Machine Learning-Driven Program Transformation to Increase Performance in Heterogeneous Architectures

S. Tamarit, G. Vigueras, M. Carro, J. Mariño

IMDEA Software Institute
and
Technical University of Madrid

10th International Parallel Tools Workshop
HLRS, Stuttgart, October 4-5, 2016

partially funded by the European Comission under FP7-ICT-2014.3.4 contract number 610686

Context and Motivation.

Heterogeneous architectures

- Different computing elements.
- Each of them better suited for a type of computation ⇒ high-performance.
- But programming paradigms differ in each component.
- Increased complexity of development, maintenance.
  - More bugs.
  - Less widespread.
- Programming heterogeneous architectures restricted to a few experts.
  - Hinders widespread adoption.

Context and Motivation.

Heterogeneous architectures

- Different computing elements.
- Each of them better suited for a type of computation ⇒ high-performance.
- But programming paradigms differ in each component.
- Increased complexity of development, maintenance.
    - More bugs.
    - Less widespread.
- Programming heterogeneous architectures restricted to a few experts.
    - Hinders widespread adoption.

Scientific code

- Performance is a must.
- Non-trivial algorithms, code very optimized for target architecture.
- Many existing algorithms, implementations.
- Deal with them in a cost-effective, safe way ⇒ mechanize.

Goal.

Develop a framework for sound, semantics-based program transformation of scientific code in order to improve non-functional characteristics.

Transformation example: from $\mathbf{c} = a\mathbf{v} + b\mathbf{v}$ to $\mathbf{c} = (a + b)\mathbf{v}$.

Initial code

```
float c[N], v[N], a, b;
for (int i = 0; i < N; i++)
  c[i] = a * v[i];

for (int i = 0; i < N;i++)
  c[i] += b * v[i];
```

Final code

```
float k = a + b;
for(i = 0; i < N; i++)
   c[i] = k * v[i];
```

Step by Step.

0 - Original

```
float c[N],v[N],a,b;
for(int i=0;i<N;i++)
  c[i] = a*v[i];

for(int i=0;i<N;i++)
  c[i] += b*v[i];
```

1 - For-Loop Fusion

```
for(int i=0;i<N;i++) {
   c[i] = a*v[i];
   c[i] += b*v[i];
}
```

2 - Aug. Addition

```
for(int i=0;i<N;i++) {
   c[i] = a*v[i];
   c[i] = c[i] + b*v[i];
}
```

3 - Join Assignments

```
for(int i=0;i<N;i++)
  c[i] = a*v[i]+b*v[i];
```

4 - Undo Distribute

```
for(int i=0;i<N;i++)
   c[i] = (a+b) * v[i];
```

5 -Inv. Code Motion

```
float k = a + b;
for(int i=0;i<N;i++)
   c[i] = k * v[i];
```

Goal (Cont.).

Capture and respect common properties

Initial code

```
Complex c[N], v[N],  a, b;

for (int i = 0; i < N; i++)
  cmp_mult(v[i], a, c[i]);

Complex aux;

for (int i = 0; i < N;i++) {
  cmp_mult(b, v[i], aux);
  cmp_add(aux, c[i], c[i]);
 }
```

Final code

```
Complex c[N], v[N],  a, b;

Complex k;

cmp_add(a, b, k);

for (int i = 0; i < N;i++)
  cmp_mult(k, v[i], c[i]);
```

Points to Address.

- **Sound.**
  - Respect functional properties.

Points to Address.

- **Sound.**
    - Respect functional properties.

- **Extensible.**
    - Different architectures.
    - Different domains.

Points to Address.

- **Sound.**
    - Respect functional properties.
- **Extensible.**
    - Different architectures.
    - Different domains.
- **Targeted.**
    - Improve (certain) non-functional properties (which depend on architecture, aims).

Points to Address.

- **Sound.**
    - Respect functional properties.

Use syntactic and **semantic** properties

- **Extensible.**
    - Different architectures.
    - Different domains.
- **Targeted.**
    - Improve (certain) non-functional properties (which depend on architecture, aims).

Points to Address.

- **Sound.**
  - Respect functional properties.

Use syntactic and **semantic** properties

- **Extensible.**
  - Different architectures.
  - Different domains.

Use (sets of) rules defining transformations

- **Targeted.**
  - Improve (certain) non-functional properties (which depend on architecture, aims).

Points to Address.

- **Sound.**
  - Respect functional properties.

Use syntactic and **semantic** properties

- **Extensible.**
  - Different architectures.
  - Different domains.

Use (sets of) rules defining transformations

- **Targeted.**
  - Improve (certain) non-functional properties (which depend on architecture, aims).

Search among applicable rules

Points to Address.

- **Sound.**
  - Respect functional properties.

Use syntactic and **semantic** properties

- **Extensible.**
  - Different architectures.
  - Different domains.

Use (sets of) rules defining transformations

- **Targeted.**
  - Improve (certain) non-functional properties (which depend on architecture, aims).

~~Search~~ **Select** applicable rule

Points to Address.

How to. . .

- **Sound.**
  - Respect functional properties.

Use syntactic and **semantic** properties

. . . infer / capture these properties?

- **Extensible.**
  - Different architectures.
  - Different domains.

Use (sets of) rules defining transformations

- **Targeted.**
  - Improve (certain) non-functional properties (which depend on architecture, aims).

~~Search~~ **Select** applicable rule

Points to Address.

How to. . .

- **Sound.**
  - Respect functional properties.

Use syntactic and **semantic** properties

. . . infer / capture these properties?

- **Extensible.**
  - Different architectures.
  - Different domains.

Use (sets of) rules defining transformations

. . . express and apply these transformations in an extensible way?

