Partial evaluation of spreadsheet functions

Preliminary results

Peter Sestoft

META 2010, Pereslavl-Zalessky
Friday 2010-07-02

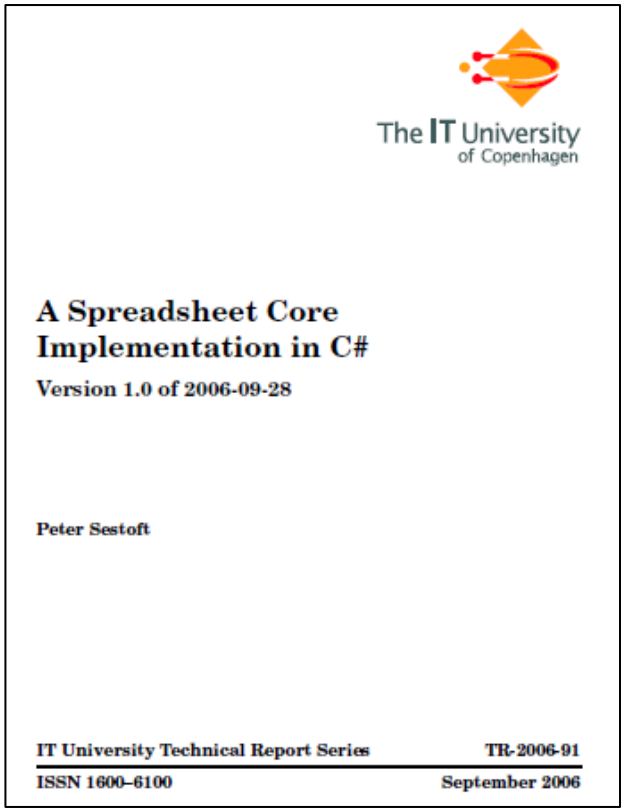
Outline

- Background
  - Why spreadsheets?
  - Sheet-defined functions
- Partial evaluation of sheet-defined functions
- Basic specialization
- Volatile and external functions
- Recursive functions
  - Generalization strategy
- Perspectives and future work

Why spreadsheet technology

- 13-25 million spreadsheet “programmers”
- Often complex models: biology, physics, games, economy, financial wizardry, ...
- A tool for *end-user software development*
- *Better tools* are desirable

- Technological opportunities
  - Spreadsheets are functional
  - Bytecode generation in .NET
  - Parallelization for multi-core
  - Parallelization for GPUs
  - Specialization of functions


The IT University of Copenhagen

A Spreadsheet Core Implementation in C#

Version 1.0 of 2006-09-28

Peter Sestoft

IT University Technical Report Series TR-2006-91

ISSN 1600–6100 September 2006

Spreadsheet model examples

- Modelling of protein structure
  - 51.8 MB file, 594,524 non-blank cells, 764 M deps
- Game character creator (role-playing)
  - 23.7 MB file, 112,754 non-blank cells, 239 M deps
- Financial model from big German bank
  - 10.4 MB file
- Actuarial computations in life insurance
  - 120 years * 12 months * N simulated lives
- Food safety sampling plans
- Pesticide kinetics in agricultural soils
- Alpha scattering (particle physics)

The trouble with functions

- One cannot define functions in a spreadsheet
- 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:** Functions ***within*** the spreadsheet metaphor (Peyton-Jones et al 2003, Nuñez 2000)

