Source for the Gottlob Proof System, with the Gottlob Paper as Comments

M. Randall Holmes

11/30/2018: The default display of Gottlob is now something like Frege's 2D notation, updated with a bug fix and fancy ASCII art. An explanation of Frege's notation is needed in the text. Bound variables are renamed when a line is constructed, preventing runaway indices. Further modifications occasioned by development in the book (see note in wish list). Revisions to theory material 9/6. The display in script files is encoded in code page 437, so it won't look right in many text editors. Notepad++ can handle it (Western European/OEM-US). I have a Python script which converts the Frege notations to Unicode. There is a way to comment out the string constant for the new ASCII art and decomment the old ones, to restore the less pretty ASCII art without encoding issues.

0.1 Preliminary remarks of various kinds

Add a comment opening at the beginning of the file to convert this to SML source.

We observe on reading the Grundgesetze that it is so precise as to be eminently implementable. So we propose to implement it: with the modifications required to make the system stratified, and so consistent. It is our thesis here that the program of the Grundgesetze is recoverable with inessential changes in Jensen's system NFU with Rosser's Axiom of Counting. We implement this higher order logic in a somewhat different way, since Frege's system is based on functions rather than sets.

wish list of possible additions:

:

1. I'd like to automate Frege's method of documenting individual steps: currently, I have to actually determine substitutions into a theorem justifying a modus pones or transitive implication step, which Frege leaves to the reader. It ought to be possible for the machine to do it.
2. 8/29 A couple of changes made as we proceed into value ranges. The backquote ‘ is now a legal special character (so that I can use it

for Frege's converse function application operator). The Raisetypes command will now raise types in closed bound variable constructions, though it still will not lower them. This actually turned out to be needed! The Raisetypes command was also bugged: fixed now.

3. 8/28 The text needs an account of the 2D notation. Explain the weird way implication is presented. Explain that Frege's tick marks are inverted (and that vertical bars can always be disambiguated: underscore to the right signals that it is a tick mark). Explain that 2D notations are closed with three spaces (our innovation), which is useful when a 2D notation is followed by a closing form. Finally, note that the for negation can still appear. We do not attempt to emulate the concavity notation for universal quanifiers (at least for now).
4. 8/24 notes as I break into the main development. I think I need the pretty-printer for sanity, and also I need sensible renumbering of bound variables. Preserving the letter shape of variables when they are bound by ugen might be nice, as well as lowering the indices.
5. The ability to pretty-print into something like Frege's concept-script is an obvious target. We claim at least that our notation is sufficiently closely analogous to Frege's to make translation possible. It may even be possible to parse something that looks like Frege's concept-script, as a stunt...
6. add line references (line numbers which are incremented as new lines are added) to fulfil the function of Frege's Greek letters. This would require some cunning about how lines are referenced.
7. I could add more letters to atomic term classes. This might include providing proposition and natural number free variables (and bound variables?)
8. bound variable and argument list as an internal component of binder terms might be useful.
9. I could record type input and output information for the bodies of variable binding operations and refine the type algorithm. I'm not sure anything in Frege needs this. A more restricted idea might be to allow typing of functions as concepts or relations (with propositional output).
10. I could allow user definition of types (as ranges of unary operators with output type `obj`) and/or updating of strongly cantorian types (by proving theorems of particular forms). Nothing in Frege will need this but the added mathematical competence would be a good. Marcel, which has the same type system in essence, does allow type declarations.
11. I could implement Marcel on top of Gottlob. The idea would be to generate Gottlob proofs in the background from sequent proofs in

the foreground. Gottlob definitions are much cleaner than Marcel definitions! Usual set notation as a variant of functions is an idea suggested by Marcel implementation, and having bearing on input and output types of binders being incorporated.

12. There is a subtle point about equality and definitions. We could have equality as a defined notion if we have the `Definition` command generate implications instead of equations. It is worth remarking on this, but probably the current behavior is better since it is what Frege actually does. The alternative behavior could be provided via a toggle or alternative commands. It is an interesting point that axiom III looks for all the world like a definition.
13. Does Frege apply his propositional rules to deeper levels of propositions than the top? I did find him doing it once, but managed to fake it with top level uses of the commands. I could add ability to apply the rules to subterms (to subcomponents as well as supercomponents), if it turns out that this is actually needed in practice. In general my propositional rules are not quite as free-form as his: for me, permutation is always bringing a subcomponent to the front and use of other rules is often specific in this sort of way, requiring extra permutation steps to do more complex things.
14. Introducing Hilbert symbol acting on bracketed function names might be fun; the binding rules are precisely such as to prevent the inadvertant assumption of Choice. But there is a problem with second order quantification in this connection: the Hilbert symbol would in this case be a new function-forming operator, something we do not otherwise have in the system.

1 Introduction

The purpose of this paper is to describe a consistent modification of Frege's logic, in the very strong sense that it includes the source code for a computer implementation of the intended system, which we intend to use to give a reasonably faithful implementation of the *Grundgesetze*. At the same time, this source file with a change of one character is the Moscow ML (version 2.01) source for the computer implementation.

The logical system presented is equivalent in strength to Jensen's variation NFU of Quine's set theory New Foundations (NF), with the addition of Rosser's Axiom of Counting. The idea of making Frege consistent by imposing the stratification restrictions of New Foundations on the formation of functions has been proposed by Nino Cocchiarella: we do not claim priority for this idea, but our implementation along these lines is improved by being closely faithful to Frege's notation and rules of inference.

It must be admitted that our notation looks superficially very different from Frege's (he has a very interesting two dimensional notation which looks forbid-

ding to the modern eye) but our notation is formally isomorphic in a suitable sense.

Sections 1-4 currently contain actual text explaining what is going on, positioned ahead of the code implementing the concepts. It's worth going past the very long parser source to read the text on the type system! The code sometimes contains further embeddied remarks, particularly in the type system. Sections 5 and 6 do not contain much text yet. Later 8/19 I have added more text about the proof environment and rules of inference. Section 7 on the NFU semantics is now drafted.

Contents

2 Frege's world introduced

The world of Frege's logic contains *objects* which include propositions, natural numbers, and value-ranges. There may be other objects as well, which we refer to as atoms.

The truth values are objects, known as the True and the False. The primitive operation − takes the True to the True and each other object to the False. The primitive operation of negation, written $\sim$ in the language we will implement, takes the True to the False and all other objects to the True. Note that double negation implements −, but there are formal reasons for us as for Frege to treat it as primitive.[1] The primitive operation of equality is defined as one would expect: $a = b$ is the True if a and b are the same object, and otherwise is the False. The primitive operation of implication $a \to b$ yields the False if a is the True and b is not the True, and otherwise yields the True.

Before we introduce further primitives producing objects, we have to discuss functions. Notations for the simplest sort of function are produced by Frege by writing a notation for an object which includes a special notation ξ standing for the object standing in its first (and only) argument place. The result of applying $f(\xi)$ to a is $f(a)$, as it were. Our notation for what Frege writes $f(\xi)$ is $[f(X)]$: a single bracket encloses a notation for a function of one argument. For example, `[~ X]` is notation for the negation function. X is our notation for an object first argument place, analogous to Frege's ξ.

Frege himself has a type system: he recognizes that functions are not of the same sort as objects (and he has further sorts of functions, as we will see). Before we proceed to further types of functions, we introduce the next fundamental primitive object construction. From a function F, we can construct an object, its *value-range*: our notation for the value range associated with `[f(X)]` is $\{x \mapsto F(x)\}$. This is our first example of a variable binding construction.

We have three further primitive object constructions. $\{Ax : \phi(x)\}$ is the format for the first-order universal quantifier. This is syntactic sugar for the application of the operator A to the function `[`ϕ`(X)]`, producing the True if all values $\phi(x)$ are the True, and the False otherwise.[2] We note in passing the identity criteria for value-ranges: $\{x \to f(x)\}$ is equal to $\{x \mapsto g(x)\}$ iff $\{Ax : f(x) = g(x)\}$ is the True. $\{If : \phi(f)\}$ is the format for the second-order universal quantifier. We use I to suggest "impredicative". This is syntactic sugar for the application of the second order universal quantifier to `[`ϕ`(F)]`, where the latter notation represents a function with a single function argument place F and object output, producing the True if $\phi(f)$ is the True for all functions f and the False otherwise. For a value range $\{x \mapsto f(x)\}$, `the`$\{x \mapsto f(x)\}$ denotes the unique object a such that $f(a)$ is the True, if there is such an object. We are indifferent to what `the` x may be when x is something other than

[1] It is worth noting that Frege himself remarks that his set of primitive notions and rules of inference is not minimal.

[2] This is somewhat overstated. Since one cannot bind into bracketed terms, it is not really possible to eliminate formalisms with bound variables in favor of formalisms using bracketed terms in this way, but it is useful to think of variable binding constructions in this way.

a value-range $\{x \mapsto x = a\}$.

We have already introduced another type of function, the type of functions taking a single function to an object, for which we introduced the first function argument place F. There are some further function types. Binary functions with object input are provided: Frege writes these $f(\xi, \zeta)$, where ζ is the second object argument-place. We use the notation `[[f(X,Y)]]` for this, Y being our notation for the second object argument place, and double brackets being used to signal a function of two arguments. We can then fill out all the function types we can represent by noting that we also provide a second argument place G representing a unary function from objects to objects, first and second argument places R and S representing binary functions taking two objects to an object, and Z, the first argument place representing a function from unary functions to objects. The use of Z is exhibited in the definition of the second-order existential quantifier as `[ ~{If:~Z(f)}]`. The presentation of orders of functions and possible numbers of arguments presented so far follows Frege exactly, except that he does not explicitly give notation for non-object argument places, and he does not need indications of scope for these notations (our brackets) because he does not actually ever embed such a notation in another notation. He remarks that such notations are not part of the concept script; we agree with him that they need never appear explicitly in theorems, but they do appear for us in definitions and as specific values assigned to function free variables during proofs, and this does correspond to features of his actual reasoning in proofs we have implemented. We do allow these notations to appear in locations where function variables are allowed; we allow these notations to contain free variables but not bound variables or argument places other than their own proper arguments.

We present the term type of the Gottlob prover. The `Prop` construction gives representations for `--` and $\rightarrow$: if `t` represents x then `Prop[t]` represents `--`x. If `t` represents x and `u` represents y, then `Prop [t,u]` represents $x \rightarrow y$. If `t` represents x and `Prop(L)`, L having length greater than 1, represents y then `Prop(t::L)` represents $x \rightarrow y$. The `Unary` constructor handles other unary operations on terms: for example, if `t` represents x, `Unary("~",t)` represents $\sim x$. The `Binary` constructor handles binary terms similarly: e.g, if `t` represents x and `u` represents y, `Binary(t,"=",u)` represents $x = y$. The construction `Function1` builds function names for functions of one argument (a single bracket is applied to the term represented by the argument). `Function2` similarly builds function names for functions of two arguments. `Constant` builds names for object constants, which are introduced by definition commands. `Freevar(t,n)` represents a free variable t_n (these can represent objects, functions of a single object, functions of two objects, and functions of a single unary function, the type being determined by the letter represented by `t`. The `App1` form represents application of unary functions, and the `App2` form represents application of binary functions.

The first numerical argument appearing in the `Boundvar`, `Binder`, `Firstarg` and `Secondarg` functions represents an additional feature of the type system of Gottlob not found in Frege: these integers represent the relative types used in the stratification criterion of New Foundations, which we call here *levels*.

The `Binder` construction handles variable binding constructions. If `T` represents the term T, then `Binder("","v",j,i,T)` represents $\{v_i^j \mapsto T\}$ and `Binder("B","v",j,i,T)` represents $(Bv_i^j : T)$. B here represents a specific binder (A for example would be the universal quantifier). v will indicate by its shape the sort of bound variable (object or function). j indicates level and i is a further index of the variable. `Boundvar(v,j,i)` represents a bound variable v_i^j, the sort (object or function) being determined by the shape of `v`. `Firstarg(S,n)` represents a first argument place S^n, S determining sort and n determining level and similarly `Secondarg(S,n)` represents a second argument place S^n. `Error` is provided to handle ill-formed terms.

```
*)

(*  BEGIN uncomment for PolyML

open PolyML;

fun desome(SOME x)=x |
desome NONE = "";

 END uncomment for PolyML*)

datatype Term =

   Prop of Term list |

   Unary of string * Term |

   Binary of Term * string * Term |

   Binder of string * string * int * int* Term |

   Function1 of Term |

   Function2 of Term |

   Constant of string |

   App1 of Term*Term |

   App2 of Term*Term*Term |

   Freevar of string * int |

   Boundvar of string * int * int |
```

```
   Firstarg of string * int |

   Secondarg of string * int |

   Error;

(*
```

3 The syntax of Gottlob

We discuss the exact form of Gottlob's notation in the context of presenting the display function and the parser.

A first consideration in developing the parser and display functions is that we chose to try to make them entirely independent of user declarations of constants, unary and binary operations, and binders, and we succeeded in doing this.

We choose to make all binary operations have the same precedence and group to the right and all unary operators to have the same precedence, stronger than that of binary operators. This is different from Frege's approach, which is left-grouping by preference, but shares the characteristic of lacking operator precedence. We depart radically from Frege by presenting implication as a binary operation in the usual sense, though our internal representation of implications is closer to his in spirit.

In the display function, this appears in the use of the functions `iscomplex1` and `iscomplex2` to identify arguments of unary operations and left arguments of binary operations, respectively, which must be enclosed in parentheses. a unary or binary or proposition term must be enclosed in parentheses when a unary operation is applied to it; binary and proposition terms left of binary operators must be enclosed in parentheses, but a unary term need only be so enclosed if its argument is an identifier serving as a user defined constant (to avoid the parser mistaking the second identifier for an infix).

Further details of the language are best discussed in connection with the code for the parser.

The lowest level (1) is suppressed in the display of bound variables, as is the lowest index 1 on free or bound variables. There are variations P, M of X and Q, N of Y which represent argument places to be occupied by propositions and natural numbers, respectively. On these argument places, level indications are suppressed. On other argument places, level is expressed, separated from the letter by a caret. The reason for this is that F and G are overloaded, used both as function bound variables and as function argument places: `F2` is a bound variable and `F^2` is an argument place, both being of level 2.

The new variant `Display1` (new as of 8/26) supports pretty-printing in something like the style of Frege's original notation. This is very useful for comparison of Gottlob results with Frege's text.

3.1 Display function ML source

```
*)

fun iscomplex1 (Unary(s,t)) = true |
iscomplex1 (Binary(t,s,u)) = true |
iscomplex1 (Prop L) = true |
iscomplex1 t = false;
```

```
fun
 iscomplex2 (Prop L) = true |
iscomplex2 (Unary(s,Constant t)) = true |
iscomplex2 (Binary(t,s,u)) = true |
iscomplex2 t = false;

fun Display (Prop [t]) = "--"^
(if iscomplex1 t then "(" else "")
^(Display t)^(if iscomplex1 t then ")" else "") |

Display (Prop [a,b]) = (if iscomplex2 a then "(" else "")^
(Display a)^(if iscomplex2 a then ")" else "")^" -> "^(Display b) |

Display (Prop (x::L)) = (if iscomplex2 x then "(" else "")
^(Display x)^(if iscomplex2 x then ")"
else "")^" -> "^(Display(Prop L)) |

Display (Prop nil) = "{Impossible Proposition}" |

Display (Unary(s,t)) = if iscomplex1 t then s^"("^(Display t)^")"
   else s^" "^(Display t) |

Display (Binary(t,s,u)) = if iscomplex2 t then
   "("^(Display t)^") "^s^" "^(Display u)
   else (Display t)^" "^s^" "^(Display u) |

Display (Binder(b,x,m,n,t)) =

"{"^b^(if b=""then "" else" ")^
(Display(Boundvar(x,m,n)))^
(if b = "" then " => " else " : ")^(Display t)^"}" |

Display (Function1 t) = "["^(Display t)^"]" |

Display (Function2 t) = "[["^(Display t)^"]]" |

Display (Constant s) = s |

Display (App1 (t,u)) = (Display t)^"("^(Display u)^")" |

Display (App2 (t,u,v))= (Display t)^"("^(Display u)^","^(Display v)^")" |

Display (Freevar (s,n)) = if n=1 andalso s<>"Z" then s else  s^(makestring n) |

Display (Boundvar(s,m,n)) = Display(Freevar(s,m))
^(if n=1 then "" else "_"^(makestring n)) |
```

```
Display (Firstarg(s,n)) =  if s = "M" orelse s = "P"
 then s else s^"^"^(makestring n) |

Display (Secondarg(s,n)) = if s = "N"
 orelse s = "Q" then s else s^"^"^(makestring n) |

Display (Error) = "?!?!";

fun OldDisplay t = Display t;

(* work on 2D style prettyprinter *)

(* Display1 = prettyprint starting at position n; Display2 will be the
genuine prettyprinter *)

fun tlength0 nil = 0 |

tlength0 (x::L) = if x= #"\n" then tlength0 L

else let val LL = tlength0 L in if LL < length L then LL else 1+LL end;

fun tlength s = tlength0 (explode s);

fun spaces n = if n<= 0 then "" else " "^(spaces (n-1));

fun indent0 x nil = nil |

indent0 x (y::L) = if y= #"\n" then [y]@x@(indent0 x L)

else (y::(indent0 x L));

fun indent x s = implode(indent0 (explode x) (explode s));

(* strings for use in improved ASCII art *)

(* BEGIN pretty version, encoding issues *)

val prettybar = " \179";  (* " |" *)

val threedashes = "\196\194\196"; (* "___" *)

val prettyneg = "\196\194\196";   (* "_|_" *)

val hook = "\n \192\196"; (* "\n |_" *)
```

```
val hookneg = "\n \192\196"^prettyneg; (* "\n |__|_" *)

(* END pretty version encoding issues *)

(* BEGIN original version

val prettybar = " |";

val threedashes = "___";

val prettyneg = "_|_";

val hook = "\n |_";

val hookneg = "\n |__|_";

 END original version *)

fun Display1 n (Prop [t]) = "--"^
(if iscomplex1 t then "(" else " ")
^(Display1 (n+3) t)^(if iscomplex1 t then ")" else "") |

(* Prop clauses to be rewritten totally *)

Display1 n (Prop [Unary("~",x),Unary("~",y)]) = threedashes^(indent (spaces n)((indent pr


Display1 n (Prop ((Unary("~",x))::L)) = threedashes^(indent (spaces n)((indent prettybar

Display1 n (Prop [x,Unary("~",y)]) =  threedashes^(indent (spaces n)((indent prettybar


Display1 n (Prop (x::L)) = threedashes^(indent (spaces n)((indent prettybar (Display1 1


Display1 n  (Prop nil) = "{Impossible Proposition}" |

Display1 n (Unary("~",Prop(x::y::L))) = (* "_|_^(indent (spaces n)((indent "   |" (Displa

Display1 n  (Unary(s,t)) = if iscomplex1 t then s^"("^(Display1 (length(explode s) + n +
   else s^" "^(Display1 (length(explode s)+n +1)  t) |

Display1 n  (Binary(t,s,u)) = if iscomplex2 t then
let val T = Display1 (n+1) t in
   "("^(T)^") "^s^" "^(Display1 (tlength ((spaces n)^"("^(T)^") "^s^" "))  u) end
```

```
   else let val T = Display1 n t in

(T)^" "^s^" "^(Display1 (tlength ((spaces n)^(T)^" "^s^" ")) u) end  |

Display1 N  (Binder(b,x,m,n,t)) =

"{"^b^(if b=""then "" else" ")^
(Display1 0 (Boundvar(x,m,n)))^
(if b = "" then " => " else " : ")^(Display1 (length(explode b)+length(explode(Display1 (

Display1 n  (Function1 t) = "["^(Display1 (n+1) t)^"]" |

Display1 n (Function2 t) = "[["^(Display1 (n+2) t)^"]]" |

Display1 n  (Constant s) = s |

Display1 n (App1 (t,u)) =
let val T = Display1 n t in

(T)^"("^(Display1 (tlength((spaces n)^T^"(")) u)^")" end |

Display1 n  (App2 (t,u,v))=

let val T = Display1 n t in

let val U = Display1 (tlength((spaces n)^T^"(")) u in

(T)^"("^U^","^(Display1 (tlength((spaces n)^(T)^"("^U^",")) v)^")"  end end |

Display1 N (Freevar (s,n)) = if n=1 andalso s<>"Z" then s else  s^(makestring n) |

Display1 N  (Boundvar(s,m,n)) = Display1 0 (Freevar(s,m))
^(if n=1 then "" else "_"^(makestring n)) |

Display1 N  (Firstarg(s,n)) =  if s = "M" orelse s = "P"
 then s else s^"^"^(makestring n) |

Display1 N  (Secondarg(s,n)) = if s = "N"
 orelse s = "Q" then s else s^"^"^(makestring n) |

Display1 N  (Error) = "?!?!";

fun Display t = Display1 0 t;

(*
```

3.2 Discussion of the syntax

The parser code follows. Here we will describe the language of Gottlob in some detail.

