Towards a Semantics-Aware Code Transformation Toolchain for Heterogeneous Systems[*]

Salvador Tamarit   Julio Mariño   Guillermo Vigueras   Manuel Carro

Universidad Politécnica de Madrid
Campus de Montegancedo 28660
Boadilla del Monte, Madrid, Spain
{salvador.tamarit,julio.marino}@upm.es

IMDEA Software Institute
Campus de Montegancedo 28223
Pozuelo de Alarcón, Madrid, Spain
{guillermo.vigueras,manuel.carro}@imdea.org

Obtaining good performance when programming heterogeneous computing platforms poses significant challenges. We present a program transformation environment, implemented in Haskell, where architecture-agnostic scientific C code with semantic annotations is transformed into functionally equivalent code better suited for a given platform. The transformation steps are represented as rules which can be fired when certain syntactic and semantic conditions are fulfilled. These rules are not hard-wired into the rewriting engine: they are written in a C-like language and are automatically processed and incorporated by the rewriting engine. That makes it possible for end-users to add their own rules or to provide sets of rules which are adapted to certain specific domains or purposes.

Keywords: Rule-based program transformation, Semantics-aware program transformation, High-performance, Heterogeneous platforms, Scientific computing, Domain-specific language, Haskell, C.

1   Introduction

There is a strong trend in high-performance computing towards the integration of heterogeneous computing elements: vector processors, GPUs, FPGAs, etc. Each of these components is specially suited for some class of computations, which makes the resulting platform able to excel in performance by mapping computations to the unit best suited to execute them. Such platforms are proving to be a cost-effective alternative to more traditional supercomputing architectures [8, 17] in terms of performance and energy consumption. However, this specialization comes at the price of additional hardware and, notably, software complexity. Developers must take care of very different features to make the most of the underlying computing infrastructure. Thus, programming these systems is restricted to a few experts, which hinders its widespread adoption, increases the likelihood of bugs, and greatly limits portability. For these reasons, defining programming models that ease the task of efficiently programming heterogeneous systems has become a topic of great relevance and is the objective of many ongoing efforts (for example, the POLCA project http://polca-project.eu, which focuses on scientific applications).

Scientific applications sit at the core of many research projects of industrial relevance which require, for example, simulating physical systems or to numerically solve differential equations. One distinguishing characteristic of many scientific applications is that they rely on a large base of existing algorithms. These algorithms often need to be ported to new architectures and exploit their computational strengths to the limit, while avoiding pitfalls and bottlenecks. Of course, these new versions have to to preserve the functional properties of the original code. Porting is often carried out by transforming or replacing

---

Contribution to:
PROLE 2016

0 - original code	1 - FOR-LOOPFUSION	2 - AUGADDITIONASSIGN

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

3 - JOINASSIGNMENTS	4 - UNDODISTRIBUTE	5 - LOOPINVCODEMOTION
`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];`	`float k = a + b;` `for(int i=0;i<N;i++)` `  c[i] = k * v[i];`

Figure 1: A sequence of transformations of a piece of C code to compute $\mathbf{c} = a\mathbf{v} + b\mathbf{v}$.

certain fragments of code to improve their performance in a given architecture while preserving their semantics. Unfortunately, (legacy) code often does not clearly spell its meaning or the programmer's intentions, although scientific code usually follows patterns rooted in its mathematical origin.

Our goal is to obtain a framework for the transformation of (scientific), architecture-agnostic C code. The framework should be able to transform existing code into a functionally equivalent program, only better suited for a given platform. Despite the broad range of compilation and refactoring tools available [2, 25, 21], no existing tool fits our needs by being flexible enough to flexibly recognize specific source patterns and generate code better adapted to different architectures (Section 2), so we decided to implement our own transformation framework. Its core is a code rewriting engine, written in Haskell, which works at the *abstract syntax tree* (AST) level. The engine executes transformation rules written in a C-like, domain-specific language (STML, inspired by CTT [4] and CML [6]). This makes understanding the meaning of the rules and defining additional rulesets for specific domains or targets easy for C programmers.

The tool does not have hard-wired strategies to select which rules are the most appropriate for each case. Instead, it is modularly designed to use external oracles that help in selecting which rules have to be applied. In this respect, we are developing human interfaces and machine learning-based tools which advice on the selection of the most promising transformation chain(s) [24]. The tool also includes an interactive mode to allow for more steering by expert users. When code deemed adequate for the target architecture is reached, it is handed out to a *translator* in charge of adapting it to the programming model of the target platform.

Fig. 1 shows a sample code transformation sequence, containing an original fragment of C code along with the result of stepwise applying a number of transformations. Although the examples presented in this paper are very simple, the tool is able to transform much more complex, real-life code, including code with nested loops (both for collapsing them in a single loop and creating them from non-nested loops), code inlining, and others. Some of these transformations are currently done by existing optimizing compilers. However, they are usually performed internally, at the *intermediate representation* (IR) level, and without any possibility for user intervention or tailoring, which falls short to cater for many relevant situations which we want to address:

- Most compilers are designed to work with minimal (if any) external intervention. While this situation is optimal when it can be applied, in many cases static analysis cannot discover the underlying properties that a programmer knows. For example, in Fig. 1, a compiler would rely on native knowledge of the properties of multiplication and addition. If these operations were
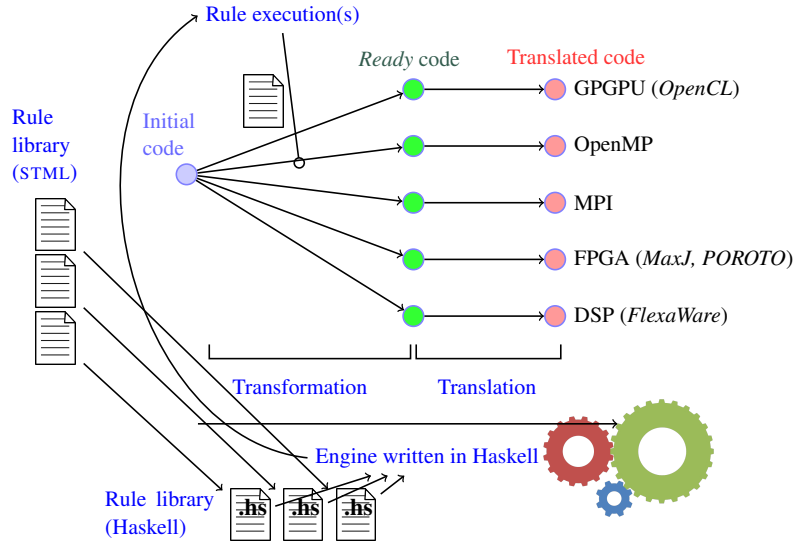
Figure 2: Architecture of the transformation tool.

substituted by calls to functions implementing operations with the same properties (distributivity, associativity, commutativity), such as operations on matrices, the transformation presented would be feasible but unlikely to be performed by a compiler relying solely on static analysis.

- Most compilers have a set of *standard* transformations which are useful in common cases for usual architectures — usually Von Neumann–based CPU architectures. However, when CPU-generic code is to be adapted for a specific architecture (e.g., FPGA, GPGPU) the transformations to be made are not trivial and fall outside those usually implemented in standard compilers. Even more, compilers such as ROCCC [14], which accept a subset of the C language and generate executables or lower-level code for a specific architecture, need the input code to follow specific coding patterns, which our tool can help generate.

- Transformations to generate code amenable to be compiled down to some hybrid architecture can be sometimes complex and are better expressed at a higher level rather than inside a compiler's architecture. That could require users to come up with transformations which are better suited for a given coding style or application domain. Therefore, giving programmers the possibility of defining transformations at a higher level as plugins for a compiler greatly enlarges the set of scenarios where automatic program manipulation can be applied.

Fig. 2 shows an overview of the tool, designed to work in two stages: a *transformation* phase (Section 3) and a *translation* phase (Section 4). The transformation phase rewrites the original input code into a form which follows coding patterns closer to what is appropriate for the destination architecture. That code can be given to compilers which accept C code adapted to the targeted architecture [14]. Additionally, this transformation phase can be used to other purposes, such as sophisticated code refactoring. The translation phase converts transformed code into code which can be compiled for the appropriate architecture by tools which do not accept (sequential) C code. For example, in our case MaxJ [18] code can be generated, as well as C code with OpenMP annotations or with MPI calls.

Our efforts have focused so far on the transformation phase [22]. Our initial work on the translation phase shows encouraging results and points to next steps which we present in more detail in Section 6.

2   Related Work

Some related approaches generate code from a mathematical model (automatic code synthesis), while others use (mathematical) properties to transform existing code. The former can in many cases generate underperforming code because of its generality. The latter usually requires that the initial code is in some "canonical" form.

An example of code generation based on mathematical specifications is [11], which focuses on synthesizing code for matrix operations. The starting point is a mathematical formula which is transformed (not automatically) using rewriting rules to generate another formula which can be implemented in a hopefully more efficient way. This kind of approaches are often very domain-dependent and restricted to a certain kind of formulas or mathematical objects, which makes their application to general domains not straightforward, if possible at all. Given that the starting point is a mathematical formula, applying them to legacy code is not easy. Also, their code generation is usually based on composing blocks which correspond to patterns for the basic formulas, where inter-optimization is often not exercised.

There are some language-independent transformation tools that share some similarities with our approach. The most relevant ones are Stratego-XT [25], TXL [7], DMS [3] and Rascal [15]. Stratego-XT is more oriented to strategies than to rewriting rules, and its rule language is too limited for our needs (e.g., rule firing does not depend on semantic conditions that express when applying a rule is sound). This may be adequate for a functional language which features referential transparency, but not for a procedural language. Besides, it is not designed to add analyzers as plugins, it does not support pragmas or keeps a symbol table, which means that, for some cases, it is not possible to decide whether a given rule can be soundly applied. The last two disadvantages are shared with TXL. DMS is a powerful, industrial transformation tool that is not free and there is not too much information of how it works internally; its overall open documentation is scarce. Since it transforms programs by applying rules until a fix point is reached, the rules should be carefully defined to ensure that they do not produce loops in the rewriting stage. Finally, Rascal is still in alpha state and only available as binary, and the source code is not freely and immediately accessible.

CodeBoost [2], built on top of Stratego-XT, performs domain-specific optimizations to C++ code following an approach similar in spirit to our proposal. User-defined rules specify domain-specific optimizations; code annotations are used as preconditions and inserted as postconditions during the rewriting process. However, it is a mostly abandoned project which additionally mixes C++, the Stratego-XT language, and their rule language, which causes it to have a steep learning curve. Concept-based frameworks such as Simplicissimus [21] transform C++ based on user-provided algebraic properties. The rule application strategy can be guided by the cost of the resulting operation. This cost is defined at the expression level (and not at the statement level), which makes its applicability limited. Besides, their cost is defined using "arbiters" which do not have a global view of the transformation process, which makes it possible to become trapped in local minima (Sec. 3.4.2).

Handel-C [6] performs program transformations to optimize parallelism in its compilation into a FPGA. It is however focused on a subset of C enriched with extensions to express synchronicity, and therefore some of its assumptions are not valid in more general settings.

