High-performance sheet-defined functions in spreadsheets

Peter Sestoft
IT University of Copenhagen

Microsoft Development Center 2009-02-24

With thanks to Thomas S Iversen, Daniel Cortes, Morten Hansen, Poul Serek, Morten Poulsen, Hui Xu, Mainul Liton

The speaker

• MSc 1988 computer science and mathematics and PhD 1991, Copenhagen University
• KU, DTU, KVL and ITU; and AT&T Bell Labs, Microsoft Research UK
• Interests: Programming languages, program generation, software development
• Open source software
  – Moscow ML implementation, 1994...
  – C5 Generic Collection Library, with Niels Kokholm, 2006...
Why spreadsheet technology

- Spreadsheets are used by 100 million people
- Often complex models: biology, physics, games, economy, finance, ...
- A tool for *end-user software development*
- *Better tools are desirable*

- Technological opportunities
  - Spreadsheets are functional
  - Bytecode generation in .NET
- CoreCalc implementation
- Several MSc thesis projects

The trouble with functions

- Many (Excel) built-in functions are bad:
  - Week numbers: two kinds, but not ISO standard
- To define new functions, “experts” use VBA
- Often *very poorly*, witness newsgroup microsoft.public.excel.programming
- Possible answers to this mess:
  - “People should not use spreadsheets”
  - “Only computer scientists should define functions”
  - “All necessary functions should be built in”
  - **Or:** Define functions within the spreadsheet metaphor, without VBA
Example: TRIAREA

- Area of triangle with sides a, b, c is \(\sqrt{s(s-a)(s-b)(s-c)}\) where \(s = \frac{a+b+c}{2}\)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Area of a triangle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a =</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>b =</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>c =</td>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>18</td>
<td>s =</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>area =</td>
<td>800</td>
<td></td>
<td>TRIAREA</td>
</tr>
</tbody>
</table>

Hypotheses

- Sheet-defined functions are **useful**
  - Users understand them better than VBA
  - Users make fewer mistakes defining them
  - Because they can be developed as spreadsheet formulas

- Sheet-defined functions can be **fast**
  - As fast as Excel built-in functions
  - Much faster than VBA functions
  - Because of runtime code generation

- Function libraries will be developed and shared

- Hoped-for impact and dissemination
  - Can this be used from MS Excel via VS2010/Office bridge?
  - Or could OpenOffice or Gnumeric adopt it?
  - Or inspire somebody else with an open source component?
What’s new

- New concepts
  - Sheet-defined function
  - A value may be a function (closure), in addition to number, text, array, and error values

- New built-in spreadsheet functions
  - MAKEFUN(“triarea”, B6, B2, B3, B4)
  - GETFUN(“bulletPV”, 4)
  - APPLY($A$1, 4.5)
  - EXTERN(“System.Math.Sinh$(D)D”, 2.3)
  - BENCHMARK($A$1, 1000000)

Runtime code generation

My compiler

.NET JIT compiler

Result: A very fast, portable spreadsheet implementation
Challenge: Avoid value wrapping

- Spreadsheet formulas are dynamically typed: \( \text{IF}(A1, 11, \text{"twelve"}) \) is either number or text
- Must wrap values to distinguish types

Problem: computing \( a * b + c \) might require
  - test and unwrap \( a \)
  - test and unwrap \( b \)
  - compute \( t1 = a*b \)
  - wrap \( t1 \)
  - test and unwrap \( t1 \)
  - test and unwrap \( c \)
  - compute \( t2 = t1 + c \)
  - wrap \( t2 \) as result

Solution part 1: distinguish computation contexts

- Compile()
  - Compile for context that needs a wrapped Value
  - Code leaves a wrapped Value on the stack top
- CompileToDoubleOrNan()
  - Compile for context that needs a “naked” double
  - If result is a number, leave it on stack top
  - If result is an error:
    - Leave a NaN: not-a-number as per IEEE 754 standard
    - Encode information about the error inside the NaN
  - The code performs no wrapping/unwrapping inside arithmetic expressions
IEEE NaNs to represent errors

- Floating point bits: 1 sign + 11 exp. + 52 significand
  
  Sign  Exponent  Significand

- The bits of a NaN (not-a-number):
  
