The role of domain-specific languages for cyber-physical systems

Jeronimo Castrillon
Chair for Compiler Construction (CCC)
TU Dresden, Germany

Seminar series: Design and Programming Cyber-Physical Systems and IoT applications
Virtual from Dresden, Germany. October 2020
Cyber-physical systems

- Cyber-Physical Systems (CPS): Integration of computing with physical processes. Embedded computers monitor and control physical processes, usually with feedback loops (physical processes affect computations and vice versa).

- Special requirements
  - Reactivity
  - Adaptivity
  - Time Sensitivity
  - Safety Criticality

- Even more demanding computational power (inference, data processing, ...)

Edward A. Lee, “Cyber physical systems: Design challenge”. ISORC’08
Evolution of computing

Evolution of computing: Heterogeneity is mainstream

- Heterogeneous many-cores, scalable platforms, complex memory hierarchies, domain-specific accelerators, emerging technologies...
- Plus: Stringent application constraints

1999

SoC programming: Evolution (required)

- Sequential: Auto-parallelization, pragmas, ...
- Formal model-based code/HW generation
- Higher-level programming abstractions

Application Architecture

Optimization Results

\[
A = \text{placeholder}((m, h), \text{name}='A')
\]
\[
B = \text{placeholder}((\infty, h), \text{name}='B')
\]
\[
C_{ij} = \sum_{k=1}^{\infty} A_{ki} B_{kj}
\]

\[
C = \text{compute}((m, h), \text{lambda } i, j:\ \text{sum}(A[k, i] * B[k, j], \text{axis}=k))
\]
Languages, tools & frameworks

- Heterogeneity not for nices
  - Embedded expert wouldn’t expect compiler to recognize an FFT written in C!

```c
void fft(CArray &x)
{
    // DFT
    unsigned int N = x.size(), k = N, n;
    double thetaT = 3.14159265358979323846264338328L / N;
    Complex phiT = Complex(cos(thetaT), -sin(thetaT)), T;
    while (k > 1)
    {
        n = k;
        k >>= 1;
        phiT = phiT * phiT;
        T = 1.0i;
        for (unsigned int l = 0; l < k; l++)
        {
            for (unsigned int a = l; a < N; a += n)
            {
                unsigned int b = a + k;
                Complex t = x[a] - x[b];
                x[a] += t * T;
                x[b] = t * T;
            }
            T = phiT;
        }
        // Decimate
        unsigned int m = (unsigned int)log2(N);
        for (unsigned int a = 0; a < N; a++)
        {
            unsigned int b = a;
            // Reverse bits
            b = (((b & 0xaaaaaaaa) >> 1) | ((b & 0x55555555) << 1));
            b = (((b & 0xCCCCCCCC) >> 2) | ((b & 0x33333333) << 2));
            b = (((b & 0xF0F0F0F0) >> 4) | ((b & 0x0F0F0F0F) << 4));
            b = (((b & 0x0F0F0F0F) >> 8) | ((b & 0xF0F0F0F0) << 8));
        }
    }
}
```
Languages, tools & frameworks

- Heterogeneity not for nices
  - Embedded expert wouldn’t expect compiler to recognize an FFT written in C!

- In CPS and computing in general
  - Changing HW substrate
  - Wider range of programmer backgrounds

- Tools, methodologies and frameworks more important than ever!

- High-level tools: Select the right abstraction when possible
  - More optimization, stronger semantics
  - Domain-specific SW for domain-specific HW

- Low-level tools: Legacy, expert coders & target of high-level flows
Background
Languages as abstractions

- Languages evolve, formalizing powerful design patterns (abstractions)
  - Some of them too common, so we do not notice it

- Examples
  - From calling conventions to procedures

```assembly
calc:
push EBP ; save old frame pointer
mov EBP,ESP ; get new frame pointer
sub ESP,localsize ; reserve place for locals
... ; perform calculations, leave result in EAX
mov ESP,EBP ; free space for locals
pop EBP ; restore old frame pointer
ret paramsize ; free parameter space and return
```

Source: https://en.wikipedia.org/wiki/Calling_convention

f(x) {...}
Languages as abstractions

- Languages evolve, formalizing powerful design patterns (abstractions)
  - Some of them too common, so we do not notice it