Identifiers for Gottlob are either strings of special characters (see `isspecial` for the list), strings of letters of which the first may be capitalized, or strings of digits (numerals) . Tokens are either identifiers or punctuation (comma, colon, parentheses, braces, brackets , underscores and carets; `=>`, which is an identifier for the tokenizer, is treated as punctuation by the parser). To parse a string into Gottlob notation, first resolve it into tokens. Whitespace between tokens is ignored, except that a token always ends at whitespace.

Certain identifiers are reserved.

1. `=>` is treated as punctuation: it is used in the notation for value-ranges.
2. `a,b,c` are reserved for object free variables.
3. `f,g,h` are reserved for unary function free variables.
4. `x,y,z` are reserved for object bound variables.
5. `X,Y,M,N,P,Q` are reserved for object argument places.
6. `F,G,H` are reserved for function bound variables and argument places.
7. `r,s,t` are reserved for binary function free variables.
8. `R,S` are reserved for binary function argument places.
9. `Z` is reserved for free variables or argument places representing functions from unary functions to objects. These have narrow uses in the definitions of second-order variable binding operators and the axiom for the second order universal quantifier.

The form of a free variable is the appropriate letter followed by a numeral read as an index (not a level). The index 1 may be omitted, except in `Z1`.

The form of a bound variable in fullest generality is the appropriate letter followed by a numeral (its level) followed by an underscore followed by a numeral (an index). The level may be omitted if it is 1. The index may be omitted, along with its underscore, if it is 1.[3]

The form of an argument place is the appropriate letter followed by a caret followed by the level. The caret and level are omitted if the letter is one of `M,N,P,Q`. We identify the argument places.

Note that when an operator is said to have input of a particular sutype of objects (propositions or natural numbers) it will actually take any object as input, but this object is then coerced into the stated input type by applying a retraction (`--` in the case of propositions, `#` in the case of natural numbers).

[3]A refinement which probably should be implemented is the option of omitting the level on a function bound variable if it is the minimum value 2. The parser does read a function bound variable without explicit level as having level 2, but this level is then displayed.

first proposition argument: P, second proposition argument: Q.

first natural number argument: M; second natural number argument: N.

first object argument: X; second object argument: Y.

first unary function argument: F; second unary function argument: G.

first binary function argument: R; second binary function argument: S.

first unary function taking a unary function argument to an object: Z.

Some identifiers are declared as primitive unary or binary operators or binders.

`--` and $\sim$ are unary operations taking propositions to propositions. `--` is represented differently internally.

`#` is a unary operation taking natural numbers to natural numbers.

`the` is a unary operation taking general objects to general objects.

`->` acts like a binary operation taking two propositions to a proposition, but is represented differently internally.

`=` is a binary operation taking two general objects to a proposition.

`A` is the first order universal quantifier (binding an object variable).

`I` is the second order universal quantifier (binding a function variable).

Other identifiers are available for user definitions as object constants, unary and binary operators with object input and output, and as first order or second order binders. Where an operation is specified to have input less general than all objects, it actually can take arbitrary object input, but the object input is supposed coerced to the target sort (for example, all non-truth values are coerced to the False by operations with propositional input). A facility for additional user defined sorts may be added.

The variables, argument places, and declared identifiers constitute the atomic terms.

A general term consists of a series of non-infix terms (there might be just one) connected by identifiers serving as binary operators. General terms are always grouped to the right, so the form is non-infix term + identifier + general term.

The non-infix terms are of the following forms:

1. an atomic term.

2. a general term in parentheses.

3. a unary operator followed by a non-infix term. Where a term begins an identifier, followed by an identfier the sequel of which resolves as a term, the second identifier is interpreted as an infix. This expedient allows us to parse terms without consulting declarations: the effect is that a unary operation applied to an identifier representing a constant must be enclosed in parentheses if it appears to the left of a binary operator.

4. a value-range term $\{x{=>}T\}$, where x is a bound variable and T is a general term.

5. a general binding term $\{Bx : T\}$ where B is an identifier, x is a bound variable, and T is a general term.

6. an application term $T(U)$, where T is a non-infix term and U is a general term: this represents application of a function with one argument.

7. an application term $T(U, V)$, where T is a non-infix term and U, V are general terms.

8. a single bracket term $[T]$ where T is a general non-bracketed term.

9. a double bracketed term $[[T]]$ where T is a general non-bracketed term.

3.3 The parser ML source

```
*)

fun islower c = #"a" <= c andalso c <= #"z";

fun isupper c = #"A" <= c andalso c <= #"Z";

fun isnumeral c = (#"0" <= c andalso c <= #"9") orelse c= #"'";

fun isspecial  c = c= #"~" orelse c= #"`"

orelse c = #"@" orelse c = #"#" orelse c = #"$"
orelse c = #"%" orelse c = #"&"
 orelse c = #"*" orelse c = #"-" orelse c = #"+"
 orelse c = #"=" orelse c = #"|" orelse c = #";"
orelse c = #"." orelse c = #"<"
orelse c = #">" orelse c = #"?" orelse c = #"/"
 orelse c = #"!" orelse c = #".";


(* get first identifier from a list of characters *)

fun getident nil = nil |
```

```
getident [c] = if islower c orelse isupper c orelse isnumeral c
orelse isspecial c orelse c = #"," orelse c= #":"
orelse c = #"(" orelse c = #")" orelse c= #"[" orelse c= #"]"
 orelse c= #"{" orelse c= #"}"then [c] else nil |

(* I could fiddle with allowed shapes of identifiers here *)

getident (a::(b::L)) =

if a = #"," orelse a = #":" orelse a = #"(" orelse a = #")"
orelse a = #"[" orelse a = #"]"
orelse a= #"{" orelse a= #"}" orelse a= #"_" orelse a= #"^"  then [a] else

if a = #" " orelse a= #"\n" orelse a= #"\\" then getident (b::L)
     else if isupper a

         then if islower b (* orelse isnumeral b *)

         then a::(getident(b::L))

         else [a]

     else if islower a

         then if islower b (* orelse isnumeral b *)

         then a::(getident(b::L))

         else [a]

     else if isnumeral a

         then if isnumeral b

         then a::(getident(b::L))

         else [a]

     else if isspecial a

         then if isspecial b

         then a::(getident(b::L))

         else [a]
```

```
     else nil;

(* the rest of the stream of characters after the first identifier is read *)

fun restident nil = nil |

restident [c] = nil |

restident (a::(b::L)) =

if a = #"," orelse a = #":" orelse a = #")" orelse a = #"("
orelse a= #"[" orelse a= #"]" orelse a= #"{" orelse a= #"}"
 orelse a= #"_" orelse a= #"^" then b::L else

if a = #" " orelse a= #"\n" orelse a= #"\\" then restident (b::L)
     else if isupper a

         then if islower b (* orelse isnumeral b *)

         then restident(b::L)

         else (b::L)

     else if islower a

         then if islower b (* orelse isnumeral b *)

         then restident (b::L)

         else (b::L)

     else if isnumeral a

         then if isnumeral b

         then restident(b::L)

         else b::L

     else if isspecial a

         then if isspecial b

         then restident(b::L)
```

```
         else b::L

     else nil;

(* utility for tokenization *)

fun testidentlist nil = nil |

testidentlist L = (implode(getident(L))::(testidentlist (restident L)));

(* get a list of tokens (identifiers and punctuation) from a string *)

fun tokenize s = testidentlist(explode s);

(* parser utilities *)

fun Item 0 L = nil |

Item n nil = nil |

Item 1 (x::L) = [x] |

Item n (x::L) = Item (n-1) L;

fun Initial 0 L = nil |

Initial n nil = nil |

Initial n (x::L) = x::(Initial (abs n -1) L);

fun Final 0 L = L |

Final n nil = nil |

Final n (x::L) = Final (n-1) L;

fun Hd nil = "" |
Hd (x::L) = x;

fun Tl nil = nil |
Tl (x::L) = L;

(* token classification *)

(* punctuation *)
```

```
fun ispunct a = a = "," orelse a = ":"
orelse a = ")" orelse a = "("
orelse a= "[" orelse a= "]"
orelse a= "{" orelse a= "}"
 orelse a= "_" orelse a= "^" orelse a="=>";

(* numerals, with evaluation function *)

fun isnum s = s <> "" andalso isnumeral(hd (explode s))
 andalso (length(explode s)=1 orelse isnum(implode(tl(explode s))));

(* fresh index for variables *)

val NEWNUM = ref 1;

fun Numvalue s = if isnum s then
if length(explode s)=1 then ord(hd(explode s))-48 else
(ord(hd(rev(explode s)))-48)+
10*(Numvalue(implode(rev(tl(rev(explode s))))))
else ~1;

fun max(x,y) = if x>y then x else y;

fun numvalue s = let val N = Numvalue s in
(NEWNUM:= max(!NEWNUM,N+1);N) end;

(* variables and arguments *)

fun isvarorarg s = s="a" orelse s = "b" orelse s="c"
 orelse s="x" orelse s="y" orelse s="z" orelse s="f"
 orelse s = "g" orelse s="h" orelse s ="F" orelse s="G"
 orelse s="H" orelse s="r" orelse s="s" orelse s="t"
 orelse s="X" orelse s="Y" orelse s="Z" orelse s="W"
orelse s="M" orelse s="N" orelse s = "P" orelse s="Q";

(* bound variables *)

fun isboundvar s = s="x" orelse s="y" orelse s="z"
orelse s="F" orelse s = "G" orelse s="H";

fun isobjectvar s = s="x" orelse s="y" orelse s="z"
orelse s="a" orelse s = "b" orelse s="c";

(* identifiers *)

fun isidentifier s = s<> "" andalso (not (ispunct s)) andalso (not (isvarorarg s));
```

```
(* outline getterm1 *)

fun Parse1 L =

let val I1 = Hd(Item 1 L) and I2 = Hd(Item 2 L)
and I3 =Hd( Item 3 L) and I4 = Hd(Item 4 L)
and I5 = Hd(Item 5 L) and I6 = Hd(Item 6 L) and
 I7 = Hd(Item 7 L)  in

(* if term begins with { and its first item is an identifier,
 second item is x/y/z/F/G/H,
third item is a numeral, fourth item is a numeral and fifth item is
:, followed by a term followed by }, then getterm1 is the
 obvious binder term and restterm1 is the tail of the restterm of the body *)

if I1 = "{" andalso isidentifier I2 andalso isboundvar I3
 andalso isnum I4 andalso I5 = "_" andalso isnum I6 andalso I7 = ":"

then let val (A,B) = Parse(Final 7 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder(I2,I3,numvalue I4,numvalue I6,A),tl B)

end

else if I1 = "{" andalso isidentifier I2
 andalso isboundvar I3 andalso isnum I4 andalso I5 = ":"

then let val (A,B)  = Parse(Final 5 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder(I2,I3,numvalue I4,1,A),tl B)

end

else if I1 = "{" andalso isidentifier I2
andalso isboundvar I3 andalso I4 = "_"
andalso isnum I5 andalso I6 = ":"
```

```
then let val (A,B)  = Parse(Final 6 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder(I2,I3,if isobjectvar I3 then 1 else 2,numvalue I5,A),tl B)

end

else if I1 = "{" andalso isidentifier I2 andalso isboundvar I3 andalso I4 = ":"

then let val (A,B)  = Parse(Final 4 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder(I2,I3,if isobjectvar I3 then 1 else 2,1,A),tl B)

end

(* if term begins with { and its first item is a string,
second item is a numeral, third item is a numeral and fourth item is
=>, followed by a term followed by }, then getterm1 is the
 obvious binder term and restterm1 is the tail of the restterm of the body *)

else if I1 = "{" andalso  isboundvar I2
 andalso isnum I3 andalso I4 = "_" andalso isnum I5 andalso I6 = "=>"

then let val (A,B) = Parse(Final 6 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder("",I2,numvalue I3,numvalue I5,A),tl B)

end

else if I1 = "{" andalso isboundvar I2 andalso isnum I3 andalso I4 = "=>"

then let val (A,B)  = Parse(Final 4 L) in

if A = Error then (Error, nil)
```

```
else if Hd B <> "}" then (Error, nil)

else (Binder("",I2,numvalue I3,1,A),tl B)

end

else if I1 = "{" andalso isboundvar I2 andalso I3 = "_"
andalso isnum I4 andalso I5 = "=>"

then let val (A,B)  = Parse(Final 5 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder("",I2,if isobjectvar I2 then 1 else 2,numvalue I4,A),tl B)

end

else if I1 = "{" andalso  isboundvar I2 andalso I3 = "=>"

then let val (A,B)  = Parse(Final 3 L) in

if A = Error then (Error, nil)

else if Hd B <> "}" then (Error, nil)

else (Binder("",I2,if isobjectvar I2 then 1 else 2,1,A),tl B)

end

(* if the term begins with ( followed by a term and ), we return
 that term and return the tail of the restterm of the term as our restterm1 *)

else if I1 = "(" then

let val (A,B) = Parse(Final 1 L) in

if A = Error then (Error,nil)

else if Hd B <> ")" then (Error, nil)

else (A,tl B)

end
```

```
(* if the term begins with a bracket followed by a term
 and a close bracket, followed by a parenthesis followed
 by a term followed by a close parenthesis, we get an App1 term *)

(* if the term begins with a bracket followed by a term
 and a close bracket, and the previous case does not hold,
 we get a Function1 term *)


else if I1 = "[" andalso I2 <> "["

then let val (A,B) = Parse(Final 1 L) in

if A = Error then (Error,nil)

else if Hd B <> "]" then (Error, nil)

else if Hd(Tl B) = "("

then let val (C,D) = Parse(Final 2 B) in

if C = Error then (Error, nil)

else if Hd D <> ")" then (Error, nil)

else (App1(Function1 A,C),Tl D)

end

else (Function1 A,Tl B)

end


(* if the term begins with a double bracket followed by a term
 and double close brackets, followed by a parenthesis followed by
 a term followed by a comma followed by a term followed by
 a close parenthesis, we get an App2 term *)

(* if the term begins with a double bracket followed by a term
 and double close brackets, and the previous case doesnt hold,
 we get a Function2 term *)

else if I1 = "[" andalso I2 = "["

then let val (A,B) = Parse(Final 2 L) in
```

```
if A = Error then (Error,nil)

else if Hd B <> "]" orelse   Hd(Tl(B)) <> "]" then (Error, nil)

else if Hd(Final 2 B) = "("

then let val (C,D) = Parse(Final 3 B) in

if C = Error then (Error, nil)

else if Hd D <> "," then (Error, nil)

else let val (E,F) = Parse(Tl D) in

if E = Error then (Error,nil)

else if Hd F <> ")" then (Error,nil)

else (App2(Function2 A,C,E),Tl F)

end

end

else (Function2 A,Final 2 B)

end

(* if the term begins with a/b/c followed by _ followed by a number,
 we get a suffixed free object variable *)

else if (I1 = "a" orelse I1 = "b" orelse I1 = "c") andalso isnum I2

then (Freevar(I1,numvalue I2),Final 2 L)

(* if the term begins with a/b/c not followed by _
 we get a1/b1/c1, a free object variable *)

else if I1 = "a" orelse I1 = "b" orelse I1 = "c"

then (Freevar(I1,1),Tl L)

(* if the term begins with x/y/z followed by a number followed by _
 followed by a number, a bound object variable with type and index *)
```

```
(* if the term begins with x/y/z followed by _ followed by a number,
 a bound object variable of type 1 with index *)

(* if the term begins with x/y/z followed by a number
 (and no earlier case holds), a bound object variable with type,
 with index 1 *)

(* if the term begins with x/y/z and no earlier case holds,
 a bound object variable of type 1 and index 1 *)

else if (I1 = "x" orelse I1 = "y" orelse I1 = "z")
andalso isnum I2 andalso I3 = "_" andalso isnum I4

then (Boundvar(I1,numvalue I2,numvalue I4),Final 4 L)

else if (I1 = "x" orelse I1 = "y" orelse I1 = "z") andalso isnum I2

then (Boundvar(I1,numvalue I2,1),Final 2 L)

else if (I1 = "x" orelse I1 = "y" orelse I1 = "z") andalso I2 = "_" andalso isnum I3

then (Boundvar(I1,1,numvalue I3),Final 3 L)

else if (I1 = "x" orelse I1 = "y" orelse I1 = "z")

then (Boundvar(I1,1,1),Tl L)

(* if the term begins with X/Y followed by a caret followed by a number,
 an object argument term with type*)

(* if the term otherwise begins with X/Y,
 an object argument term of type 1 *)

else if I1 = "X" orelse I1 = "M" orelse I1 = "P"

then if I2 = "^" andalso isnum I3

then (Firstarg(I1, numvalue I3),Final 3 L)

else (Firstarg(I1,1),Tl L)

else if I1 = "Y" orelse I1  = "N" orelse I1 = "Q"

then if I2 = "^" andalso isnum I3

then (Secondarg(I1, numvalue I3),Final 3 L)
```

```
else (Secondarg(I1,1),Tl L)


(* if the term begins with F/G/Z/W followed by a caret followed
 by a number followed by ( followed by a term followed by ),
 an App1 term *)

(* W not supported unless we implement quantification over
binary relations *)

(* if the term begins with F/G/Z/W followed by a caret
 followed by a number otherwise, a function argument term *)

else if (I1 = "F" orelse I1 = "G" orelse I1 = "Z")
 andalso I2 = "^" andalso isnum I3

then

if I4 = "("

then let val (A,B) = Parse(Final 4 L) in

if A = Error then (Error,nil)

else if Hd B <> ")"

then (Error,nil)

else if I1 = "F" orelse I1 = "Z"
then (App1(Firstarg(I1,numvalue I3),A),Tl B)
else (App1(Secondarg(I1,numvalue I3),A),Tl B)

end (* 1 *)

else  if I1 = "F" orelse I1 = "Z"
then (Firstarg(I1,numvalue I3),Final 3 L)
else (Secondarg(I1,numvalue I3),Final 3 L)

(* if the term begins with f/g/h/F/G/H/Z (possibly followed by
 a number) (possibly followed by an underscore
 followed by a number in F/G/H case) followed by ( followed by a term followed by ),
 an App1 term;  a skipped index is read as 1 and a skipped type as 2 *)

(* if the term begins with f/g/h/F/G/H/Z (possibly followed by a number
 in the F/G/H case) (possibly followed by an underscore followed by a number)
```

```
 otherwise, a free or bound unary function variable;
 a skipped index is read as 1 and a skipped type as 2 *)

else if ( I1 = "F" orelse I1 = "G" orelse I1 = "H")
andalso isnum I2 andalso I3 = "_" andalso isnum I4

then

if I5 = "("

then  let val (A,B) = Parse(Final 5 L) in

if A = Error then (Error,nil)

else if Hd B <> ")" then (Error,nil)

else  (App1(Boundvar(I1,numvalue I2,numvalue I4),A),Tl B)


end (* 3 *)

else (Boundvar(I1,numvalue I2,numvalue I4),Final 4 L)

else if (I1 = "f" orelse I1 = "g" orelse I1 = "h"
orelse I1 = "F" orelse I1 = "G" orelse I1 = "H" orelse I1 = "Z") andalso isnum I2

then

if I3 = "("

then  let val (A,B) = Parse(Final 3 L) in

if A = Error then (Error,nil)

else if Hd B <> ")" then (Error,nil)

else if isboundvar I1 then (App1(Boundvar(I1,numvalue I2,1),A),Tl B)

else  (App1(Freevar(I1,numvalue I2),A),Tl B)


end (* 4 *)

else if isboundvar I1 then (Boundvar(I1,numvalue I2,1),Final 2 L)

else (Freevar(I1,numvalue I2),Final 2 L)
```

```
else if (I1 = "F" orelse I1 = "G" orelse I1 = "H")
andalso I2 = "_" andalso isnum I3

then

if I4 = "("

then  let val (A,B) = Parse(Final 4 L) in

if A = Error then (Error,nil)

else if Hd B <> ")" then (Error,nil)

else (App1(Boundvar(I1,2,numvalue I3),A),Tl B)

end (* 5 *)

else (Boundvar(I1,2,numvalue I3),Final 3 L)

else if (I1 = "f" orelse I1 = "g" orelse I1 = "h"
 orelse I1 = "F" orelse I1 = "G" orelse I1 = "H")

then

if I2 = "("

then  let val (A,B) = Parse(Final 2 L) in

if A = Error then (Error,nil)

else if Hd B <> ")" then (Error,nil)

else if isboundvar I1 then (App1(Boundvar(I1,2,1),A),Tl B)

else  (App1(Freevar(I1,1),A),Tl B)

end (* 6 *)

else if isboundvar I1 then (Boundvar(I1,2,1),Final 1 L)

else (Freevar(I1,1),Final 1 L)

(* if the term begins with r/s/t (possibly followed by an underscore
 followed by a number) followed by ( followed by a term followed by
```

```
 a comma followed by a term followed by ), an App2 term;
 a skipped index is read as 1 and a skipped type as 2 *)

(* if the term begins with r/s/t (possibly followed by an underscore
 followed by a number) otherwise, a free binary function variable
[I am omitting binary function bound  variables and argument forms
 for the moment, and their associated App2 terms] *)

else if (I1 = "r" orelse I1 = "s" orelse I1 = "t") andalso I2 = "_" andalso isnum I3

then if I4 = "(" then

let val (A,B) = Parse (Final 4 L)

in

if A = Error then (Error,nil)

else if Hd B <> "," then (Error,nil)

else let val (C,D) = Parse (Tl B) in

if C = Error then (Error,nil)

else if Hd D <> ")" then (Error,nil)

else (App2(Freevar(I1,numvalue I3),A,C),Tl D)

end (* 7 *)

end (* 8 *)

else (Freevar(I1,numvalue I3),Final 3 L)

else if (I1 = "r" orelse I1 = "s" orelse I1 = "t")

then if I2 = "(" then

let val (A,B) = Parse (Final 2 L)

in

if A = Error then (Error,nil)

else if Hd B <> "," then (Error,nil)
```

```
else let val (C,D) = Parse (Tl B) in

if C = Error then (Error,nil)

else if Hd D <> ")" then (Error,nil)

else (App2(Freevar(I1,1),A,C),Tl D)

end

end

else (Freevar(I1,1),Final 1 L)

else if (I1 = "R" orelse I1 = "S") andalso I2 = "^" andalso isnum I3

then if I4 = "(" then

let val (A,B) = Parse (Final 4 L)

in

if A = Error then (Error,nil)

else if Hd B <> "," then (Error,nil)

else let val (C,D) = Parse (Tl B) in

if C = Error then (Error,nil)

else if Hd D <> ")" then (Error,nil)

else if I1 = "R" then (App2(Firstarg(I1,numvalue I3),A,C),Tl D)
else (App2(Secondarg(I1,numvalue I3),A,C),Tl D)

end (* 7 *)

end (* 8 *)

else if I1="R" then (Firstarg(I1,numvalue I3),Final 3 L)
else (Secondarg(I1,numvalue I3),Final 3 L)

else if (I1 = "r" orelse I1 = "s" orelse I1 = "t")

then if I2 = "(" then
```

```
let val (A,B) = Parse (Final 2 L)

in

if A = Error then (Error,nil)

else if Hd B <> "," then (Error,nil)

else let val (C,D) = Parse (Tl B) in

if C = Error then (Error,nil)

else if Hd D <> ")" then (Error,nil)

else (App2(Freevar(I1,1),A,C),Tl D)

end

end

else (Freevar(I1,1),Final 1 L)


(* if the term begins with an identifier, followed by an identifier
followed by a term, read just the identifier as a constant *)

(* if the term begins with an identifier, followed by a term1,
 read it as a unary operation term *)

(* if the term begins with an identifier otherwise, read it as a constant *)

else if isidentifier I1

   then if isidentifier I2

      then let val (A,B) = Parse(Final 2 L)
        in

        if A = Error
           then let val (A1,B1) = Parse1(Final 1 L)
           in
           if A1 = Error
               then (Constant I1, Tl L)
               else
                   if I1 = "--"
                       then (Prop [A1],B1)
```

```
                        else (Unary (I1,A1),B1)
          end else (Constant I1, Tl L)  (*1 *)

          end

   else  let val (A,B) = Parse1(Final 1 L) in
if A = Error then (Constant I1,Final 1 L)
else (if I1 = "--" then Prop [A] else Unary (I1,A),B) end

(* otherwise error *)

else (Error,nil)

end (* D *)

and deprop (Prop L) = L |

deprop t = [t]

and isprop (Prop L) = true |

isprop t = false

and Parse L =

let val (A,B) = Parse1 L in

if A = Error then (Error,nil)

else if not (isidentifier (Hd B))  then (A,B)

else let val (C,D) = Parse (Tl B) in

if C = Error then (Error,nil)

else  if Hd B = "->" then (Prop (A::(if isprop C
andalso length(deprop C)=1 then [C] else (deprop C))),D) else

 (Binary(A,Hd B,C),D)

end (* A *)

end (* B *)

fun parse s =
```

```
let val (A,B) = Parse (tokenize s) in

if B <> nil then Error else A end

fun parsetest s =

let val (A,B) = Parse(tokenize s) in
Display A end

(*
```