Other systems lay between both approaches. They use a high-level (declarative) language with some syntactical / semantic restrictions that is compiled down to a lower level formalism [1, 10]. While successful for their goals, they cannot directly tackle the problem of adapting existing code.

Most compilers have an internal phase to perform code transformation, commonly at the IR level. Among the well-known open-source compilers, CLang / LLVM probably has the better designed libraries / APIs to perform program manipulation. However, they were designed for compilation instead of for

```
#pragma polca scanl F INI v w
#pragma stml reads output(INI)
#pragma stml reads v in {0}
#pragma stml reads w in {0}
#pragma stml writes w in {1}
#pragma stml pure F
#pragma stml iteration_space 0 length(v)
```

```
#pragma polca map F v w
#pragma stml reads v in {0}
#pragma stml writes w in {0}
#pragma stml same_length v w
#pragma stml pure F
#pragma stml iteration_space 0 length(v)
#pragma stml iteration_independent
```

```
#pragma polca zipWith F v w z
#pragma stml reads v in {0}
#pragma stml reads w in {0}
#pragma stml writes z in {0}
#pragma stml same_length v w
#pragma stml same_length v z
#pragma stml pure F
#pragma stml iteration_space 0 length(v)
#pragma stml iteration_independent
```

```
#pragma polca fold F INI v a
#pragma stml reads v in {0}
#pragma stml reads output(INI)
#pragma stml writes a
#pragma stml pure F
#pragma stml iteration_space 0 length(v)
```

Table 1: Annotations used in the POLCA project and their translation into STML annotations.

source-to-source program transformation. We tried using them for our purposes and we determined that is not easy or even the best option. Moreover, the design documents warn that the interface can not be assumed to be stable. Additionally, code transformation routines had to be coded in C++, which made these routines verbose, full of low-level details, and their writing error-prone. Compiling rules to C++ is of course an option, but the conceptual distance between the rules and the (unstable) code manipulation API was quite large. That pointed to the a difficult compilation stage which would need considerable maintenance. Even in that case, the whole Clang project would have to be recompiled after introducing new rules, which made project development and testing cumbersome, and would make adding user-defined rules complicated.

3   Source-to-Source Transformations

The code transformation tool has two main components: a parser which reads the input program and the transformation rules and translates them into Haskell for speed, and an engine to perform source-to-source C code transformation using these rules.

The transformation rules contain patterns which have to syntactically match input code and describe the skeleton of the code to generate. They can specify, if necessary, conditions to ensure that their application is sound. These conditions are checked using a combination of static analysis and user-provided annotations (*pragmas*) in the source code with which the programmer provides additional information. The annotations can capture properties at two different levels: high-level properties which describe algorithmic structures and low-level properties which describe details of the procedural code. The decision of whether to apply a given rule depends on two main factors:

- Its application must be sound. This can be checked with the AST in simple cases. Otherwise, whether a rule is applicable or not can be decided based on information inferred from annotations in the source code. These annotations may come from external tools, such as static analyzers, or be provided by a programmer.

- The transformation should (eventually) improve some efficiency metric, which is far from trivial. An interactive mode which leaves this decision to a final user is available. While this is useful

in many cases, including debugging, it is clearly not scalable and deciding the best rule is often difficult. As a solution, we are working on a machine learning-based oracle [24] which decides which rule to apply based on estimations of the expected performance of rule application chains.

We present now the code annotations and the rule language. We will close this section with a description of the interaction between the transformation tool and external oracles.

3.1   High-Level Annotations

Listing 1: Annotations for Fig. 1.

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

#pragma polca map BODY1 v c
for(int i=0;i<N;i++)
  c[i] = a*v[i];

#pragma polca map BODY2 zip(v,c) c
for(int i=0;i<N;i++)
  c[i] += b*v[i];
```

Annotations describing semantic features of the code make it possible to capture algorithmic skeletons at a higher level of abstraction and, at the same time, to express properties of the underlying code. Our annotations follow a functional programming style. For instance, `for` loops expressing a mapping between an input and an output array are annotated with a `map` pragma such as `#pragma polca map F v w`. This annotation would indicate that the loop traverses the input array `v` and applies function `F` to each element in `v` giving as result the array `w`. For the annotation to be correct, we **assume** that `F` is pure, that `v` and `w` have the same length, and that every element in `w` is computed only from the corresponding element in `v`. As a design decision, we do not check for these properties, but we expect them to hold.[1]

The top boxes of the frames in Table 1 list some of the high-level annotations that can be currently used. For illustrative purposes, Listing 1 shows an annotated version of the code in Fig. 1. The annotation in the second loop uses the constructor `zip` to express that a pair of arrays is treated as an array of pairs —that is necessary due to the signature of `map`.

3.2   STML Properties

The transformation tool requires that some low-level, language-dependent properties hold to ensure that transformations are sound. While some of these properties can be inferred from a high-level annotation, some of them can go beyond what can be expressed in the high-level functional specifications. For example, a purely functional semantics featuring referential transparency cannot capture some aspects of imperative languages such as destructive assignment or aliasing. In our framework, these properties can be expressed in a language we have termed STML (from *Semantic Transformation Meta-Language*) which can be used both in the code annotations and in the conditions of the transformation rules.

3.2.1   Syntax and Semantics of STML Annotations

Listing 2 shows the grammar for STML annotations. An intuitive explanation of its semantics follows.

- `<code_prop>` refers to code properties expressed through STML annotations.

- `[<exp>] <exp_prop> <exp>`: `<exp_prop>` denotes properties about code expressions of the statement immediately below the annotation. Some examples are:

---

[1]However, if some available analysis gives us information to deduce that some of these assumptions does not hold, we warn the user about the discrepancy.

Listing 2: BNF grammar for STML.