- Examples
  - From instructions to expressions
  - From calling conventions to procedures
  - From label-goto to structured control flow
  - From memory layout to data types (arrays, structs, …)
  - Memory allocation/deallocation (new/delete, garbage collection)
  - From function pointers and tables to dynamic dispatch
  - …
Domain specific languages (DSLs)

- DSLs help bridge the gap between problem domain and general purpose languages

Problem domain → General purpose language → Machine code

Adapted from lecture: “Concepts of Programming Languages”, Eelco Visser, TU Delft
Domain specific languages (DSLs)

- DSLs help bridge the gap between problem domain and general purpose languages

Problem domain  ->  DSL  ->  General purpose language  ->  Machine code

- Natural vocabulary for concepts are fundamental to problem domain
- Faster way to write common concepts (concrete syntax)
- Optimization potential due to domain-specific information
- A DSL can be disguised as library or framework

Adapted from lecture: “Concepts of Programming Languages”, Eelco Visser, TU Delft
DSLs and semantic model

- DSLs offer a way of manipulating an abstraction (or semantic model)
  - Other example of abstraction: APIs
  - A DSL can be evolve from an API: more flexible manipulation

When defining a DSL, the hardest and most important part is the definition of the semantic model (the rest is engineering)

Basics of PL: Formal (dynamic) semantics

- Relation between syntax (e.g., as context-free grammar), states and values
  - States: Can be seen as the state of the machine it runs on
  - Values: Actual values of symbols during execution

- Operational semantics
  - Set of inference rules: Describe how syntactic constructs update the state
  - Small (detailed via transitions systems) vs Big (fewer transitions in derivation tree)
  - Commonly based on lambda-calculus

- Denotational semantics
  - Direct notation: meaning provided by functional style (aka state transformers)
  - Compositionality: Program execution as the composition of functions

Quite complex (useless?) for C++, but great asset for DSLs!
Operational semantics (nutshell)

- General form
  - If I can prove that expression $E$ in state $s$ evaluates to value $V$
  - Then program $L := E$ in that state will update the state, giving $L$ the value $V$

- Simple, though complex notation, enables analysis in DSLs

$$\langle E, s \rangle \Rightarrow V$$

$$\langle L := E, s \rangle \rightarrow (s \uplus (L \mapsto V))$$

- $[\text{ass}_{ns}]$
  - $x := a, s \rightarrow s[x \mapsto A[a]s]$

- $[\text{skip}_{ns}]$
  - $\langle \text{skip}, s \rangle \rightarrow s$

- $[\text{comp}_{ns}]$
  - $S_1, s \rightarrow s', S_2, s' \rightarrow s''$
  - $\langle S_1; S_2, s \rangle \rightarrow s''$

- $[\text{if}^{tt}_{ns}]$
  - $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'$
  - if $B[b]s = \text{tt}$

- $[\text{if}^{ff}_{ns}]$
  - $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'$
  - if $B[b]s = \text{ff}$

- $[\text{while}^{tt}_{ns}]$
  - $\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''$
  - if $B[b]s = \text{tt}$

- $[\text{while}^{ff}_{ns}]$
  - $\langle \text{while } b \text{ do } S, s \rangle \rightarrow s$ if $B[b]s = \text{ff}$

http://softlang.wikidot.com/rlaemmel:home
Big-step operational semantics: Derivation tree

- Using assign and compose
- Program:
  - \texttt{z:=x; x:=y; y:=z}
  - Initial state: \(x=5\), \(y=7\), \(z=0\)

\[
\begin{align*}
\langle z := x, s_0 \rangle &\rightarrow s_1 & \langle x := y, s_1 \rangle &\rightarrow s_2 \\
\langle z := x; x := y, s_0 \rangle &\rightarrow s_2 & \langle y := z, s_2 \rangle &\rightarrow s_3 \\
\langle z := x; x := y; y := z, s_0 \rangle &\rightarrow s_3
\end{align*}
\]

\[
\begin{align*}
s_0 &= \{x\mapsto 5, y\mapsto 7, z\mapsto 0\} \\
s_1 &= \{x\mapsto 5, y\mapsto 7, z\mapsto 5\} \\
s_2 &= \{x\mapsto 7, y\mapsto 7, z\mapsto 5\} \\
s_3 &= \{x\mapsto 7, y\mapsto 5, z\mapsto 5\}
\end{align*}
\]

http://softlang.wikidot.com/rlaemmel:home
Denotational semantics

- (recall) Direct notation: meaning provided by functional style
  - Semantic domains
    - Store transformation:
    - Store observation:

- Semantic functions: mapping from syntax to semantics
  - Semantics of statements
  - Semantics of expressions

\[
\begin{align*}
storeT &= \text{store} \rightarrow \text{store} \\
storeO &= \text{store} \rightarrow \text{value}
\end{align*}
\]

\[
\begin{align*}
S : \text{stmt} &\rightarrow \text{storeT} \\
E : \text{expr} &\rightarrow \text{storeO}
\end{align*}
\]
Denotational semantics (2)

- **Semantic functions**

\[
S : \text{stmt} \rightarrow \text{store}\ T \\
\mathcal{E} : \text{expr} \rightarrow \text{store}\ O
\]

- **Example**
  - Underlined functions are semantic combinators: combine meanings, not syntax

\[
\begin{align*}
S[\text{skip}] &= \text{skip} \\
S[\text{assign}(x, e)] &= \text{assign } x (\mathcal{E}[e]) \\
S[\text{seq}(s_1, s_2)] &= \text{seq }(S[s_1]) (S[s_2]) \\
S[\text{if}(e, s_1, s_2)] &= \text{if } (\mathcal{E}[e]) (S[s_1]) (S[s_2]) \\
S[\text{while}(e, s)] &= \text{while } (\mathcal{E}[e]) (S[s]) \\
\mathcal{E}[\text{intconst}(i)] &= \text{intconst } i \\
\mathcal{E}[\text{var}(x)] &= \text{var } x \\
\mathcal{E}[\text{unary}(o, e)] &= \text{unary } o (\mathcal{E}[e]) \\
\mathcal{E}[\text{binary}(o, e_1, e_2)] &= \text{binary } o (\mathcal{E}[e_1]) (\mathcal{E}[e_2])
\end{align*}
\]

http://softlang.wikidot.com/rlaemmel:home
Example 1: Halide

- DSL for image processing pipelines
  - Composition of multiple stencils

- Abstraction
  - No explicit loops
  - Declarative approach to define filters as operation between functions
  - Functions: map coordinates to pixels, i.e., \( f(i,j) \) returns the pixel at position \( i,j \)

```plaintext
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

Halide: The power of abstraction

- Automatically play tradeoffs: storage+compute

```plaintext
UniformImage in(UInt(8), 2)
Var x, y
Func blur(x, y) = in(x-1, y) + in(x, y) + in(x+1, y)
Func out(x, y) = blur(x, y-1) + blur(x, y) + blur(x, y+1)

alloc blur[2048][3072]
for each y in 0..2048:
    for each x in 0..3072:
        blur[x][y] = in[y][x-1] + in[y][x] + in[y][x+1]
alloc out[2046][3072]
for each y in 1..2047:
    for each x in 1..3072:
        out[y][x] = blur[y-1][x] + blur[y][x] + blur[y+1][x]
```

Halide: The power of abstraction

- Automatically play tradeoffs: storage+compute

```plaintext
UniformImage in(UInt(8), 2)
Var x, y
Func blur(x, y) = in(x-1, y) + in(x, y) + in(x+1, y)
Func out(x, y) = blur(x, y-1) + blur(x, y) + blur(x, y+1)

alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blur[x-1, i]
    for each i in -1..1:
      blur[x] = in[y-1+i][x-1] + in[y-1+i][x] + in[y-1+i][x+1]
```

Halide: The power of abstraction

- Automatically play tradeoffs: storage + compute

```c
alloc out[2046][3072]
for each ty in 0..2048/32:
  for each tx in 0..3072/32:
    alloc blurx[-1..33][32]
    for y in -1..33:
      for x in 0..32:
        blurx[y][x] = in[ty+32+y][tx+32+x]-1
        + in[ty+32+y][tx+32+x]
        + in[ty+32+y][tx+32+x+1]

for y in 0..32:
  for x in 0..32:
    out[ty*32+y][tx*32+x] = blurx[y-1][x]
    + blurx[y][x]
    + blurx[y+1][x]
```

Example 2: Tensorflow

- Dataflow representation for the entire machine learning process
  - Preprocessing, inference, training, backing-up, …
  - For large-scale distributed processing

- On top of pure dataflow
  - Allow controlled global state (parameters) \(\rightarrow\) vertices can modified shared state
  - Explicit special queues (with known access patterns)
  - Edges are tensors (multi-dimensional arrays)
  - Include symbolic differentiation for training

### Tensorflow: API