4 Augmenting the type system of Frege with levels in the style of Quine

We next present the refinement of the type system which allows us to avoid the contradictions found in the original system. The refinement is motivated by the stratification criterion for instances of the axiom of comprehension in Quine's set theory New Foundations.

We have already assigned types to each term of our language. We provide a succinct notation for these basic types. Let 0 denote the type of objects. For each type τ for which we have first argument place terms, we have the type (τ) of functions taking a single argument of type τ to an object. For each pair of types σ, τ for which we have a first argument place of type σ and second argument places of type τ, we have a type (σ, τ) inhabited by functions of two arguments, the first of type σ and the second of type τ, with object output.

We now describe the refinement. Types can further be qualified with indications of *level*, either a positive integer or an indeterminate value which we write ∞; object types may be qualified as `prop` (proposition) or `nat` (natural number) which are special cases of ∞. Note that adding the same integer to each integer component of a refined type yields a name for the same type: this polymorphism reflects the fact that all the levels are actually the same type, as in New Foundations. An extension to allow user definition of more specific types of object is feasible. These refinements can be applied to each input type of a function and to its output type, so notations are extended to have a third argument representing the output. The basic type of negation is (0); the refined type is $(\texttt{prop}, \texttt{prop})$. The basic type of equality is $(0, 0)$; the refined type is $(1, 1, \texttt{prop})$ (or any $(n, n, \texttt{prop})$). Further, each function type has a further feature, a level of its own which may be expressed as a superscript, either a natural number or ∞: the superscripted level must match the level of each component (input or output) of the type.

We explain notation and details of matching of arguments of functions with their formal types. The notation for type 0 (objects) qualified with level n is simply n (∞ for the indeterminate type, and `prop`, `nat` for the specific types). Object types match if they are not distinct integers (so ∞ and the specific types match any type). A unary function of level $n+1$ is of type (n, n) or a type matching this compnentwise. Functions of type (m, n) with $m \neq n$ can be defined (for example, `the` is of type $(2, 1)$) but such functions cannot have functions applied to them and are not assigned levels: the superscript on a type $(2, 1)$ (for example) must be ∞. Binary functions of level $n+2$ are of type (n, n, n) or a type matching this componentwise, and with level superscript either $n+2$ or ∞[4]. Again, binary functions of types with distinct integer components do exist but cannot be assigned levels (the level superscript would be ∞). Finally, unary functions of level $n+2$ sending unary functions to objects have types $((n, n), n+1)$ or a type matching this componentwise, again with

[4]The displacement of 2 may be surprising here, but is strongly suggested by the way Frege chooses to correlate binary functions with objects.

level superscript n or ∞. Again, functons of this basic type can only appear as arguments if they have this refined type. All of this reflects stratification criteria for functions in the usual mathematical sense in New Foundations.

The result of application of an entity of one type to an entity or pair of entities of another type or types is now described. Take the type being applied and displace its integer components uniformly by a constant amount so that the input type or types of the applied entity match the type or types of the types of the entities to which it is being applied [but this displacement is only permitted where the superscript attached to the type of the applied entity is ∞]. If this is not possible, the application is not permitted (the term is rejected as ill-formed, not by the parser but in a second pass by the type algorithm).

We now discuss the typing of actual terms. Free object variables are typed ∞. Object bound variables and argument places are typed as their level. All free unary function variables have type $(1,1)^{\infty}$; a bound unary function variable has type $(n,n)^{n+1}$ where $n+1$ is its level. The effect of this on a bound function variable is that its type cannot be displaced before matching when it is applied: the type of `F^n` is frozen at $(n-1,n-1)$. A free function variable can be viewed as of type $(1,1)$ with the usual privileges of displacement. The types of other free function variables are their usual types with free displacement (so the attached level, as it were, is ∞). The types of the other function argument places are treated as having an affixed level, and so are frozen: the type of `R^n` is $(n-2,n-2,n-2)^n$ and the type of `Z^n` is $((n-2,n-2),n-1)^n$. [5]

Typing of application terms follows the algorithm described above, working from the types of the applied entity and the entities to which it is applied, with the note that if the applied entity has frozen level, the displacement of the applied type to get a match cannot be used. It should further be noted that when a frozen function (say of frozen level $n+1$ and type (n,n), for example) is applied to an object term or terms of type ∞ (or of type `prop`, `nat`), the result is of type n, not of type ∞.

Typing of unary and binary operation terms is handled as application, in effect: each unary operator is typed as a unary function (whose arguments may be at different levels, and whose level superscript is ∞, allowing displacement) and each binary operator is typed as a binary function with the same qualities.

1. $-$ and $\sim$ have type $(\texttt{prop},\texttt{prop})$.
2. $\rightarrow$ has type $(\texttt{prop},\texttt{prop},\texttt{prop})$.
3. $=$ has type $(1,1,\texttt{prop})$.
4. `the` has type $(2,1)$.

[5] The typing of R^n , which will very seldom be an issue, is intended to support the correlation (identification?) of binary functions with their double value ranges; if there were bound binary function variables, this type differential might be adopted as well. A type differential of 1 between binary functions and their arguments is also perfectly supportable, but 2 agrees with Frege's preferred method of correlation of such functions with objects.

Superscripts of ∞ are not expressed above.

Typing of binder terms is handled similarly, with the proviso that $\{Bx : T\}$ is read as $B([T'])$, where T' is the result of replacing x with X in T [this has to be viewed with caution, because this expansion cannot be carried out if T includes any bound variables other than x, because as we shall see bracketed terms cannot contain bound variables, but the intention here is not to actually carry out such expansions but to use them to motivate typing.] The value-range notation is handled as a special case with B being named by the empty string. We state the types of the primitive binders.

1. the value range operation has type $((1, 1), 2)$.
2. the first order universal quantifier A has type $((1, \texttt{prop}), \texttt{prop})$, though it is currently internally represented in Gottlob as simply $(1, 1), \texttt{prop})$.
3. the second order universal quantifier I has type $(((1, 1), \texttt{prop}), \texttt{prop})$, though it is currently internally represented as having type $(((1, 1), 2), \texttt{prop})$.

Again, each of these types has unexpressed superscript ∞.

The final subtleties occur in the typing of bracketed terms and in the bodies of variable binding terms. In either context, there is a notion of whether a bound variable or argument place actually contributes to the type of the term: an occurrence of an atomic term x in a term T does not contribute to its type if it is bound in the term in the usual sense or if it occurs as the only variable contributing type to a subterm of T whose type is ∞ or which appears as an argument of a unary or binary relation in a position assigned input type ∞ (or its specializations such as `prop`, `nat`). A bound variable term $\{Bx : T\}$ is assigned type ∞ if no variable other than x contributes to the type of T (and in particular if $\{Bx : T\}$ is a closed term). Computation of input types is done by looking at all instances of the atomic term as an argument of an operation: if all such instances are in argument places with the same type more specific than ∞, the variable is assigned that type as an input. An input variable which makes no contribution to type and cannot be assigned a more specific type is assigned type ∞. Bracketed terms are ill-typed if they contain any bound variables. They must contain occurrences of exactly one argument place of each appropriate kind (and it must be noted, following Frege, that a term $\{x \mapsto T\}$ or $\{Bx : t\}$ must have x occurring in T, contrary to modern practice). The input type associated wth an argument place will be ∞ if it does not contribute to the type of the body of the bracketed term, or a more specific subtype of ∞ if it occurs only in argument places of operations assigned that type.

A simplification of the type system of Gottlob already alluded to is that function input types assigned to bracketed terms are coerced to the form (n, n), as seen in the types of the universal quantifiers. A refinement of the type system of Gottlob to allow specification of ∞ or its specializations as input and/or output of function input types is possible but would make things a bit more complex.

It should be noted here that defined unary and binary operations and binders of both first and second order variables can neatly be defined simply by equating their names to a bracketed term (containing no free variables) from which their types can be computed (with the indicated coercion of input types in the case of binders).

It is further important to note that, just as in NF, the levels actually do not reflect any sorting of objects. All constant object terms are assigned type ∞ if well-formed, and all constant unary function terms (capable of occurring as values of function variables) are of type $(1,1)^\infty$. The type distinction between functions and objects, and between various different sorts of functions, remains real as in Frege, though as in Frege this is obviated by the precise embedding of all unary functions of type $(1,1)^\infty$ into the objects by the value-range operation. But some unary and binary functions considered by Frege (`the` is an example among the primitives) are seen not to be identified with any function in the range of the second order quantifiers when the stratification conditions are imposed.

In relation to NF(U), it is worth noting that Gottlob does not allow the expression of unstratified assertions which can be expressed in the language of NF(U). All operations definable in Gottlob are stratified, though they can be inhomogeneous (bracketed terms can be assigned types which have more than one integer level among their arguments and outputs, which are stratified but inhomogeneous and in terms of NF represent stratified operations not implementable by functions which are sets). The unstratified content of the Axiom of Counting is expressed by the provision that the operator `#` (which is defined by an additional axiom in the implementation, not presented here, as sending each natural number to itself and each other object to 0) produces terms which can be uniformly shifted in type: this causes many unstratified assertions about natural numbers to become in effect stratified.

4.1 The ML source for the datatype of types of our logic system, with additional discussion

```
*)

(* more complex type scheme *)

(* object types *)

datatype Kind1 =

prop1 |  (* propositions *)

nat1 |  (* natural numbers *)

obj1 |  (* constants or unspecified s.c. type *)
```

```
level1 of int;  (* type object variable or argument expressions *)

(* function types.  level3 are functions from unary
functions to objects, needed for second order quantificaiton. *)

datatype Kind2 =

fun2 |  (* function constant or free variable *)

rel2 |  (* binary function constant or free variable *)

level2 of int |  (* typed unary functiion variable or argument terms *)

level3 of int |  (* typed unary function from unary functions to objects:
    only first arguments. Second order binders have this argument type
for their input.  *)

fun3 |  (* type of constants of the just previous sort *)

blevel2 of int ;  (* only assigned to relation argument terms,
unless we introduce binary function binders, in which case we
would also need blevel3 as their input type. *)

datatype Argtype =

arg1 of Kind1 |

arg2 of Kind2;

(* these would be the types assigned to general terms:  atype types assigned
to non bracketed terms, utype and btype types assigned to bracketed terms
and operators.  Some atypes are assigned to functions, but only to stratified functions.

(* binders in effect have things with utypes or btypes as arguments, but
only homogeneous things.  The right word is homogeneous, not stratified, because
we actually cannot express unstratified abstractions at all. *)

datatype Kind =

atype of Argtype |  (* types of arguments *)

utype of (Argtype*Kind1) |  (* type of unary operators *)

btype of (Argtype*Argtype*Kind1) |  (* types of binary operators *)

terror;  (* type error *)
```

```
(*
```

Here is the type system of Gottlob.

Type `Kind1` comprises the object types, these being propositions, natural numbers, "belongs to an unspecified strongly cantorian set determinable from context (this includes "is a constant or free variable")" (`obj1`, type ∞ in the discussion above) and relative types indexed by integers for stratification purposes: level n is the type of a bound object variable coding an nth order function (of level type, a function from order $n-1$ functions to order $n-1$ functions). Note to self: use "level" for the types of Gottlob to avoid confusion with ML types in the paper. The terms to watch are "order" and "level".

Type `Kind2` comprises the function and relation types assignable to arguments. These include the types `fun2`, `rel2`, and `fun3` of function constants (unary functions from objects to objects, binary functions, and unary functions from unary functions to objects, and the relative types `level2 n`, `blevel2 n` and `level3 n` which correspond to these. The binary function indexed types are assigned only to binary function argument terms, as we do not bind binary functions. I'm contemplating additional types to be inhabited by functioins of unspecified type with strongly cantorian range. In terms of the discussion above, `fun2` is $(1,1)^\infty$, `rel2` is $(1,1,1)^\infty$, and `fun3` is $((1,1),2)^\infty$. `level2 n` is $(n-1,n-1)^n$. `blevel n` is $(n-2,n-2,n-2)^n$. `level3 n` is $((n-2,n-2),n-1)^n$.

Type `Kind` contains types of arguments and the further types to be assigned to unary and binary function bracketed terms, operators, and binders. Notice that the arguments of these latter types are of type `Argtype`, a disjoint union of `Kind1` and `Kind2`, while the output type is of type `Kind1`. This has the merit of excluding unused options. It does mean we need to do some type casting in the code to get from one type of type to another.

Frege does draw distinctions of type: his account of orders of functions and the discussion on p. 48 describe a type system, in effect. Here we refine his system of orders into a finer system of levels corresponding to the relative types used in New Foundations. Special additional types represent "strongly cantorian" sorts on which relative types can be freely raised and lowered. That propositions are such a type is provable in NFU. That natural numbers are such a type is a strong axiom consistent with NFU, known as Rosser's Axiom of Counting. We must assume this in order to justify the assertion that the set $\{1,\ldots,n\}$ has n elements, which he uses to prove Infinity and which is not a theorem of NFU.

4.2 ML source code for functions implementing the type system, with a fair amount of text comment

```
*)

val TYPES = ref [("~",utype(arg1 prop1,prop1)),
("=",btype(arg1 (level1 1),arg1(level1 1),obj1)),
```

```
("",utype(arg2(level2 2),level1 2)),
("A",utype(arg2(level2 2),prop1)),
("the",utype(arg1(level1 2),level1 1)),
("I",utype(arg2(level3 3),prop1)),
("#",utype(arg1(nat1),nat1))];

val DEFS = ref[("a",Freevar("a",1))];

val _ = DEFS := nil;

fun find s default nil = default |
find s default ((t,x)::L) = if s=t then x else find s default L;

fun findtype s = (* find type associated with an identifier *)

find s terror (!TYPES);

fun finddef s = (* find body of definition of s *)

find s Error (!DEFS);

fun foundin x nil = false |

foundin x (y::L) = x=y orelse foundin x L;

fun foundin2 x nil = false |

foundin2 x ((y,z)::L) = x=y orelse foundin2 x L;

fun purge s nil = nil |

purge s (t::L) = if s=t then purge s L else t::(purge s L);

fun purge2 s nil = nil |

purge2 s ((t,u)::L) = if s=t then purge2 s L else (t,u)::(purge2 s L);

(* list has a single item (of object type if an argument)  in it; a bound function
variable can be present *)

fun purgeargs nil = nil |

purgeargs (Firstarg(s,n)::L) = purgeargs L |

purgeargs (Secondarg(s,n)::L) = purgeargs L |
```

```
purgeargs (t::L) = t::(purgeargs L);

(* list has no more than one item in it *)

fun oneitem L = L=nil orelse (purge(hd L)L) = nil;

(*
```

Here we have the declaration list of types (and probably some other declaration lists presently) and some basic functions for manipulating lists.

In addition the initial value of the list TYPES contains all the declarations of primitive notions. There is a new primitive notion `#` made necessary by Frege's implicit assumption of the axiom of counting (in his proof of infinity).

As of 8/15 the output type of equality was revised to `obj`, so that we can actually implement the proof of IIIi in the book. It is probably better eventually to reinstall the obvious `prop` output type, in which case IIIi might become quite hard to prove, but is nothing one would want to say (`--(a=b)` automatically simplifies), while preserving the proof that works with `obj` typing. Another alternative is to have a toggle which suppresses automatic simplifications during substitution, to make it easier to exhibit reasoning not using these substitutions, or using them in restricted ways. As of 8/16. substitution was refined to allow lazier substitution in such a way that the proof of IIIi works with equality assigned output type `prop` (though this seems silly, since the system *knows* that $a = b$ is a proposition). My observation also has bearing, that with a slight tweak equality would be a defined notion whose output was *obviously* `prop`, and in fact this is roughly how the proof works.