```
<code_prop_list> ::= "#pragma stml" <code_prop> |
                     "#pragma stml" <code_prop> <code_prop_list>
<code_prop>  ::= <loop_prop> | <exp_prop> <exp> | [<op>] <op_prop> <op> |
                 "write("<exp>") =" <location_list> | "same_length" <exp> <exp> |
                 "output("<exp>")" | <mem_access> <exp> ["in" <offset_list>]
<loop_prop>  ::= "iteration_independent" | "iteration_space" <parameter> <parameter>
<exp_prop>       ::= "appears" | "pure" | "is_identity"
<op_prop>        ::= "commutative" | "associative" | "distributes_over"
<mem_access>     ::= "writes" | "reads" | "rw"
<location_list> ::= "{" <c_location> {"," <c_location>} "}"
<offset_list>    ::= "{" <INT> {"," <INT>} "}"
<exp>            ::= <C_EXP> | <C_VAR> | <polca_var_id>
<op>             ::= <C_OP> | <C_VAR> | <polca_var_id>
<c_location>     ::= <C_VAR> | <C_VAR>("["<C_EXP>"]")+
<parameter>      ::= <c_location> | <polca_var_id> | <INT>
```

- appears <exp>: there is at least one occurrence of <exp> in the statement below.
- pure <exp>: expression <exp> is pure, i.e. it has neither side effects nor writes on any memory location.
- is_identity <exp>: <exp> is an identity element. High-level annotations that define the group or field in which <exp> is the identity element must have appeared before.

- [<op>] <op_prop> <op>: <op_prop> is an operator property (maybe binary). Some examples are:

  - commutative <op>: <op> has the commutative property: if <op> $= f$, then $\forall x, y. \ f(x,y) = f(y,x)$.
  - associative <op>: <op> has the associative property: if <op> $= f$, then $\forall x, y, z.$ $f(f(x,y),z) = f(x,f(y,z))$.
  - <op> distributes_over <op>: The first operator distributes over the second operator: if the operators are f and g, then $\forall x, y, z. \ g(f(x,y),z) = f(g(x,z),g(y,z))$.

- "write("<exp>")="<location_list>: the list of memory locations written on by expression <exp> is <location_list>, a list of variables (scalar or array type) in the C code. For example, write(c = a + 3)= {c} and write(c[i++] = a + 3)= {c[i], i}

- <mem_access> <exp> ["in"<offset_list>]: <mem_access> states properties about the memory accesses made by the statement(s) that immediately follow the expression <exp>. When <exp> is an array, "in"<offset_list> can state the list of positions accessed for reading from or writing to (depending on <mem_access>) the array. Some examples are:

  - writes <exp>: the set of statements associated to the annotation writing into a location identified by <exp>.
  - writes <exp> "in"<offset_list>: this annotation is similar to the previous one, but for non-scalar variables within loops. It specifies that for each i-th iteration of the loop, an array identified by <exp> is written to in the locations whose offset with respect to the index of the loop is contained in <offset_list>.

```
#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;}
```

- `reads <exp>`: the set of statements associated with the annotation read from location `<exp>`.

- `reads <exp> "in"<offset_list>t`: similar to `writes <exp> "in"<offset_list>` but for reading instead of writing. An example follows:

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

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

- `rw <exp>`: the set of statements associated to the STML annotation reads and writes from / to location `<exp>`.

- `rw <exp> "in"<offset_list>`: similar to `writes <exp> "in"<offset_list>` but for reading or writing.

- `<loop_prop>`: this term represents annotations related with loop properties:

    - `"iteration_space"<parameter> <parameter>`: this annotation states the iteration space limits of the `for` loop associated with the annotation. An example would be:

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

    - `"iteration_independent"`: this annotation is used to state that there is no loop-carried dependencies in the body of the loop associated to this annotation.

- `"same_length"<exp> <exp>`: the two C arrays given as parameters have the same length.

- `"output("<exp>")"`: `<exp>` is the output of a block of code.

3.2.2   Translation from High-Level to STML Annotations

As mentioned before, annotated code is assumed to follow the semantics given by the annotations. Using this interpretation, lower-level STML properties can be inferred for annotated code and used to decide which transformations are applicable. For example, let us consider the loop annotated with `map BODY1 v c` in Listing 1. In this context the assumption is that:

- `BODY1` behaves as if it had no side effects. It may read and write from/to a global variable, but it should behave as if this variable did not implement a state for `BODY1`. For example, it may always write to a global variable and then read from it, and the behavior of other code should not depend on the contents of this variable.

- `v` and `c` are arrays of the same size.

- For every element of `c`, the element in the i-th position is computed by applying `BODY1` to the element in the i-th position of `v`.

$$\begin{array}{l} \texttt{for}\,(l{=}e_{ini}; rel(l, e_{end}); mod(l)) \quad \{s_1\} \\ \texttt{for}\,(l{=}e_{ini}; rel(l, e_{end}); mod(l)) \quad \{s_2\} \end{array} \Rightarrow \quad \texttt{for}\,(l{=}e_{ini}; rel(l, e_{end}); mod(l)) \quad \{s_1; s_2\}$$

when *rel* pure, $(s_1; s_2) \nmapsto \{l, e_{ini}, e_{end}\}$, writes$(mod(l)) \subseteq \{l\}$, $s_1 \underset{-a[l]}{\nmapsto} s_2, s_2 \underset{-a[l]}{\nmapsto} s_1, s_2 \overset{<}{\underset{a[l]}{\nmapsto}} s_1$

(FOR-LOOPFUSION)

$l \;{+}{=}\; e; \;\Rightarrow l \;=\; l \;+\; e;$

when l pure (AUGADDITIONASSIGN)

$s_1; \; l = e_1; \; s_2; \; l = e_2; \; s_3; \Rightarrow s_1; \; s_2; \; l = e_2[e_1/l]; \; s_3;$

when l, e_1 pure, $s_2 \nmapsto \{l, e_1\}, s_2 \nleftarrow l, s_2 \nmapsto e_1$ (JOINASSIGNMENTS)

$f(g(e_1, e_3), g(e_2, e_3)) \Rightarrow g(f(e_1, e_2), e_3)$

when e_1, e_2, e_3 pure, g distributes_over f (UNDODISTRIBUTE)

$\texttt{for}\;(e_1; e_2; e_3)\,\{s_b\} \Rightarrow l = e_{inv}; \; \texttt{for}\;(e_1; e_2; e_3)\,\{s_b[l/e_{inv}])\}$

when l fresh, e_{inv} occurs in s_b, e_{inv} pure, $\{s_b, e_3, e_2\} \nmapsto e_{inv}$ (LOOPINVCODEMOTION)

Table 2: Source code transformations used in the example of Fig. 1.

- The applications of `BODY1` are not assumed to be done in any particular order: they can go from `v[0]` upwards to `v[length(v)-1]` or in the opposite direction. Therefore all applications of `BODY1` should be independent from each other.