```python
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])  # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))  # 784x100
b_1 = tf.Variable(tf.zeros([100]))
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_1)  # Output of linear layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))  # 100x10
b_2 = tf.Variable(tf.zeros([10]))
layer_2 = tf.matmul(layer_1, W_2) + b_2  # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdamOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    for step in range(NUM_STEPS):
        x_data, y_data = ...
        sess.run(train_op, {x: x_data, y: y_data})
```


**High-level known tensor operations**
MoC-based programming languages

- Models of computation (MoCs) define components and rules of how they interact
- For programming, MoCs also define possible transitions a system may follow
- Many examples: Synchronous dataflow, Khan Process Networks, Reactors, ... (see lecture by Prof. Edward Lee)

```cpp
#include "PnTransform.h"
#include "PnTVPUtg.h"
#include "PnTVPUmap.h"
#include "clang/AST/ASTContext.h"
#include "PnStreamFactory.h"

using namespace clang;

void clang::PnTransform(TransTarget transTarget, bool traces, const std::string &strMappingFileName, ASTContext &Ctx, Sema &S, const llvm::sys::Path &BasePath) {
    assert(transTarget != TransInvalid);
    TranslationUnitDecl *D = Ctx.getTranslationUnitDecl();
    PnTransformSizeof(D, S);
    PnTransformTask(D, S);
    switch (transTarget) {
        case TransSystemC:
        case TransVPUtg:
        case TransVPUmap:
            PnTCopying(D, S);
            break;
        default:
            ;
    }
    PnTransformPthreads(D, S, traces);
    ErasePnDefs(D);
    break;
    case TransSystemC:
        PrintForSystemC(D, S, traces, streamFactory);
        ErasePnDefs(D);
        break;
    case TransVPUtg:
        PrintForVPUtg(D, S, streamFactory);
        ErasePnDefs(D);
        break;
    case TransVPUmap:
        PrintForVPUmap(D, S, strMappingFileName, streamFactory);
        ErasePnDefs(D);
        break;
    case TransInvalid:
        assert(false);
        break;
    }
```
Graphical programming environments

LabVIEW multi-rate diagram


Node-RED (actors, devices, …)

https://flows.nodered.org

For ENSI students: Talk to Chadlia about this
Example 3: C for process networks

- **FIFO Channels**

  ```c
  typedef struct { int i; double d; } my_struct_t;
  __PNchannel my_struct_t S;
  __PNchannel int A = {1, 2, 3}; /* Initialization */
  __PNchannel short C[2], D[2], F[2], G[2];
  ```

- **Processes & networks**

