An Update on the Coconut Project

Christopher Kumar Anand
Michal Dobrogost
Maryam Moghadas
Jessica Pavlin
Yuriy Toporovskyy
Wolfram Kahl
COCONUT
COde CONstructing User Tool

- DSLs embedded in Haskell
- *strong* types (detect errors in model and code)
- *high* level
  - start with objective function
  - differentiate symbolically
  - simplify expressions
  - recognize recurrence relations
  - generate code
- verification
  - SIMD & distributed parallelization (private and shared-memory)
  - linear-time verification
Models

define variables

```haskell
> let x = var3d (16,16,16) "x"
> let y = var3d (16,16,16) "y"
> ft (x +: y)

\[ \text{FT}(x(16,16,16)+:y(16,16,16)) \]
```

define an objective function to minimize

\[ \| \text{ft}(x + iy) \|_2 \]

and take it’s derivative

```haskell
>> diff (mp ["X"] ) (norm2 (ft (x +: y)))

\[
((\text{Re}(\text{FT}((dX[16][16][16])+:0.0[16,16,16])))))(\text{Re}(\text{FT}((X[16][16][16])+:Y[16][16][16])))
+ ((\text{Re}(\text{FT}((dX[16][16][16])+:0.0[16,16,16])))))(\text{Re}(\text{FT}((X[16][16][16])+:Y[16][16][16])))
+ ((\text{Im}(\text{FT}((dX[16][16][16])+:0.0[16,16,16])))))(\text{Im}(\text{FT}((X[16][16][16])+:Y[16][16][16])))
+ ((\text{Im}(\text{FT}((dX[16][16][16])+:0.0[16,16,16])))))(\text{Im}(\text{FT}((X[16][16][16])+:Y[16][16][16])))
\]
more complicated models

- e.g., \text{conservation of mass} in material transport models is a function

\[
\text{massConservation} (\text{ThreeD } vx, \text{ThreeD } vy, \text{ThreeD } vz) \\
= \text{norm2} \left( \text{conv3Zip1ZM com } vx \text{ vy vz } \right)
\]

where
\[
\text{com} (vX, vY, vZ) = (vX[1,0,0] - vX[-1,0,0]) \\
+ (vY[0,1,0] - vY[0,-1,0]) \\
+ (vZ[0,0,1] - vZ[0,0,-1])
\]
what was that $d(X)$, and where’s $\nabla f$?

- $d(X)$ is a vector of differential forms
- comes from *implicit* derivative
- we can extract $\nabla f$ by simplifying
simplify rules

- commute \( d(x) \) to LHS of dot
- move linear operators to RHS via adjoint
Fourier Transform of a complex vector. We can calculate the exterior derivatives using Automatic Discretization structure, to enable more efficient function and gradient computations and preserving the multi-dimensional awareness of an implementation capable of calculating derivatives with respect to vector variables, and of collecting subexpressions common to both the input and output expressions in the original function. Although existing symbolic calculators have only implemented first-order differentiation, but higher-order derivatives could be similarly computed. Since we are targeting large problems, algebraic simplifications to put the resulting expressions into normal forms, or a functional equivalent involving implicit differentiation, must be performed. However, as taught in vector calculus, instead, we must use the simplification DSL, although the semantics are subtly different from which gradients can be extracted. While first-class symbolic vectors have many advantages, it is not possible to implement exterior derivatives, although the semantics are subtly different.

Consider the scalar function $f(x) = x^7$. We can compute $f^{(7)}(x)$ using the Fast Fourier Transform (FFT) and its inverse ($f^{-1}$), as follows:

$$f^{(7)}(x) \mapsto x$$

This operation can be implemented in the simplification DSL as:

```
Just (x) <- (M.invFt 'o' M.ft) exprs node = x exprs
```

- **Match** inverse FT composed with FT of $x$
- **Replace** with $x$

Composition is designed to have syntax very close to that of the algebraic simplification DSL, making it easier to port existing symbolic calculators. The development of a second DSL for term graph rewriting, with two components, will simplify code generation than would otherwise be expected using Automatic Discretization. Currently, composition can be chained together using Haskell pattern guards, and alternative matches are chained together using composition syntax.
Type Checking

(|| work)

• with stronger typing, compilers find more mistakes
• in C, all dynamic arrays (float*) look alike
• others check size and dimension
• we can do better
Arrays of Samples
Arrays of Samples

- number of samples
- dimension
- frame of reference
- resolution (including units)
- units of measurement
Formalization

- in Haskell types define
  - physical units
  - array size and dimension
  - frame of reference