  Fixed  Variable: 51 bits for error codes

- Arithmetic operators propagate NaNs and preserve error bits
- If \( \text{LOG} \("\text{foo}\") \) gives an ArgTypeError, then so does \( \text{Math.Sqrt}(2 + 7 \times \text{LOG} \("\text{foo}\") \)
- Alternative: throw exception, but very slow on .NET

Global error message table

- We use 32 error bits as index into error table
- ErrorValue objects are allocated dynamically

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;#ERR: NumError&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;#ERR: ArgCount&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;#ERR: ArgType&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;#ERR: NoSuchFunction&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;#ERR: Ref&quot;</td>
</tr>
<tr>
<td>5</td>
<td>&quot;#ERR: Name&quot;</td>
</tr>
<tr>
<td>6</td>
<td>&quot;#ERR: MyCustomError&quot;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- Example: the error "ArgType" as a double:

  Error code=2

  Sign  Exponent  Significand

  0 1111111111 10000000000000000000000000000000000000000000000000000000000000010
Solution part 2: Type analysis
Avoid inter-cell wrapping of doubles

- Type analysis of the function definition
- Compute the “natural type” of each cell

Example: NORMDISTCDF

- Normal distribution N(0,1) cumulative distribution function
- More accurate than Excel’s built-in NORMDIST(z,0,1,1)
• 250 ns/call on 1.6 GHz Pentium M
• C#: 153 ns/call; C: 204 ns/call; VBA: 3400 ns/call

**Challenge: Compiling comparisons A1<0**

Excel uses a three-valued logic:
IF(B8>37, 11, 22) is ErrorValue if B8 is ErrorValue or infinite or NaN or wrong type

Solution: CompileToDoubleProper(ifProper,ifOther)
- Compile to context that needs finite non-NaN
- ifProper: generates code for the “good” case: when result is a finite non-NaN double
- ifOther: generates code for the “error” case: operand is error, or infinite, or NaN, or ...

Effect: Comparison e0<e1 compiles operands in a context where the error case is handled explicitly

```csharp
es[0].CompileToDoubleProper(ilg, new Generate(delegate { return es[1].CompileToDoubleProper(ilg, new Generate(delegate { ilg.Emit(OpCodes.Clt); ifProper.Gen(ilg); return ilg.Emit(OpCodes.Pop); ifOther.Gen(ilg); return ilg; })});
```

Possibly multiple uses
Avoiding bytecode duplication

- Problem: A code generator may be used 0, 1 or many times, possibly causing code duplication
- Solution: Label the code, and share by jumps

```csharp
public class Generate {
    private readonly Act generate;
    private Label? label; // Invariant: generated implies label.HasValue
    private bool generated;

    public Label GetLabel(ILGenerator ilg) {
        if (!label.HasValue)
            label = ilg.DefineLabel();
        return label.Value;
    }

    public void Gen(ILGenerator ilg) {
        if (generated)
            ilg.Emit(OpCodes.Br, GetLabel(ilg));
        else {
            ilg.MarkLabel(GetLabel(ilg));
            generated = true;
            generate();
        }
    }
}
```

Using Generate functionality

- Default implementation of CompileCondition(ifTrue, ifFalse, ifOther):

```csharp
CompileToDoubleProper(ilg, new Generate(delegate {
    ilg.Emit(OpCodes.Ldc_R8, 0.0);
    ilg.Emit(OpCodes.Beq, ifFalse.GetLabel(ilg));
    ifTrue.Gen(ilg);
    if (!ifFalse.Generated) {
        Label endLabel = ilg.DefineLabel();
        ilg.Emit(OpCodes.Br, endLabel);
        ifFalse.Gen(ilg);
        ilg.MarkLabel(endLabel);
    }
}), ifOther);
```

- This slightly complicated logic is needed only four places in the compiler
Challenge: Logical expressions, value or decision?