It is still instructive to convert the output type of equality to `obj1` and run the proof book. The simplification algorithm otherwise suppresses various dashes and makes some hypotheses in the argument appear trivial. 8/23 output type of equality set back to `obj1` for faithfulness to Frege's development.

```
*)

(* types of bound variables *)

fun vartype x m n  = (* compute type of a double indexed variable *)

if x = "x" orelse x = "y" orelse x = "z" then atype (arg1 (level1 m))

else if x = "F" orelse x = "G" orelse x = "H" then  atype (arg2  (level2 m))

else terror;

(* types of argument terms, both from internal data
```

```
and from such a term itself *)

fun argumenttype s n = (* compute type of argument term s^n *)

if s = "X" orelse s= "Y" then atype (arg1  (level1 n))

else if s = "M" orelse s="N" then atype (arg1  (nat1))

else if s="P" orelse s="Q" then atype (arg1  prop1)

else if s="F" orelse s="G" then atype (arg2 (level2 n))

else if s ="R" orelse s = "S" then atype (arg2(blevel2  n))

else if s = "Z" then atype (arg2 (level3 n))

else terror;

fun argumenttype0 (Firstarg(s,n)) = argumenttype s n |
argumenttype0 (Secondarg(s,n)) = argumenttype s n |
argumenttype0 x = terror;

(* types of free variables *)

fun freevartype(s,n) = (* compute type of free variable sn *)

if s = "a" orelse s = "b" orelse s="c" then atype (arg1 obj1)

else if s="f" orelse s="g" orelse s="h" then atype (arg2 fun2)

else if s="r" orelse s="s" orelse s="t" then atype (arg2 rel2)

else if s = "Z" then atype(arg2 fun3)

else terror;

(*
```

Here are functions for computing types of variables and argument terms.

```
*)

(* cast types to argument or object types *)
```

```
fun  acast (atype x) = x |

acast x = (arg1  (level1 0));

fun otype (atype (arg1 x)) = true |

otype x = false;

fun  ocast (atype (arg1 x)) = x |

ocast x = (level1 0);

(* the data type of type intervals *)

datatype typediff =

freetdiff |  (* unknown type interval *)

badtdiff |  (* impossible type interval *)

tdiff of int;  (* concrete integer type interval *)

(* unification of type intervals *)

fun tdiffmerge badtdiff x = badtdiff |

tdiffmerge x badtdiff = badtdiff |

tdiffmerge freetdiff x = x |

tdiffmerge x freetdiff = x |

tdiffmerge (tdiff m) (tdiff n)  = if m=n then tdiff m else badtdiff;

(* move argument types by a constant *)

(* moving types by the unknown interval has the
effect of ``floating" them *)

fun tadd (tdiff n) ((arg1  (level1 m))) = (arg1 (level1 (m+n))) |

tadd (tdiff n) ((arg2 (level2 m))) = (arg2(level2(m+n))) |

tadd (tdiff n) ((arg2 (level3 m))) = (arg2(level3(m+n))) |

tadd freetdiff (arg1 (level1 n)) = arg1 obj1 |
```

```
tadd freetdiff (arg2(level2 n)) = arg2 fun2 |

tadd freetdiff (arg2(blevel2 n)) = arg2 rel2 |

tadd freetdiff (arg2(level3 n)) = arg2 fun3 |

tadd n x = x;

(* float a type, discarding stratification
information *)

fun typefloat (atype a) = atype(tadd freetdiff a) |

typefloat x = x;

(* determine difference between argument types
of the same kind  *)

fun typecompare (arg1 (level1 m)) (arg1(level1 n)) = tdiff (n-m)  |

typecompare (arg1 x) (arg1 y) = freetdiff |

typecompare (arg2 (level2 m)) (arg2(level2 n)) = tdiff (n-m) |

typecompare (arg2 fun2) (arg2 (level2 m)) = freetdiff |

typecompare (arg2 (level2 n)) (arg2 fun2) = freetdiff |

typecompare (arg2 (blevel2 m)) (arg2 (blevel2 n)) = tdiff (n-m) |

typecompare (arg2 rel2) (arg2 (blevel2 n)) = freetdiff |

typecompare (arg2(blevel2 n)) (arg2 rel2) = freetdiff |

typecompare (arg2 (level3 m)) (arg2 (level3 n)) = tdiff(n-m) |

typecompare (arg2 fun3) (arg2 (level3 m)) = freetdiff |

typecompare (arg2 (level3 m)) (arg2 fun3) = freetdiff |

typecompare x y = if x=y then freetdiff else badtdiff;

fun typevalue (arg1 (level1 m)) = m |

typevalue (arg2 (level2 m)) = m |
```

```
typevalue (arg2(level3 m)) = m |

typevalue x = ~1;

(* difference of level without regard to
being of the same kind *)

fun freetypecompare x y =

let val X = typevalue x and Y = typevalue y in

if X= ~1 then freetdiff
else if Y = ~1 then freetdiff

else tdiff (Y-X)

end;

(* determine whether argument types of the same kind can be
the same type *)

fun typematches t u = typecompare t u = freetdiff orelse typecompare t u = tdiff 0;

(* here we just determine whether two types can be
at the same stratification level *)

fun freetypematches t u = freetypecompare t u = freetdiff
orelse freetypecompare t u = tdiff 0;

(* this acts on general types *)

fun Typematches (atype t) (atype u) = typematches t u |

Typematches t u = t=u;

fun isfuntype (arg2 (level2 n)) = true |
isfuntype (arg2 fun2) = true |
isfuntype x = false;

(* cast a function type to an argument type, works
only when the type is homogeneous *)

fun funtypecast (utype (x,y)) =

if typematches x (arg1 y) then atype (arg2 fun2)
```

```
else if isfuntype x andalso freetypematches x (arg1 y) then

atype(arg2 fun3)

 else utype(x,y) |

funtypecast (btype(x,y,z)) = if typematches x y
andalso typematches x (arg1 z) andalso typematches y (arg1 z)

then atype (arg2 rel2) else btype(x,y,z) |

funtypecast x = x;

(*
```

Here is a block of functions for handling types.

We provide a type of intervals between types, having either an integer value, a value indicating an unknown displacement, and an error value. These objects can be unified: unification of distinct integer values produces the error value; unification of the unknown value with an integer value gives the second value; unification of a value with itself of course gives itself.

We then provide functions for determining the interval between two types for purposes of stratification, one which takes into account their being of the same kind (function or object) and one which does not, and functions for raising a type by a type interval. Notice that raising a type by the unknown interval gives the corresponding "constant" type which does not need stratification information.

The last function allow us to cast operation types to argument types when typing permts this (only homogeneous operations may appear as arguments).

It is important to note that all operations that we can represent in our language and assign type to are stratified: operations may be represented by bracketed terms which are inhomogeneous (their definition bodies are stratified bu their input and output types include distinct integers). These can only appear in applied position, not as arguments.

```
*)

fun applyutype (utype(x,y)) (atype z) =

let val D = typecompare x z in

if D = badtdiff then terror

else atype (tadd D (arg1 y))
```

```
end |

applyutype (atype (arg2 fun2)) (atype (arg1 z)) =
applyutype (utype(arg1 (level1 1),level1 1)) (atype (arg1 z)) |

applyutype(atype(arg2 (level2 n))) (atype(arg1(level1 m))) =
if n=m+1 then atype( arg1(level1 m)) else terror |

applyutype (atype(arg2(level2 n))) (atype(arg1 x)) =
atype(arg1(level1 (n-1))) |

applyutype (atype(arg2(level3 n))) (atype(arg2(level2 m))) =
if n=m+1 then (atype(arg1(level1 m))) else terror |

applyutype (atype(arg2(level3 n))) (atype(arg2(fun2)))=
atype(arg1(level1 (n-1)))  |

applyutype (atype(arg2(fun3))) (atype(arg2(level2 m))) =
atype(arg1 (level1 m)) |

applyutype (atype(arg2(fun3))) (atype(arg2(fun2)))=
atype(arg1 obj1)  |


applyutype x y =

if funtypecast y <> y then applyutype x (funtypecast y)

else terror;

(* find type of output
of unary operator of type x with input type y *)
(* also works on function variables *)

fun applybtype (btype(x,y,z)) (atype u) (atype v) =

let val D = tdiffmerge (typecompare x u) (typecompare y v)
and E = tdiffmerge (freetypecompare x y) (freetypecompare u v) in

if D = badtdiff orelse E = badtdiff then terror

else atype(tadd D (arg1 z))

end  |
```

```
applybtype (atype(arg2 rel2)) (atype u) (atype v) =

applybtype (btype(arg1(level1 1),arg1(level1 1),level1 1)) (atype u) (atype v) |

applybtype (atype(arg2(blevel2 n))) (atype u) (atype v) =

if typematches u (arg1(level1 (n-2))) andalso
typematches v (arg1(level1 (n-2))) andalso typematches u v

then atype(arg1(level1(n-2))) else terror |

applybtype x y z =

if funtypecast y <>y orelse funtypecast z <> z

then applybtype x (funtypecast y) (funtypecast z)

else terror;

 (* match formal input type of binder type x to function
 taking actual type y to actual type z and compute the
actual output type, or report type error*)

(*
```

These functions compute the output type of a unary or binary operator given the types of its inputs. The general idea is that one matches the types of the actual inputs to the formal input types of the operations and determines a type interval which if added to the formal types gives types which match the actual types, then add the type interval to the formal output type to determine the actual output type.

If bracketed terms appear as arguments then the type is cast to an argument type if possible.

```
*)

fun deatype (atype x) = x |

deatype x = arg1(level1 0);

fun bindermatch (utype(x,y)) (atype u) (atype (arg1 v)) =

if typecompare x (acast(funtypecast(utype(u,v)))) = badtdiff then terror else
let val D = freetypecompare (tadd (tdiff ~1) x) u in
```

```
if D = badtdiff then terror

else atype(tadd D (arg1 y))

end |

bindermatch x y z = terror;

(*
```

This function is used to determine the actual type of a bound variable term given the formal type of the binder, the actual type of the bound variable, and the actual type of the body of the abstraction.

Check that the abstract function argument is of the same sort as one taking the actual bound variable to the actual body.

Determine the displacement between the formal type expected of the bound variable (one level below the type of the function argument) and the actual type of the bound variable, and apply this displacement to the formal type of the output to obtain the actual type of the output.

```
*)

fun floatoutputtype0 (utype(x,y)) = typevalue (arg1 y) = ~1 |

floatoutputtype0 (btype(x,y,z)) = typevalue (arg1 z) = ~1 |

floatoutputtype0 x = false;

fun floatoutputtype s = floatoutputtype0(findtype s);

fun floatlefttype0  (btype(x,y,z)) = typevalue (x) = ~1 |

floatlefttype0 x=false;

fun floatlefttype s = floatlefttype0 (findtype s);

fun floatrighttype0  (btype(x,y,z)) = typevalue (y) = ~1 |

floatrighttype0 x=false;

fun floatrighttype s = floatrighttype0 (findtype s);

fun propinputtype0 (utype(x,y)) = x=arg1 prop1 |
```

```
propinputtype0 x = false;

fun propinputtype s = propinputtype0(findtype s);

fun proplefttype0  (btype(x,y,z)) = x=arg1 prop1 |

proplefttype0 x=false;

fun proplefttype s = proplefttype0 (findtype s);

fun proprighttype0  (btype(x,y,z)) = y = arg1 prop1 |

proprighttype0 x=false;

fun proprighttype s = proprighttype0 (findtype s);


fun typevarlist (Prop nil) = nil |

typevarlist (Prop(x::L)) = let val L1 = typevarlist x and L2 = typevarlist (Prop L) in

if oneitem L1 andalso oneitem L2 then nil

else if oneitem L1 then L2

else if oneitem L2 then L1

else L1@L2

end |

typevarlist (Unary(s, t)) =

let val T = typevarlist t in

if oneitem T  andalso floatoutputtype s then nil else

T  end |

typevarlist (Binary(t,s,u)) =

let val T = typevarlist t and U = typevarlist u in

if oneitem T andalso oneitem U andalso floatlefttype s
andalso floatrighttype s then nil
```

```
else if oneitem (T@U) andalso floatoutputtype s then nil

else if oneitem T andalso floatlefttype s then U

else if oneitem U andalso floatrighttype s then T

else T@U

end |

typevarlist (Binder(b,x,m,n,t)) = let val T =purge (Boundvar(x,m,n)) (typevarlist t)

in if oneitem T andalso floatoutputtype b then nil else T end |

typevarlist (Function1 t) = purgeargs (typevarlist t) |

typevarlist (Function2 t) = purgeargs(typevarlist t) |

typevarlist (Constant t) = nil |

typevarlist (App1(t,u)) = (typevarlist t) @ (typevarlist u) |

typevarlist (App2(t,u,v)) = (typevarlist t)@(typevarlist u)@(typevarlist v) |

typevarlist (Freevar (s,n)) = nil |

typevarlist (Boundvar(s,m,n)) = [Boundvar(s,m,n)] |

typevarlist (Firstarg(s,n)) = [Firstarg(s,n)] |

typevarlist (Secondarg(s,n)) = [Secondarg(s,n)] |

typevarlist Error = nil;

(*
```

This function computes the list of bound variables and argument terms which actually contribute information to the type of a term. To begin with we have the list of bound variables occurring free in the term: items can be eliminated from the list if they appear as the only element of the type variable list of a subterm which is in a position which allows the subterm to float in type.

```
*)
```

```
(* design decision is made here (sensibly) that bracketed terms may not
contain bound variables *)

(* only object argument terms can be eliminated by
the type variable list generator, so it is safe to use
object types as default when argument terms are not
present *)

fun isfirstarg (Firstarg (s,n)) = true |
isfirstarg x = false;

fun issecondarg (Secondarg (s,n)) = true|
issecondarg x = false;

(* list of all argument terms free in a term, without type considerations *)

(* this is needed to catch extra argument terms which are eliminated by
the type variable algorithm *)

fun argumentlist (Prop nil) = nil |

argumentlist (Prop(x::L)) = (argumentlist x)@(argumentlist (Prop L))|

argumentlist (Unary(s, t)) =

argumentlist t |

argumentlist (Binary(t,s,u)) =

(argumentlist t)@(argumentlist u) |

argumentlist (Binder(b,x,m,n,t)) = argumentlist t |

argumentlist (Function1 t) = nil |

argumentlist (Function2 t) = nil  |

argumentlist (Constant t) = nil |

argumentlist (App1(t,u)) = (argumentlist t)@(argumentlist u) |

argumentlist (App2(t,u,v)) = (argumentlist t)@(argumentlist u)@(argumentlist v) |

argumentlist (Freevar (s,n)) = nil |
```

```
argumentlist (Boundvar(s,m,n)) = nil |

argumentlist  (Firstarg(s,n)) = [Firstarg(s,n)] |

argumentlist  (Secondarg(s,n)) = [Secondarg(s,n)] |

argumentlist  Error = nil;

fun isutype (utype(x,y)) = true |
isutype x = false;

fun isbtype (btype(x,y,z)) = true |
isbtype x = false;

(* determine type of a variable from context in a term *)

(* this function is used for precise computations of
input domains of operations *)

fun typesin z (Prop nil) = nil |

typesin z (Prop(x::L)) = (if x = z then [atype(arg1 prop1)]
else (typesin z x))@(typesin z (Prop L)) |

typesin z  (Unary(s, t)) =
if not (isutype(findtype s)) then nil else

let val utype(x,y) = findtype s in
 if z=t then [atype x] else typesin z t end |

typesin z (Binary(t,s,u)) =
if not (isbtype(findtype s)) then nil else

let val btype(x,y,w) = findtype s
 in (if z=t then [atype x] else typesin z t)@
(if z=u then [atype y] else typesin z u) end

|

typesin z (Binder(b,x,m,n,t)) = typesin z t |

typesin z  (Function1 t) = nil |

typesin z  (Function2 t) = nil  |

typesin z (Constant t) = nil |
```

```
typesin z  (App1(t,u)) = (typesin z t)@(typesin z u)|

typesin z  (App2(t,u,v)) = (typesin z t)@(typesin z u)@(typesin z v) |

typesin z  (Freevar (s,n)) = nil |

typesin z  (Boundvar(s,m,n)) = nil |

typesin z   (Firstarg(s,n)) = if s="P" then [atype(arg1 prop1)]
else if s="M" then  [atype(arg1 nat1)]  else nil|

typesin z  (Secondarg(s,n)) =  if s="Q" then [atype(arg1 prop1)]
 else if s="N" then  [atype(arg1 nat1)]  else nil |

typesin z  Error = nil;

(* both arguments must actually appear in any bracketed function *)


fun firstargumenttype1 t = let val L = typevarlist t and M =argumentlist t in

if M=nil orelse  not(oneitem M) then terror else

if L = nil then

let val T = typesin (hd M) t in
if T<>nil andalso oneitem T then typefloat(hd T)

else typefloat(argumenttype0(hd M))  end
else if isfirstarg (hd L) then argumenttype0(hd L) else terror end;

fun firstargumenttype2 t = let val L = typevarlist t and M = argumentlist t  in

if oneitem M orelse  not(oneitem(purge(hd M) M)) then terror else

let val A = if isfirstarg(hd M) then hd M
else if isfirstarg(hd(purge(hd M)M)) then hd(purge(hd M)M) else Error in

if A = Error then terror

else if foundin A L then argumenttype0 A

else let val T = typesin A t in
if T<> nil andalso oneitem T then typefloat(hd T)
else
```

```
typefloat(argumenttype0 A)

end end end;

fun secondargumenttype t = let val L = typevarlist t and M = argumentlist t  in

if oneitem M orelse  not(oneitem(purge(hd M) M)) then terror else

let val A = if issecondarg(hd M) then hd M
else if issecondarg(hd(purge(hd M)M)) then hd(purge(hd M)M) else Error in

if A = Error then terror

else if foundin A L then argumenttype0 A

else  let val T = typesin A t in
if T<>nil andalso oneitem T then typefloat(hd T)

else typefloat(argumenttype0 A)

end end end;

(*
```

Here we provide functions which find the arguments in bracketed terms, checking that there is just one argument term of each appropriate kind (first or second) and determining the contribution it makes to the type information.

The `typesin` function is a refinement which allows us to determine whether an argument can have a specific s.c. type (`prop` and `nat` being the possibilities so far). Arguments of the less definite type `obj` can be determined using `typevarlist`.

```
*)

(* list of all bound variables free in a term, without type considerations *)

(* this is needed because Frege requires that bound variables actually always occur
in their scopes  *)

(* it would be a good idea to have the type function post
an error message when a bound variable doesnt appear in its
scope, since this type error defies modern expectations. *)
```

```
fun varlist (Prop nil) = nil |

varlist (Prop(x::L)) = (varlist x)@(varlist (Prop L))|

varlist (Unary(s, t)) =

varlist t |

varlist (Binary(t,s,u)) =

(varlist t)@(varlist u) |

varlist (Binder(b,x,m,n,t)) = purge (Boundvar(x,m,n)) (varlist t)|

varlist (Function1 t) = nil |

varlist (Function2 t) = nil  |

varlist (Constant t) = nil|

varlist (App1(t,u)) = (varlist t)@(varlist u) |

varlist (App2(t,u,v)) = (varlist t)@(varlist u)@(varlist v) |

varlist (Freevar (s,n)) = nil |

varlist (Boundvar(s,m,n)) = [Boundvar(s,m,n)]|

varlist  (Firstarg(s,n)) = nil |

varlist  (Secondarg(s,n)) = nil |

varlist  Error = nil;

fun typeof (Prop nil) = terror |

typeof (Prop[x]) = let val T = typeof x in
if otype T then atype(arg1 prop1) else terror end|

typeof (Prop(x::L)) = let val T1 = typeof x
and T2 = typeof (Prop L) in if otype T1 andalso otype T2
then atype(arg1 prop1) else terror end |

typeof  (Unary(s,t)) = applyutype (findtype s)(typeof t) |

typeof (Binary(t,s,u)) = applybtype(findtype s)(typeof t)( typeof u) |
```

```
(* adding requirement that bound variables always appear in any scope *)

typeof (Binder(b,x,m,n,t)) =
 let val V = varlist t in if purge(Boundvar(x,m,n))(V) = V then terror else
  let val T = bindermatch(findtype b)(vartype x m n)(typeof t)
 in if typevarlist  (Binder(b,x,m,n,t)) = nil
then typefloat T
else T
  end end |

typeof (Function1 t) = if varlist t <> nil then terror else

 let val T = firstargumenttype1 t and U = typeof t in

 if T = terror orelse not(otype U)  then terror

(* else

if Typematches T U then atype(arg2 fun2) *)

else (utype(acast T,ocast(U))) end |

typeof (Function2 t) = if varlist t <> nil then terror else

let val T =
firstargumenttype2 t and U = secondargumenttype  t and V = typeof t in

if T = terror orelse U = terror orelse not (otype V) then terror

(* else if Typematches T U andalso Typematches T V andalso Typematches U V

then atype (arg2 rel2) *)

else (btype(acast T,acast U,ocast(V))) end |


typeof (Constant t) = findtype t |

typeof (App1 (t ,u)) = applyutype (typeof t)(typeof u) |

typeof (App2 (t,u,v)) = applybtype (typeof t)( typeof u)(typeof v) |

typeof (Freevar(x,n)) = if n>0 then freevartype(x,n) else terror|

typeof (Boundvar(x,m,n)) = if m>0 andalso n>0 then vartype x m n else terror|
```

```
typeof (Firstarg(s,n)) =if n>0 then  argumenttype s n else terror|

typeof (Secondarg(s,n)) = if n>0 then argumenttype s n else terror |

typeof Error = terror;

fun typetest s = typeof(parse s);

(*
```

And finally we present the type function itself. It makes use of the function `varlist` to enforce the condition that bound variables must occur in their scopes.