Example: Area of a triangle

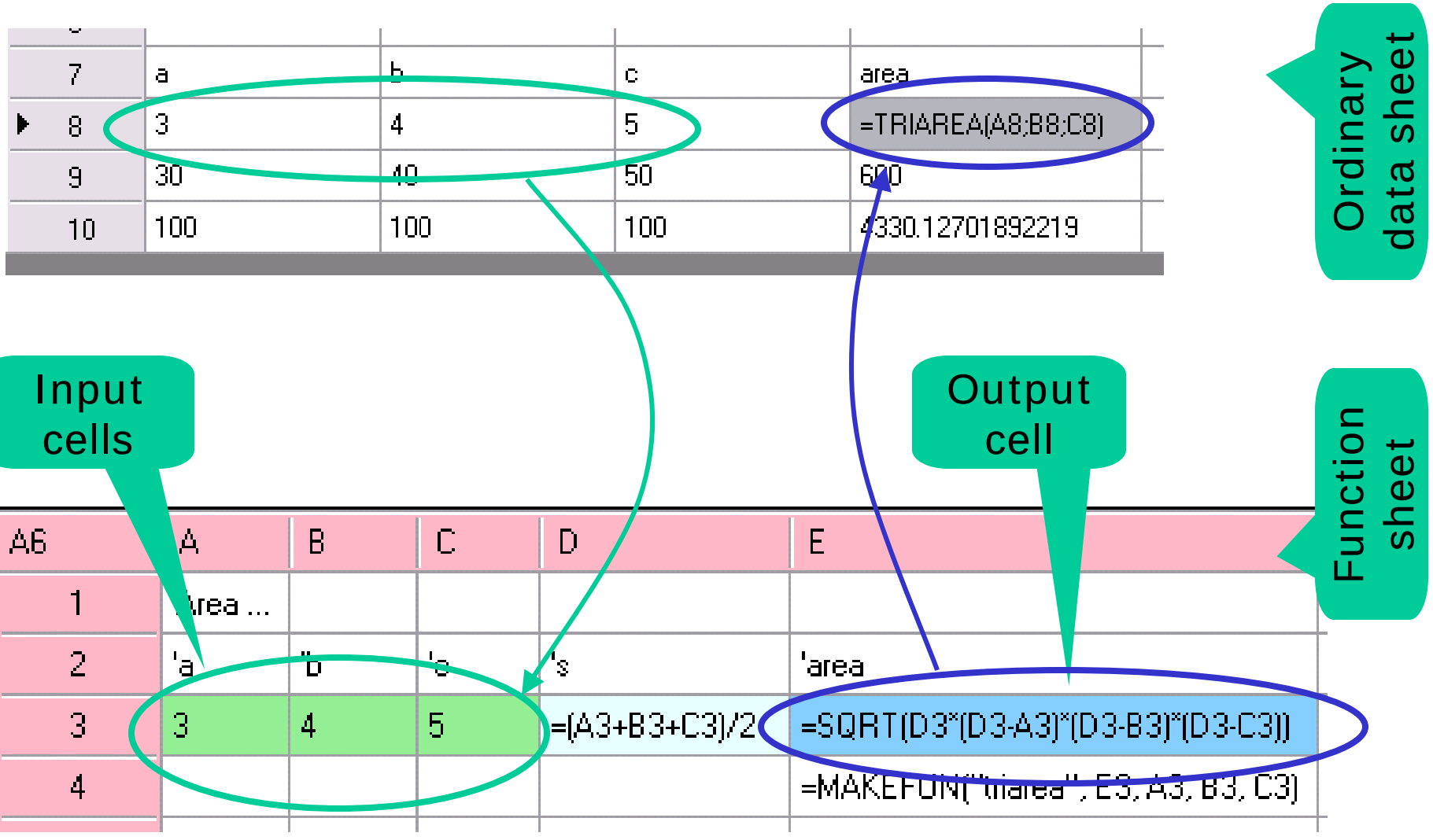
- Area of triangle with sides a, b, c is SQRT(s(s-a)(s-b)(s-c)) where s = (a+b+c)/2
- Either add extra column for s:

	a	b	c	s	area
6					
7	a	b	c	s	area
8	3	4	5	=(A8+B8+C8)/2	=SQRT(D8*(D8-A8)*(D8-B8)*(D8-C8))
9	30	40	50	=(A9+B9+C9)/2	=SQRT(D9*(D9-A9)*(D9-B9)*(D9-C9))
10	100	100	100	=(A10+B10+C10)/2	=SQRT(D10*(D10-A10)*(D10-B10)*(D10-C10))
11					

- Or inline s formula in area formula:

	a	b	c	area
6				
7	a	b	c	area
8	3	4	5	=SQRT((A8+B8+C8)/2*((A8+B8+C8)/2-A8)*((A8+B8+C8)/2-B8)*((A8+B8+C8)/2-C8))
9	30	40	50	=SQRT((A9+B9+C9)/2*((A9+B9+C9)/2-A9)*((A9+B9+C9)/2-B9)*((A9+B9+C9)/2-C9))
10	100	100	100	=SQRT((A10+B10+C10)/2*((A10+B10+C10)/2-A10)*((A10+B10+C10)/2-B10)*((A10+B10+C10)/2-C10))
11				

Solution:
Sheet-defined function TRIAREA



Spreadsheet “semantics”

