Refuting Tarski and Gödel with a Sound Deductive Formalism

The conventional notion of a formal system is adapted to conform to the sound deductive inference model operating on finite strings. Finite strings stipulated to have the semantic value of Boolean true provide the sound deductive premises. Truth preserving finite string transformation rules provide the valid deductive inference. Sound deductive conclusions are the result of these finite string transformation rules.

Axioms, rules-of-inference, syntax, and truth conditional semantics are all fully integrated together into the single formalism of finite string transformation rules.

The {domain of discourse} of the **Sound Deductive Formalism** (SDF) is the body of **Analytical_Knowledge** defined as follows: The set of knowledge that can be expressed using language and verified as true entirely on the basis of stipulated relations between expressions of language.

Validity and Soundness https://www.iep.utm.edu/val-snd/

A deductive argument is said to be *valid* if and only if it takes a form that makes it impossible for the premises to be true and the conclusion nevertheless to be false. Otherwise, a deductive argument is said to be *invalid*.

A deductive argument is *sound* if and only if it is both valid, and all of its premises are *actually true*. Otherwise, a deductive argument is *unsound*.

It seems self-evident that any formal system conforming to the above Sound Deductive Inference Model (SDIM) that applies truth preserving finite string transformation rules to a set of finite strings that are stipulated to have the semantic value of Boolean true would have a universal Truth(X) predicate on the basis of its universal Provable(X) predicate thus refuting both Tarski and Gödel for the domain of discourse of Analytical_Knowledge.

To provide a simple intuitive grasp of the **Sound Deductive Formalism** (SDF) we define a very simple formal system named **Simple_Arithmetic**.

All that **Simple_Arithmetic** does is evaluate relational_expressions comprised of a pair of arithmetic_expressions. These expressions have the exact same syntax as the "C" programming language. The arithmetic_expressions are limited to the operations of addition and multiplication of unsigned integer literals comprised entirely of the ASCII digits [0-9].

The only divergence from the "C" standard is that these unsigned integer literals are of arbitrary length and all arithmetic operations are performed directly on these strings of ASCII digits. This formal system would have a single Boolean Evaluate() function.

Evaluate("((((2 + 3) * 7) + 9) == 44)") evaluates to true which indicates that a set of finite string transformation rules derives: "44" from: "(((2 + 3) * 7) + 9)" thus satisfying: "==".

$\forall F \in \text{Sound\_Deductive\_Formalism}\ \forall X \in \text{WFF}(F)\ (\text{True}(F, X)) \leftrightarrow \text{Provable}(F, X))$

```
/**
  Truth-conditional semantics is an approach to semantics of natural
  language that sees meaning (or at least the meaning of assertions)
  as being the same as, or reducible to, their truth conditions.
  https://en.wikipedia.org/wiki/Truth-conditional_semantics

  This program demonstrates [truth conditional semantics] for a very
  tiny subset of analytic knowledge. It can only evaluate finite strings
  matching this AWK regular expression: /[0-4]\+[0-4]=[0-8]/

  The finite string transformation rules specified by this source-code
  provide the means to formally prove whether a finite string of the
  language of the Tiny_Arithmetic formal system has the semantic property
  of Boolean true.
**/

#include <cstdio>
#include <string>

char SumDigits[58][58];

void InitSumDigits()
{
  for (char row = '0'; row <= '4'; row++)
    for (char col = '0'; col <= '4'; col++)
    {
      SumDigits[row][col] = ((row-'0')+(col-'0'))+'0';
//    printf("%c + %c = %c\n", row, col, SumDigits[row][col]);
      printf("%c+%c=%c\n", row, col, SumDigits[row][col]);
    }
}


bool ValidateInput(const char* Input)
{
int ErrorCode = 0;
  if (strlen(Input) != 5)
    ErrorCode = 1;

  if (Input[0] < '0' || Input[0] > '4')
    ErrorCode = 2;

  if (Input[1] != '+')
    ErrorCode = 3;

  if (Input[2] < '0' || Input[2] > '4')
    ErrorCode = 4;

  if (Input[3] != '=')
    ErrorCode = 5;

  if (Input[4] < '0' || Input[4] > '8')
    ErrorCode = 6;
  if (ErrorCode)
    printf("ValidateInput(%s) ErrorCode(%d)\n", Input, ErrorCode);
  return(ErrorCode == 0);
}
```

```
bool ProveInput(const char* Input)
{
  char d1  = Input[0];
  char d2  = Input[2];
  char sum = Input[4];
  if (SumDigits[d1][d2] == sum)
    return true;
  else return false;
}


bool RegressionTest()
{
int ErrorCount = 0;
char Input[6];
  Input[1] = '+';
  Input[3] = '=';
  Input[5] = 0;
  for (char row = '0'; row <= '4'; row++)
    for (char col = '0'; col <= '4'; col++)
    {
      Input[0] = row;
      Input[2] = col;
      Input[4] = SumDigits[row][col];
      if (!ProveInput(Input))
        ErrorCount++;
    }
  if (ErrorCount != 0)
    printf("RegressionTest() ErrorCount(%d)\n", ErrorCount);
  return(ErrorCount == 0);
}


// main() returns void to reduce clutter
void main(int argc, char *argv[])
{
  InitSumDigits();
  RegressionTest();
  if (argc != 2)
  {
    printf("Must entire an equality expression of the form:\n");
    printf("Digit[0-4] \"+\" Digit[0-4] \"+\" Digit[0-8]\n");
    return;
  }
  ValidateInput((const char*)argv[1]);
  if (ProveInput((const char*)argv[1]))
    printf("Input string is proven and true!\n");
  else
    printf("Input string is unproven and untrue!\n");
}
```