This is an all purpose type system, which handles typing of operators as well as terms.

Another essay! I'll talk about the intended semantics, and how the syntax and types support them. Frege says that the objects are propositions, natural numbers, and courses of values. In our scheme, there are necessarily also some atoms (lots of them): Frege wants all notations to be total, and with this in mind explains what the results of applying propositions as functions are. In our models, there are many many objects which are not actually courses of values: they can of course be applied as if they were courses of values, and, if I remember correctly, have constant extensions equal to the default value of the definite description operator, given Frege's definition of application.

The definition of $f`x$, application of objects, is

$$\texttt{the}(y \mapsto (\exists F : f = (z \mapsto F(z)) \wedge y = F(x))).$$

So applications of non-functions give the default object, which by convention we have taken to be the empty set (the constant course-of-values of the False). In knwn models of the theory we are implementing, there will be lots of non-functions: if the Axiom of Choice were added to our theory, it would be a theorem that there are many such non-functions (which we call atoms). It is curious that the definition of object function application requires the definition description operator. The definition of membership does not: $x \in y$ is defined as $(\exists F : y = (z \mapsto -F(z)) \wedge F(x))$. Notice that it is entirely natural that the interpreted set theory contains atoms, since anything other than a concept (unary function with proposition values) is treated as an urelement. The reasons why it is hard to have an extensional theory of object function application are much less obvious. It is amusing to observe that "the set x such that $\phi(x)$" is actually definable without recourse to definite description, as "the set of all z such that there is exactly one set which contains exactly those x such that $\phi(x)$, and z belongs to some set which contains exactly those x such that $\phi(x)$". The set of all x such that $\phi(x)$ is just $(x \mapsto -\phi(x))$. Further, one could develop the theory of functions with set outputs in this style without using a

primitive definite description operator, anachronistically using one of the well known implementions of functions as sets.

```
*)

fun freshen() = let val N = !NEWNUM in
(NEWNUM:=(!NEWNUM)+1;N) end;

(* list of all free variables in a term *)

fun freevarlist (Prop nil) = nil |

freevarlist (Prop(x::L)) = (freevarlist x)@(freevarlist (Prop L))|

freevarlist (Unary(s, t)) =

freevarlist t |

freevarlist (Binary(t,s,u)) =

(freevarlist t)@(freevarlist u) |

freevarlist (Binder(b,x,m,n,t)) =  (freevarlist t)|

freevarlist (Function1 t) = freevarlist t |

freevarlist (Function2 t) = freevarlist t  |

freevarlist (Constant t) = nil|

freevarlist (App1(t,u)) = (freevarlist t)@(freevarlist u) |

freevarlist (App2(t,u,v)) = (freevarlist t)@(freevarlist u)@(freevarlist v) |

freevarlist (Freevar (s,n)) = [Freevar(s,n)] |

freevarlist (Boundvar(s,m,n)) = nil |

freevarlist  (Firstarg(s,n)) = nil |

freevarlist  (Secondarg(s,n)) = nil |

freevarlist  Error = nil;

(* shift types in all bound variables uniformly *)
```

```
fun typeshift n (Prop nil) = Prop nil |
typeshift n (Prop(x::L)) = Prop((typeshift n x)::(deprop (typeshift n (Prop L)))) |
typeshift n (Unary (s,t)) = Unary(s, typeshift n t) |
typeshift n (Binary(t,s,u)) = Binary(typeshift n t,s,typeshift n u) |
typeshift n (Binder(b,x,m,N,t)) =
if n<0 andalso varlist (Binder(b,x,m,N,t)) = nil andalso argumentlist (Binder(b,x,m,N,t)
then (Binder(b,x,m,N,t)) else
 Binder(b,x,m+n,N,typeshift n t) |
typeshift n (Boundvar(x,m,N)) = Boundvar(x,m+n,N) |
typeshift n (App1(s,t)) = App1(typeshift n s,typeshift n t) |
typeshift n (App2(s,t,u)) = App2(typeshift n s, typeshift n t, typeshift n u) |
typeshift n t = t;
(* bound variable renumbering function *)
val RENUMBERS = ref[(0,0)];
val NEXTRENUMBER = ref 1;
val _ = RENUMBERS:=nil;
fun Renumber0 (Prop nil) = Prop nil |
Renumber0 (Prop(x::L)) = Prop((Renumber0 x)::(deprop(Renumber0 (Prop L)))) |
Renumber0 (Unary(s,t)) = Unary (s,Renumber0 t) |
Renumber0 (Binary(s,t,u)) = Binary(Renumber0 s,t,Renumber0 u) |
Renumber0 (Binder(b,x,m,n,t)) = let val N = find n ~1 (!RENUMBERS) in
if N<> ~1 then (Binder(b,x,m,N,Renumber0 t))
else (RENUMBERS:= (n,!NEXTRENUMBER)::(!RENUMBERS);
```

```
NEXTRENUMBER:=1+(!NEXTRENUMBER);
(Binder(b,x,m,find n 0 (!RENUMBERS),Renumber0 t)))

end | (*1*)

Renumber0 (App1(s,t)) = App1(Renumber0 s,Renumber0 t) |

Renumber0 (App2(s,t,u)) = App2(Renumber0 s,Renumber0 t, Renumber0 u) |

Renumber0 (Function1 t) = Function1(Renumber0 t) |

Renumber0 (Function2 t) = Function2(Renumber0 t) |

Renumber0 (Boundvar(x,m,n)) = let val N = find n ~1 (!RENUMBERS) in

if N<> ~1 then Boundvar(x,m,N)

else (RENUMBERS:= (n,!NEXTRENUMBER)::(!RENUMBERS);
NEXTRENUMBER:=1+(!NEXTRENUMBER);
(Boundvar(x,m,find n 0 (!RENUMBERS))))

end | (*2*)

Renumber0 x =x;

fun Renumber t = (RENUMBERS:=nil;NEXTRENUMBER:=1;Renumber0 t);

val OLDNEWNUM = ref 0;

fun Subs0 check x T (Prop nil) = Prop nil  |

Subs0 check  x T (Prop [t]) = if typeof t = atype(arg1 prop1)
then
Subs0 check x T t else Prop[Subs0 check x T t] |

Subs0 check x T (Prop ((Prop[u])::L)) = Subs0 check x T (Prop(u::L)) |

Subs0 check x T (Prop [a,b]) = let val A = Subs1 check x T a and
B = Subs1 check x T b in
if A = Error orelse B = Error then Error else
if isprop B then
Prop(A::(deprop B)) else Prop[A,B] end |

Subs0 check x T (Prop (y::L)) = let val Prop M = Subs0 check x T (Prop L) in

Prop((Subs1 check x T y)::M) end |
```

```
Subs0 check x T (Unary(s,Prop[t])) =

if propinputtype s then Unary(s,Subs1 check x T t)

else Unary(s,Subs1 check x T (Prop[t])) |

Subs0 check x T  (Unary (s,t)) = Unary(s,Subs1 check x T t) |

Subs0 check x T (Binary(Prop[t],s,u)) =
if proplefttype s  orelse typeof t = atype(arg1 prop1)
then Subs0 check x T (Binary(t,s,u))
else  Binary(Prop[Subs1 check x T t],s,Subs1 check x T u) |

Subs0 check x T (Binary(t,s,Prop[u])) =
if proprighttype s orelse typeof u = atype(arg1 prop1)
then Subs0 check x T (Binary(t,s,u))

else Binary(Subs1 check x T t,s,Prop[Subs1 check x T u]) |

Subs0 check x T (Binary(t,s,u)) = Binary(Subs1 check x T t,s,Subs1 check x T u) |

Subs0 check x T (Binder(b,y,m,n,t)) =
let val N = freshen() in Binder(b,y,m,N,
Subs1 check x T (Subs2 (Boundvar(y,m,n))
(Boundvar(y,m,N)) t)) end |

Subs0 check x (Function1 T)  (App1(s,t)) =

if x=s then let val L = argumentlist T in

let val A = hd L in

let val M = typevalue(acast(typeof t)) and N = typevalue(acast(typeof A)) in

if M= ~1 then Subs2 A t T

else Subs2 A t (typeshift (M-N) T)

end

end
end

else App1(Subs1 check x (Function1 T) s,Subs1 check x (Function1 T) t) |
```

```
Subs0 check x (Function2 T) (App2(s,t,u)) =

if x = s then

let val L = argumentlist T in

let val A = hd L

in
let val B = hd(purge A L) in

if isfirstarg A then

let val M = typevalue(acast(typeof A)) and M2 = typevalue(acast(typeof t))
and N = typevalue(acast(typeof B)) and N2 = typevalue(acast(typeof u)) in

if M2 = ~1 andalso N2 = ~1 then

Subs2 A t(Subs2 B u T)

else if M2 = ~1 then

Subs2 A t(Subs2 B u (typeshift (N2 - N) T))

else if N2 = ~1 orelse N2-N = M2 - M then

Subs2 A t(Subs2 B u(typeshift (M2 - M) T))

else Error

end

else let val M = typevalue(acast(typeof A)) and M2 = typevalue(acast(typeof u))
and N = typevalue(acast(typeof B)) and N2 = typevalue(acast(typeof t)) in

if M2 = ~1 andalso N2 = ~1 then

Subs2 A u(Subs2 B t T)

else if M2 = ~1 then

Subs2 A u(Subs2 B t (typeshift (N2 - N) T))

else if N2 = ~1 orelse N2-N = M2 - M then

Subs2 A u(Subs2 B t(typeshift (M2 - M) T))
```

```
else Error

end


end

end

end


else App2(Subs1 check x (Function2 T) s,
Subs1 check x (Function2 T) t,Subs1 check x (Function2 T) u) |

Subs0 check x T (App1(s,t)) =

App1(Subs1 check x T s,Subs1 check x T t) |

Subs0 check x T (App2(s,t,u)) =

App2(Subs1 check x T s,Subs1 check x T t,Subs1 check x T u) |

Subs0 check (Freevar(s,n)) T (Function1 U) =
Function1 (Subs1 check (Freevar(s,n))  T U) |

Subs0 check  (Freevar(s,n)) T (Function2 U) =
Function2 (Subs1 check (Freevar (s,n))  T U) |

Subs0 check (Constant t) T (Function1 U) =
Function1 (Subs1 check (Constant t)  T U) |

Subs0 check  (Constant t) T (Function2 U) =
 Function2 (Subs1 check(Constant t)  T U) |

Subs0 check x T (Function1 U) = Function1 U |

Subs0 check x T (Function2 U) = Function2 U |

Subs0 check x T U = if x = U then

if typeof x = atype(arg1 prop1) andalso typeof T <> atype(arg1 prop1)
then Prop [T] else

if typeof x = atype(arg1 nat1) andalso typeof T <> atype(arg1 nat1)
```

```
then Unary("#",T)

else T else U

and Subs1 check x T U = if (not check)
orelse(foundin x (varlist U) orelse foundin x (argumentlist U)
orelse foundin x (freevarlist U)) then Subs2 x T U else U

and Subs2 x t U = Subs0 false x t U

and Subs x T U =

(OLDNEWNUM:=(!NEWNUM);

let val X = Renumber(Subs0 true x T U)

in (NEWNUM:=max(!NEXTRENUMBER,!OLDNEWNUM);X) end);

fun substest x y z = let val Z = parse z and Y = parse y in
if typeof Y <>terror andalso typeof Z <>terror then
Display(Subs (parse x) (parse y) (parse z)) else "Type error" end;

fun ren s = OldDisplay(Renumber(parse s));


(*
```

Here is the substitution function. The term replaced is always an atomic term (variable, argument term, or constant).

We avoid bound variable collisions by the aggressive strategy of always renaming bound variables in the target term by applying a fresh index. This could be tidied up as we have done in other systems to avoid a profusion of larger and larger indices.

Notice that substitution of bracketed terms for atomic function terms causes substitution into the body of the bracketed term. Only free variables and constants may be replaced in substitutions into bracketed terms: argument terms are bound there, and bound variables should not appear. Moreoever, all types in the body of the bracketed term are shifted so that the type of the argument place(s) agrees with the type of the actual inputs (if the input types do not float).

In principle it is possible for substitution to fail for type shifting reasons, if something whose type is too low is substituted into the body of a bracket term, forcing variables into negative types. I believe this will never happen in any prover operation, because no application of a bracket term to arguments whose

types cannot float (and which thus contain free occurrences of bound variables) is ever actually made.

This function only checks type information in one place (when the term replaced in the substitution is an argument term of concrete type), but it is necessary to check that terms are well typed before making a substitution, as otherwise infinite loops are possible.

The substitution function carries out simplifications of propositional terms, but not in top level proper subterms which do not contain the atomic term being replaced. This is a refinement supporting the ability to prove theorems which are not in simplified form. Note that substitution of a fresh free variable for itself is a clever way to do simplifications of propositional structures, just at the top level.

5 The definition facility of Gottlob

Next comes declaration and use of definitions.

There is not a lot to say about this feature. It is entertaining that Frege's original type system makes it possible to tell exactly which of the various kinds of definitions (object constants, unary and binary operations taking objects to objects, term binding operations binding object or function variables with object output) a term is intended to define strictly from its type, so it is only necessary to give a fresh identifier and choose a term as its referent to to effect a definition. A definition may not contain a free variable.

Definition expansion (except in the trivial case of defined constants) is effected by substitution into application terms using the bracketed term referred to by the constant or operator defined. This may seem quite dangerous for variable binding operations, but it will work, noting that it is important that the `Defexpand` function is only applied to terms not containing free occurrence of bound variables or argument places.

The `setlog` command will open a log file (with extension `.gob`) which is both readable, with useful information, and executable. The `closelog` command closes the log file. When a log created with the setlog command is executed (which can be done with the ML use command) it starts by clearing the proof environment all the way down to the axioms, so one should use the same log file for an entire session, unless one edits out the part which clears the environment.

```
*)

(* log file commands needed for user commands to
be recorded with their outputs *)

val LOGFILE = ref (TextIO.openOut("default"));

fun Flush() = (TextIO.flushOut(TextIO.stdOut);TextIO.flushOut(!LOGFILE));

fun say s = (TextIO.output(TextIO.stdOut,"\n"^s^"\n\n");
Flush();TextIO.output(!LOGFILE,"\n"^s^"\n\n");Flush());

fun comment s = say("comment \""^s^"\";");

fun setlog s = (LOGFILE:= TextIO.openOut(s^".gob");
say "LINES:=nil;";
say "THEOREMS:=(!AXIOMS);";
say "DEFS:=nil;";
say "NEWNUM:=2;");

fun closelog() = (TextIO.flushOut(!LOGFILE);
TextIO.closeOut(!LOGFILE);LOGFILE:=TextIO.openOut("default"));
```

```
fun saycomment s = say ("(* ERROR:  "^s^" \n*)");

fun versiondate() = say "Gottlob Proof System 8/28/2018";


fun Define s t =  if foundin2 s (!DEFS) then "Already defined"

else let val D = parse t in

if freevarlist D <> nil then "Free variables present"

else let val T = typeof D in

if T = terror then "Badly typed"

else (DEFS:= (s,D)::(!DEFS); TYPES:=(s,T)::(!TYPES);

say ("Define \""^s^"\" \""^t^"\";");
say ("(*\n"^s^" := "^(indent(spaces(length(explode(s^" := "))))) (Display D))^" \n*)") ;

 s^" := "^(Display D))

end end;

fun isbracketfun (Function1 t) = true |

isbracketfun x = false;

fun thefirstarg (x,m,n) =

if x = "x" orelse x = "y" orelse x= "z" then Firstarg("X",m)
else if x = "F" orelse x = "G" orelse x = "H" then Firstarg("F",m)
else Error;


fun Defexpand (Constant s) =

if finddef s = Error then Constant s else finddef s |

Defexpand (Unary(s,t)) = if finddef s = Error then Unary(s,t) else
let val N = freshen() in
Subs (Freevar("f",N)) (finddef s) (App1(Freevar("f",N),t)) end |
```

```
Defexpand (Binary(t,s,u)) = if finddef s = Error then Binary(t,s,u) else
let val N = freshen() in
Subs (Freevar("r",N))(finddef s)(App2(Freevar("r",N),t,u)) end |

Defexpand (Binder(b,x,m,n,t)) =

if not (isbracketfun (finddef b)) then  (Binder(b,x,m,n,t)) else

let val Function1 B = finddef b in

let val F = Function1 (Subs (Boundvar(x,m,n)) (thefirstarg(x,m,n)) t)

in Subs (hd(argumentlist B)) F B

end end |

Defexpand (App1((Function1 t),u)) =
let val L = argumentlist t in Subs (hd L) u t end |

Defexpand (App2((Function2 t),u,v)) =
let val L = argumentlist t in
if isfirstarg(hd L) then Subs(hd L) u(Subs(hd(purge(hd L)L)) v t)
else Subs(hd L) v(Subs(hd(purge(hd L)L)) u t) end |

Defexpand t = t;

fun deftest s = Display(Defexpand(parse s));

(*
```

6 The proof environment of Gottlob

Now we get into entirely new headaches with the logical rules.

We summarize Frege's rules and remark on how we handle them.

1. The rule of "fusion of horizontals" (elimination of dashes in various contexts) is handled by the substtution function implicitly. A subtle point is that the substitution function does *not* carry out fusion of horizontals in top level subterms which do not contain the atomic subterm being subsituted for: this enables us to prove theorems which involve non-simplified terms.

2. in a term $x_1 \to x_2 \to \ldots \to x_n \to y$ (grouped to the right), Frege calls each x_i a *subcomponent* and each subterm $x_i \to x_2 \to \ldots \to x_n \to y$ (and y itself) a *supercomponent*.

3. Frege provides a rule that subcomponents can be permuted freely. We implement a rule which allows the subcomponent at a given position in a given line in the current proof environment to be brought to the front of the term. A small number of applications of this move will implement any desired permutation. In this and all my rules the result of the operation is added as a new front line to the environment, not replacing the line acted upon.

4. Frege gives a quite free notion of contrapositive: we provide a rule which replaces a given supercomponent of a given line in the book with its contrapositive in the usual sense. Experience suggests that it is not expensive to implement Frege's contrapositive moves using this move. It should be noted that inversion of the truth value of a term in Frege's sense involves automatic application of double negation.

5. Frege allows fusion of identical subcomponents in a line. We provide a command that fuses subcomponents at given positions in a given line. This is the first of several prover functions which require a function which determines whether two internal representations of terms are "the same". Our `Sameterms` function identifies terms which are the same up to renaming of bound variables and shifts of types by a constant amount in closed terms, and also automatically takes redundant dashes into account.

6. Frege allows universal generalization of supercomponents. We provide a function which replaces a given free variable in a given line with a bound variable (of course adding a universal quantifier to bind it) in a given supercomponent of a given line; the bound variable must be assigned a given type (and this can fail for stratification reasons). The free variable acted on cannot occur in the line outside the given supercomponent.

7. Frege provides three familiar rules of inference, which we can express exactly as they would be expressed in a modern system as long as we provide that we move relevant subcomponents to the front before applying

the rule. In all of these rules, the `Sameterms` function is used to check required identifications of propositions.

(a) modus ponens: if two given previous lines are of the form A and $A \to B$, we get a new line B.

(b) Two previous lines $A_1 \to \ldots \to A_n \to B$ and $B \to C$ allow us to deduce $A_1 \to \ldots \to A_n \to C$. We call this "transitivity of implication", though it might be thought a trifle more general than that.

(c) From $A \to C_1 \to \ldots \to C_m \to B$ and $\sim A \to D_1 \to \ldots \to D_n \to B$ we can deduce $C_1 \to \ldots \to C_m \to D_1 \to \ldots \to D_n \to B$. We call this "reasoning by cases", though again it might be thought to be more general than that.

8. We may replace a free variable of any kind uniformly in a line which has been proved: this includes replacement of functions with bracket terms, causing complicated further substitutions.

9. Generate an equation using Defexpand to implement reasoning with definitions. I am strongly tempted to generate implications instead to save steps.

10. Renaming of bound variables is automatically taken care of, because of our automatic generation of fresh variables during subsitution and our term identification function alluded to above.

The code which follows immediately implements the Gottlob proof environment, which consists of a list of lines to which new items are added at the beginning: line 1 is always the most recently generated command, and one should use the `Displaylines` command to check references which are not to the immediately preceding line or two, coimbined with a list of named theorems.