- Cell values are *consistent* with formulas
- Calculation order is unspecified
- Each cell exhibits at most one value
- Errors propagate from argument to result
  - Also in A1>37 and in IF(A1>37, 11, 22)*42
- Some functions are *volatile*: NOW, RAND
  - They are re-evaluated in every recomputation
- Some functions are *nonstrict*: IF, CHOOSE
- Cyclic dependencies must be discovered

Compiling a spreadsheet function

- Basically, generate cell bindings:

```
v_D3 = (v_A3+v_B3+v_C3)/2.0;
v_E3 = Math.Sqrt(v_D3*(v_D3-v_A3)*...));
```

- Complications are caused by
  - Errors must be tolerated and propagated
  - Value boxing should be avoided for efficiency
  - Due to non-strict IF and recursion, evaluation must be guarded by an *evaluation condition*

```
if (v_B67 != 0.0)
  v_B68 = REPT4(v_B66, v_B67/2);
v_B69 = ... v_B68 ...;
```

- Evaluation conditions are complicated by volatile functions

Compiler: runtime code generation

Spreadsheet formula

```
=SQRT(a1*a1+a2*a2)
```

My compiler

.NET bytecode

```
ldloc 2
ldloc 2
mul
ldloc 3
ldloc 3
mul
add
call Math.Sqrt
```

.NET JIT compiler

x86 machine code

```
fldl   0xfffffff0(%ebp)
fldl   0xfffffff0(%ebp)
fmulp  %st,%st(1)
fldl   0xffffffe8(%ebp)
fldl   0xffffffe8(%ebp)
fmulp  %st,%st(1)
faddp  %st,%st(1)
fsqrt
```

Result:
A fast, portable spreadsheet implementation

Example: MONTHLEN(Y,M)

```
MONTHLEN(Y,M)=
 CHOOSE(M,31,28+OR(AND(NOT MOD(Y,4)),MOD(Y,100)),NOT(MOD(Y,400))),
          31,30,31,30,31,31,30,31,30,31)
```

Leap year adjustment

```
0000 ldarg V_0
0004 call ToDoubleOrNan
0009 stloc.3
000a ldarg V_1
000e call ToDoubleOrNan
0013 stloc.0
0014 ldloc.0
0015 call IsInfinity
001a brtrue 0221
001f ldloc.0
0020 call IsNaN
0025 brtrue 0221
002a ldloc.0
002b conv.i4
002c ldc.i4 1
0031 sub
0032 switch (0076, 0089, ...)
0067 ldc.i4 7
006c call ErrorValue.FromIndex
0071 br 021c
0076 ldc.r8 31
007f call NumberValue.Make
0084 br 021c
0089 ldc.r8 28
0092 ldloc.3
0093 ldc.r8 4
009c call ExcelMod
00a1 stloc.0
00a2 ldloc.0
00a3 call IsInfinity
00a8 brtrue 0118
00ad ldloc.0
00ae call IsNaN
00b3 brtrue 0118
00b8 ldloc.0
00b9 ldc.r8 0
00c2 beq 011e
00c7 ldloc.3
00c8 ldc.r8 400
00d1 call ExcelMod
00d6 stloc.0
00d7 ldloc.0
00d8 call IsInfinity
00dd brtrue 0118
00e2 ldloc.0
00e3 call IsNaN
00e8 brtrue 0118
00ed ldloc.0
00ee ldc.r8 0
00f7 beq 010a
00fc ldc.r8 0
0105 br 0113
010a ldc.r8 1
0113 br 0119
0118 ldloc.0
0119 br 0158
011e ldloc.3
011f ldc.r8 100
0128 call ExcelMod
012d stloc.0
012e ldloc.0
012f call IsInfinity
0134 brtrue 0118
0139 ldloc.0
013a call IsNaN
013f brtrue 0118
0144 ldloc.0
0145 ldc.r8 0
014e beq 00c7
0153 br 010a
0158 add
0159 call NumberValue.Make
015e br 021c
0163 ldc.r8 31
...
```

Higher-order functions and GETFUN

- A function can be partially applied
- GETFUN("MONTHLEN", 2010) is a function that computes month lengths in year 2010
- It is called as APPLY(G120, 2) to get eg. 28

- As usual in a functional language, the partial application simply creates a *closure*, that is, a package MONTHLEN(2010)