- **Targeted.**
  - Improve (certain) non-functional properties (which depend on architecture, aims).

~~Search~~ **Select** applicable rule

Points to Address.

How to. . .

- **Sound.**
  - Respect functional properties.

Use syntactic and **semantic** properties

. . . infer / capture these properties?

- **Extensible.**
  - Different architectures.
  - Different domains.

Use (sets of) rules defining transformations

. . . express and apply these transformations in an extensible way?

- **Targeted.**
  - Improve (certain) non-functional properties (which depend on architecture, aims).

~~Search~~ **Select** applicable rule

. . . identify the right rule(s)?

Rules: Extensible Program Transformations

Example: Undo Distributive Property.

```
for(int i=0;i<N;i++)
  c[i] = a*v[i]+b*v[i];
```

```
for(int i=0;i<N;i++)
   c[i] = (a+b) * v[i];
```

Transformation rule:

```
undo_distributive {
   pattern: {
     (cexpr(e1) * cexpr(e2)) + (cexpr(e1) * cexpr(e3));
   }
   condition: {
     pure(cexpr(e1));
     pure(cexpr(e2));
     pure(cexpr(e3));
   }
   generate: {
     cexpr(e1) * (cexpr(e2) + cexpr(e3));
   }
}
```

Example: Undo Distributive Property.

```
for(int i=0;i<N;i++)
  c[i] = a*v[i]+b*v[i];
```

```
for(int i=0;i<N;i++)
   c[i] = (a+b) * v[i];
```

Transformation rule:

```
undo_distributive {
   pattern: {
     (cexpr(e1) * cexpr(e2)) + (cexpr(e1) * cexpr(e3));
   }
   condition: {
     pure(cexpr(e1));
     pure(cexpr(e2));
     pure(cexpr(e3));
   }
   generate: {
     cexpr(e1) * (cexpr(e2) + cexpr(e3));
   }
}
```

Syntactical pattern

Example: Undo Distributive Property.

```
for(int i=0;i<N;i++)
  c[i] = a*v[i]+b*v[i];
```

```
for(int i=0;i<N;i++)
    c[i] = (a+b) * v[i];
```

Transformation rule:

```
undo_distributive {
   pattern: {
     (cexpr(e1) * cexpr(e2)) + (cexpr(e1) * cexpr(e3));
   }
   condition: {
     pure(cexpr(e1));
     pure(cexpr(e2));
     pure(cexpr(e3));
   }
   generate: {
     cexpr(e1) * (cexpr(e2) + cexpr(e3));
   }
}
```

Syntactical pattern

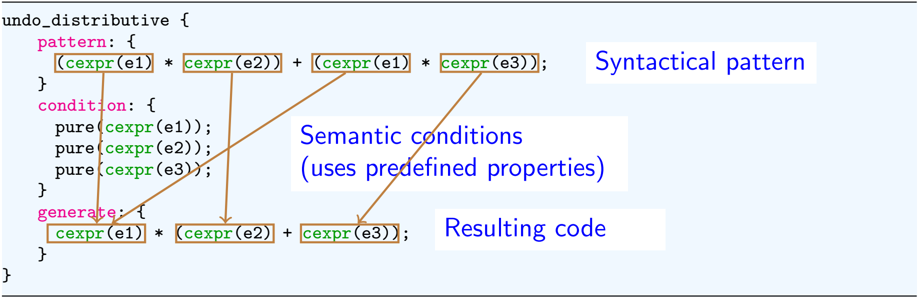
Semantic conditions
(uses predefined properties)

Example: Undo Distributive Property.

```
for(int i=0;i<N;i++)
  c[i] = a*v[i]+b*v[i];
```

```
for(int i=0;i<N;i++)
   c[i] = (a+b) * v[i];
```

Transformation rule:



```
undo_distributive {
   pattern: {
    (cexpr(e1) * cexpr(e2)) + (cexpr(e1) * cexpr(e3));
   }
   condition: {
    pure(cexpr(e1));
    pure(cexpr(e2));
    pure(cexpr(e3));
   }
   generate: {
    cexpr(e1) * (cexpr(e2) + cexpr(e3));
   }
}
```

Syntactical pattern

Semantic conditions
(uses predefined properties)

Resulting code

Example: Invariant Code Motion.

```
for(int i=0;i<N;i++)
   c[i] = (a+b) * v[i];
```

```
float k = a + b;
for(int i=0;i<N;i++)
   c[i] = k * v[i];
```

Transformation rule (simplified):

```
loop_inv_code_motion{
   pattern:{
      for (cexpr(ind) = cexpr(ini); cexpr(cond); cexpr(mod)){
         cexpr(e3) = cexpr(e1) * cexpr(e2); }
   }
   condition:{
      pure(cexpr(e2));
      no_reads(cexpr(e2), cexpr(ind));
      no_reads(cexpr(e2), cexpr(e3));
      no_reads_in_written(cexpr(e2), cexpr(mod));
      no_reads_in_written(cexpr(e2), cexpr(e1));
      no_reads_in_written(cexpr(e2), cexpr(e3));
   }
   generate:{
      cdecl(ctype(cexpr(e2)),cexpr(aux));
      cexpr(aux) = cexpr(e2);
      for (cexpr(ind) = cexpr(ini); cexpr(cond); cexpr(mod)){
         cexpr(e3) = cexpr(e1) * cexpr(aux);}
   }
}
```

Syntactical pattern

Semantic conditions (uses predefined properties)

Resulting code

The STML Rule Language.

```
rule_name {
   pattern:  {...}
   condition: {...}
   generate: {...}
}
```

- Inspired by CML [BrownLukKelly 2005], an evolution of CTT [BoekholdKarkowskiCorporaal 1999].
- STML rules expressed in a subset of C.
- Sections:
  - `pattern`:
    - Matches (localizes) code.
    - Meta-expressions (`cexpr`, `cstmt`, ...) substituted by actual symbols before transformation.
  - `condition`:
    - Code properties required by rule (soundness).
  - `generate`: New code, replaces matched section.
  - Not shown: (new) properties of generated code.