User commands to save a line as a theorem and to show a numbered line or theorem are provided. User commands to state axioms as theorems and instances of definitions as lines are given.

The commands to substitute a value for a free variable in a line or theorem are given: the result becomes line 1.

A command is given which uniformly moves types in a line. Our belief is that this is never actually needed.

It should be noted that prover lines and theorems cannot include free occurrences of bound variables or argument places.

```
*)

(* recently proved lines *)

val LINES = ref[parse "a=a"];
```

```
val _ = LINES:= nil;

fun pptest s = Display1 0 (parse s);

(* display the recent lines *)

fun Displaylines0 n nil = "\n\n" |

Displaylines0 n (x::L) =

(Displaylines0 (n+1) L)^"\n"^(makestring n)^".  "^(OldDisplay x)^"\n";

fun Displaylines2() = TextIO.output(TextIO.stdOut,Displaylines0 1 (!LINES));

(* Display the recent lines using the Frege-style pretty printer *)

fun Displaylines01 n nil = "\n\n" |

Displaylines01 n (x::L) =

(Displaylines01 (n+1) L)^"\n"^(makestring n)^".  "^(indent (spaces(length(explode((makes

fun Displaylines() = TextIO.output(TextIO.stdOut,Displaylines01 1 (!LINES));

fun getline0 n  L =

if n<1 then Error

else if L = nil then Error

else if n=1 then hd L

else getline0 (n-1) (tl L);

fun getline n = getline0 n (!LINES);

fun dropline0 n L =

if L = nil then nil

else if n=1 then tl L

else (hd L)::(dropline0 (n-1) (tl L));

fun dropline n = LINES:=dropline0 n (!LINES);
```

```
(* Theorem list *)

val THEOREMS = ref [("Id",parse "a=a")];

val _ = THEOREMS := nil;

val LEMMAS = ref (!THEOREMS);

fun thethm s = find s Error (!THEOREMS);

fun thelemma s = find s Error (!LEMMAS);

(* user command to introduce an axiom *)

fun Axiom s t = if thethm s <> Error
then "Theorem name already in use"

else let val T = parse t in

if T = Error then

(saycomment("Badly formed term");
say("(*\n"^s^":  "^t^" \n*)");
"Badly formed term"
)

else let val U = typeof T in

if U = terror orelse varlist T <> nil orelse argumentlist T <> nil
then "Badly typed term or bad atomic subterms"

else (THEOREMS:=(s,T)::(!THEOREMS);

say("Axiom \""^s^"\" \""^t^"\";");

say("(*\n"^s^":  "^t^" \n*)");

s^":  "^t) end end;

(* user command to record a line as a theorem *)

fun Theorem s n = if thethm s <> Error
then (saycomment "Theorem name already in use";
saycomment ("(*\n"^s^":  "^(indent(spaces(length(explode(s^":  "))))(Display (thethm s))
"Theorem name already in use")
```

```
else (THEOREMS:=(s,getline n)
::(!THEOREMS); LINES:= [getline n];

say("Theorem \""^s^"\" "^(makestring n)^";");

say ("(*\n"^s^":  "^(indent(spaces(length(explode(s^":  "))))(Display (thethm s)))^" \n*

s^":  "^(Display (thethm s)));

fun Lemma s n = (LEMMAS:=(s,getline n)
::(purge2 s (!LEMMAS)); LINES:= [getline n];

say("Lemma \""^s^"\" "^(makestring n)^";");

say ("(*\n"^s^":  "^(indent(spaces(length(explode(s^":  "))))(Display (thelemma s)))^" \

s^":  "^(Display (thelemma s)));


fun showline n = (makestring n)^":  "^(indent (spaces(length(explode((makestring n)^":

fun showthm s = s^":  "^(indent(spaces(length(explode(s^":  "))))(Display(thethm s)));

fun showlemma s =  s^":  "^(indent(spaces(length(explode(s^":  "))))(Display(thelemma s)

(* user command to introduce a definition expansion as a line *)

fun Definition t = let val T = parse t in

if T = Error then

(saycomment "Badly formed term";
saycomment ("Definition \""^t^"\";");
"Badly formed term" )

else

let val U = Defexpand T in

if U = Error orelse T=U orelse varlist U <> nil
orelse argumentlist U <> nil
then

(saycomment  "Unsuccessful typing or definition expansion";
saycomment ("Definition \""^t^"\";");
 "Unsuccessful typing or definition expansion")
```

```
else (LINES := (Binary(U,"=",T))::(!LINES);

say ("Definition \""^t^"\";");

say("(*\n"^(showline 1)^" \n*)"); (*1 *)

showline 1)

end end;

(* user commands to introduce specifications of free variables
in lines or theorems as lines;  this will also simply copy lines
or copy theorems to lines *)

fun Specifyline x t n = let val X = parse x and T = parse t in

if typeof T = terror then "Badly typed source" else

if varlist T <> nil orelse argumentlist T <> nil
then
(saycomment  "Bad atomic terms in source";
say("(*\n"^(showline (n))^" \n*)");
saycomment ("Specifyline \""^x^"\" \""^t^"\" "^(makestring n)^";");
"Bad atomic terms in source"
)
else

let val U = Subs X T (getline n) in

if U = Error orelse typeof U = terror
then (

saycomment "Unsuccessful substitution";
say("(*\n"^(showline (n))^" \n*)");
saycomment("Specifyline \""^x^"\" \""^t^"\" "^(makestring n)^";");
"Unsuccessful substitution"
)

else (LINES:=((Renumber U)::(!LINES));

say("Specifyline \""^x^"\" \""^(ren t)^"\" "^(makestring n)^";");

say("(*\n"^(showline (n+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)"); (* 2 *)
```

```
showline 1)

end

end;

fun Specifythm x t s  = let val X = parse x and T = parse t in

if typeof T = terror then "Badly typed source" else

if varlist T <> nil orelse argumentlist T <> nil
then (saycomment "Bad atomic terms in source";

saycomment("Specifythm \""^x^"\" \""^t^"\" \""^s^"\";");

"Bad atomic terms in source") else

let val U = Subs X T (thethm s) in

if U = Error orelse typeof U = terror
then (  saycomment"Unsuccessful substitution";

saycomment("Specifythm \""^x^"\" \""^t^"\" \""^s^"\";");

"Unsuccessful substitution")

else (LINES:=((Renumber U)::(!LINES));
say("Specifythm \""^x^"\" \""^(ren t)^"\" \""^s^"\";");

say("(*\n"^(showline 1)^" \n*)"); (* 3 *)

showline 1)

end

end;


fun Specifylemma x t s  = let val X = parse x and T = parse t in

if typeof T = terror then "Badly typed source" else

if varlist T <> nil orelse argumentlist T <> nil
then (saycomment "Bad atomic terms in source";
```

```
saycomment("Specifylemma \""^x^"\" \""^t^"\" \""^s^"\";");

"Bad atomic terms in source") else

let val U = Subs X T (thelemma s) in

if U = Error orelse typeof U = terror
then (  saycomment"Unsuccessful substitution";

saycomment("Specifylemma \""^x^"\" \""^t^"\" \""^s^"\";");

"Unsuccessful substitution")

else (LINES:=((Renumber U)::(!LINES));

say("Specifylemma \""^x^"\" \""^(ren t)^"\" \""^s^"\";");

say("(*\n"^(showline 1)^" \n*)"); (* 3 *)

showline 1)

end

end;

(* user command which uniformly shifts types in a line *)

fun Raisetypes n m =

let val T = typeshift m (getline n) in

if T = Error orelse typeof T = terror

then (
saycomment "Unsuccessful type shift";
say("(*\n"^(showline (n))^" \n*)");
saycomment  ("Raisetypes "^(makestring n)^" "^(makestring m)^";");
"Unsuccessful type shift")

else (LINES:= (Renumber T)::(!LINES);

say ("Raisetypes "^(makestring n)^" "^(makestring m)^";");

say("(*\n"^(showline (n+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)"); (* 4 *)
```

```
showline 1) end;

(*
```

Here we create the proof environment, which consists of a list of lines currently being manipulated and a repository of named theorems.

I have now added an additional repository of lemmas, the temporarily named lines in Frege's proofs, which reduces the need for distant backward line references. Also, all but the last line is discarded when a lemma or theorem is proved: short term memory display and management will be easier, and the whole structure will look more like what Frege describes.

6.1 The axioms

```
*)



(* the axioms *)

val _ = Axiom "Ia" "a->b->a";

val _ = Axiom "Ib" "a->a";

val _ = Axiom "IIa" "{A x:f(x)} -> f(a)";

val _ = Axiom "IIb" "{I F: Z1(F)} -> Z1(f)";

val _ = Axiom "III" "g(a=b) -> g({IF:F(b)->F(a)})";

val _ = Axiom "IV" "~(--a = ~b) -> (--a = --b)";

val _ = Axiom "V" "({x=>f(x)}={x=>g(x)}) = {A x:f(x)=g(x)}";

val _ = Axiom "VI" "a=the{x=>a=x}";

(*
```

Here we state the axioms from the Grundgesetze.

Now we need to develop rules of inference. Our forms of the rules may be eccentric.

The axioms implementing fusion of horizontals are actually provable, except for `Fusion4`, which really is needed as an axiom unless we allow more detailed

input types for function types (the prover does not know enough to automatically handle leading dashes in the body of a universal quantification).

```
*)

(* axioms implementing fusions of horizontals *)

val _ = Axiom "Fusion1" "--(--a) = --a";

val _ = Axiom "Fusion2" "--(--a -> -- b) = a -> b";

val _ = Axiom "Fusion3"  "--(~(--a))= ~a";

val _ = Axiom "Fusion4"  "--{A x:--f(x)} = {A x:f(x)}";

val _ = Axiom "Fusion5" "--{IF:--Z1(F)}={IF:Z1(F)}";

val AXIOMS = ref(!THEOREMS);

(*
```

6.2 The rules of inference

In this subsection, we actually define the notions of subcomponent and supercomponent and implement the rules of inference not yet handled along the lines indicated above.

A possible weakness of our approach is that we envison application of the rules only at the top levels of lines (though many of them allow application to numbered supercomponents). If Frege makes a lot of use of application of the rules to more deeply embedded subterms, we might need to make the functions a bit more sophisticated. There is no question but that we have adequate logical power in principle: the issue, which can only be tested by looking at what Frege actually does, is whether our rules might need to be augmented to be faiithful to what Frege does. Our aim is to have a very small uniform blowup factor in number of rules applied.

```
*)

(* hypotheses *)

fun subcomponent n (Prop nil) = Error |

subcomponent n (Prop([x])) = Error |
```

```
subcomponent n (Prop(x::L)) =

if n<1 then Error

else if n=1 then x

else subcomponent (n-1) (Prop L) |

subcomponent n T = Error;

fun dropsub n (Prop nil) = Error |

dropsub n (Prop([x])) = Error |

dropsub n (Prop(x::L)) =

if n<1 then Error

else if n = 1 then Prop L

else let val T  = dropsub(n-1)(Prop L) in
if T = Error then Error else

Prop(x::(deprop (T))) end|

dropsub n T = Error;

fun bringtofront n t =

let val T = subcomponent n t

and U = dropsub n t in

if T=Error orelse U=Error then

Error

else Prop(T::(deprop U)) end;

(* user command to bring the nth hypothesis to
the front in the mth line, as a new line *)

fun Permline m n =

let val T = bringtofront n (getline m)
```

```
in if T = Error then (
saycomment "Permutation error";

saycomment(Displaylines0 1 (!LINES));

saycomment("Permline "^(makestring m)^" "^(makestring n)^";");

"Permutation error")

else (LINES:=(T)::(!LINES);

say("Permline "^(makestring m)^" "^(makestring n)^";");

say("(*\n"^(showline (m+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)");

showline 1) end;

fun supercomponent n (Prop(x::L)) =

if n<0 then Error
else if n=0 then Prop(x::L)
else if n=1 andalso L<>nil then Prop L
else supercomponent (n-1) (Prop L) |

supercomponent n T =
if n=0 then T else Error;

fun subcomponents n (Prop(x::L)) =

if n<0 then nil
else if n=0 then nil
else if n=1 andalso L<>nil then [x]
else (x::(subcomponents (n-1) (Prop L))) |

subcomponents n T = nil;


fun invert (Prop[t]) = invert t |
invert (Unary("~",t)) = t |
invert t = Unary("~",t);

 fun contra1 t = let val T = subcomponent 1 t and
U = supercomponent 1 t in
```

```
if T = Error orelse U = Error then Error
else Prop[invert U,invert T] end;

 fun contra n (Prop nil) = Error |

contra n (Prop(x::L)) =

if n<1 then Error

else if n=1 then contra1 (Prop(x::L))

else let val T = contra (n-1)(Prop L) in

if T = Error then Error else

Prop(x::(deprop T)) end |

contra n x = Error;

(* use command to perform contrapositive
maneuver at a position indicated by n in the  mth line *)

fun Contra m n =

let val T = contra n (getline m) in

if T = Error then (
saycomment"Contrapositive error";
saycomment(Displaylines0 1 (!LINES));
saycomment ("Contra "^(makestring m)^" "^(makestring n)^";");
"Contrapositive error")
else (LINES:= (T)::(!LINES);

say("Contra "^(makestring m)^" "^(makestring n)^";");

say("(*\n"^(showline (m+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)"); (* 5 *)

showline 1) end;

(* equivalent of terms up to renaming of bound variables;
this could be enhanced to get stronger ability to recognize
equivalent terms *)

fun sameterms (Prop nil) (Prop nil) = true |
```

```
sameterms (Prop(x::L)) (Prop(y::M)) =

sameterms x y andalso sameterms (Prop L) (Prop M) |

sameterms (Prop [x]) y =  typeof y = atype(arg1 prop1) andalso sameterms x y |

sameterms y (Prop[x]) = typeof y = atype(arg1 prop1) andalso sameterms x y |

sameterms (Unary(s,t)) (Unary(u,v)) =

s=u andalso sameterms t v |

sameterms (Binary(s,t,u)) (Binary(f,g,h)) =
sameterms s f andalso t=g andalso sameterms u h |

sameterms (Binder(s,x,m,n,t)) (Binder(s2,x2,m2,n2,t2)) =

if s=s2 andalso m<>m2 andalso varlist (Binder(s,x,m,n,t))  = nil
andalso varlist (Binder(s2,x2,m2,n2,t2)) = nil
then if m>m2
then sameterms (Binder(s,x,m,n,t))
(typeshift (m-m2) (Binder(s2,x2,m2,n2,t2)))
else if m2>m then sameterms (typeshift (m2-m) (Binder(s,x,m,n,t)))
 (Binder(s2,x2,m2,n2,t2))

else false

else s=s2 andalso

m=m2 andalso

let val N = freshen() in

sameterms (Subs (Boundvar(x,m,n)) (Boundvar(x,m,N)) t)
 (Subs (Boundvar(x2,m2,n2)) (Boundvar(x,m,N)) t2) end |

sameterms (Function1 t) (Function1 u) = sameterms t u |

sameterms (Function2 t) (Function2 u) = sameterms t u |

sameterms (App1(t,u)) (App1(t2,u2)) =
sameterms t t2 andalso sameterms u u2 |

sameterms (App2(t,u,v)) (App2(t2,u2,v2)) =
sameterms t t2 andalso sameterms u u2 andalso sameterms v v2 |
```

```
sameterms Error t = false |

sameterms t Error = false |

sameterms t u = t=u;

fun Sameterms t u =

let val OLD = !NEWNUM in let val N = freshen() in

let val T = sameterms (Subs (Freevar("a",N))
 (Freevar("a",N)) t)
(Subs (Freevar("a",N)) (Freevar("a",N)) u)
in (NEWNUM:=OLD;T) end end end;

(* user command to
fuse hypotheses m and n if identical in line r *)

fun Samehyps r m n =

if m=n then (
saycomment "Same indices, nothing to do";

saycomment("Samehyps "^(makestring r)^" "^(makestring m)^" "^(makestring n)^";");

"Same indices, nothing to do")

else let val T = subcomponent m (getline r)
and U = subcomponent n (getline r) in

if T = Error orelse U = Error
then (
saycomment "Badly numbered hypotheses";

saycomment(Displaylines0 1 (!LINES));

saycomment("Samehyps "^(makestring r)^" "^(makestring m)^" "^(makestring n)^";");

"Badly numbered hypotheses")

else if not(sameterms T U)
then ( saycomment "Hypotheses do not match";

saycomment(Displaylines0 1 (!LINES));
```

```
saycomment("Samehyps "^(makestring r)^" "^(makestring m)^" "^(makestring n)^";");

"Hypotheses do not match")

else (LINES:= ((dropsub m (getline r))):(!LINES);

say("Samehyps "^(makestring r)^" "^(makestring m)^" "^(makestring n)^";");

say("(*\n"^(showline (r+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)"); (* 6 *)

showline 1) end;

fun samelists nil x = false |

samelists x nil = false |

samelists [x] [y] = sameterms x y |

samelists (x::L) (y::M) = sameterms x y andalso samelists L M;

fun getboundvar m (Freevar(s,n)) =
let val N = freshen() in if s="a" orelse s="b"
orelse s="c" then Boundvar("x",m,N)
else if s="f" orelse  s="g" orelse s="h" then Boundvar("F",m,N)
else Error end |
getboundvar m T = Boundvar("?!?!",0,0);

fun getquantifier (Freevar(s,n)) = if s="a" orelse s="b" orelse s="c" then "A"
else if s ="f" orelse s="g" orelse s="h" then "I"
else "?!?!"|

getquantifier T = "?!?!";

fun ugen1 x n t = if foundin x (freevarlist t) then
let val XX = getboundvar n x in
let val T = Subs x (XX) t in
if typeof T = terror then Error else
let val Boundvar(X,M,N) = XX in
Binder(getquantifier x,X,M,N,T) end end end
else Error;

fun ugen x n m (Prop(y::L)) =

if m<0 then Error
```

```
else if m=0 then ugen1 x n (Prop(y::L))

else if foundin x (freevarlist y) then Error

else let val T = ugen x n (m-1) (Prop L) in
if T = Error then Error else
Prop(y::(deprop T)) end|

ugen x n m T =

if m=0 then ugen1 x n T else Error;

(* user command to carry out universal generalization
in line r replacing free variable x with a bound variable
of type n in the mth supercomponent *)

fun Ugen r x n m  =

let val X = parse x in

let val T = ugen X n m (getline r) in

if T = Error then
(saycomment "Bad generalization";

saycomment(Displaylines0 1 (!LINES));

saycomment("Ugen "^(makestring r)^" \""^x^"\" "^(makestring n)^" "^(makestring m)^";");

"Bad generalization")

else (LINES := (Renumber T)::(!LINES);

say("Ugen "^(makestring r)^" \""^x^"\" "^(makestring n)^" "^(makestring m)^";");

say("(*\n"^(showline (r+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)");

showline 1)end end;

fun mp t (Prop(x::L)) =

if (sameterms t x orelse sameterms t (Prop[x]) )
andalso L<>nil then if length L = 1
```

```
then hd L else Prop L else Error |

mp t u = Error;

(* user command, modus ponens *)

fun Mp m n =

let val T = mp (getline m) (getline n) in

if T=Error then (
saycomment  "Not modus ponens";
saycomment(Displaylines0 1 (!LINES));
saycomment("Mp "^(makestring m)^" "^(makestring n)^";");

 "Not modus ponens")

else (LINES:=  (T)::(!LINES);

say("Mp "^(makestring m)^" "^(makestring n)^";");

say("(*\n"^(showline (m+1))^" \n*)");

say("(*\n"^(showline (n+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)");

showline 1) end;

fun transimp (Prop(x::L)) (Prop(y::M)) n =

if sameterms y (supercomponent n (Prop(x::L)))
  orelse length(deprop(supercomponent n (Prop(x::L)))) = 1
andalso sameterms y (hd (deprop(supercomponent n (Prop(x::L)))))

then  Prop((subcomponents n (Prop(x::L)))@M)  else Error |

transimp t u n = Error;

(* user command, transitivity of implication *)

fun Transimp m n n2 =

let val T = transimp (getline m) (getline n) n2 in

if T = Error then (
```

```
saycomment "Not transitive implication";
say("(*\n"^(showline (m))^" \n*)");

say("(*\n"^(showline (n))^" \n*)");
saycomment(Displaylines0 1 (!LINES));
saycomment("Transimp "^(makestring m)^" "^(makestring n)^" "^(makestring n2)^";");

"Not transitive implication")

else (LINES:=((T)::(!LINES));

say("Transimp "^(makestring m)^" "^(makestring n)^" "^(makestring n2)^";");

say("(*\n"^(showline (m+1))^" \n*)");

say("(*\n"^(showline (n+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)");

showline 1) end;

fun cases (Prop((Unary("~",t))::L1)) (Prop(u::L2)) n  =

let val n2 = n + (length L2) - (length L1) in

if sameterms t u andalso sameterms
(supercomponent (n-1) (Prop L1)) (supercomponent (n2-1) (Prop L2)) then

let val L = (subcomponents (n-1) (Prop L1))@
 (subcomponents (n2-1) (Prop L2))@
(deprop(supercomponent (n-1) (Prop L1))) in
if length L = 1 then hd L else Prop L end

else Error end |

cases t u n  = Error;

(* user command, reasoning by cases *)

fun Cases m n n2 =

let val T = cases (getline m) (getline n) n2 in

if T = Error then (
saycomment "Not reasoning by cases";
say("(*\n"^(showline (m))^" \n*)");
```

```
say("(*\n"^(showline (n))^" \n*)");
saycomment(Displaylines0 1 (!LINES));
saycomment("Cases "^(makestring m)^" "^(makestring n)^" "^(makestring n2)^";");

"Not reasoning by cases" )
else  (LINES:=(T)::(!LINES);

say("Cases "^(makestring m)^" "^(makestring n)^" "^(makestring n2)^";");

say("(*\n"^(showline (m+1))^" \n*)");

say("(*\n"^(showline (n+1))^" \n*)");

say("(*\n"^(showline 1)^" \n*)");


showline 1) end;

(*
```

One important observation is that our system of levels restricts the use of universal generalization. A type must be proposed by the user for the free variable to be replaced with a bound variable, and it is quite possible that a free variable in a general proposition may be impossible to type uniformly: such a general theorem cannot be converted to a universally quantified assertion.