- In the context $10 + \text{AND}(x, y)$, the logical expression evaluates to 0 or 1
- In the context $\text{IF}($AND$(x, y)$, 11, 22), the logical expression should compile to a jump
- Answer: CompileCondition(ifTrue, ifFalse, ifOther)
  - Compile to a context that needs a decision
  - ifTrue.Gen(): generate code for the true case
  - ifFalse.Gen(): generate code for the false case
  - ifOther.Gen(): generate code for the error case
- Compiling $\text{IF}(e0,e1,e2)$:

```csharp
public override void CompileToDoubleOrNan(ILGenerator ilg) {
    es[0].CompileCondition(ilg,
        new Generate(delegate { es[1].CompileToDoubleOrNan(ilg); }),
        new Generate(delegate { es[2].CompileToDoubleOrNan(ilg); }),
        new Generate(delegate { ilg.Emit(Opcodes.Ldloc, testDouble); }));
}
```

Example: Leap-year test as function $\text{OR}($AND$(\text{NOT}(\text{MOD}(B27, 4)), \text{MOD}(B27, 100)), \text{NOT}(\text{MOD}(B27, 400)))$
One pass optimizations

- Compiling a number constant as double:

  ```csharp
  public override void CompileToDoubleOrNan() {
      ilg.Emit(OpCodes.Ldc_R8, number.value);
  }
  ``

- Compiling number const to proper double:

  ```csharp
  public override void CompileToDoubleProper(Generate ifProper, Generate ifOther) {
      if (double.IsInfinity(number.value) || double.IsNaN(number.value)) {
          ilg.Emit(OpCodes.Ldc_R8, number.value);
          ilg.Emit(OpCodes.Stloc, testDouble);
          ifOther.Gen(ilg);
      } else {
          ilg.Emit(OpCodes.Ldc_R8, number.value);
          ifProper.Gen(ilg);
      }
  }
  ``

- Compiling NOT(e₀) as condition:

  ```csharp
  void CompileCondition(Generate ifTrue, Generate ifFalse, Generate ifOther) {
      es[0].CompileCondition(ilg, ifFalse, ifTrue, ifOther);
  }
  ``

Comparison: Excel’s Data > Table

- One- or two-argument formula-based “functions”
- Example: B5 = 100*(1+B3)^B4

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>Illustration of data tables in Excel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f(x) = (1 + 0.5 * x)²</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>FY = 156.2969</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>156.2969</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The computed data table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>7</td>
<td>156.2969</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>1%</td>
<td>101.01</td>
<td>102.43</td>
</tr>
<tr>
<td>9</td>
<td>2%</td>
<td>103.05</td>
<td>104.91</td>
</tr>
<tr>
<td>10</td>
<td>3%</td>
<td>105.52</td>
<td>107.46</td>
</tr>
<tr>
<td>11</td>
<td>4%</td>
<td>107.45</td>
<td>110.07</td>
</tr>
</tbody>
</table>

- Excel formulae in B8:D11 display as `{=TABLE(B3,B4)}`
Problems with Excel Data Tables

- Cannot “call” from inside an expression
- A data table cannot depend on its own output
  - Results are silently wrong
  - Recalculating (F9) doesn’t help
- One data table’s input can depend on another data table’s output ...
  - ... but only to around eight levels
    - Then it silently stops recalculating correctly
    - Recalculating (F9) vigorously does help
- In summary, a half-baked mechanism

Comparison: VBA functions

- VBA code is 10-15 times slower than FunCalc
- A call from Excel to VBA is very slow