STML Constructs.

For all sections

Construct	Description
bin_op(E_{op},E_1,E_2)	represents the binary operation E_1 E_{op} E_2
una_op(E_{op},E)	represents the unary operation E_{op} E

In the condition section

Construct	Description
is_identity(E_{op},E)	E is identity element for E_{op}
no_writes(E_v,(S|[S]|E))	E_v is not written onto in (S|[S]|E)
no_reads(E_v,(S|[S]|E))	E_v is not read from in (S|[S]|E)
no_rw(E_v,(S|[S]|E))	E_v is neither read from nor written onto in (S|[S]|E)
pure((S|[S]|E))	There are no assignments in (S|[S]|E)
is_const(E)	There are variables inside E
is_block(S)	S is a block of statements
is_commutative(E_{op})	Operation E_{op} is commutative
is_associative(E_{op})	Operation E_{op} is associative
not(E_{cond})	E_{cond} is false

STML Constructs (Cont.).

In the generate section

<table>
<tr><th>Construct</th><th>Description</th></tr>
<tr><td>subs((S|[S]|E),$\mathrm{E_f}$,$\mathrm{E_t}$)</td><td>Replace each occurrence of $\mathrm{E_f}$ in (S|[S]|E) for $\mathrm{E_t}$</td></tr>
<tr><td>if_then:{$\mathrm{E_{cond}}$;(S|[S]|E);}</td><td>If $\mathrm{E_{cond}}$ is true, then generate (S|[S]|E)</td></tr>
<tr><td>if_then_else:{$\mathrm{E_{cond}}$; $\mathrm{(S|[S]|E)_t}$;$\mathrm{(S|[S]|E)_e}$;}</td><td>If $\mathrm{E_{cond}}$ is true, then generate $\mathrm{(S|[S]|E)_t}$, else generate $\mathrm{(S|[S]|E)_e}$</td></tr>
<tr><td>gen_list:{[(S|[S]|E)];}</td><td>Each statement/expression in [(S|[S]|E)] produces a different rule consequent</td></tr>
</table>

Extracting Code Properties

Inferring and Expressing Properties.

```
loop_inv_code_motion{
   pattern:{...}
   condition:{
      pure(cexpr(e2));
      no_reads(cexpr(e2), cexpr(ind));
      no_reads(cexpr(e2), cexpr(e3));
      no_reads_in_written(cexpr(e2), cexpr(mod));
      no_reads_in_written(cexpr(e2), cexpr(e1));
      no_reads_in_written(cexpr(e2), cexpr(e3));
   }
   generate:{...}
}
```

1. Which properties are needed?
2. How can we determine whether they hold?

1. Low-level properties, capture characteristics of imperative languages.
   - Destructive assignment & its effects.
   - Aliasing.
   - Memory management
2. Obtained from:
   - Automatic program analysis.
   - User-provided pragmas.

Program Analysis.

- Cetus: source-to-source C compiler written in Java.
- Extensive set of compiler passes working on a high-level IR.
- Analyses and transformations:
  - Dependence analysis.
  - Loop parallelizer.
  - Source program in canonical form.
  - Loop outlining (procedural abstraction of loops).
- Read and Write pragmas (OpenMP and STML — explained later)
- Modifications to Cetus:
  - Generation of STML annotations.
  - Rewriting pass to adapt input code to Cetus (C99).
  - Modified Artistic License $\Longrightarrow$ change Cetus without restrictions.

STML Pragmas.

- Not all properties always automatically inferred.
- `#pragma`s in code.

```
#pragma stml writes c in {0}
for (i = 0; i < N; i++)
  c[i] = i*2;

#pragma stml writes c in {-1,0}
for (i = 1; i < N; i++) {
  c[i-1] = i;
  c[i]   = c[i-1] * 2;
}
```

```
#pragma stml reads c in {-1,0,+1}
for (i = 0; i < N; i++)
  a += c[i-1]+c[i+1]-2*c[i];
```

```
#pragma stml iteration_space 0 N-1
for (i = 0; i < N; i++)
  c[i] = i*2;
```

High-Level Annotations.

- Stemming from POLCA project
- Functional programming flavor.
- Capture algorithmic skeletons:
  - Summarily capture properties of underlying code.
  - Also, can help determine transformation strategies.

```
#pragma polca map F v w

#pragma polca fold F INI v e

#pragma polca itn F INI n w

#pragma polca zipWith F u v w

#pragma polca scanl F INI v w
```

```
#pragma polca map B v c
for(int i=0;i<N;i++)
#pragma polca def B
#pragma polca input v[i]
#pragma polca output c[i]
   c[i] = a*v[i];
```

Information from Annotations.

What map tells us

```
#pragma polca map B v c
for(int i=0;i<N;i++)
#pragma polca def B
#pragma polca input v[i]
#pragma polca output c[i]
   c[i] = a*v[i];
```

- `v` is input to `map`, `c` is output.
- For every `v[i]`, `c[i]` is produced using only `v[i]`.
  - No global variables, no dependencies across iterations.
- Computation of `c[i]` enclosed inside `B`.

⇒ Pragma as summary of simpler properties.

⇒ These are dealt with by rules in program transformation tool.

Translating High-level Annotations to STML.