  ```c
  __PNkpn AudioAmp __PNin(short A[2]) __PNout(short B[2])
  __PNparam(short boost){
      while (1)
      __PNin(A) __PNout(B) {
          for (int i = 0; i < 2; i++)
            B[i] = A[i]*boost;
      }
  __PNprocess Amp1 = AudioAmp __PNin(C) __PNout(F) __PNparam(3);
  __PNprocess Amp2 = AudioAmp __PNin(D) __PNout(G) __PNparam(10);
  ```
- Extended application specification
  - Selected processes are algorithmic kernels with **algorithmic parameters**
- Extended platform model
  - SW/HW accelerated kernels and their **implementation parameters**

---

Algorithmic description

- FFT HW ACC
  - Points
  - Data format

---

**Types & parameters**

---

**Latency equations**

---

**Interfacing**

---


---

© Prof. J. Castrillon. Programming CPS and IoT. 2020
Deep dive: TensorDSLs
Tensors: Multi-dimensional arrays

- Tensors are common to different areas: ML, quantum chemistry, physics sims.
- Algorithmic kernels concisely expressed with tensors turn into deep loop nests
  - Challenging to optimize for embedded devices (inference in CPS)
Tensor expressions typically occur in numerical codes

\[ \mathbf{v}_e = (\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A}) \mathbf{u}_e \]

\[ v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'e} \]

- Matrixes are small, so libraries like BLAS do not always help
- Expressions result in deeply nested for-loops
- Performance highly depends on the shape of the loop nests
Lowering → Tensor IR → Codegen

\[
\begin{align*}
\mathbf{v}_e &= (\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A}) \mathbf{u}_e \\
\end{align*}
\]

Example: Interpolation operator

- Interpolation: \( v_e = (A \otimes A \otimes A) u_e \)
  \[
  v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn}
  \]

- Three alternative orders (besides naïve)
  1. \( v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))) \)
  2. \( v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn}) \)
  3. \( v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn})) \)

TeML: Meta-programming for Tensor Optimizations

- Generalization across domains for tensor expressions
- Clean expression language: **Index-free notation**
- **Formal semantics** for correctness and transformations
  - An expression is a Tree (T)
  - Expressions can be implemented as loop-nests (L)
  - The state maps identifiers to either T or L

\[ S = \text{identifier} \rightarrow (T + L) \]

A Susungi, N. A. Rink, A. Cohen, J. Castrillon, C. Tadonki, "Meta-programming for Cross-Domain Tensor Optimizations". GPCE 18,
Recall

\[ S = \text{identifier} \rightarrow (T + L) \]

In TeML

- \( P_{\text{prog}} : \text{program} \rightarrow (S \rightarrow S) \),
- \( P_{\text{stmt}} : \text{stmt} \rightarrow (S \rightarrow S) \),
- \( E_t : \text{Texpression} \rightarrow (S \rightarrow T) \),
- \( E_l : \text{Lexpression} \rightarrow (S \rightarrow L) \).

A Susungi, N. A. Rink, A. Cohen, J. Castrillon, C. Tadonki, "Meta-programming for Cross-Domain Tensor Optimizations". GPCE 18,
TeML: Meta-programming

Formally defined transformation primitives

Higher-level transformations via composition

A Susungu, N. A. Rink, A. Cohen, J. Castrillon, C. Tadonki, "Meta-programming for Cross-Domain Tensor Optimizations". GPCE 18,
TelL: Safe code generation

- Formalized core tensor primitives
- Showed flaws in widespread languages
- Proved no out of bound accesses (using Coq)

\[
C_{ij} = \sum_{k=1}^{h} A_{ki} B_{kj}
\]

A = placeholder((m,h), name='A')
B = placeholder((n,h), name='B')
k = reduce_axis((0, h), name='k')
C = compute((m,n), lambda i, j:
    sum(A[k, i] * B[k, j], axis=k))

N. A. Rink, J. Castrillon, "TelL: a type-safe imperative Tensor Intermediate Language". ARRAY'19
Emerging memories

- Non-volatile memories provide high energy efficiency (think embedded)
- Racetrack-memories provide unprecedented density (embedded inference)


© Prof. J. Castrillon. Programming CPS and IoT. 2020
**Layout optimization: By hand**

Latency comparison vs SRAM

- Un-optimized and naïve mapping: Even worse latency than SRAM
- 24% average improvement (even with very conservative circuit simulation)
Energy comparison vs SRAM

- Higher savings due to less leakage power
- 74% average improvement

A. Ali Khan, N. A. Rink, F. Hameed, J. Castrillon, "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads". In LCTES'19
Compiler optimization

- Recent work on automatic identification of loop patterns

- Working on (easier) optimization from Tensor DSLs

Closing remarks
Summary

- CPS challenges: Heterogeneity, changing HW substrate, interconnectivity, ...
- Background: DSL principles, formal foundations and examples
- Deep-dive: Tensor DSLs in general and at our lab @ TU Dresden
  - Formalization for transformations (enable search space)
  - Formalization for correctness proofs
  - Current work transformations for emerging NVMs
- Role of DSLs (and tools) in CPS programming
  - Productivity boost (specially as coders' backgrounds widens)
  - Correctness of the specification (try to avoid having to debug!)
  - Enabler of more powerful (higher-level) optimizations
References


A. Susungi, N. A. Rink, J. Castrillon, et al "Towards Compositional and Generative Tensor Optimizations", GPCE 17,

A Susungi, N. A. Rink, A. Cohen, J. Castrillon, C. Tadonki, "Meta-programming for Cross-Domain Tensor Optimizations", GPCE 18,