```
join_assignments {
  pattern: {
    cstmts(s1);
    cexpr(l) = cexpr(e1);
    cstmts(s2);
    cexpr(l) = cexpr(e2);
    cstmts(s3);
  }
  condition: {
    no_write(cstmts(s2),
    {cexpr(l),cexpr(e1)});
    no_read(cstmts(s2),{cexpr(l)});
    pure(cexpr(e1));
    pure(cexpr(l));
  }
  generate: {
    cstmts(s1);
    cstmts(s2);
    cexpr(l) =
        subs(cexpr(e2),cexpr(l),
        cexpr(e1));
    cstmts(s3);
  }
}
```

Figure 3: JOINASSIGNMENTS STML rule.

The STML properties inferred from some high-level annotations are shown in Table 1. Focusing on the translation of map, the STML annotations mean that:

- Iteration *i*-th reads from v in the position *i*-th (it actually reads from the set of positions $\{i + 0\text{-th}\}$, since the set of offsets it reads from is $\{0\}$).

- Iteration *i*-th writes on w in the position *i*-th.

- v and w have the same length.

- F behaves as if it did not have side effects.

- F is applied to v and w in the indexes ranging from 0 to $length(\texttt{v})$.

Table 1 shows the STML properties inferred from other high-level annotations (explained more in depth in [20]). Fig. 4 shows the translation of the code in Listing 1 into STML. All rules used in the transformation in Fig. 1 are shown in Table 2, and the conditions they need are described in Table 3.

3.2.3 External Tools

Besides the properties provided by the user, external tools can automatically infer additional properties, thereby relieving users from writing a large number of annotations which capture several low-level

$s \not\mapsto l$	statements s do not write into location l: $l \notin \text{writes}(s)$
$s \not\leftarrow l$	statements s do not read the value in location l
$s_1 \not\mapsto s_2$	statements s_1 do not write into any location read by s_2
$s_1 \not\leftarrow s_2$	statements s_1 do not read from any location written by s_2
$s_1 \underset{-a[l]}{\not\mapsto} s_2$	same predicate as the previous one but not taking into account locations referred through arrays.
$s_1 \overset{<}{\underset{a[l]}{\not\mapsto}} s_2$	statements s_1 do not write into any previous location corresponding to an index array read by s_2
e pure	expression e is *pure*, i.e. does not have side effects nor writes any memory locations.
$\text{writes}(s)$	set of locations written by statements s.
g distributes_over f	$\forall x, y, z.\ g(f(x,y),z) \approx f(g(x,z), g(y,z))$
l fresh	l is the location of a *fresh* identifier, i.e. does not clash with existing identifiers if introduced in a given program state.

Table 3: Predicates used to express conditions for the application transformation rules in Table 2.

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

#pragma polca map BODY1 v c
#pragma stml reads v in {0}
#pragma stml writes c in {0}
#pragma stml same_length v c
#pragma stml pure BODY1
#pragma stml iteration_space 0 length(v)
#pragma stml iteration_independent
for(int i = 0; i < N; i++)
#pragma polca def BODY1
  c[i] = a*v[i];
```

```
#pragma polca map BODY2 zip(v,c) c
#pragma stml reads (v in {0}, c in {0})
#pragma stml writes c in {0}
#pragma stml same_length zip(v,c) c
#pragma stml pure BODY2
#pragma stml iteration_space 0 length(zip(v,c))
#pragma stml iteration_independent
for(int i = 0; i < N; i++)
#pragma polca def BODY2
  c[i] += b*v[i];
```

Figure 4: Translation of high-level annotations in Fig. 1 into STML.

details. These properties can be made available to the transformation tool by writing them as STML annotations. We are currently using Cetus [9] to automatically produce STML annotations. Cetus is a compiler framework, written in Java, to implement source-to-source transformations. We have modified it to add some new analyses and to output the properties it infers as STML pragmas annotating the input code. If the annotations automatically inferred by external tools contradict those provided by the user, the properties provided by the user are preferred to those deduced from external tools, but a warning is issued nonetheless.