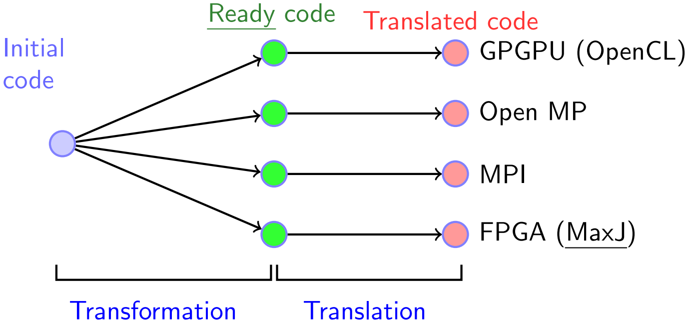
STML properties can be inferred for high-level annotations.

```
#pragma polca map B v c
for(int i=0;i<N;i++)
#pragma polca def B
#pragma polca input v[i]
#pragma polca output c[i]
   c[i] = a*v[i];
```

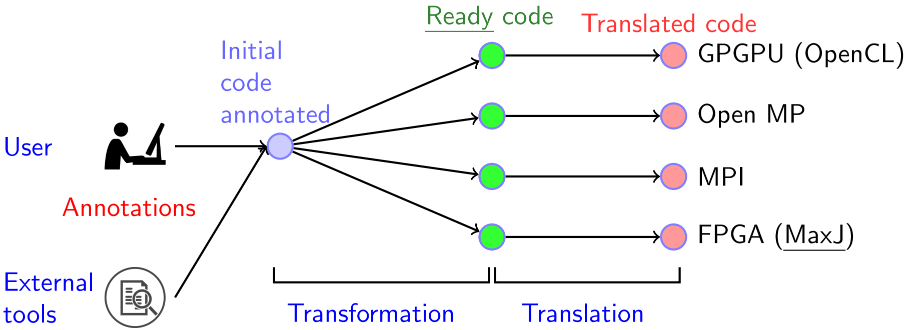
→

```
#pragma stml reads v in {0}
#pragma stml writes c in {0}
#pragma stml same_length v c
#pragma stml pure B
#pragma stml iteration_space 0 length(v)
#pragma stml iteration_independent
for(int i = 0; i < N; i++)
#pragma polca def B
#pragma polca input v[i]
#pragma polca output c[i]
   c[i] = a*v[i];
```

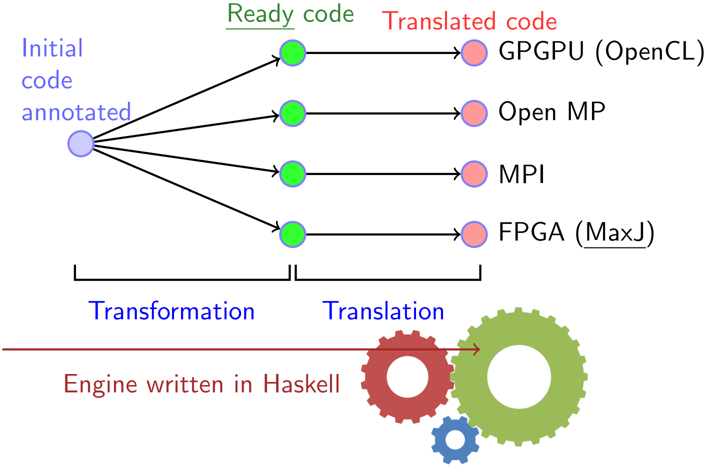
Components of the Transformation Tool.



Components of the Transformation Tool.



Components of the Transformation Tool.



Ready code

Translated code

Initial code annotated

GPGPU (OpenCL)

Open MP

MPI

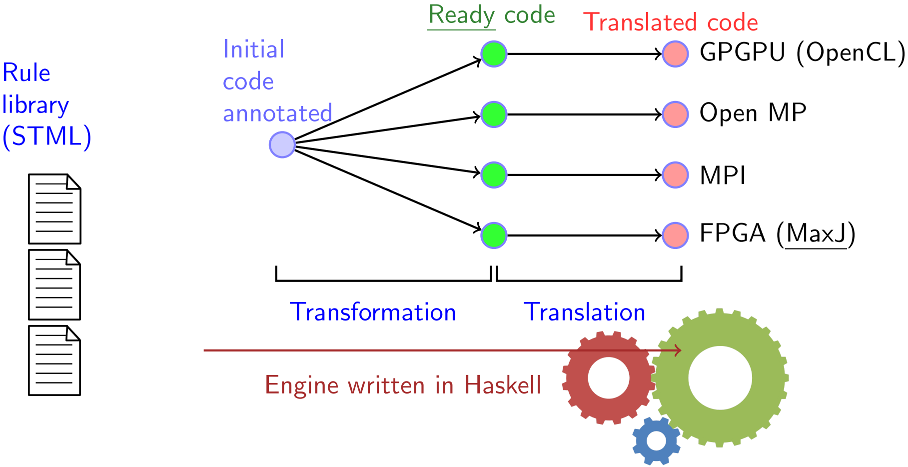
FPGA (MaxJ)

Transformation

Translation

Engine written in Haskell

Components of the Transformation Tool.



