = (\lambda x.f(x)) \wedge f(X) = z))].$$

Signature $(2, 1, 1)$.

The reason that we use the application notation of PM is to make a distinction between application of a second order function f to an object x, for which we already use the modern notation $f(x)$ and the application of a first order Function y to an object x, for which we write $y‘x$. The connection is that $(\lambda z.f(z))‘(x) = f(x)$.

constant function: The unary operator K is defined as $[(\lambda x.X)]$. Signature $(2, 1)$.

The relevant equation is $Kx‘(y) = x$.

ordered pair: The binary operator , is defined as $[(\lambda f.K^2(f‘(KX))‘Y))]$. We always write $\langle x, y\rangle$ instead of (x, y). Signature $(1, 1, 4)$. This is similar but not identical to Frege's pair: it is defined this way so that it will be typed correctly.

Alternative ordered pairs $\langle x, y\rangle_{+n}$ and $\langle x, y\rangle_{-n}$, where the integer subscript is the difference between the type of y and the type of x, are defined as $\langle K^n x, y\rangle$ (note that here the type of y must be n higher than the type of x) and $\langle x, K^n y\rangle$ (note that here the type of x must be n higher than the type of y). Strictly speaking these are binary operator terms. One can suppose that the true form of $\langle x, y\rangle_n$ for any integer n (of the metatheory) is $x,_n y$, but putting a subscript on a comma is weird. These are used when the types of the two arguments of an ordered pair are different. $\langle x, y\rangle_0$ is defined as $\langle x, y\rangle$.

double value range: We will not use this unless we go quite deep into Frege's development, but it's useful in connection with understanding the difference between our pair and Frege's to know that the double value range of a function f of two arguments (understood to be of the same type if neither is type-free) will be $(\lambda x.(\lambda y.f(x`\mathbf{f}, y)))$ instead of $(\lambda x.(\lambda y.f(x, y)))$: the reason, again, is that the modified definition is well-typed.

The relevant equation is $((\lambda x.(\lambda y.f(x`\mathbf{f}, y)))`(a))`(b) = f(a`\mathbf{f}, b)$, or perhaps more informatively $((\lambda x.(\lambda y.f(x`\mathbf{f}, y)))`(Ka))`(b) = f(a, b)$ In this construction, the variable x does not actually represent the argument of f in which we are interested, but a function which takes the value of interest at $\mathbf{f}$, which we may thriftily suppose to be the constant function of the value of interest (compare with the use of constant functions in the definition of ordered pair).

To give a relevant example, the converse operation on value ranges takes a function f used as a value range to $(\lambda x.(\lambda y.f(y, x`\mathbf{f})))$: the function $(\lambda f.(\lambda x.(\lambda y.f(y, x`\mathbf{f}))))$ is well typed and implements this conversion operation.

projection operators: The unary operator π_1 is defined as $[((X`(\lambda x.(\lambda y.x`\mathbf{f})))`\mathbf{f})`\mathbf{f}]$. The equation of interest is $\pi_1(\langle x, y\rangle) = x$.

The unary operator π_2 is defined as $[((X`(\lambda x.(\lambda y.y)))`\mathbf{f})`\mathbf{f}]$. The equation of interest is $\pi_2(\langle x, y\rangle) = y$.

The relevant equations are $\pi_1 \langle x, y\rangle = x$ and $\pi_2 \langle x, y\rangle = y$. The type signatures are $(4, 1)$.

We define alternative projection operators $\pi_{1,+n}$ and $\pi_{2,+n}$, $\pi_{1,-n}$ and $\pi_{2,-n}$. A prerequisite for their definition is to define `atnull` as $[X`\mathbf{f}]$. We can then define $\pi_{1,+n}$ as $[\texttt{atnull}^n \pi_1 X]$ and $\pi_{2,+n}$ as π_2, and define $\pi_{1,-n}$ as π_1 and $\pi_{2,-n}$ as $[\texttt{atnull}^n \pi_2 X]$. $\pi_{1,0}$ and $\pi_{2,0}$ are synonymous with π_1, π_2, respectively.

set builder notation: We can define $\{x : [A](x)\}$ as $(\lambda x.\text{–}[A]x)$.