A part of the program presented here must be an attempt to address the issue of whether the stratification restriction on function and concept formation can be justified on a philosophical basis. We do believe that this can be done, but it is of course very tricky.

The function `Sameterms` checks whether terms are the same in a suitable sense. It checks for identity mod bound variable renaming and mod uniform shifting of types in closed terms, and in addition checks for identity mod propositional simplification (after elimination of redundant dashes).

7 Semantics for our formal system in NFU with the Axiom of Counting

In this section (mercifully free of computer code) we discuss the interpretation of what we have done above in Jensen's set theory NFU with the addition of Rosser's Axiom of Counting.

Quine's New Foundations is the first order theory with equality and membership whose axioms are

extensionality: $(\forall xy : (\forall z : z \in x \leftrightarrow z \in y) \rightarrow x = y))$

stratified comprehension: For each formula ϕ for which there is a function σ from variables appearing in ϕ (bound or free, as items of syntax) to natural numbers such that when $u \in v$ appears in ϕ we have $\sigma(u) + 1 = \sigma(v)$ and when $u = v$ appears in v we have $\sigma(u) = \sigma(v)$, we say that ϕ is *stratified*, we call σ a *stratification* of ϕ, and we adopt $(\exists A : (\forall x : x \in A \leftrightarrow \phi))$ as an axiom, where A does not appear in ϕ.

In this and the subsequent theory, free variables may occur in axioms, and are understood to be implicitly universally quantified. Note that this is analogous to the treatment of free variables in Frege's own logic.

The consistency of this system is a vexed question. However, the following modification was shown to be consistent by Jensen:

weak extensionality: $(\forall xyw : (w \in x \wedge (\forall z : z \in x \leftrightarrow z \in y)) \rightarrow x = y))$

stratified comprehension: For each formula ϕ for which there is a function σ from variables appearing in ϕ (bound or free, as items of syntax) to natural numbers such that when $u \in v$ appears in ϕ we have $\sigma(u) + 1 = \sigma(v)$ and when $u = v$ appears in v we have $\sigma(u) = \sigma(v)$, we say that ϕ is *stratified*, we call σ a *stratification* of ϕ, and we adopt $(\exists A : (\forall x : x \in A \leftrightarrow \phi))$ as an axiom, where A does not appear in ϕ.

This system NFU modifies the original NF to allow many objects with no elements. It is usual to specify a constant $\emptyset$ with no elements to serve as the empty *set* and to call all objects which either have elements or are equal to $\emptyset$ *sets* and call all other objects *atoms* or *urelements* (thus the U in the name of the theory).

We note some modifications of stratification. First of all, we can restrict the domains of stratifications to bound variables. If a formula becomes an axiom of comprehension by distinguishing some free occrrences of some variables, notice that we can deduce thie offending formula as a theorem by asserting the instance of the comprehension scheme obtained by distinguishing occurrences of free variables, then reimposing the identifications by using the fact that free variables are implicitly universally quantified and making substitutions of variables to restore the original formula from the axiom.

Secondly, we can introduce terms with stratification rules. A simple way to do this is to introduce a definition description operator. Define $\phi[(\theta x : \psi)]$ contextually as

$$((\exists y : (\forall x : x = y \leftrightarrow \psi)) \wedge \phi[y/x]) \vee (\neg(\exists y : (\forall x : x = y \leftrightarrow \psi)) \wedge \phi[\emptyset/x]).$$

This has the exact effect of allowing $(\theta x : \psi)$ to denote the unique x such that ψ, or the empty set if there is no such x. It is then clear that the conditions for stratification of the extended language with definite descriptions constitute extension of stratifications to all terms, with the condition that the type assigned to $(\theta x : \psi)$ is the same as that assigned to x; now all operations defined in set theory can be regarded as abbreviations for definite descriptions. The idea of stratification for general term constructions is then that a term $f(x_1, \ldots x_n)$ will be assigned a value under a stratification with a constant displacement δ_i from the values assigned by the stratification to each x_i (with the effect that the difference between the values assigned to x_i and x_j must be $\delta_i - \delta_j$).

A particular case of definite description is the usual notation for set abstracts: $\{x : \phi\}$ can be read as $(\theta A : (\forall x : x \in A \leftrightarrow \phi))$ (our choice of default object makes this work correctly for empty extensions where we are not guaranteed a unique witness).

We can then define $\{x\}$ as $\{y : y = x\}$ (one type higher than x), (x, y) as $\{\{x\}, \{x, y\}\}$ (two types higher than x and y, which must be of the same type), $(\lambda x : T)$ as $\{p : (\exists x : p = (x, T))\}$ (three types higher than x and T), and $f[x]$ as $(\theta y : (x, y) \in f)$, the same type as x and three types lower than f, and $f(x)$ as $(\theta y : f = (\lambda u : f[u]) \wedge (x, y) \in f)$, also the same type as x and three times lower than f: the latter form of application has the exact behavior for non-functions that we want. This gives a nonextensional theory of functions under which all non-functions map everything to the empty set. We define $\mathbf{t}$ as the universal set and $\mathbf{f}$ as the empty set, then invite the reader into the theory whose primitives are the primitives of first order logic, stratified λ abstraction, β-reduction, equality, definite description ($\mathtt{the}f$ to be defined as the unique x such that $f(x) = \mathbf{t}$, if there is one, and otherwise $\mathbf{f}$: this is stratified and of the type of the arguments of f, not of f itself), and the assertion that $\mathbf{t}$ and $\mathbf{f}$ are distinct. Notice that in this theory all values of stratifications can be taken to be multiples of three...and so we can conveniently divide them by three.[6]

In this theory of functions, we can interpret all of the primitive notions of Frege's logic (or we are given them already) in stratified forms. $\sim x$ is defined as $\mathbf{t}$ if $x = \mathbf{t}$ and $\mathbf{f}$ otherwise. There is a function which implements $\sim$ in the underlying set theory. Similarly there is a function which implements `--`. We read $x = y$ as a term equal to $\mathtt{t}$ if $x = y$ in the usual sense and $\mathtt{f}$ if $x \neq y$; this is definable in terms of the underlying set theory in a stratified way, and we will see below how to represent it as a function. We read $x \to y$ as a term equal to

[6]The usual method for achieving a type differential of one between functions and their arguments, via postulating the Quine type-level pair, is irrelevant to our purposes. It is interesting to see that this type differential can be achieved without any particular attention to the nature of the pair: in fact, the Kuratowski pair used to build the functions in our interpretation disappears froim view as soon as we pass to the interpretation.

$\mathbf{f}$ if x is equal to $\mathbf{t}$ and y is not equal to $\mathbf{t}$, and otherwise equal to $\mathbf{t}$, which is again definable in stratified terms.

The type distinction, crucial to Frege, between functions and objects, is actually entirely ignored in our semantics: the interpretation of $[f(X)]$ is just $(\lambda x : f(x))$. We do not have a pair here, so we interpret $[\phi(X, Y)]$ more deviously. We first redefine $\{x\}$ as $(\lambda x : x = \mathbf{t})$ (and more generally can reinterpret $\{x : \phi\}$ as $(\lambda x : \phi = \mathbf{t})$). The notation $x = y$ of course denotes $\mathbf{t}$ if $x = y$ in the usual sense and $\mathbf{f}$ otherwise. To see that this allows us to recover set theory in our theory of functions, note that a term p whose value is a proposition can be freely raised or lowered in type. Define $--p$ as $\mathtt{t}$ if p is $\mathbf{t}$ and as $\mathbf{f}$ otherwise. We can define the function $P(x)$ as $(\theta y : (x = \mathbf{t} \wedge y = (\lambda z : \mathbf{t})) \vee (x \neq \mathbf{t} \wedge y = (\lambda z : \mathbf{f})))$. $(\lambda x : P(x))$ exists, and we can briefy call this function P. Now observe that $P(p) = (\lambda u : --p)$ for *any* p, and $P(p) = (\lambda u : p)$for a truth value p. We can also define $P^{-1}(x) = (\theta y : P(y) = x)$: $P^{-1}(\lambda u : p)$ is then $--p$ (the default object for unsatisfied definite descriptions is $\mathbf{f}$ on our current definitions; but we could define P^{-1} more carefully to get this default value anyway), and so equal to p if p is a truth value. This means that we can raise the type of a proposition valued expression p by replacing it with $P(p)(\emptyset)$ or lower it by replacing p with $P^{-1}(\lambda u : p)$, without changing its meaning (in propositional contexts, we arrange for p to be replaceable by $\mathtt{--}p$). This gives sense to our assignment of type ∞ to proposition-valued terms.

We interpret $[f(X, Y)]$ as $(\lambda x : (\lambda y : f(\{x\}, y)))$, using the new definition of singleton set: we identify functions of two variables with the abstractions denoting a stratified version of what Frege calles their double value ranges.[7] Note that we have entirely avoided any appeal to a notion of pairing in the development in our theory of functions. The functions of higher orders which implement the quantifiers (and can implement other defined variable binding operations in the system of Frege) collapse to λ abstractions in ways a reader can work out via the collapse of their higher order domains to the first order objects of our theory of functions. The first order quantifier becomes

$$\forall = (\lambda x : (\lambda y : x(y)) = \mathbf{t}) = (\lambda y : \mathbf{t})).$$

The second order quantifier can be interpreted as

$$(\lambda x : (\lambda y : x((\lambda u : y(u)))) = (\lambda y : \mathbf{t})) :$$

one has to recall that the second order quantifier acts only on functions from unary functions to truth values.

Natural numbers can be defined in the theory of functions presented exactly as Frege defines them (which is essentially the way they are defined in NFU in modern treatments). We define the map # sending each natural number to itself and each non-natural number to 0, and then we implicitly assert Rosser's

[7]We were charmed, years ago, to discover that Frege is the inventor of currying. His use of double value ranges is the reason that we declare that binary functions are assigned type 2 higher than that of their arguments: a displacement of 1 could easily be justified but we follow his preferred representation.

Axiom of Counting by asserting that the types of terms $\#(N)$ can be raised or lowered freely. If the Axiom of Counting holds, there is a function defined by a stratified recursion which sends each natural number n to $\{n\}$: this function can be used to demonstrate that terms with natural number values can be freely raised and lowered in type. The original form of the axiom is the assertion that $|\{1, \ldots, n\}| \in n$, which is not stratified, but which becomes stratifiable and provable if each n is qualified with $\#$. This is very similar to the statement Frege uses to prove Infinity. The Axiom of Counting is a consequence of unstratified mathematical induction, but is not nearly as strong. Jensen's original paper on consistency of NFU establishes the existence of ω-models of NFU, in which Counting holds. It has been shown that the consistency strength of NFU + Counting is that of Zermelo set theory with the existence of $\beth_n$ for each concrete natural number n, which is quite a lot of set theory.

It should not be particularly hard to see (though we will work on filling in more details) that we have described an exact implementation of the logic of Gottlob in NFU + Counting, with the unexpected feature that all the second- and higher-order types collapse to subtypes of the type of objects.

8 Philosophical motivation for stratification, and possible consequences for the logicist program of Frege

This section (also innocent of computer code) is very much under construction.

It is a fact of mathematical and philosophical significance that the original program of Frege failed. Frege did not make the error which to our minds underlies "naive set theory" naively presented, of overlooking the distinction in sort between objects and predicates of objects (or operations taking objects to objects). He acknowledges the distinction of sort and explicitly postulates an embedding of the operations (and so the predicates) into the domain of objects, sending operations with distinct extensions to distinct objects, which turns out on examination to be impossible. It is an interesting question whether this correlation, postulated as Axiom V, is actually a logical hypothesis at all. It certainly cannot be a logical principle, since it turns out to be false!

It is a mathematical fact that the program of Frege as modified by stratification restrictions avoids paradox. Whether this is a fact of philosophical interest requires discussion. We do not modify Axiom V in our approach (another neo-Fregean approach is to dump Axiom V in favor of the much weaker Hume's Principle, which supports the exposition of the Grundgesetze but gives a much weaker logical system); what we actually modify is the ways in which operations and predicates applying to objects can be defined. It is a mathematical fact that our restriction averts paradox. For this to be a fact of philosophical interest, an account needs to be given as to why this restriction of the formation of operations and predicates makes sense. One should note that a successful explanation of this kind would also bear on the question of the philosophical interest of Quine's set theory New Foundations.

It is important to note that while Frege restricts the scope of his work in such a way as to quantify only over objects (first-order) and unary functions from objects to objects (second-order) he does not deny the presence of entitities of higher order. We can give an account of what he did as an effort to collapse discourse about all orders to discourse about two; moreover, he indicates fairly clearly that this was what he was doing. In terms of a scheme with all orders, we can explain Axiom V as a daring mathematical hypothesis which turned out to be false – but which was excessive for his purposes. A more modest hypothesis along the same lines does everything he needs for hsi mathematics, preserving the ability to collapse discourse about all orders into two, and results in our stratified version of his system. Neither his hypothesis nor ours has the necessary character of a logical principle, his being necessarily false and ours merely possible. But ours *is* possible.

We start by describing a system with orders indexed by all natural numbers with the same conceptual resources as Frege's system. There are orders indexed by the natural numbers. Order $n+1$ objects are to be understood as functions from order n objects to order n objects. We provide basic notions of function application and abstraction: $T^{n+1}[U^n]$ represents application of an order $n+$

1 term to an order n term (note the use of brackets: this is a preliminary implementation of application which we improve below). $(\lambda x^n : T^n)$ represents a function of a variable x^n which sends x^n to T^n for each value of x^n. The symbol $=$ refers to equality of order 1 objects. $X^{n+1} = Y^{n+1}$ is defined as holding iff $X^{n+1}(Z^n) = Y^{n+1}(Z^n)$ for each Z^n (we finesse the question of whether extensional identity criteria actually hold for our functions). We provide a primitive operation $\texttt{if}X = Y\texttt{then}T\texttt{else}U$ which yields T if $X = Y$ holds and U otherwise. Note that we can define negation, implication, and quantifiers using this operation mod choice of distinct objects **t** and **f** of each order: choose two distinct objects **t** and **f** of order 1, and use their respective iterated constant functions in higher orders in this role. Finally, we provide $\texttt{the}\ (X, Y)$, which returns a Z with the unique extension such that $X(Z) = Y(Z)$ if there is such a Z and otherwise returns the value **f** appropriate to the order (each **f** being the constant function of the one of the next lower order). We define $F^{n+1}(X^n)$ as $\texttt{the}(\lambda Y : (\exists X' : X = X' \wedge Y = F(X')))$: this modifies the application operation to return the default value if the function F does not respect our extensional notion of equality.

It is important to note that Frege himself uses functions between different orders. This is easily managed: a function F from order n to order 1 (we can stick to this case because all functions Frege uses are of this form) can be coded by the order $n + 1$ function taking each x of order n to $K^{n-1}[F(x)]$, where $K[x] = (\lambda y : \texttt{if}y = y\texttt{then}x\textbf{else}x)$, the constant function of x.

The system described above is adequate for a lot of mathematics, including a good deal of technical set theory.

The claim behind Frege's Axiom V in the form he uses can then be stated economically: he postulates the existence of a γ mapping order 2 to order 1 (which we would encode as an order 3 function sending order 2 objects to constant functions of order 1 objects, but we will write $\gamma(F^2)$ to refer to the intended order 1 image), with the property that if $\gamma(x) = \gamma(y)$, x and y are coextensional (note that Frege supplies no notion of equality except for order 1 objects). The advantage of this is that the entire structure described above collapses to orders 1 and 2. We define $f`x$, application of an order 1 object f to an order 1 object x, as $\texttt{the}(\lambda y : (\exists F^2 : \gamma(F^2) = f \wedge y = F^2(x)))$. Frege's use of this definition is the reason why he cannot eliminate all the orders. We can then use the defined application and use $\gamma(F)$ to represent each function F in our reasoning, with $(\lambda x : f`x)$ as a specific preimage $\gamma^{-1}(f)$ for each f of order 1 which actually is an image under γ. If we have defined γ_n mapping order n into order 1, and γ^{-n} sending each element of order 1 which is an image under γ_n back to a preimage (and each other object to a default), then we can define $\gamma_{n+1}(F^{n+1})$ as the map sending $\gamma_n(X^n)$ to $\gamma_n(F^{n+1}(X^n))$ for each x, that is, $\gamma_{n+1}(F^{n+1})$ is defined as $\gamma(\lambda x : \gamma_n(F^{n+1}(\gamma^{-n}(x))))$. This map is clearly injective up to coextensionality, so $\gamma^{-(n+1)}$ can be defined conveniently using the definite description operator. The point of this scheme is that all need for orders (much) higher than 2 is eliminated. Frege does need to refer to functions of order 3 in some generality to handle second order quantifiers and to a specific order 4 function (the second order universal quantifier itself) but all quantifiers are over orders 1 and

2 (and he really only needs to mention order 2 because of the way he defines application).

However, this lovely hypothesis, far from being a logical truth as he conjectured, is false. One needs only to consider $(\lambda x : \neg x`x)$ to obtain the paradox of Russell.

There things stopped: Frege gave up and abandoned his work.

However, his despair was premature. In fact, there is no need to suppose that γ itself is a function found in order 3 at all. It can be postulated as an external operation taking order 2 objects to order 1 objects, not represented by an actual function of order 3. We can then see that the Russell "function" will not be a function at all. We do require than that some functions defined in terms of the operator γ actually be functions: the exact requirement is that every function F definable in the language of the system with all orders taking a vector or parameters $x_i^{\tau(i)}$ of order $\tau(i)$ to output y of order τ[8] corresponds to a function of order 2 taking the vector whose ith component is $\gamma_{\tau(i)}(x_i^{\tau(i)})$ to $\gamma_\tau(y)$ (this does require of course that the functions γ_n be total, and it requires that the ranges of the maps γ_n be concepts, needed for definition of default behavior for other input vectors). This does not allow definition of the Russell function because the Russell function has no such analogy to a function definable in the system with all orders. This is a function based way of expressing the stratification criterion of New Foundations, of course, and it is implemented above. And, if it works, it allows expression of all the considerable mathematics implementable in the system with all orders in just 2 to 4 orders (depending on what one considers Frege's actual commitments to be).

Does this make sense philosophically? We would say yes, if functions and concepts are understood intensionally: the idea that there might be a purely arbitrary correlation of type 1 objects to type 2 objects that our mathematical understanding underlying the theory with all orders does not allow us to capture does not seem absurd. Does it make sense as a hypothesis? It can be motivated much as Quine motivated NF, by the observation that the system with all orders has the feature that any theorem or defined object in one order is mirrored in all higher orders. Is it a logical truth that there is such a correlation? We would say not. Is it possible? Jensen showed this to be true.