Rule library (STML)

Initial code annotated

Ready code

Translated code

GPGPU (OpenCL)

Open MP

MPI

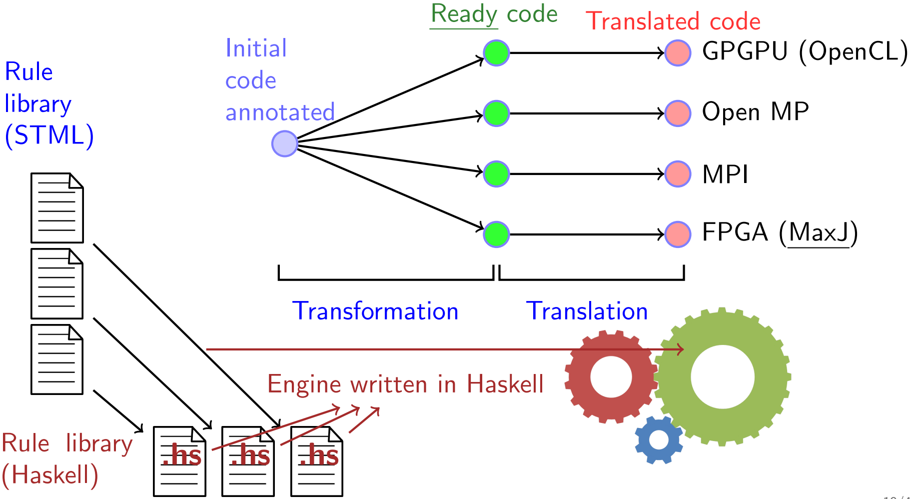
FPGA (MaxJ)

Transformation

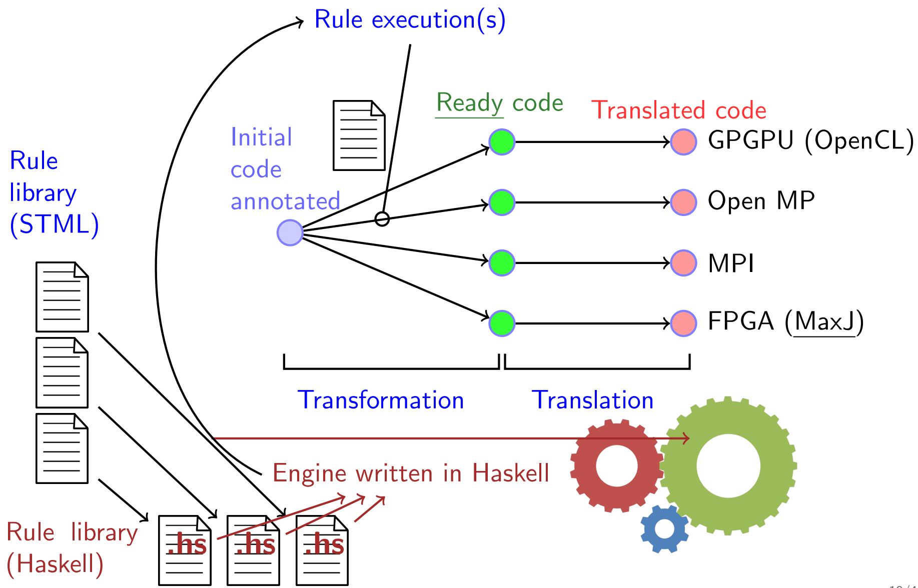
Translation

Engine written in Haskell

Components of the Transformation Tool.



Components of the Transformation Tool.



Rule Testing.

- How to ensure rules are correct?
  - Formal proofs possible (and ideal), but also resource-consuming.
- A set of automatic testers has been developed to ease the checking of the rules behavior.
- Perform several tests over the resulting code of each transformation step.
- Several strategies at the moment:
  - *n* random steps.
  - Random order, using each rule at most once.
  - Similar to last one, backtracking to previous states that could enable non-applied rules.
- Output:
  - Applied rules.
  - Not applied rules:
    - Because patterns or conditions not met.
    - Because they were not chosen (but could be applied).
  - In case of error: The failing rule and the concrete transformation step.

Rule Selection.

- Generally, several rules can be applied at a number of code locations.
- At any point, candidate rules can be:
  - applicable,
  - maybe (not) applicable,
  - definitely not applicable
- Selected rule(s) should improve code.

Interactive Rule Selection

- User chooses the transformation steps to apply.
- Meld and other specific tools (e.g. POGADE) help in this process.
- Useful to refine rules, perform specific transformations, do aggressive program refactoring.
- Scalability an issue.

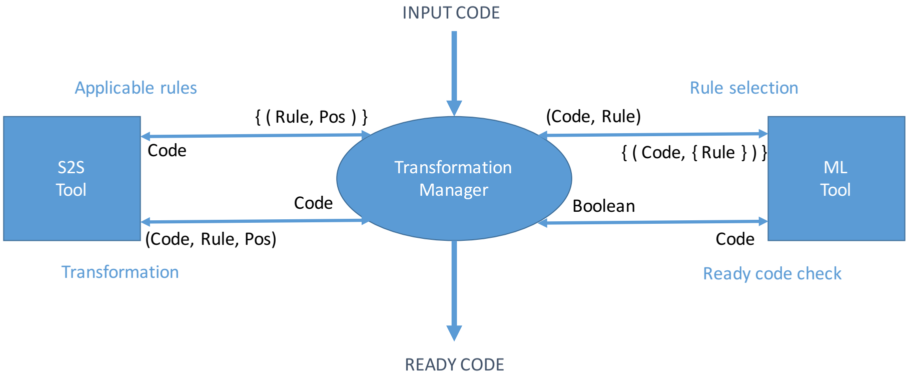
Rule Selection.

- We need a way to mechanize rule selection.
  - Select at each step the rule which reduces some metric → local minima
  - Explore a bounded number of possible rule applications
    - Too small → local minima
    - Too big → exponential explosion
- Additional issues
  - Inverse/involutive rules → infinite loops
  - Rules that duplicate code (e.g. unfolding) → same rule/different states

External Oracle to Select Rules

- Given code and set of applicable transformation steps, return which transformation step should be applied.
- Given code, return whether transformation can be assumed successfully finishd.

Source-to-Source Tool Interface.