We could add a primitive or even a defined operator # so that $\#n$ is n for each natural number n, with output type `number`, which would enable us to define the notation of a syntactical number analogously to the notion of a syntactical proposition. This would enable Rosser's Axiom of Counting to

hold for reasons intrinsic to the type system, which would allow one to carry out Frege's proof of Infinity in the system. Mathematical fluency could be increased by having a larger domain for #: a sensible set of axioms would assert that this operation fixes 0, commutes with successor, fixes $\aleph_0$ and commutes with the operator $\kappa \to 2^\kappa$ on cardinal numbers. This much is also justifiable on the basis of the axiom of counting.

2 The definition of substitution

Where A is an object term, we indicate how to compute $[A](u)$ or $[A](u, v)$, where this represents the result of substituting u for the first argument variable and v for the second argument variable in A [restricting of course to ingredients that are present]. We simplify our scheme by making all arguments objects and casting Function terms to the corresponding functions to effect substitutions of values for function argument variables.

A rule $t \Rightarrow u$ means that a term of the form t may be reduced to a term of the form u in the computation of a substitution.

Here are rules for the argument variables. Type indices on argument variables X_i, Y_i, F_i, G_i play no explicit role in these computations so are not expressed.

$[X](t, u) \Rightarrow [X](t) \Rightarrow t$

$[Y](t, u) \Rightarrow u$

$[P](t, u) \Rightarrow [P](t) \Rightarrow -t$, unless t is a syntactical proposition (either a quantified term or an operator term with the output type of the operator `prop`), in which case $[P](t, u) \Rightarrow [P](t) \Rightarrow a$.

$[t](u, v) \Rightarrow [t](u) \to t$ when t contains no argument variables.

$[Q](t, u) \Rightarrow -u$, unless u is a syntactical proposition, in which case $[Q](t, u) \Rightarrow u$

Here are rules for function application terms.

$[F(A)](u) \Rightarrow u`([A](u))$ when u is not a Function term.

$[F(A)](u, v) \Rightarrow u`([A](u, v))$ when u is not a Function term.

$[G(A)](u, v) \Rightarrow v`([A](u, v))$ when v is not a Function term.

$[F(A)](\lambda x.[B](x)) \Rightarrow [B]([A](\lambda x.[B](x)))$

$[F(A)](\lambda x.[B](x), v) \Rightarrow [B]([A](\lambda x.[B](x), v))$

$[G(A)](u, \lambda x.[B](x)) \Rightarrow [B]([A](u, \lambda x.[B](x)))$

$[f(A)](u) \Rightarrow f([A](u))$ when f is any other function term other than an argument variable (including bracketed terms)

$[f(A)](u,v) \Rightarrow f([A](u,v))$ when f is any other function term other than an argument variable (including bracketed terms).

Here are the rules for unary and binary operation terms. The rule for free function variable application terms with two arguments can be deduced from the binary operation rules by construing the function as a binary operation with different syntax.

$[?A](u) \Rightarrow ?([A](u))$

$[?A](u,v) \Rightarrow ?([A](u,v)$

$[A \circ B](u) \Rightarrow [A](u) \circ [B](u)$

$[A \circ B](u,v) \Rightarrow [A](u,v) \circ [B](u,v)$

Here are the rules for variable binding forms.

$[(\forall x.[A](U,x))](u) = (\forall w.[A](u,w))$ where U is any first argument variable, and w does not appear on the left side.

$[(\forall f.[A](U,(\lambda y.f(y))))](u) = (\forall g.[A](u,(\lambda y.g(y))))$, where U is any first argument variable and g does not appear on the left side.

$[(\lambda x.[A](U,x))](u) = (\lambda w.[A](u,w))$ where U is any first argument variable and w does not appear on the left side.

Here are special rules for projection operators. These are applied before the operator rules which would otherwise be applied. The letter n denotes any integer, positive, negative or zero.