```vba
Public Function square(ByVal x As Double) As Double
    Let square = x * x
End Function
```

- Evaluate the formula: =SQUARE(RAND())
- Timing via full sheet recalculation

<table>
<thead>
<tr>
<th>Call Type</th>
<th>Time (ns/call)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excel-to-VBA call</td>
<td>190,000</td>
</tr>
<tr>
<td>FunCalc interpreted EXTERN</td>
<td>3,900</td>
</tr>
<tr>
<td>FunCalc compiled EXTERN</td>
<td>570</td>
</tr>
<tr>
<td>FunCalc sheet-defined fcn SQUARE</td>
<td>400</td>
</tr>
</tbody>
</table>


333 x faster
Calendrical functions

- Excel’s calendar functions are poor
  - Wrong before 1900, no ISO week numbers,
    cannot easily find first Monday of month, no holidays
- Easy to implement as sheet-defined functions
- Example: Easter in a given year:

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td><code>easter(yyyy)</code></td>
<td>Input: year</td>
</tr>
<tr>
<td>23</td>
<td><code>yyyy = 2000</code></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td><code>century = FLOOR(B122/100,1)*1</code></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td><code>unified-quotient = MOD(14+11*MOD(B122,19)/FLOOR(3*B124/4,1)+FLOOR(5+B124/7,1),30)</code></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td><code>adjusted-quotient = IF((OR(B126=0) AND(B124=1,19 &lt; MOD(B122,19)),B124+1,B124)</code></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td><code>moon-solar = FLOOR(B122,19)/19</code></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td><code>easter-fixed = FLOOR(365/4)-28*(1+MID(LEFT(22,2),1,2)+B125)</code></td>
<td>Output: fixdate of Easter</td>
</tr>
</tbody>
</table>

- Some other functions:
  - Fixdate to/from day-month-year
  - Fixdate to/from ISO week and ISO year
  - Last Monday (etc) before given date
  - First Monday (etc) after given date
  - Nth Monday (etc) before given date
  - Nth Monday (etc) after given date

By MSc students Xu and Liton, following Dershowitz & Reingold
(3rd ed, Cambridge U)

Compiling AND(e₁,...,eₙ) for any n>=0

- CompileCondition(ilg, ifTrue, ifFalse, ifOther)
  can be implemented like this:

```csharp
for (int i = es.Length - 1; i >= 0; i--)
    ifTrue = new Generate(delegate { es[i].CompileCondition(ilg,
        ifTrue, ifFalse, ifOther); });
ifTrue.Gen(ilg);
```

- Idea: eᵢ is a evaluated
  - with the AND-expression’s false- and other-continuations, and
  - with a true-continuation that evaluates eᵢ₊₁ with a true-continuation that ... with AND’s true-cont.
- Generates beautiful and fast code
Example: Fast binomial coefficients

- \( \text{BINOM}(a,b) = \frac{(a+b)!}{(a!)!(b!)!} \)
- Exists only in Analysis Toolpak in Excel
- Define as sheet-defined function like this:
  - Tabulate \( \log(n!) \) in ordinary sheet and compute as \( \exp(\log((a+b)!)-\log(a!)-\log(b!)) \)
  - Built-in \text{INDEX}(B3:B54,a,1) indexes into that table

- Speed: 600 ns per call to \text{BINOM}(a,b), despite argument wrapping and unwrapping

Sheet-defined functions as values

- New built-ins to manipulate functions
  - \text{GETFUN}("name", a1, ...) evaluates to a function value, a partially applied sheet-defined function
  - \text{APPLY}(f, b1, ...) applies a function value

- Example function “ndie”, a general n-side die

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>General n-side die</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>n =</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>eyes = =FLOOR(RAND()*B31)+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Defining & rolling 6-sided and 20-sided dice

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>=GETFUN(&quot;ndie&quot;,6)</td>
<td>=GETFUN(&quot;ndie&quot;,20)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>=APPLY(A624)</td>
<td>=APPLY(B624)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>=APPLY(A624)</td>
<td>=APPLY(B624)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>=APPLY(A624)</td>
<td>=APPLY(B624)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>=APPLY(A624)</td>
<td>=APPLY(B624)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>=APPLY(A624)</td>
<td>=APPLY(B624)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aside: partial evaluation?

- Specialization of sheet-defined functions by static evaluation of IF’s should be fairly easy
- Should also be possible to give good estimates of possible time savings
- For recursive functions, additional challenges

Example: Numerical equation solving

- GOALSEEK(f,r,a) finds x that solves f(x)=r, starting with guess x=a
- Exists in a dialog-based form in Excel
- Cumbersome to use for multiple rows

- Higher-order: function f as argument
- Implementable as a sheet-defined function
GOALSEEK as sheet-defined function

- “Bounded iteration” by careful use of IF
- 30 rows give better precision than Excel

<table>
<thead>
<tr>
<th></th>
<th>a0</th>
<th>b0</th>
<th>f(a0)</th>
<th>f(b0)-target</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>f(a0)</td>
<td>f(b0)-target</td>
</tr>
<tr>
<td>5</td>
<td>=IF(b2=0)</td>
<td>=IF(AND(b2&lt;0.8, b2&gt;0.6))</td>
<td>=f(b2)</td>
<td>=apply(b2, b2)-0.8</td>
</tr>
<tr>
<td>6</td>
<td>Invariant: f(a0)&lt;=target&lt;=f(b0)</td>
<td>ai</td>
<td>bi</td>
<td>x=(a+b)/2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>=IF(b1&lt;0.8, 0.0, 0.5)</td>
<td>=IF(b1&lt;0.6, 0.0, 0.5)</td>
<td>=b1+c/2</td>
<td>=apply(b1, 0.0)-0.8</td>
</tr>
<tr>
<td>9</td>
<td>=IF(b3&lt;0.1, 0.0, 0.1)</td>
<td>=IF(b3&lt;0.0, 0.0, 0.1)</td>
<td>=b3+c/2</td>
<td>=apply(b3, 0.0)-0.1</td>
</tr>
<tr>
<td>10</td>
<td>=IF(b3&lt;0.0, 0.0, 0.0)</td>
<td>=IF(b3&lt;0.0, 0.0, 0.0)</td>
<td>=b3+c/2</td>
<td>=apply(b3, 0.0)-0.0</td>
</tr>
<tr>
<td>11</td>
<td>=IF(b1&lt;0.1, 0.1, 0.1)</td>
<td>=IF(b1&lt;0.1, 0.1, 0.1)</td>
<td>=b1+c/2</td>
<td>=apply(b1, 0.1)-0.1</td>
</tr>
<tr>
<td>12</td>
<td>=IF(b1&lt;0.1, 0.1, 0.1)</td>
<td>=IF(b1&lt;0.1, 0.1, 0.1)</td>
<td>=b1+c/2</td>
<td>=apply(b1, 0.1)-0.1</td>
</tr>
<tr>
<td>13</td>
<td>=IF(b1&lt;0.1, 0.1, 0.1)</td>
<td>=IF(b1&lt;0.1, 0.1, 0.1)</td>
<td>=b1+c/2</td>
<td>=apply(b1, 0.1)-0.1</td>
</tr>
</tbody>
</table>

Example use: Effective interest rate of a 10 year bullet loan at price 90 and coupon 4%
- Can be solved only by numeric approximation
- Computes a solution in 26,500 ns

Example: Adaptive integration

- INTEGRATE(f,a,b) = the integral of f on [a,b]
- Compute Simpson and midpoint formula
- If they differ, integrate on [a,m] and [m,b]
- Naturally recursive and higher-order

![Diagram of adaptive integration](image)
INTEGRATE sheet-defined function

- \( \text{INTEGRATE}(f,a,b) = \text{the integral of } f \text{ on } [a,b] \)

- Accurate and quite fast
- Impossible to define in Excel or VBA

```
=INTEGRATE(D9^2,0,1)/2
=apply(S9:M9,
   =INTEGRATE(D10^2,0,1)+
   =INTEGRATE(D11^2,0,1))
=apply(S9:M9,
   =INTEGRATE(D10^2,0,1)+
   =INTEGRATE(D11^2,0,1))
```

Another iteration-free function

- \( \text{REPT}(s,n) = \text{string consisting of } n \text{ copies of } s \)
- Efficient implementation for \( n < 1024 \):