We are here drafting an account of the logic which makes it clear how the non-function embedding from order 2 into order 1 is handled. The sorts we consider are the same ones that Frege uses: order 1 objects, order 2 functions from order 1 to order 1, order 3 functions taking an order 2 argument to an order 1 argument, and order 3 functions taking two order 1 arguments to an order 1 argument. Our primitives are the same as his: the order 1 constants

[8]Note that such a function can be represented in our system of orders in spite of the heterogeneous types of the inputs and outputs by coding: raise all inputs and outputs to the maximum type present by iterating the constant function operation; further, functions of multiple arguments can be represented by a stratified variant of currying as functions of a single argument: $f(x_1, x_2, \ldots, x_{n-1}, x_n)$ can be read not as $f(x_1)(x_2)\ldots(x_{n-1})(x_n)$ but as $f(K^{n-1}[x_1])(K^{n-2}[x_2])\ldots(K^1[x_{n-1}])(x_1)$. Frege of course uses (invented?) currying in his notion of double value range.

t and **f**, the order 1 negation function, the order 3 implication and equality functions, the order 3 first order universal quantifier, and the order 4 second order quantifier (only constants of order 4 are ever considered). We view `the` as order 3, taking $[a = X]$ to a instead of taking $(x \mapsto a = a)$ to a. Functions can be constructed from the primitives using bracketed terms as above.

We then add to our system the external operator γ taking order 2 functions $[F(x)]$ to order 1 objects (intended to be $(x \mapsto F(x))$). Unrestricted use of γ to build functions is not permitted, but certain functions defined in terms of γ do exist. We provide in addition the operator γ^{-1} sending each object $\gamma(F)$ to a function coextensional with F and each other object to the constant function of **f**: the intention is that $\gamma^{-1}(f) = [f`X]$.

We note that it is advantageous to suppose that γ sends $[\mathbf{t} = X]$ to **t** and $[\mathbf{f} = X]$ to **f**. This is not the same as Frege's assignment of extensions to the truth values, but it is handy to do things this way to avoid the need for extra definitions by cases.

We define other operators in terms of γ. To any third order function Z of a second order argument there corresponds a second order function $\gamma_1(Z)$ defined as $[Z(\gamma^{-1}(X))]$, and to any second order function f there corresponds a third order function $\gamma_1^{-1}(f)$ defined as $[f(\gamma(F)]$ (f being the argument). The assertion that these are genuine functions in either case has real content, as they are defined in terms of γ and γ^{-1}. Functions from order 2 functions to objects (if they respect coextensionality) are precisely correlated with functions to objects from the representations of order 2 functions as objects vai γ.

We define an operation allowing an order 1 object to be in effect shifted in level. $\gamma_0(x)$ is defined as $\gamma[x = X]$ (this is the singleton set operation). γ_0^{-1} is exactly Frege's operation `the` (noting that for us `the` is not a function, as it shifts level; we use **f** as the default object which is the value of `the` at non-singletons). We provide that for every order 2 function F, $\gamma_2(F) = [\gamma_0(F(\gamma_0^{-1}(X)))]$ and $\gamma_2^{-1}(F) = [\gamma_0^{-1}(F(\gamma_0(X)))]$ are functions: there is a precise correlation between the action of functions on singletons and the action of functions on general objects. These operations can be iterated, of course.

Other orders of function need associated operators. Let r be an order 3 function of two order 1 arguments. That $\gamma[r(a, X)]$ is equivalent to $r^*(\gamma_0(a))$ for an order 2 function r^* appears to require a postulate: this ensures that double value ranges work as intended (in a stratified version). Since there is no abstraction over binary relations in Frege's development, it seems sufficient to cash out talk of any specific r with talk of r^*, but we could provide that functions on binary functions (if we consider any) have exact correlates acting on the correlated unary functions. It is worth noting that this postulate will be needed in justifying the ability to bind variables into value range terms in general, in a full justification of our typed logic along the lines we are developing here.

This account is still incomplete: I am working on developing an account roughly in Frege's own terms which justifies the typed logic developed above via introduction of an external operation γ from order 2 to order 1 and careful specification of how γ, which is not itself a function, can be used to define

functions. A motivation of the whole system along similar lines already exists via a wholesale translation of everything that is being done into NFU, but this is of course anachronistic.

9 Appendix: old embedded notes

note from Frege text: pp. 5-6 and footnote on page 6. Contra the footnote on p.6, I find it useful to provide notation for functions with one and two arguments, with argument place-holders of the same general kind as ξ, ζ but distinguished by level and order, and I do regard these as an essential part of my implementation of the concept-script. Frege refers to these notations later as names of functions: including them in our notation then seems natural. It is also the case that the bracket notations (I enclose function of one argument in one bracket, functions of two arguments in two brackets) will not appear in theorems: but they are an essental device in definitions, and they are used in specification of function variables in proofs (as Frege actually uses them in his own discussion!)

note from Frege text re rules: Permutability of lines is implemented in the narrow form of bringing a single subcomponent to the front. A small number of such operations will handle any general permutation encountered in practice. A more profound difficulty arises if Frege makes much use of rules for calculation with propositions inside subcomponents: the rules as we have implemented them are formally adequate but we might need to expand our functions to emulate such reasoning, if it occurs. We do provide parameters for user commands to indicate that they act on a particular supercomponent.

note from Frege text re names: Starting on page 43 there is a discussion of proper names for functions of one and two arguments which I believe justifies my inclusion of notations for functions and concepts in my formal language. There are quantifiers over functions, so we ought to be able to write names of particular functions! p. 48 he quite accurately enumerates the orders of function which a faithful formalization of his system requires. It is interesting that the four flavors of function correspond precisely to basic syntactical constructions, first level functions of one argument to unary operators, first level functions of one argument to binary operators, second level functions with one first level function as an argument to first-order quantifiers and analogous variable binding operations, and second level functions with a function from unary functions to arguments to the second-order quantifier and analogous variable binding operations (in the latter two cases, the type described is not the type of the variable binding operator but the type of the function which invisibly serves as its argument). We need free variables and a first argument place term of the last kind, though not bound variables, in order to support statement of the axiom on the second order quantifier and to make definitions of second order variable binding operations possible.

definition commands: It is fun that Gottlob requires only a single definition command to handle definitions of object constants, unary and binary operations on objects, and first and second order variable binders. For all that is required is an equation between a fresh identifier and a term, and the type of the term unequivocally indicates the sort of construction being defined! The elegance of the type system which makes this possible is entirely down to Frege himself.

a block of notes I wrote after I wrote the basic type definition for terms and before I wrote anything else: some of this I changed as I wrote further code:

Please note that this section was written before the code and some of the concepts in it changed at least in detail as the implementation proceeded. In particular, there is no type `cls` in the implementation: this is resolved into finer-grained types of unary and binary relations (the same types assigned to operations).

We summarize the type system. Each term has a sort: object, unary function, or binary function, and a type. Objects are of type `prop`, `nat`, `obj`, or a positive integer. Unary functions are of type `fun`, `cls`, or a positive integer greater than 1. Binary functions are of type `rel`, `cls`, or a positive integer greater than 1. It is worth noting that all functions have object output, and that constant object terms never have positive integer type. All references to type `cls` in this section are replaced in the implementation by references to actual function types with specified input types and an object output type: what distinguishes things typed with `cls` in this discussion is that they are inhomogeneous operations in the stratified context (more than one integer level appears among their input and output types).

note from Frege text re signs for functions: `--` is our eventual notation for Frege's dash function, though we use ! for this purpose in some discussion below. $\sim$ is our notation for negation. Equality is provided with the same notation Frege uses: we note that a very slight modification of his scheme would allow equality to be *defined* rather than primitive. In his exact scheme it cannot be so, because use of definitions relies on equality: but use of definitions could equally well be founded on implication. We denote the universal quantifier as $\{Av : \phi\}$, $\{IF : \phi\}$, I for "impredicative" being our notation for the second order universal quantifier. We use the notation $\{x \mapsto T\}$ for value ranges. Note that we use the same bound variables in universal quantifications (and other object variable binding constructions: we allow such to be defined) that we use in value-range notations: it is straightforward to see that this is harmless, particularly as my presentation of types is based on the analysis of $\{Bx : T\}$ for any binder B as something like $B(\{x \mapsto T\})$, though the argument is not really taken to be a value-range, but a function. We follow Frege in requiring that a bound variable always occur in its scope. For the definite description operator $\backslash$ we use the notation `the`, and we note that our implementation of the semantics using NFU precludes the definition of $\backslash f$ as the unique object x such that $f(x)$ is the True, and otherwise x: this definition is unstratified, and it is a theorem of NFU that there can be no such operation. Fortunately, this stipulation is no part of Frege's axiomatization, and we are free to provide that $\backslash$ is $\{x \mapsto\sim (x = x)\}$ (the empty set) when there is not a unique x such that $f(x)$ is the True. We use the notation $x \to y$ for material implication. We note that the notations $--$ and $\to$ are not handled in our internal representations of terms as standard unary and binary operators: the internal representation is via an operator which acts on a list $[t_1, \ldots, t_n]$ and returns notation for $(t_1 \to (t_2 \to \ldots \to t_n))$, and acts on a list $[t]$ and returns notation for $--t$. Further, an implication may not

occur as the final item in an implication notation (a dash notation may, but is automatically simplified away by the substitution operation). This means that our notation, though it does not look like Frege's concept-script for propositions, does have certain formal features which it has.

old material resumes

The `Proposition` form denotes nested implications: $[p_1, p_2, \ldots, p_n]$ represents $p_1 \to (p_2 \to \ldots \to p_n)$: note the direction in which parentheses group. We are conscious of the abuse of use and mention in our nonce notation here. This notation implements his charming two dimensional notation in a linear form: we suppose that a complete version of Gottlob would be enhanced by pretty printing of the two dimensional notation. The type of such a term is `prop` (see below for discussion of types) and all that is required of the listed items is that they have object types (`prop` (preferred), `nat`, `obj`, or a positive integer). A `Proposition` will not be asserted as a theorem if it contains any free variables (which rules out a positive integer type and some occurrences of type `obj`).

The `Unary`, `Binary` and `Binder` forms represent unary and binary operations and variable binding constructions in the obvious way. The string and integer in the binder form are the innards of the bound variable.

Primitive unary operations include $\sim$ (negation), ! (coercion to a proposition: sends the True to the True and everything else to the False; actually definable as double negation, and reasonably expressed as unit list, not unary at all; this is changed back to `--` in the implementation, similar to Frege's original notation), # (coercion to a natural number: sends each natural number to itself and everything else to 0), $\backslash$ (the definite description operator: sends courses-of-values of concepts under which one object falls to that object, and everything else to the constant course-of-values of the False; this is not Frege's definition, but Frege's is intrinsically unstratified; this is expressed as `the` in the implementation).

Primitive binary operations include $=$ (equality, taking two object arguments and returning a proposition). Implication is not actually expressed using the `Binary` form, though we could allow the parser to read $\to$ as a binary operator and express it as a `Proposition` form.

New unary and binary operations can be defined by binding strings to functions using a user command to be implemented below.

With each unary or binary operator, we need to associate input and output types. The types of objects are `prop`, `nat` and positive integers: there is an additional special type `obj` of objects whose type can be left free. The type of a unary operator is the pair of its input type and its output type. The type of a binary operator is the triple of its first input type, its second input type, and its output type. The meaning of an operator type is not changed by adding the same integer constant to all positive integers present in it, and the conventional form of the type of a unary or binary operator is chosen so that the minimal integer present (if there is one) is 1 (one can subtract as well as add as long as no nonpositive types are created).

The type of $\sim$ and ! is $[\texttt{prop}, \texttt{prop}]$. The type of # is $[\texttt{nat}, \texttt{nat}]$. The type of $\backslash$ is $[2, 1]$. The type of $=$ is $[1, 1, \texttt{prop}]$. The type of $\to$ would be $[\texttt{prop}, \texttt{prop}, \texttt{prop}]$.

In a unary or binary term, the types of the arguments (actual types) must

match the type of the operator (formal types), with the proviso that a match can be effected by adding the same integer constant to all the integer formal types, and that `prop` and `nat` match integer types with no need for adjustment (either as actual or formal types), and `obj` matches any type: an actual argument which matches a formal type of `prop` or `nat` may be supposed decorated with ! or # as appropriate, and this will be supplied if the definition of the operator has to be expanded. We indicate below ways in which `obj` may appear as the formal type of an argument or output. The output type found in the formal type of the operator as adjusted to match the actual types of the inputs is the actual type of the term.

The primitive binders are the empty string (courses of values, expressed $(x_i^j \mapsto T)$ where x is the bound object variable of index i and positive integer type j and T is the body), A and I. I for "impredicative" is our notation for the second order quantifier over functions of one argument. `(Ax:phi)` is our notation for $(\forall x : \phi)$ and `(IF:phi)` is our notation for $(\forall^2 F : \phi)$ (the superscript indicating second order quantification: x and F here will have indices and type superscripts).

A binder is typed as a unary operator, as suggested by rewriting $(Bx : T)$ as $B(x \mapsto T)$: the $(x \mapsto T)$ is a function, not a course of values, as it must be for second-order binders (binding function variables). This has the advantage that a binder type cannot be confused with a binary function type. As with unary and binary operators, all integers in a type may have the same constant added to their absolute values to effect a match, and the actual type of the output is the output type component of the formal type of the operation as adjusted to effect the match. There is a final qualification that the type of a binder term with integer type in which no bound object variable [not immediately decorated with ! or # or other unary operator with non-integer output type] occurs free is taken to be `obj`, a special type (not occurring as a formal type?) which matches positive integer types, `prop`, or `nat` freely. If the bracketed qualification is adopted, some provision should be made for `obj` to occur as a formal as well as an actual type, as it will be a possible output (and, with more effort, input) type of defined operations. [9]

Binders may be defined by binding strings to suitable abstractions. Only the first input type may be of sort other than object: the body type and the output type must be object types. The typing of these abstractions is subtle: a function taking the bound variable type to the body type is mapped to the output type. Underneath, we always read $(Bx : T)$ as $B(x \mapsto T)$. Below we will see that this requires us to take rather elaborate types into account to handle definition of second order quantifiers.

The `Function1` and `Function2` forms represent function constants of one and two arguments in Frege's sense. A `Function1` term with body T is written $[T]$; a `Function2` term with body T is written $[[T]]$. In either case, T must be of object type (so there is no ambiguity in the double bracket notation).

[9] The meaning of type `obj` is "is constant or varies in some unspecified strongly cantorian set": this footnote is for NF-istes.

The type assigned to $[T]$ is `fun`, which matches any unary function type other than `cls`, or `cls` (conditions under which it must be typed as `cls` are described below) as long as T is typable and contains occurrences of no more than one first argument variable not included in a proper nonatomic function subterm of T (there may be more than one occurrence of the same such first argument variable) and no second argument variable not included in a proper nonatomic function subterm of T. The type assigned to $[[T]]$ is `rel`, which matches any binary function type other than `cls`, or `cls` (conditions under which it must be typed as `cls` are described below) as long as T is typable and contains occurrences of no more than one first argument variable not included in a proper nonatomic function subterm of T (there may be more than one occurrence of the same such first argument variable) and contains occurrences of no more than one second argument variable not included in a proper nonatomic function subterm of T (there may be more than one occurrence of the same such second argument variable).

We assign type `cls` to `Function1` or `Function2` terms with non-object arguments: we do maintain the requirement that they have one first argument and one second argument (if appropriate) in the exact sense described above, and that the body of the term has object type.

`App1` and `App2` implement application of functions of one and two arguments, respectively.

In a `App1` or `App2` term in which the function part is a `Function1` or `Function2` term, the type of the first argument must match the type of the first argument variable appearing in the function part, unless the first argument variable is absent or immediately decorated with a unary operation with non-integer output type, and the type of the second argument must match the type of the second argument variable, unless the second argument variable is absent or immediately decorated with a unary operation with non-integer output type. The match may be effected by adding the same constant (positive or negative) to the absolute values of the types of any argument variables with integer types. In a `App1` or `App2` term in which the function part is atomic (and so has integer type), the successors of of any integer types of arguments and the type of the function part must all be the same. Note that this means that `Function1` or `Function2` terms with object arguments in which the integer type of the body does not match the integer type of an argument (not decorated immediately with unary operations with non-integer output type) or the integer types of the arguments are different (and not decorated immediately with unary operations with non-integer output type) cannot instantiate function variables: such terms are assigned type `cls`. Type `cls` does not match any other type (in particular, it does not match integer unary or binary function types).

The `Freevar` and `Boundvar` forms represent free and bound variables respectively. Free variables have a string body and an index (a positive integer; index 1 is not expressed). Bound variables have a string body, an index, and a type (a positive integer: one is not expressed). The type is a new feature not found in Frege's system, of course. A free variable a_i, b_i, c_i may be assigned a type `obj` to be construed as an indefinite integer: for purposes of type matching,

`obj` can match any integer, `prop` or `nat`. A bound variable x_i^j, y_i^j, z_i^j is assigned the object type j. A variable f, g, h is a free function variable of one argument and may be taken as having type `fun`, which will match any unary function type other than `cls`. A variable F_i^j, G_i^j, H_i^j is a bound unary function variable of unary function type j. A variable r_i, s_i, t_i is a free function variable of two arguments and may be taken as having type `rel`, matching any binary relation type other than `cls`. A variable R_i^j, S_i^j, T_i^j is a bound binary function variable of binary function type j (not used by Frege, but we provide it for completeness).

The `Firstarg` and `Secondarg` forms represent arguments of functions, in Frege's sense (analogous to his ξ, ζ, but we implement this in more generality). The string and the integer are both indications of type. X^i is a first object argument of type i. Y^i is a second object argument of type i. P, Q (the superscript 1 not expressed) are first and second proposition arguments (convenient abbreviations for $!X, !Y$, assigned type `prop`). M, N (superscript 1 not expressed) are first and second natural number arguments (convenient abbreviations for $\#X, \#Y$, assigned type `nat`). F^i, G^i are first and second unary function arguments (with one argument themselves) of type i. R^i, S^i are first and second binary function arguments (with two arguments themselves) of type i, an option which we provide because Frege mentions it, though we do not construct such things ourselves. A non-subscripted X or Y (hiding a one) of one of these shapes, occurring always with the same unary operation of a non-integer output type (including `obj`!) may be assigned type `obj` (the idea here is that the inputs associated with the same outputs in the s.c. range of the operator are in effect identified).

See the last paragraph for additional first argument terms Z^i, W^i of more involved types.

The only function terms are free and bound function variables and the `Function1` and `Function2` terms (which are expressed by enclosing their bodies in single or double brackets). Since no defined unary or binary operation can have a function as an argument or value, nor can any binder have a function as body or value (the only way a function appears as a subterm of a complex term is as a free or bound function variable), it follows that bracketed terms can *only* appear as input to definition commands or in the function role in an `App1` or `App2` term[10]which seems to agree with Frege's uses of functions. Our intention is that `App1` and `App2` terms will be automatically eliminated by the substitution operation. The existential quantifier, for example, appears as $[\sim (\forall x \sim F^2(x))]$: the abstraction is a function of a unary function variable with type one higher than the bound variable type with output the body type. We have to do a little extra work (as Frege did) to be able to define the second order existential quantifier (or any other second order quantifier). We actually need first argument terms (we supply the shapes Z^{i+2}, W^{i+2} of type $i+2$) which take a unary (resp. binary) function input of type $i+1$ and return an object output of type i. So the second order existential quantifier $\exists_2$ is bound to the third level function

[10]or as an argument to a function which takes function arguments, but in this case it must be of type `fun` and so actually stratified

$[\sim (\forall F_0^2 :\sim Z^3(F_0^2))]$ (the subscript on F is expressed to indicate that it is not a first argument term; this is handled, but differently, in the Gottlob syntax). We do not need second argument terms of these involved types as we are not committed to quantifying over vectors of arguments. It is amusing to observe that these function terms are actually typable.

The semantics is simpler than it appears, because ultimately the second order objects (functions) are actually *identified* with their corresponding courses-of-values (when they have them) which causes our formally second-order (or even third-order) theory to collapse to a first-order theory. The same would have been true of Frege's original system, were it of any actual interest.

In the implementation we follow Frege in not allowing binding of binary function variables. But we do support bracket terms which can take binary functions as arguments, so that we can define operations taking binary functions to objects.

Here is the display function. At least initially, we are giving all operators the same precedence and having them group to the right. No more parentheses are displayed then are needed. `Prop` terms are displayed as implication terms with the expected binary notation. Single term `Prop`'s are adorned with $-$. Note that first and second arguments are always adorned with their types, separated by a caret. Thus they cannot be confused with free variables or bound variables with only one index. A bound variable is followed first by its type with no separator, then by its index with an underscore as separator; neither is expressed if it is 1 and if only one is expressed the presence or absence of the separator indicates which one it is.

On the proposition and natural number argument terms, types are suppressed by the display, because they are meaningless. On F and G the types must be displayed even if the minimal value 2 is taken, to maintain distinction from the bound variables F and G. On X and Y we retain display of the minimal type just because it is easier.

Unary operations bind slightly more tightly than binary operations.

see p. 64: of Frege text for his discussion of these issues. We see no reason to emulate his usage, but we notice that it is similar in spirit in giving all operations the same precedence, but he groups to the left not the right. Our usage is roughly that of the ancient computer language APL.