$[\pi_{1,n}(X)](\langle u,v\rangle_n, w) \Rightarrow [\pi_{1,n}(X)](\langle u,v\rangle_n) \Rightarrow u$

$[\pi_{2,n}(X)](\langle u,v\rangle_n, w) \Rightarrow [\pi_{1,n}(X)](\langle u,v\rangle_n) \Rightarrow v$

$[\pi_{1,n}(Y)](u, \langle v,w\rangle_n) \Rightarrow v$

$[\pi_{2,n}(Y)](u, \langle v,w\rangle_n) \Rightarrow w$

$[\pi_{1,n}(X)`A](\langle(\lambda x.[B](x)), v\rangle_n) \Rightarrow [B]([A](\langle\lambda x.[B](x), v\rangle_n)$

$[\pi_{2,n}(X)`A](\langle u, (\lambda x.[B](x))\rangle_n) \Rightarrow [B]([A](\langle\lambda x.[B](x), v\rangle_n)$

$[\pi_{1,n}(X)`A](\langle(\lambda x.[B](x)), v\rangle_n, w) \Rightarrow [B]([A](\langle\lambda x.[B](x), v\rangle_n, w)$

$[\pi_{2,n}(X)`A](\langle u, (\lambda x.[B](x))\rangle_n, w) \Rightarrow [B]([A](\langle\lambda x.[B](x), v\rangle_n, w)$

$[\pi_{1,n}(Y)`A](w, \langle(\lambda x.[B](x)), v\rangle_n) \Rightarrow [B]([A](w, \langle\lambda x.[B](x), v\rangle_n)$

$[\pi_{2,n}(Y)`A](w, \langle u, (\lambda x.[B](x))\rangle_n) \Rightarrow [B]([A](w, \langle\lambda x.[B](x), v\rangle_n)$

Notice that no rule is provided for reduction of a bracketed term with two arguments built from a variable binding expression. The solution to this problem is to convert an expression $[A](U,V,u)$ (U, V being first and second argument variables of unspecified order and n being the difference between the type of V and the type of U, or 0 if one or both is type-free) to which a binding operator is being applied to $[A^*](X,u)$, where A^* is obtained from A by replacing U with $\pi_{1,n}(X)$, V with $\pi_{2,n}(X)$, and u with Y or G as

appropriate. Note that $F(C)$ would be replaced with $\pi_{1,n}(X)`C$, for example. It is then possible to evaluate the substitution $[(Bu.[A](U,V,u)](s,t)$ as $[(Bw.[A^*](X,w))](\langle s,t\rangle_n)$, which can be read sensibly as $[[(Bw.[A^*](X,w))](\langle X,Y\rangle_n)](s,t)$: this gives us a constant notation for the desired function of two variables (and also implicitly says exactly what A^* is). The special rules for projection operators are installed to make this work correctly. Of course this could have been handled by allowing functions of any number of variables, but it is interesting to see that Frege's restriction to functions of two variables can be justified using essentially his own machinery (mod unanticipated details due to attention to type).

One can write $A(U,V,u) = [A](\pi_1(X),\pi_2(X))[V/u]$. The notation $A[V/u]$ for substitution of the second argument variable V for all occurrences of u needs to be defined.

We also want to provide some special rules for substitutions of terms $-u$ or $\neg u$ into particular contexts. $-u$ should be simplified to u when substituted into an argument place of an operator typed `prop`. Substitution of $\neg u$ into the context $\neg X$ or $\neg Y$ or $\neg P$ or $\neg Q$ should give $-u$, or just u if either u is a syntactical proposition or u is being substituted into an argument place of an operator with associated input type `prop`. I will eventually write out all the rules implicitly described here, but they will be numerous.

3 Axioms and rules of inference

We will use sequent calculus, though we will actually interpret sequents as sentences of our language.

$\emptyset$ is our notation for an empty list of propositions. p,Γ is our notation for the result of prepending p to the list Γ.

$\emptyset \vdash p$ is a notation for $-p$.

$p,\Gamma \vdash q$ is a notation for $\Gamma \vdash p \rightarrow q$.

It is neither the case that this is an entirely perverse way to read a sequent, nor that the kind of sentence which results is alien to Frege's approach.

We now state our rules of inference.

We may assert $\Gamma \vdash p$ if p appears as an item in Γ.

We may assert $\Gamma \vdash p$ iff we may assert $\Gamma' \vdash p$, if the set of sentences in Γ is the same as the set of sentences in Γ'.

We may assert $\Gamma \vdash p \rightarrow q$ iff we may assert $p.\Gamma \vdash q$ (of course this is obvious because they are notations for the same sentence).

We may assert $\Gamma \vdash \neg p$ iff we may assert $p, \Gamma \vdash \mathbf{f}$.