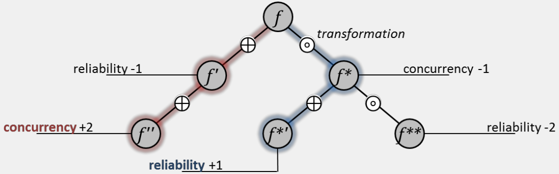
- All the communications are done using JSON.
- The s2s tool interface is also used by POGADE.
- A serialization of the AST, the applicable transformations and other relevant data is done when the s2s interfaces are used.

Machine Learning: Deciding Strategy

Problems to be Addressed.

- Rule-based transformation $\Longrightarrow$ **state-space exploration** problem.
  - Efficiently explore (**prune**) search space.
  - Define a **stop criteria**.
  - Improve **quality** of code.

Possible solution: heuristics to drive the search



- Heuristics which monotonically increase some metric would disallow sequences which temporarily generate code of lesser quality.
- Heuristics should “plan for” whole sequences or at least series of steps.

Non-Local Heuristics and Machine Learning.

- Don't come up with heuristics: likely complex and brittle.
- Synthesize (learn) them from existing, good examples.

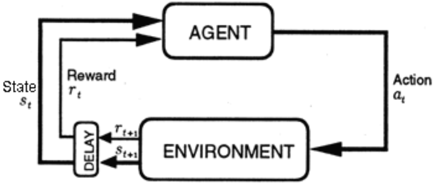
- ML generally operates on descriptions of real world.
- Program abstractions as descriptions of actual programs.
  - Should capture features of states (codes) in the search space.
  - Should reflect the changes performed by actions.
- Based on code features related with:
  - Control flow,
  - Data layout,
  - Data dependencies. . .
  - Also on code annotations (externally provided.)
- Note: previous works apply ML to e.g. compilation.

Abstraction: vector of code features (to be further enriched)

$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{N}$	$\mathbb{B}$	$\mathbb{N}$	$\mathbb{B}$	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$
max. nested loop depth	number of function calls	number of array write shifted within `for` loops	any irregular loop? (e.g., `break`, `continue`)	any global variable	number of `if` statements	`for` loop limits static for all loops?	number of SIMD loop	any `for` loop with `loop-schedule`?	number of loop invariant vars	number of loop hoisted var modifications	number of non-1D array	number of aux. var to access arrays	number of `for` loops	Non-normalized `for` loops

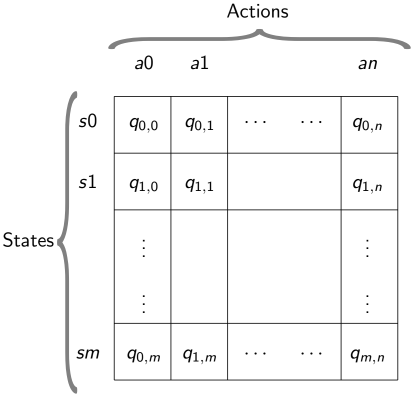
Reinforcement Learning.

- Large area of machine learning.
- Learn from sequences of states and transformations which may contain "counterproductive" steps.
- Reinforcement learning (RL):
  - Well-suited for problems with long-term (vs. short-term) rewards.
- Used by the transformation engine:
  - Reduce search space by applying "good" transformation sequences.
  - Handle sequences with intermediate "bad" steps (delayed reward).



RL Basics.

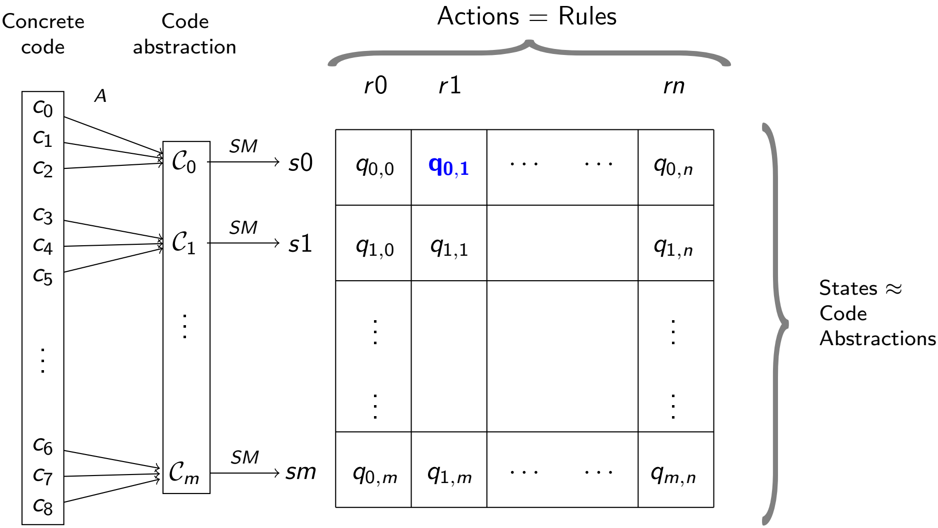
- RL learns from examples.
  - Sequences of stepwise changes from some initial state to some final state.
- Result: action-value matrix $Q(s_t, a_t)$
  - Determines, for each state, the relative profit of applying every action.
  - Action to apply: the one with highest profit.
- Q matrix filled with learning model
  - α: learning rate (new information)
  - γ: discount factor (future rewards)



	$a0$	$a1$		an
$s0$	$q_{0,0}$	$q_{0,1}$	$\cdots$ $\cdots$	$q_{0,n}$
$s1$	$q_{1,0}$	$q_{1,1}$		$q_{1,n}$
	$\vdots$			$\vdots$
	$\vdots$			$\vdots$
sm	$q_{0,m}$	$q_{1,m}$	$\cdots$ $\cdots$	$q_{m,n}$

$$Q(s_t, a_t) = \begin{cases} Q(s_t, a_t) + \alpha \cdot (r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) & \text{if } s_t \textit{ not final} \\ Q_{init}(s_t, a_t) & \text{otherwise} \end{cases}$$

Code, State/Action Table, Rule Selection.