```haskell
canalSample2 :: Discretization1D (F "CanalFrame")
(NAT 12)
[Float | 0.02 | ]
```

Trying to add those two samplings to each other will cause a compile time error:

```
No instance for (Add (Discretization1D (F "CanalFrame")
(NAT 12)
(Floating 'Pos 2 (Exp 2))
SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
[Float] )
(Discretization1D (F "CanalFrame")
(NAT 12)
(Floating 'Pos 1 (Exp 2))
SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
[Float] )
```

The error message indicates that the discretizations have different types and adding them is not a valid operation. Ideally, we would prefer the error message to say the sample spacing needs to be identical but it would not be difficult for a domain expert to understand this error.
No instance for 
(Add 
  (Discretization1D 
    (F "CanalFrame")
    (NAT 12)
    (FLOAT 'Pos 1 (E-2))
    (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
    [Double])
  (Discretization1D 
    (F "CanalFrame")
    (NAT 12)
    (FLOAT 'Pos 2 (E-2))
    (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('Mol 0) ('K 0))
    [Double])

array sizes match, so try to add them …
Type Inference Across Linear Operations

• Fourier Transforms are very picky
• Nyquist Sampling Theorem
  • a *theorem* about how you can
• frame of reference
• resolution (including units)
• units of measurement
Classy Proofs

• Properties are Classes