3.3 Rules in STML

Let us see one example: Fig. 3 shows the STML version of rule JOINASSIGNMENTS. Rules can be applied when the code being transformed matches the `pattern` section and fulfills the `condition` section. When the rule is activated, code is generated according to the template in the `generate` section, where expressions matched in the `pattern` are replaced in the `generated` code. In this case one assignment is removed by propagating the expression in its *right hand side* (RHS).

STML uses tagged meta-variables to match components of the initial code and specify which kind of component is matched. For example, a meta-variable `v` can be tagged as `cexpr(v)` to denote that it can only match an expression, `cstmt(v)` for a statement, or `cstmts(v)` for a sequence of statements. In Fig. 3, `s1`, `s2` and `s3` should be (sequences of) statements, and `e1`, `e2` and `l` are expressions.

Additional conditions and primitives (Tables 4 and 5) help write descriptive rules which can at the

Function	Description
`no_write((S|[S]|E)`$_1$`, (S|[S]|E)`$_2$`)`	True if `(S|[S]|E)`$_1$ does not write in any location read by `(S|[S]|E)`$_2$.
`no_write_except_arrays` `((S|[S]|E)`$_1$`,(S|[S]|E)`$_2$`,E)`	As the previous condition, but not taking arrays accessed using `E` into account.
`no_write_prev_arrays` `((S|[S]|E)`$_1$` (S|[S]|E)`$_2$`, E)`	True if no array writes indexed using `E` in `(S|[S]|E)`$_1$ access previous locations to array reads indexed using `E` in `(S|[S]|E)`$_2$.
`no_read((S|[S]|E)`$_1$`, (S|[S]|E)`$_2$`)`	True if `(S|[S]|E)`$_1$ does not read in any location written by `(S|[S]|E)`$_2$.
`pure((S|[S]|E))`	True if `(S|[S]|E)` does not write in any location.
`writes((S|[S]|E))`	Locations written by `(S|[S]|E)`.
`distributes_over(E`$_1$`,E`$_2$`)`	True if operation `E`$_1$ distributes over operation `E`$_2$.
`occurs_in(E,(S|[S]|E))`	True if expression `E` occurs in `(S|[S]|E)`.
`fresh_var(E)`	Indicates that `E` should be a new variable.
`is_identity(E)`	True if `E` is the identity.
`is_assignment(E)`	True if `E` is an assignment.
`is_subseteq(E`$_1$`,E`$_2$`)`	True if $E_1 \subseteq E_2$

Table 4: Rule language functions for the `condition` section of a rule.

Function/Construction	Description
`subs((S|[S]|E),E`$_f$`,E`$_t$`)`	Replace each occurrence of `E`$_f$ in `(S|[S]|E)` for `E`$_t$.
`if_then:{E`$_{cond}$`; (S|[S]|E);}`	If `E`$_{cond}$ is true, then generate `(S|[S]|E)`.
`if_then_else:{E`$_{cond}$`; (S|[S]|E)`$_t$`; (S|[S]|E)`$_e$`;}`	If `E`$_{cond}$ is true, then generate `(S|[S]|E)`$_t$ else generate `(S|[S]|E)`$_e$.
`gen_list: {[(S|[S]|E)];}`	Each element in `[(S|[S]|E)]` produces a different rule consequent.

Table 5: Rule language constructions and functions for `generate` rule section.

same time be sound. In these tables, `E` represents an expression, `S` represents a statement, while `[S]` represents a sequence of statements. The function `bin_oper(E`$_{op}$`,E`$_l$`,E`$_r$`)` matches or generates a binary operation `(E`$_l$` E`$_{op}$` E`$_r$`)` and can be used in the sections `pattern` and `generate`. The section `generate` can also state, using `#pragmas`, new properties which hold in the resulting code.

3.4 Rule Selection

In most cases, several (often many) rules can be safely applied at multiple code points in every step of the rewriting process. Deciding which rule has to be fired should be ultimately decided based on whether that rule contributes to an eventual increase in performance. As mentioned before, we provide two ways to perform rule selection: a human-driven one and an interface to communicate with external tools.

3.4.1 Interactive Rule Selection

An interface to make interactive transformations possible is available: the user is presented with the rules that can be applied at some point together with the piece of code before and after applying some selected rule (using auxiliary programs, such as [26], to clearly show the differences). This is useful to refine/debug rules or to perform general-purpose refactoring, which may not be related to improving performance of adapting to a given platform.

$AppRules(Code) \rightarrow \{(Rule, Pos)\}$
$Trans(Code_i, Rule, Pos) \rightarrow Code_o$

$SelectRule(\{(Code_i, \{Rule_i\})\}) \rightarrow (Code_o, Rule_o)$
$IsFinal(Code) \rightarrow Boolean$

Figure 5: Functions provided by the transformation tool.

Figure 6: Functions provided by the oracle.

3.4.2   Oracle-Based Rule Selection

In our experience, manual rule selection is very fine-grained and in general not scalable, and using it is not realistic even for small programs with a reduced set of rules. Therefore, mechanizing as much as possible this process is a must, keeping in mind that our goal is that the final code must improve the original code. A straightforward possibility is to select at each step the rule improves more some metric. However, this may make the search to be trapped in local minima. In our experience, it is often necessary to apply transformations which temporarily reduce the quality of the code because they enable the application of further transformations.

A possibility to work around this problem is to explore a bounded neighborhood. The size of the bounded region needs to be decided, since taking too few steps would not make it possible to leave a local minima. Given that in our experience the number of rules that can be applied in most states is high (typically in the order of the tens), increasing the diameter of the boundary to be explored can cause an exponential explosion in the number of states to be evaluated. This would happen even considering some improvements such as partial order reduction-like when rule commutativity holds.