Opportunity for specialization

- Better: *Specialize* the MONTHLEN function with respect to the given argument, eg. Y=2010
- Use *partial evaluation* (Futamura 1970)
- The given arguments are called *static*, the missing ones *dynamic*

- General partial evaluation has not caught on
  - Problems with loop unrolling, side effects, etc
- But spreadsheets are mostly declarative, have explicit data and control dependencies, etc
- So specialization can be easy and worthwhile

Specializing MONTHLEN to Y

```
MONTHLEN(Y,M)=
 CHOOSE(M,
  31,
  28+OR(AND(NOT(MOD(Y,4)),
       MOD(Y,100)),
       NOT(MOD(Y,400))),
  31,30,31,30,
  31,31,30,31,30,31)
```

When Y is known, this can be computed

```
0000: ldarg V_0
0004: call ToDoubleOrNan
0009: stloc.0
000a: ldloc.0
...
0016: call IsNaN
001b: brtrue 0150
0020: ldloc.0
0021: conv.i4
0022: ldc.i4 1
0027: sub
0028: switch (006c, 007f, 0092, ...)
005d: ldc.i4 7
0062: call ErrorValue.FromIndex
0067: br 014b
006c: ldc.r8 31
0075: call NumberValue.Make
007a: br 014b
007f: ldc.r8 28
0088: call NumberValue.Make
008d: br 014b
0092: ldc.r8 31
...
015a: ret
```

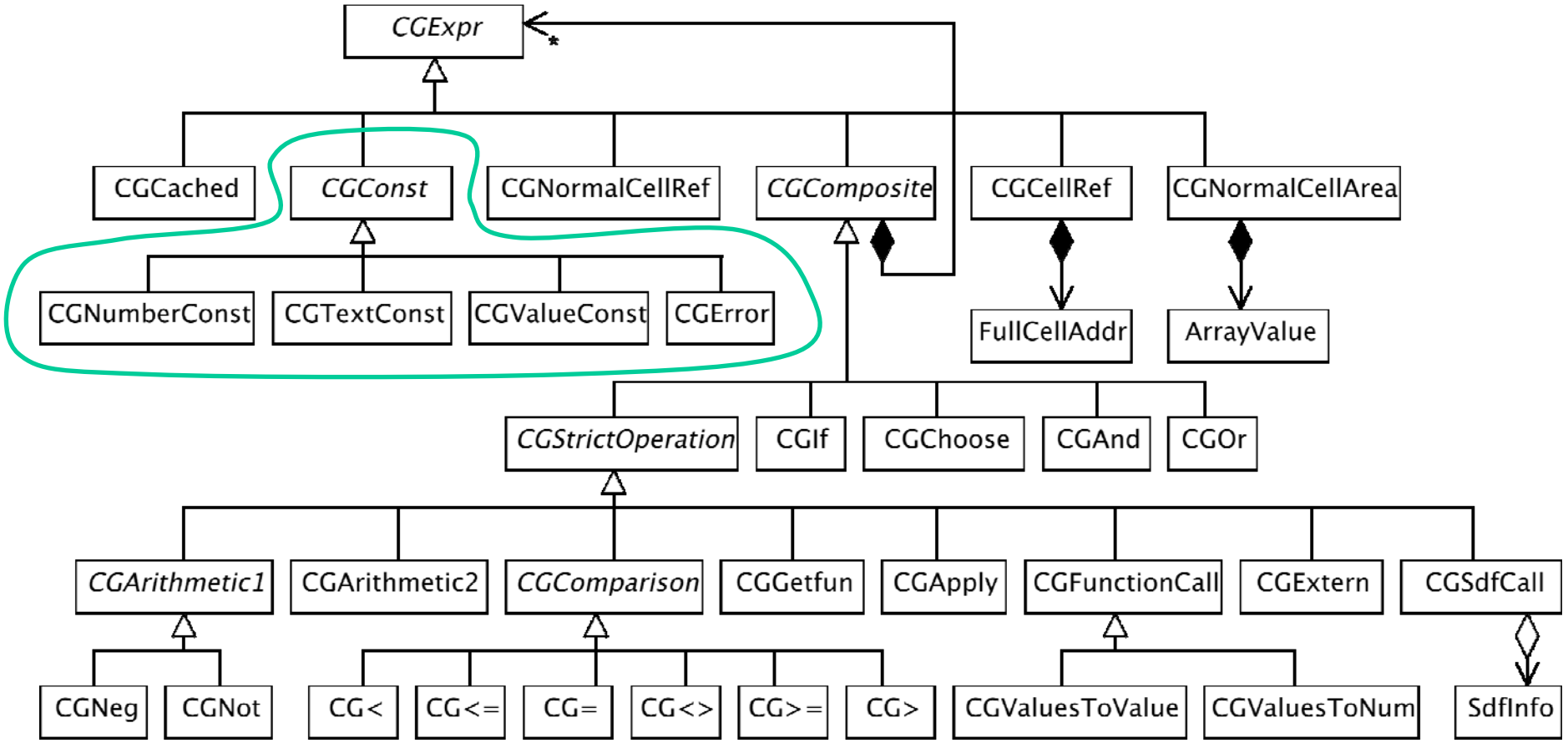
Specialized MONTHLEN(2010)