We may assert $p \to q, \Gamma \vdash r$ iff we may assert both $\neg r, \Gamma \vdash p$ and $q, \Gamma \vdash r$.

We may assert $p \to q, \Gamma \vdash \mathbf{f}$ iff we may assert both $\Gamma \vdash p$ and $q, \Gamma \vdash \mathbf{f}$.

We may assert $\neg p, \Gamma \vdash q$ iff we may assert $\neg q, \Gamma \vdash p$.

We may assert $\neg p, \Gamma \vdash \mathbf{f}$ iff we may assert $\Gamma \vdash p$.

We may assert $\Gamma \vdash (\forall x.[A](x))$ iff we may assert $\Gamma \vdash [A](y)$, where y is a free object variable not appearing in the first sequent.

For any object term t, we may assert $(\forall x.[A](x)), \Gamma \vdash r$ iff we can assert $[A](t), (\forall x.[A](x), \Gamma \vdash r$.

We may assert $\Gamma \vdash (\forall f.[A](\lambda x.f(x)))$ iff we may assert $\Gamma \vdash [A](\lambda x.g(x))$, where g is a free function variable not appearing in the first sequent.

For any Function term t, we may assert $(\forall f.[A](\lambda x.f(x))), \Gamma \vdash r$
iff we can assert $[A](t), (\forall x.[A](x), \Gamma \vdash r$.

We may assert $\Gamma \vdash t = t$ for any object term t.

We may assert $s = t, \Gamma \vdash [A](s)$ iff we may assert $s = t, \Gamma \vdash [A](t)$, for any terms s and t.

For any term q, we may assert $\Gamma \vdash p$ iff we may assert $q, \Gamma \vdash p$ and $\neg q, \Gamma \vdash p$ (cut rule).

We may assert the following axioms for any terms of the appropriate sorts If `axiom` is an axiom what we actually claim is that we may always assert $\Gamma \vdash$ `axiom`, independently of what Γ may be.

$\neg(-a = \neg b) \to -a = -b$ (this might be provable from logic, I am not certain).

$((\lambda x.f(x)) = (\lambda x.g(x))) = (\forall x.f(x) = g(x))$. This is the infamous Axiom V.

$a = \theta(\lambda x.x = a)$. This establishes that θ can serve as a definite description operator. We further provide the assertion that $(\forall x.(\forall a.\neg(x = (\lambda y.y = a))) \to \theta x = \mathbf{f})$. Frege's default behavior for θ is not included in his axioms and not appropriate for a stratified system.

Some basic manipulations of truth values and propositional operations are needed. $\neg\mathbf{t} = \mathbf{f}; \neg\mathbf{f} = \mathbf{t}; -t = \neg\neg t$. – should freely be added or removed when applied to syntactical propositions or to argument places of operator terms with type `prop`. Such things may be provable from the rules above; I am not certain. They may also be handled by defining some primitives in terms of others. These issues could also be dispelled by making `prop` a completely separate type. In this case, the terms typed `prop` would be the quantified terms and the operator terms with output type `prop` (and function application terms where the applied variable is a predicate variable).

An operator term could have only proposition terms in argument places with associated input type `prop`, and could have arguments of propositional type only in those places. There would be predicate variables. The set-builder construction would be primitive (binding proposition valued terms to produce objects). There would be a separate definition of membership in a set analogous to the given definition of application of a Function. Type casting operations from proposition to general object and back would be needed. Define – as $[X = \{x : x = x\}]$; this casts objects to their corresponding truth values. The reverse type coercion, for which we will provide the nonce notation `deprop`, would be effected by

$$[\theta(\lambda z.P \to z = \{x : x = x\} \wedge \neg P \to z = \{x : \neg x = x\})].$$

Behind this typed view would lie the assumption that the proposition $\mathbf{t}$ is to be identified with $\{x : x = x\} = \{x : \mathbf{t}\}$ and similarly the proposition $\mathbf{f}$ is to be identified with $\{x : \neg x = x\} = \{x : \mathbf{f}\}$, and further the sets $\{x : p(x)\}$ are to be identified with the corresponding Functions $(\lambda x.\texttt{deprop}(p(x)))$. This last identification could be posited as an axiom, as the identified entities are both objects.