Therefore we need a mechanism which can make local decisions taking into account global strategies — i.e., a procedure able to select a rule under the knowledge that it is part of a sequence of rule applications that improves code performance for a given platform. We are exploring the use of machine learning techniques based on *reinforcement learning* [24]. From the point of view of the transformation engine, the selection tool works as an *oracle* which, given a code configuration and a set of applicable rules, returns which rule should be applied. We will describe now an abstract interface to an external rule selector, which can be applied not only to the current oracle, but to other similar external oracles.

The interface of the transformation tool (Fig. 5) is composed by functions *AppRules* and *Trans*. Function *AppRules* determines the possible transformations applicable to a given code and returns, for a given input *Code*, a set of tuples containing each a rule name *Rule* and the code position *Pos* where it can be applied (e.g., the identifier of a node in the AST). Function *Trans* applies rule *Rule* to code $Code_i$ at position *Pos* and returns the resulting code $Code_o$ after applying the transformation.

The API from the external tool (Fig. 6) includes operations to decide which rule has to be applied and whether the search should stop. Function *SelectRule* receives a set of safe possibilities, each of them composed of a code fragment and a set of rules that can be applied to it, and returns one of the input code fragments and the rule that should be applied to it. Function *IsFinal* is used to know whether a given code *Code* is considered ready for translation or not.

The function that defines the interaction between the transformation engine and the external oracle is *NewCode* (Fig. 7), which receives an initial $Code_i$ and a set of rules and returns (a) $Code_o$ which results from applying one of the rules from $\{Rule_i\}$ to $Code_i$, and (b) $Rule_o$ that should be applied in the next transformation step, i.e., the next time *NewCode* is invoked with $Code_o$. The rationale is that the first time *NewCode* is called, it receives all the applicable rules as candidates to be applied, but after this first application $\{Rule_i\}$ is always a singleton. *NewCode* is called repeatedly until the transformation generates a code for which *IsFinal* returns true.

This approach makes it unnecessary for the external oracle to consider code positions where a trans-

Header

$NewCode(Code_i, \{Rule_i\}) \rightarrow (Code_o, Rule_o)$

Definition

$NewCode(c, rls) = SelectRule(\{(c', \{r' \mid (r', \_) \in AppRules(c')\})$
$\mid c' \in \{Trans(c, r, p) \mid (r, p) \in AppRules(c), r \in rls\} \})$

Complete derivation

$NewCode(c_0, AllRules) \rightarrow^* (c_n, r_n)$
when $IsFinal(c_n)$ and $\forall i, 0 < i < n.$
$(c_i, r_i) = NewCode(c_{i-1}, \{r_{i-1}\})$
when $\neg IsFinal(c_i)$

Figure 7: Interaction between the transformation and the oracle interface.

formation can be applied, since that choice is implicit in the selection of a candidate code between all possible code versions obtained using a single input rule. Furthermore, by selecting the next rule to be applied, it takes the control of the next step of the transformation. The key here is the function *SelectRule*: given inputs $Code_i$ and $Rule_i$, *SelectRule* selects a resulting code between all the codes that can be generated from $Code_i$ using $Rule_i$. The size of the set received by function *SelectRule* corresponds to the total number of positions where $Rule_i$ can be applied. In this way, *SelectRule* is implicitly selecting a position.

4   Producing Code for Heterogeneous Systems

In the second phase of the tool (Fig. 1), code for a given platform is produced starting from the result of the transformation process. The destination platform of a fragment of code can be specified using annotations that make this explicit (e.g., `#pragma polca mpi`). This information helps the tool decide what transformations should be applied and when the code is ready for translation.

The translation to code for a given architecture is in most cases straightforward as it needs only to introduce the "idioms" necessary for the architecture or to perform a syntactical translation. As a consequence, there is no search or decision process: for each input code given to the translation, there is only one output code which is obtained via predefined transformations or glue code injection.

Some of the translations need specific information: for instance, knowing if a statement is performing I/O is necessary when translating to MPI, because executing this operation might need to be done in a single thread. It is often the case that this can be deduced by syntactical inspection, but in other cases (e.g., if the operation is part of a library function) it may need explicit annotations.

5   Implementation Notes

The transformation phase, which obtains C code that could be easily translated into the source language for the destination platform, is a key part of the tool. As a large part of the system was experimental (including the definition of the language, the properties, the generation of the final code, and the search / rule selection procedures), we needed a flexible and expressive implementation platform. Therefore we decided to use a declarative language and implement the tool in Haskell. Parsing the input code is done by means of the `Language.C` [12] library, which returns the AST as a data structure which is easy to manipulate. In particular, we used the Haskell facilities to deal with generic data structures through the *Scrap Your Boilerplate* (SYB) library [16]. This allows us to easily extract information from the AST or modify it with a generic traversal of the whole structure.

The rules themselves are written in a subset of C and are parsed using `Language.C`. After reading these rules in, they are automatically compiled into Haskell code (contained in the file `Rules.hs` —see

Fig. 2) which performs the traversal and (when applicable) the transformation of the AST. This module is loaded together with the rest of the tool, therefore avoiding the extra overhead of interpreting the rules.

When it comes to rule compilation, STML rules can be divided into two classes: those which operate at the expression level and those which can manipulate both expressions and sequences of statements. In the latter case, sequences of statements (`cstmts`) of an unknown size have to be considered: for example, in Fig. 3, `s1`, `s2`, and `s3` can be sequences of any number of statements (including the empty sequence), and the rule has to try all the possibilities to determine if there is a match which meets the rule conditions. For this, Haskell code that explicitly performs an AST traversal needs to be generated. Expressions, on the other hand, are syntactically bound and the translation of the rule is much easier.