$r0$	$r1$		rn
$q_{0,0}$	$\mathbf{q_{0,1}}$	$\cdots$ $\cdots$	$q_{0,n}$
$q_{1,0}$	$q_{1,1}$		$q_{1,n}$
$\vdots$ $\vdots$			$\vdots$ $\vdots$
$q_{0,m}$	$q_{1,m}$	$\cdots$ $\cdots$	$q_{m,n}$

Search Space: Stopping.

- Stopping encoded in state-action table.
- Determine **stop criteria** for training sequences:
  - Is the current code ready to be translated into the target architecture?
  - If so, is the current code of quality enough?
- One possibility: at each step, check if transformed code can be used to generate correct platform-specific code.
  - Involves calling translator, platform-specific compiler.
  - Not practical – might take an exceedingly long time (e.g., in FPGAs).
  - Can not determine / measure optimality.
- Classification methods learned to classify good states to stop transforming.
  - Not a canonical form, but a shape which looks good enough.

Learning Example.

- A transformation sequence on a simple example.
  - Initial state: 2D convolution kernel.
  - Final code:
    - Well-suited to generate MaxJ (FPGA/DFE) code.
    - Learned by a classifier as a good final state for the translation phase.
  - Abstraction (vector of features) shown for each code state.
  - Transformation steps provided by expert programmer.
  - Code transformed in each step highlighted as:
    - before transformation
    - after transformation

Learning Example (Cont.).

- Initial code

```
// ABSTR: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 3, 2, 4, 0]

int dead_rows = K / 2;
int dead_cols = K / 2;

int normal_factor = K * K;

for (r = 0; r < N - K + 1; r++) {
  for (c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++) {
      for (j = 0; j < K; j++) {
        sum += input_image[r+i][c+j] * kernel[i][j];
      }
    }
    output_image[r+dead_rows][c+dead_cols] = (sum / normal_factor);
  }
 }
```

Learning Example (Cont.).

- Step 1 (Rule: ArrayFlatten)

```
// ABSTR: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 2, 2, 4, 0]

int dead_rows = K / 2;
int dead_cols = K / 2;

int normal_factor = K * K;

for (r = 0; r < N - K + 1; r++) {
  for (c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++) {
      for (j = 0; j < K; j++) {
        sum += input_image[(r+i)*(N - K + 1) + (c+j)] * kernel[i][j];
      }
    }
    output_image[r+dead_rows][c+dead_cols] = (sum / normal_factor);
  }
 }
```

Learning Example (Cont.).

- Step 2 (Rule: ArrayFlatten)

```
// ABSTR: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 2, 4, 0]

int dead_rows = K / 2;
int dead_cols = K / 2;

int normal_factor = K * K;

for (r = 0; r < N - K + 1; r++) {
  for (c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++) {
      for (j = 0; j < K; j++) {
        sum += input_image[(r+i)*(N - K + 1) + (c+j)] * kernel[i*K+j];
      }
    }
    output_image[r+dead_rows][c+dead_cols] = (sum / normal_factor);
  }
 }
```

Learning Example (Cont.).

- Step 3 (Rule: ArrayFlatten)

```
// ABSTR: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 2, 4, 0]

int dead_rows = K / 2;
int dead_cols = K / 2;

int normal_factor = K * K;

for(r = 0; r < N - K + 1; r++) {
  for(c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++) {
      for (j = 0; j < K; j++) {
        sum += input_image[(r+i)*(N - K + 1) + (c+j)] * kernel[i*K+j];
      }
    }
    output_image[(r+dead_rows)*(N-K+1) + (c+dead_cols)] = (sum / normal_factor);
  }
 }
```

Learning Example (Cont.).

- Step 4 (Rule: CollapseFoorLoops)

```
// ABSTR: [2, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 2, 3, 0]

int dead_rows = K / 2;
int dead_cols = K / 2;

int normal_factor = K * K;

for (z = 0; z < (N - K + 1)*(N - K + 1); z++) {
    sum = 0;
    for (i = 0; i < K; i++) {
      for (j = 0; j < K; j++) {
        sum += input_image[((z / (N - K + 1))+i)*(N - K + 1) + ((z % (N - K +
              1))+j)] * kernel[i*K+j];
      }
    }
    output_image[((z / (N - K + 1))+dead_rows)*(N - K + 1) +
          ((z % (N - K + 1))+dead_cols)] = (sum / normal_factor);
}
```

Learning Example (Cont.).

- Mapping of states (S) to abstractions (A)

$S_0 = A(C_0)$: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 3, 2, 4, 0] (Initial)
$S_1 = A(C_1)$: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 2, 2, 4, 0]
$S_2 = A(C_2)$: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 2, 4, 0]
$S_3 = A(C_3)$: [3, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 2, 4, 0]
$S_4 = A(C_4)$: [2, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 2, 3, 0] (Goal)

- Expert sequence (transition matrix)

	R_0	R_1
S_0	S_1	S_0
S_1	S_2	S_1
S_2	S_3	S_2
S_3	S_3	S_4
S_4	S_4	S_4

- Initial action-value table (Q)

	R_0	R_1	Best
C_0	1.0	1.0	-
C_1	1.0	1.0	-
C_2	1.0	1.0	-
C_3	1.0	1.0	-
C_4	1.0	1.0	-

Learning Example (Cont.).