Should be easy to use

- The old GETFUN("MONTHLEN", 2010)
  - Creates an *unoptimized* closure
- A new function PEFUN("MONTHLEN", 2010)
  - Performs specialization at runtime
  - Creates an *optimized* closure

- The two closures are called in exactly the same way: APPLY(G120, 2)
  - The specialized one is likely to be faster

The expression class hierarchy



Basic online partial evaluation

- The result of partial evaluation is an expression; possibly a *constant* expression
- To partially evaluate e1+e2:
  - Partially evaluate e1 to get r1
  - Partially evaluate e2 to get r2
  - If both r1 and r2 are constants, compute their sum; the result is a new constant number
  - If one or both are non-constants, the result is the non-constant *residual* expression (r1+r2)

Static value

Dynamic value

- A constant is a subclass of CGConst
- A non-constant is everything else

External and volatile functions

- Partial evaluation of RAND()
  - Does not result in a number, such as 0.4212
  - Instead it is residualized as an expression RAND()

- References to *data sheets*, calls to *external* functions, and calls to *volatile* functions RAND, NOW are always residualized
- This makes partial evaluation agree with spreadsheet semantics: any recomputation should re-evaluate volatile functions
- This makes partial evaluation *deterministic*, important for termination and loop creation

Multistage specialization of ADD3(x,y,z) = x+y+z

129 ns

```
0000 ldarg V_0
0004 call ToDoubleOrNan
0009 ldarg V_1
000d call ToDoubleOrNan
0012 add
0013 ldarg V_2
0017 call ToDoubleOrNan
001c add
001d call NumberValue.Make
0022 ret
```

x=101

84 ns

```
0000 ldc.r8 101
0009 ldarg V_0
000d call ToDoubleOrNan
0012 add
0013 ldarg V_1
0017 call ToDoubleOrNan
001c add
001d call NumberValue.Make
0022 ret
```

y=201

68 ns

```
0000 ldc.r8 302
0009 ldarg V_0
000d call ToDoubleOrNan
0012 add
0013 call NumberValue.Make
0018 ret
```

z=301

53 ns

```
0000 ldc.r8 603
0009 call NumberValue.Make
000e ret
```

Example: quasi-Monte Carlo Black-Scholes (qmcbs)

- Pricing of European call options
- QMCBS(p0,d0,r,delta,sigma,T,p)
- Fixed parameters: p0,d0,r,delta,sigma,T
- Variable parameter: p

- Specialization wrt fixed gives speedup from 650 ns to 508 ns, or 22%
- But only 8% speedup of the simulation due to other overheads (sampling)

Recursive functions

- Specializing non-recursive functions is easy
- If we do not specialize recursive calls, a specialized program cannot have loops: sad
- Specialization of recursive calls is dangerous
  - (A) Creation of an infinite number of specialized versions
  - (B) Creation of an infinitely large expression
  - (C) Infinite loop while computing a constant
- (A) is prevented by *generalization*: considering a constant to be non-constant
- (B) is prevented by *not unfolding* and (A)
- (C) is OK if the original program would loop

Static and dynamic control

- A function call is under *dynamic* control if it appears inside some IF(e1, F(...), ...) whose condition e1 does not evaluate to a constant
- Otherwise it is under *static control*

- Similarly for CHOOSE(e0, e1...en), and shortcut AND(e1,e2) and OR(e1,e2)