```
class FT a b | a → b, b → a where
  ft :: a → b
  invFt :: b → a
```

```
class MultD3 f2 f1 f0 e2 e1 e0 g2 g1 g0 | f2 f1 f0 e2 e1 e0 → g2,
  f2 f1 f0 e2 e1 e0 → g1,
  f2 f1 f0 e2 e1 e0 → g0 where
```
Proofs are Instances

\textbf{instance} (  
AssertDualFrames frame1 frame2, Frame frame1, Frame frame2,  
IsFloat stepSize2, IsFloat stepSize1, ToFloat numSamp ~ numSampF,  
) ⇒  
FT (Discretization1D frame1 numSamp stepSize1 rangeU [Complex Double])  
(Discretization1D frame2 numSamp stepSize2 rangeU [Complex Double])  
where  
ft (Discretization1D x) = (Discretization1D $ FFT.fft x)  
invFt (Discretization1D x) = (Discretization1D $ FFT.ifft x)

\textbf{instance} (  
Times f0 e0 p00h p00l, Times f1 e0 p10h p10l,  
Times f2 e0 D0 p20l, Times f0 e1 p01h p01l,  
Times f1 e1 D0 p11l, Times f2 e1 D0 D0,  
Times f0 e2 D0 p02l, Times f1 e2 D0 D0,  
Times f2 e2 D0 D0,  
Add3 p00h p10l p01l c1h c1l,  
Add5 p20l p01h p11l p02l c1h D0 c2l)  
⇒ MultD3 f2 f1 f0 e2 e1 e0 c2l c1l p00l where
Readable Errors

- type inference fails on modelling errors
- ghc *loves* to throw up thousand-line errors
- we tamed it

No *instance* for (*DualUnits*
  
  (SIUnit (’M 1) (’S 0) (’Kg 0) (’A 0) (’Mol 0) (’K 0))
  (SIUnit (’M 0) (’S 0) (’Kg 0) (’A 0) (’Mol 0) (’K 0))))
Multi-Core = ILP Reinvented

<table>
<thead>
<tr>
<th>Instruction Level Parallelism</th>
<th>Multi-Core Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Chip</td>
</tr>
<tr>
<td>Execution Unit</td>
<td>Core</td>
</tr>
<tr>
<td>Load/Store Instruction</td>
<td>DMA</td>
</tr>
<tr>
<td>Arithmetic Instruction</td>
<td>Computational Kernel</td>
</tr>
<tr>
<td>Register</td>
<td>Buffer / Signal</td>
</tr>
</tbody>
</table>
The Catch: Soundness

- on CPUs hardware maintains OOE
- instructions execute out of order
- hardware hides this from software
  - ensures order independence
- in our Multi-Core virtual CPU
- compiler inserts synchronization
  - soundness up to software
  - uses asynchronous communication
Asynchronous

• no locks
  • locking is a multi-way operation
  • a lock is only local to one core
    • incurs long, unpredictable delays
• use asynchronous messages
  • matches efficient hardware
Async Signals

No reads or writes to buffer until DMA completion is confirmed. No writes to buffer until DMA completion is confirmed. Other operations. Reorder Window. Hazard. Reorder Window. No reads or writes to buffer until past barrier WaitData.
# Multi-Core Language

**AVOps**

<table>
<thead>
<tr>
<th>Computation operation bufferList</th>
<th>do a computation with local data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SendData</strong> localBuffer remoteBuffer tags</td>
<td>start DMA to send local data off core</td>
</tr>
<tr>
<td><strong>WaitData</strong> localBuffer tag</td>
<td>wait for arrival of DMAed data</td>
</tr>
<tr>
<td><strong>WaitDMA</strong> tag</td>
<td>wait for locally controlled DMA to complete</td>
</tr>
<tr>
<td><strong>SendSignal</strong> core signal</td>
<td>send a signal to distant core</td>
</tr>
<tr>
<td><strong>WaitSignal</strong> signal</td>
<td>wait for signal to arrive</td>
</tr>
<tr>
<td><strong>Loop</strong> n π body</td>
<td>body; π(body); π(π(body))…</td>
</tr>
</tbody>
</table>
**locally Sequential Program**

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SendSignal ( s \rightarrow c2 )</td>
<td>WaitSignal ( s )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>SendSignal ( s \rightarrow c2 )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>WaitSignal ( s )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- total order for instructions
- easier to think in order
- send precedes wait(s)
NOT sequential

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SendSignal $s \rightarrow c2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
<tr>
<td></td>
<td>second signal overlaps the first, only one registered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
</tbody>
</table>

• can execute out of order
does NOT imply order independent

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>computation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>using</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>wrong assumptions</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SendSignal $s \rightarrow c2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
</tbody>
</table>
Linear-Time Verification

- must show
  - results are independent of execution order
  - no deadlocks
- need to keep track of all possible states
- linear in time = one-pass verifier
  - constant space
    - = max possible states at each instruction
Proof State

- state of buffers (valid, waiting for DMA, ...)
- active signals
- $\Phi$ follows map $\Phi_n(c_1, c_2)$
- records last instruction on core 1 known to complete before the last instruction on core 2 completing before instruction $n$
Algorithm

- maintain the state one instruction at a time
- flag indeterminate states as errors

Proof

- show that any indeterminacy and/or deadlock would have been flagged
Impact

• no parallel debugging !!
• every optimization trick used for ILP can be adapted
Loops (Simple Repetition)

- locally sequential after 2 iterations
  $\Rightarrow$ locally sequential

- no hazards after 2 iterations
  $\Rightarrow$ no hazards

- proof: $\Phi$... are periodic after 1st iteration
Loops (With Rewriting)

- rewriting = permutation per iteration
- if rewritten $\Phi \ldots$ repeats a state & locally sequential $\Rightarrow$ locally sequential
- if rewritten $\Phi \ldots$ repeats a state, & hazard free $\Rightarrow$ hazard free
- proof: $\Phi \ldots$ deja vu (all over again)
But...

• assumes you have signals
• what about shared memory?
• still lock-free synchronization?
• let’s try it on x86
Single-Reader, Single-Writer AVOp Ring Buffers (SRSWARB)
Limit Hazards

- AVOps can only conflict if they can fit in the Ring Buffers at the same time
- on each core, AVOps are sequential, therefore safe
- reads on different cores are safe
- while write AVOp is on a core, check that no other core reads or writes
Nest Step

• signals across nodes

• SRSWARB system for multicore

• new system for GPU
Details ...


Jessica L M Pavlin and Christopher Kumar Anand, Symbolic Generation of Parallel Solvers for Inverse Imaging Problems, CAS-14-05-CA.

Maryam Moghadas, Yuriy Toporovskyy, Christopher Kumar Anand, Type-Safety for Inverse Imaging Problems, CAS-14-04-CA.
It Works!

• used to generate, from the objective function, a multi-core (shared memory) image reconstruction software for parallel Magnetic Resonance Imaging for AllTech Medical Systems America
Thanks

Stephen Adams
Curtis d’Alves
Kevin Browne
Shiqi Cao
Nathan Cumpson
Saeed Jahed
Damith Karunaratne
Clayton Goes

Gabriel Grant
William Hua
Fletcher Johnson
Wei Li
Nick Mansfield
Maryam Moghadas
Mehrdad Mozafari
Adam Schulz

Anuroop Sharma
Sanvesh Srivastava
Wolfgang Thaller
Gordon Uszkay
Christopher Venantius
Paul Vrbik
Fei Zhao

Robert Enenkel

IBM Centre for Advanced Studies, CFI, OIT, MITACS, NSERC, OCA Inc. and Apple Canada for research support.