- Simple training: 10 iterations, 3 interactions/iteration
- Learning params: $\alpha = 0.5$, $\gamma = 0.5$, $r_{t+1} = 1$ (final) | 0 (not final)

Iteration 1

	R_0	R_1	Best
C_0	0.9955	1.0	R_1
C_1	0.995	1.0	R_1
C_2	1.0	1.0	-
C_3	1.0	1.0	-
C_4	1.0	1.0	-

Iteration 3

	R_0	R_1	Best
C_0	0.9955	0.9881	R_0
C_1	0.995	1.0	R_1
C_2	1.0	1.0	-
C_3	1.0	1.4434	R_1
C_4	1.0	1.0	-

Iteration 5

	R_0	R_1	Best
C_0	0.9955	0.9881	R_0
C_1	0.9925	0.9955	R_1
C_2	1.2145	1.0910	R_0
C_3	1.0	1.4434	R_1
C_4	1.0	1.0	-

Iteration 10

	R_0	R_1	Best
C_0	1.1490	0.9881	R_0
C_1	1.2216	0.9905	R_0
C_2	1.5942	1.0910	R_0
C_3	0.995	1.9466	R_1
C_4	1.0	1.0	-

Implementation Details.

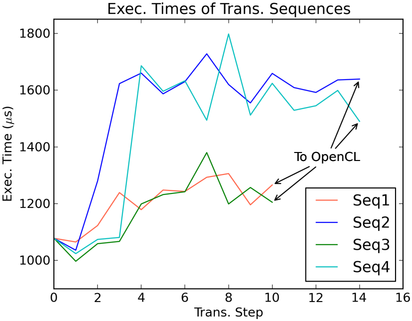
- Implemented in Python using two packages
- Scikit-learn: classification methods
  - Several machine learning algorithms
  - Good support and ample documentation
  - Widely used by the scientific community.
- PyBrain: reinforcement learning
  - Modular structure in classes (environment, actions, ...)
  - Extension of PyBrain classes to implement our approach

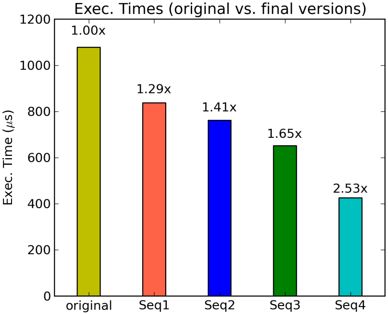
Preliminary Evaluation.

- Using cases from POLCA and UTDSP benchmark suite as training set.
- Featuring different algorithmic patterns (image processing):
  - Compress (JPG compression).
  - Edge detection (convolution).
  - RGB filter (image filter).
  - Threshold (image filter).
- Manually identified transformation sequences leading to **OpenCL**
- Simple **reward scheme** linked to efficiency (much higher for the best code).
- Evaluation using different examples, sharing patterns with training set:
  - 3D rotation.
  - Image difference.
  - Brightness change.
- Different aspects were satisfactorily evaluated:
  - Transformation process terminates, “good” final states reached.
  - RL learned different sequences which might share intermediate steps.
  - Transformations can start at any point in the learned sequences

Preliminary Evaluation.

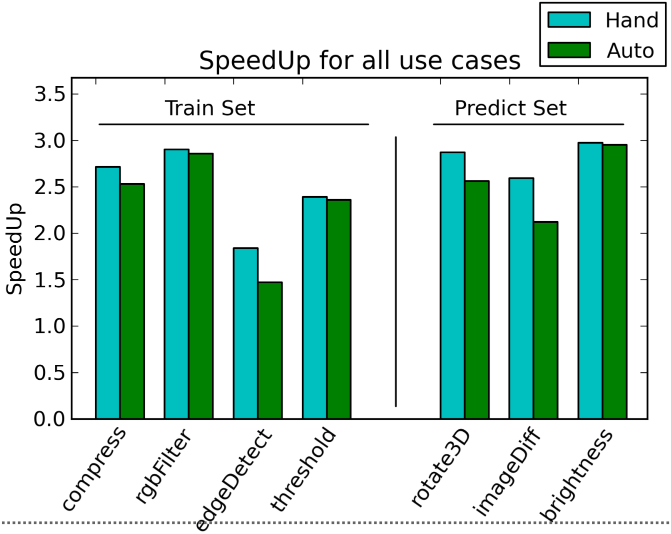
- Results to show the non-monotonic behavior of transformation sequences





Preliminary Evaluation.

- Evaluation of OpenCL code mechanically generated from learnt transformation sequences



Conclusions.

- Extensible, flexible rule-based framework for program transformation.
- Interface to external tools to select transformations to be enacted.
- Using machine learning-based oracle to guide rule selection.
- Preliminary evaluation (more around the corner!) satisfactory.

Future Steps.

- Add more (complex) transformation rules, evaluate on larger examples.
- Add more interfaces to external analysis tools.
  - Dependence analysis (e.g., polytope-based analyzers).
  - Reasoning over heap pointers (e.g., separation logic).
- Integrate profiling techniques → ease evaluation and give feedback.
- Increase training set, enhance code abstractions as needed.
- Use richer, non-functional measurements for RL.
  - Measures of optimality to better discriminate between possible final states based on performance.
- Explore using ML to decide best platform for a piece of code
  - E.g., use a single action-value table with several goals, select the best platform among the most reinforced sequences

Machine Learning-Driven Program Transformation to Increase Performance in Heterogeneous Architectures

S. Tamarit, G. Vigueras, M. Carro, J. Mariño

IMDEA Software Institute
and
Technical University of Madrid

10th International Parallel Tools Workshop
HLRS, Stuttgart, October 4-5, 2016

partially funded by the European Comission under FP7-ICT-2014.3.4 contract number 610686