```
=REPT(F9,F49)
```

- Perhaps more clever than transparent...
Defining REPT recursively

- A more general implementation of REPT
- Uses an accumulating parameter, and

\[ s^0 = "" \]
\[ s^{2n+1} = s \& s^{2n} \]
\[ s^{2n} = (s\&s)^n \]

Too fragile, currently (Mark III)

- Important that the recursive call is under IF
- An attempt to get rid of the accumulating parameter loops infinitely:

Here using that \( s^{2n+1} = s \& (s^n)^2 \) and \( s^{2n} = (s^n)^2 \)
Reuse libraries: call external code

- Need a mechanism to call external methods
  
  \texttt{EXTERN("System.String.Format\$(\text{T})\text{O}\text{T}", \text{"x=\{0\:F6\}\text{\", RAND()}\})}

- Also instance methods:
  
  \texttt{EXTERN("System.String.StartsWith\$(\text{T})\text{Z}\", \text{B30, B31})}

- Runtime cost of an external call:
  - Moderate in ordinary sheet formulas: 3,200 ns
  - Very low in sheet-defined functions: 185 ns
    By eliminating wrapping of numbers and strings

- EXTERN calls are as dangerous as macros:
  
  \texttt{EXTERN("System.IO.File.Delete\$(\text{T})\text{V}\", \"thesis-final.tex\")}

Better support for recursion: Conditional evaluation of cells

- Example

  \begin{tabular}{|c|c|}
    \hline
    A & B \\
    \hline
    1 & 17 \\
    2 & =B1+2 \\
    3 & =B1*3 \\
    4 & =IF(B2>0, B3, 42) \\
    \hline
  \end{tabular}

  \begin{itemize}
    \item Build \textit{conditional} dependency graph:
  \end{itemize}
Hypothetical generated code

```
B1 = <input>
B2 = B1+2
cond4 = B2>0
if (cond4)
  B3 = B1*3
B4 = cond4 ? B3 : 42
```

- Thanks to absence of side effects, evaluation can be reordered
- A condition must be evaluated at most once
  - Because it may involve volatile functions, such as RAND() or NOW()

Augmented dependency graph

- Let cells(e) = cells directly referred from e
- For IF(e1, e2, e3), add edge c2 --> c1 for each c1 in cells(e1) and c2 in cells(e2), unless there is a path c1 \rightarrow* c2 already
- In the latter case, c2 must have been evaluated before the IF anyway
- No new dependency cycles are introduced
- Hence can generate code in augmented dependency order
Using the augmented dependency graph

• For a given function sheet cell c
  – Let P be the set of labelled dependency paths from the output to c
  – For each path p ∈ P, let b_p be the conjunction of labels/conditions along p
  – Let b = OR { b_p | p ∈ P }

• Then cell c must be evaluated under the condition b

• Clearly, logical simplifications and identities will be useful...

Function to benchmark functions

=BENCHMARK(GETFUN("NORMDISTCDF",-3), 10000000)

• Arguments: function to call; number of calls
• Result: Average number of ns per call

• Measures function body’s execution time, not the spreadsheet recalculation overhead
• What we need for improving code generation
Stateful function sheets?

- A random number generator needs state
- Idea: persistent cell \( =\text{DELAY}(\text{init}, \text{next}) \)
  - The persistent cell exhibits its previous value
  - Each evaluation computes its next value
  - A dependency cycle must involve a persistent cell

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>=\text{DELAY}(117, \text{MOD}(A1*16807, 2147483647))</td>
</tr>
<tr>
<td>2</td>
<td>=A1/2147483647</td>
</tr>
</tbody>
</table>

Spreadsheet parallelism

- Soon PCs will have 8, 16, 32, ... CPUs
- General programming languages have
  - Shared mutable data
  - Complex and implicit data flow and control flow
  - So, difficult to parallelize for multiple CPUs
- Spreadsheets have
  - Functional semantics, no shared mutable data
  - Near-explicit data flow and control flow
  - So, easy to parallelize for multiple CPUs
  - (Lazy languages are not, complex control flow)

- Old idea by Mani Chandy: *Concurrent programming for the masses*, PODC 1984.
Future work

- Better compilation of IF-expressions
  - Makes recursive programming easier
  - Makes some non-recursive functions faster too
- Multiple return values from sheet-defined functions
- Better types for sheet-defined functions
  - To avoid the current argument un/wrapping
  - Makes GOALSEEK and INTEGRATE faster
- Better types to distinguish proper/improper doubles
- Matrix values and types
- Further investigation of sheet-defined functions as programming model
- Exploit multicore CPUs; declarative model and explicit dependencies help scheduling

Related work

- Nuñez, MSc thesis 2000
  - Scheme-based spreadsheet implementation including sheet defined functions
- Peyton-Jones, Blackwell, Burnett ICFP 2003
  - Usability of sheet-defined functions; recursion and higher-order functions explicit excluded
- Resolver One, commercial spreadsheet
  - Python-based