We provide $(-x)(y) = -x$. Truth values are their own constant Functions. Another approach would be to identify $\mathbf{f}$ with $(\lambda x.\mathbf{f})$ and $\mathbf{t} = (\lambda x.x = \mathbf{t})$; this works better with the natural way of constructing models of this theory directly. There is not even any particular need to identify the truth-values with any courses of values.

This set of axioms and rules is not the same as Frege's, but it is quite straightforward to see that each set of logical axioms and rules justifies the other (mod the effects of typing restrictions on terms).

Note that while we do provide second-order quantification (quantification over functions) we do this only for functions with one argument (in this we follow Frege, who notes the formal possibility of quantification over functions of two arguments but never uses it), and in fact we provide it for a restricted subset of the functions, namely those functions $[A]$ such that $(\lambda x.[A](x))$ is well-formed (the restriction being that the first argument variable in A must be of the same type as A if both are typed).

4 The intended semantics

The theory presented is consistent. It can be interpreted in NFU + "there is a type-level ordered pair". In this theory, we can define the pair so that it is type level, so that function application $f(x)$ assigns x and y the same type and f one type higher, as here. We can then interpret $\mathbf{t}$ and $\mathbf{f}$ as two functions, which we can arrange to be their own constant functions by a Rieger-Bernays permutation. Application of objects $a`x$ and of functions to objects are both to be interpreted as the function application of NFU. The propositional operators have the obvious interpretations (with the extra remark that they coerce non-truth-value inputs to $\mathbf{f}$). A function $[A]$ and a Function $(\lambda x.[A](x))$ are interpreted as precisely the same object. Any function or Function we can express will actually exist because its definition will be weakly stratified; this is the point of our restriction of our language to typed formulas, even though we do not end up partitioning the objects into corresponding sorts.

In fact, the theory (before the proof of Infinity) can be interpreted in NFU without any assumption of Infinity. The type displacement of $f(x)$ when the Kuratowski pair is used is 3. Function application and lambda abstraction are the only operations used in the NFU interpretation: we interpret relative type $3n$ in terms of NFU as type n in terms of our version of the Frege system. So we can see that the system as it stands cannot prove Infinity (because NFU cannot).

To prove Infinity in the style of Frege it is necessary to adjoin Rosser's Axiom of Counting, which is equivalent to the unstratified fact that Frege uses to prove infinity: the size of $\{0, \ldots, n-1\}$ is n. Rosser's axiom is considerably stronger than Infinity, in fact (in the context of NFU). In a full treatment, we would add type-free number variables to our toolkit, which would make the proof of Counting from suitable axioms seem more natural. With Counting, the unstratified theorem which Frege uses to prove Infinity goes through.

This repair of Frege's theory preserves all his mathematics (unlike the approach of the "neo-Fregeans" who study weak subsystems satisfying Hume's Principle; the theory presented here is very strong). The most serious modification needed is the adjustment of the definition of double value ranges. What we might still feel that we need is a philosophical explanation of why this is a reasonable thing to do. Here is a brief outline of such a justification. A Function term (a course of values) is an object representing a function

(a first-order object representing a second-order object). But a Function applied to a Function representing a second-order object must in fact be representing a third-order object (a function taking second order functions to second order functions). This can be seen in a term like $(x`y)`z$. If z is representing a first-order object, $x`y$ and y represent second order objects and x represents a third order object. The punchline is that the type of any bound variable in a term codes the order of function that it represents. There is some typical ambiguity: if all types in a term are raised uniformly, it is still well-typed. Bracket abstractions or operators can be applied to objects at any type, as long as the relative differences between types are correct. Propositions (and numbers, if the Axiom of Counting is used) can have their types shifted freely.

Now the philosophical motivation is that the choice of which first-order objects represent which second-order objects is arbitrary and carries no real information about the objects actually being represented. So identifications between first-order objects (or second-order objects) appearing with different types so representing objects of different orders do not capture real properties we are interested in of the objects they are representing in those different roles. We are only interested in properties of and and relations between fixed objects *in fixed roles*. Only the well-typed (stratified) functions or predicates are needed for mathematics, and it is not so surprising that the use of "functions" or "predicates" which do not represent genuine information about the objects considered (such as $[\neg(X`X)]$) will lead to disastrous results; in any event, we have a formal reason to reject them which is prior to the discovery that they lead to paradox. It is not simply accidentally the case that stratification (our typing system) averts paradox; there is a metaphysically prior reason why we should only have been interested in stratified predicates and functions in the first place. The paradoxes are mistakes, not challenges to the foundations of reason.