When generating Haskell code, the rule sections (`pattern`, `condition`, `generate`, `assert`) generate the corresponding LHS's, guards, and RHS's of a Haskell function. If the conditions to apply a rule are met, the result is returned in a triplet (`rule_name`, `old_code`, `new_code`) where the two last components are, respectively, the matched and transformed sections of the AST. Note that `new_code` may contain new properties if the `generate` section of the rule defines them.

The tool is divided into four main modules:

- **`Main.hs`** implements the main workflow of the tool: it calls the parser on the input C code to build the AST, links the pragmas to the AST, executes the transformation sequence (interactively or automatically) and outputs the transformed code.

- **`PragmaLib.hs`** reads pragmas and links them to their corresponding node in the AST. It also restores or injects pragmas in the transformed code.

- **`Rul2Has.hs`** translates STML rules (stored in an external file) into Haskell functions which actually perform the AST manipulation. It also reads and loads STML rules as an AST and generates the corresponding Haskell code in the **`Rules.hs`** file.

- **`RulesLib.hs`** contains supporting code used by `Rules.hs` to identify whether some STML rule is or not applicable (e.g., there is matching code, the preconditions hold, etc.) and to execute the implementation of the rule (including AST traversal, transformation, . . .).

6   Conclusion

We have presented a transformation toolchain that uses semantic information, in the form of user- or machine-provided annotations, to produce code for different platforms. It has two clearly separated phases: a source-to-source transformation which generates code with the style appropriate for the destination architecture and a translation from that code to the one used in the specific platform. We have focused until now in the initial phase, which included the specification of a DSL (STML) to define rules and code properties, a translator from this language into Haskell, a complete engine to work with these rules, and an interface to interact with external oracles (such as a reinforcement learning tool which we are developing) to guide the transformation. The translation phase is still in a preliminary stage. However, and while it is able to translate some input code, it needs to be improved in order to support a wider range of programs. We have compared, using several metrics, the code obtained using our tool and the corresponding initial code and the results are encouraging.

We plan to improve the usability of the STML language and continue modifying Cetus to automatically obtain more advanced / specific properties, and we are integrating profiling techniques in the process to make it easier to evaluate the whole transformation system and give feedback on it. Simultaneously, we are investigating other analysis tools which can be used to derive more precise properties. Many

of these properties are related to data dependencies and pointer behavior. We are considering, on one hand, tools like PLuTo [5] and PET [23] (two polytope model-based analysis tools) or the dependency analyzers for the Clang/LLVM compiler. However, since they fall short to derive dependencies (e.g., alias analysis) in code with pointers, we are also considering tools based on separation logic [19] such as VeriFast [13] which can reason on dynamically-allocated mutable structures.

References

[1] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. Cλash: Structural descriptions of synchronous hardware using haskell. In S. López, editor, *DSD*, pages 714–721. IEEE, 2010.

[2] O. S. Bagge, K. T. Kalleberg, E. Visser, and M. Haveraaen. Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003*, pages 65–75. IEEE, 2003.

[3] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[4] M. Boekhold, I. Karkowski, and H. Corporaal. Transforming and parallelizing ANSI C programs using pattern recognition. In *High-Performance Computing and Networking*, pages 673–682. Springer, 1999.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[6] A. Brown, W. Luk, and P. Kelly. Optimising Transformations for Hardware Compilation. Technical report, Imperial College London, 2005.

[7] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[8] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[9] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. P. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42(11):36–42, 2009.

[10] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus:(via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2012.

[11] F. Franchetti, Y. Voronenko, and M. Püschel. FFT Program Generation for Shared Memory: SMP and Multicore. In *Supercomputing (SC)*, 2006.

[12] B. Huber. The Language.C Package. `https://hackage.haskell.org/package/language-c`, 2014.

[13] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, USA, April 18-20, 2011.*, pages 41–55, 2011.

[14] Jacquard Computing Inc. *ROCCC 2.0 User's Manual*, revision 0.74 edition, June 2012. http://roccc.cs.ucr.edu/UserManual.pdf.

[15] P. Klint, T. Van Der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[16] R. Lammel, S. P. Jones, and J. P. Magalhaes. The SYB Package. `https://hackage.haskell.org/package/syb`, 2009.

[17] O. Lindtjorn, R. G. Clapp, O. Pell, H. Fu, M. J. Flynn, and O. Mencer. Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. *IEEE Micro*, 31(2):41–49, 2011.

[18] Maxeler Technologies. Max Compiler MPT. https://www.maxeler.com/solutions/low-latency/maxcompilermpt/, March 2016.

[19] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[20] D. Rubio, C. W. Glass, J. Kuper, and R. de Groote. Introducing and exploiting hierarchical structural information. In *IEEE International Conference on Cluster Computing (CLUSTER), 2015*, pages 777–784. IEEE, 2015.

[21] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.

[22] S. Tamarit, G. Vigueras, M. Carro, and J. Mariño. A Haskell Implementation of a Rule-Based Program Transformation for C Programs. In E. Pontelli and T. C. Son, editors, *International Symposium on Practical Aspects of Declarative Languages*, number 9131 in LNCS, pages 105–114. Springer-Verlag, June 2015.

[23] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012.

[24] G. Vigueras, M. Carro, S. Tamarit, and J. Mariño. Towards Automatic Learning of Heuristics for Mechanical Transformations of Procedural Code. In A. Villanueva, editor, *Proceedings of XIV Jornadas sobre Programación y Lenguajes (PROLE 2016)*, September 2016.

[25] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer, D. Batory, C. Consel, and M. Odersk, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.

[26] K. Willadsen. Meld. http://meldmerge.org/, February 2016. Retrieved on February 2016.