Generalization and unfolding

- Generalization rule:
  - Assume we meet a call f(e1...en) during partial evaluation of f(a1...am), n>=m
  - If the call is under *static control*, specialize wrt static prefix of e1...em
  - If the call is under *dynamic control*, specialize wrt only those static values of e1...em that equal a1...am
- Current unfolding rule:
  - Don't unfold, even if all arguments are static
  - (For simplicity. Due to volatile functions, the result may be a residual expression)

Properties of the generalization strategy

- Good
  - Accumulating static parameters under dynamic control are generalized, so harmless
  - Dynamic-control recursive branches do not give rise to exponentially large residual programs
  - Structural descent under static control (power, interpreters, …) not generalized, so no loss
  - Unchanging static “configuration” parameters are not generalized, so no loss
- Bad
  - Sometimes generalizes where we do not want it
- Termination
  - Well …
  - Understanding and formalization wanted

Generalization example: Ackermann's function, take one

- One way to define Ackermann's function

```
ackA(m,n) =
  IF(m=0, n+1,
          IF(n=0, ackA(m-1, 1),
                  ackA(m-1, ackA(m, n-1))))
```

- Poor result of specializing wrt m=2:

```
ackA2(n) =
  IF(n=0, ackA(1,1), ackA(1, ackA2(n-1)))
```

- Because ackA(1,ackA(2,n-1)) under dynamic control (n=0)

Ackermann's function, better

- A different definition of Ackermann:

```
ackB(m,n) =
  IF(m=0, n+1,
          ackB(m-1, IF(n=0, 1, ackB(m, n-1))))
```

- Better result of specializing wrt m=2:

```
ackB2(n) = ackB1(IF(n=0, 1, ackB2(n-1)))
ackB1(n) = ackB0(IF(n=0, 1, ackB1(n-1)))
ackB0(n) = n+1
```

Generalization example

- F(P,N)=IF(1<=P, ERR("P"),IF(RAND()<P,F(P,N+1),N)
- Specialize wrt P=0.8 and N=0
- Both P and N static but RAND is dynamic, so dynamic control

```
0000: call ExcelRand
0005: ldc.r8 0.8
000e: bge 0035
0013: ldsfld sdfDelegates
0018: ldc.i4 30
001d: ldelem.ref
001e: castclass Fun`2
0023: ldsfld NumberValue.ONE
0028: tail.
002a: call Invoke
002f: ret
0030: br 003a
0035: ldsfld NumberValue.ZERO
003a: ret
```

Specialized F(0.8, 0)

```
0000: call ExcelRand()
0005: ldc.r8 0.8
000e: bge 0048
0013: ldsfld sdfDelegates
0018: ldc.i4 30
001d: ldelem.ref
001e: castclass Fun`2
0023: ldarg V_0
0027: call ToDoubleOrNan
002c: ldc.r8 1
0035: add
0036: call NumberValue.Make
003b: tail.
003d: call Invoke
0042: ret
0043: br 004c
0048: ldarg V_0
004c: ret
```

Specialized F(0.8)

- Speedup from 1060 ns to 665 ns

Example: REPT5(n,s)

- REPT5(n,s) = n copies of s concatenated
- Unspecialized REPT5 is 123 bytecode instructions
- REPT5(10,”abc”) takes 2430 nanoseconds

- Specialized REPT5(10) uses REPT5(5), REPT5(2), REPT5(1), REPT5(0); in total 11+13+11+13+2 = 50 bytecode instructions
- Application to “abc” takes 1048 nanoseconds

Perspective: Parallelization for graphics cards (GPU)

- Partial evaluation primarily
  - Decides conditional jumps early, eg by loop unrolling
  - Performs some operations early, eg arithmetics
- On modern CPUs (superscalar, with branch prediction, etc) this gives very little speedup
- **But** graphics processors are fast for straightline arithmetic code, slow for tests and jumps
- **So** elimination of tests and jumps by partial evaluation should give large speedups on GPUs

- Garbos and Videbæk BSc project, ITU 2010:
  - *Spreadsheet optimization on GPUs*
- See also Berlin and Weise 1990

Future work

- Improve flexibility of specialization
  - Not only prefix of arguments
- Formalize a semantics of spreadsheets
- Show PE conforms to this semantics
- Study synergy with GPU parallelization
- Enable some unfolding
  - Would be good for GPU application too
- Prove that generalization strategy ensures termination of partial evaluation
  - Unless original would loop on given inputs
- Develop more case studies