Here is a longer version of the story. We suppose that we have a universe of objects which include $\mathbf{t}$, $\mathbf{f}$, and the von Neumann ordinals $\leq \omega$ (remember that these are distinct from the natural ordinals defined as equivalence classes of well-orderings). There is a domain of functions f, which are mapped to courses of values $(\lambda x.f(x))$, chosen arbitrarily, except that the truth values map to their own constant functions and each von Neumann ordinal α maps to the characteristic function of the set of von Neumann naturals below it. We consider that we can use expressions to define not only first order objects and second order functions, but also third order and higher functions. But

the device of courses of values enable us to collapse second order objects into the first order objects and so indirectly collapse each higher order: if order n has been coded into order 1 then an order $n+1$ function is clearly coded as an order 2 function, which can then be collapsed to its order 1 course of values.

Now we take seriously the fact that the correlation between second order objects and their courses of values is essentially arbitrary. Let π be a permutation of the universe of first order objects, fixing the truth values and von Neumann ordinals $\leq \omega$. We suppose that we reshuffle the first order objects used to represent second order functions, replacing $(\lambda x.f(x))$ with $\pi(\lambda x.f(x))$. This gives an alternative semantics for our notation: the type 2 order 1 notation $(\lambda x.f(x))$ could be read as denoting the former referent of $\pi(\lambda x.f(x))$ after the reshuffle. A type 2 order 2 variable still represents the same function, since it acts on unaffected type 1 order 1 terms. We generalize this: after the reshuffle, a type n order 1 term t should denote what was previously denoted $\pi_n(t)$. We know that π_0 is the identity and π_1 is π. We consider a type $n+1$ order 2 term F. If this sends t to u (type n order 1) before the reshuffle, it sends $\pi_n(t)$ to $\pi_n(u)$ afterward, so $F^*(\pi_n(t)) = \pi_n(F(t))$ whence $F^*(t) = \pi_n F(\pi_n^{-1}(t))$. Now a type $n+1$ order 1 term $(\lambda x.F(x))$ represents what was represented by $\pi(\lambda x.\pi_n(F(\pi_n^{-1}(x))))$ before the reshuffle. The full map π_{n+1} is a permutation of the entire universe of order 1 objects which sends $(\lambda x.f(x))$ to $\pi(\lambda x.\pi_n(f(\pi_n^{-1}(x))))$ for each second-order f. Now every term of our language has reshuffled semantics in which each term of type n order 1 or 2 is transformed in the ways indicated above. What is not invariant under these permutations is not a mathematical fact but a fact about the exact way that functions are coded as objects. Note that propositions and selected von Neumann ordinals are unperturbed by π. Now what happens is that ill-typed terms, in which bound variables appear with more than one type, have the identifications between variables of different types subverted by the reshuffling maneuver, so such terms do not represent genuine operations on the mathematical objects being represented. For example, $[X`X]$ transforms to $[\pi(X)`X]$, which is not of the same form. An additional technical point is that not just any permutation π will do: the rule is that the conjugate $F^*(t) = \pi_n F(\pi_n^{-1}(t))$ of any second order function F must be a second order function for each n (this corresponds to Forster's setlike permutations). This condition will hold for all π which actually are second order functions by reasonable closure conditions on second order functions, but in fact we want invariance under all functions satisfying the more gen-

eral condition: some things which cannot be functions are invariant under all second order function permutations. This invariance under permutations of the arbitrary features of the notation characterizes the formulas of our typed language [there is actually an exact mathematical theorem to this effect] and we allow abstraction only from such terms, which represent genuine properties and relations between objects considered under only a single order each.

The general idea here is that there is a reason why we can reject $[\neg X`X]$ and similarly "paradoxical" ill-typed things as abstractions *a priori*, before we discover that they are impossible in any case.

The transition from using von Neumann ordinals to Frege's numerals is straightforward: the cardinal of a variable von Neumann ordinal is a type-free expression which can be manipulated as a type-free Frege numeral variable.