A Haskell Implementation of a Rule-Based Program Transformation for C Programs[⋆]

Salvador Tamarit[1], Guillermo Vigueras[2], Manuel Carro[1,2], and Julio Mariño[1]
{guillermo.vigueras,manuel.carro}@imdea.org
{salvador.tamarit,julio.marino}@upm.es

[1] Universidad Politécnica de Madrid. [2] IMDEA Software Institute.

Abstract. Obtaining good performance when programming heterogeneous computing platforms poses significant challenges for the programmer. We present a program transformation environment, implemented in Haskell, where architecture-agnostic scientific C code is transformed into a functionally equivalent one better suited for a given platform. The transformation rules are formalized in a domain-specific language (STML) that takes care of the syntactic and semantic conditions required to apply a given transformation. STML rules are compiled into Haskell function definitions that operate at AST level. Program properties, to be matched with rule conditions, can be automatically inferred or, alternatively, stated as annotations in the source code. Early experimental results are described.

Keywords: High-Performance Computing, Scientific Computing, Heterogeneous Platforms, Rule-Based Program Transformation, Domain-Specific Language, Haskell.

1 Introduction

There is currently a strong trend in high-performance computing towards the integration of various types of computing elements: vector processors, GPUs being used for non-graphical purposes, FPGA modules, etc. interconnected in the same architecture. 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 able to execute them and is proving to be a cost-effective alternative to more traditional supercomputing architectures [4]. 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.

Defining programming models that ease the task of efficiently programming heterogeneous systems is the goal of the ongoing European research project POLCA.[3] The project specifically targets scientific programming on heterogeneous platforms, due to the performance attained by certain hardware components for some classes of computations – e.g., GPUs and linear algebra – and to the energy savings achieved by heterogeneous computing in scientific applications characterized by high energy consumption [4,5]. Additionally, most scientific applications rely on a large base of existing algorithms that must be ported to the new architectures in a way that gets the most out of

[⋆] Work partially funded by EU FP7-ICT-2013.3.4 project 610686 *POLCA*, Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software* and MINECO Projects TIN2012-39391-C04-03 and TIN2012-39391-C04-04 (*StrongSoft)* and TIN2013-44742-C4-1-R (*CAVI-ROSE*).

[3] *Programming Large Scale Heterogeneous Infrastructures*, http://polca-project.eu.

original 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];
```

For-LoopFusion

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

AugAdditionAssign

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

JoinAssignments

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

UndoDistribute

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

LoopInvCodeMotion

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

Fig. 1: A sequence of transformations of a C code that computes $\mathbf{c} = a\mathbf{v} + b\mathbf{v}$.

their computational strengths, while avoiding pitfalls and bottlenecks, and preserving the meaning of the original code. Porting is carried out by transforming or replacing certain fragments of code to improve their performance in a given architecture while preserving their meaning. Unfortunately, (legacy) code often does not spell its meaning or the programmer's intentions clearly, although scientific code usually follows patterns rooted in its mathematical origin.

Our proposal is to develop a framework for semantic-based program transformation of scientific code where the validity of a given transformation is guided by high-level annotations expressing the mathematical foundation of the source code. Fig. 1 shows a sample code transformation sequence, containing the original fragment of C code along with the result of applying *loop-fusion*, reorganizing assignments, algebraic rewriting based on *distributivity* and moving *invariant* expressions out of a loop body. Some of these transformations are currently done by existing compilers. However, they are performed internally, and we need them to be applied at the source code level, since they may enable further source code-level transformations.

Due to space limitations, we are not showing the code annotations required to associate algebraic properties with variables and operators, etc.[4] – in this paper we will focus on the design of the tool implementing them.

The decision of whether to apply a given transformation depends on many factors. First, it is necessary to ensure that applying a rule at a certain point is sound. Several sources of information can be used here, from static analyzers (e.g., to extract data dependency and type information) to inline code annotations provided by the programmer. Second, whether the transformation may improve efficiency, which is far from trivial: *Cost models* for different target architectures are needed, as the transformation process will eventually be guided by estimations of the final performance (which may bring problems such as local optima). Finally, the transformed code may contain new derived annotations that can affect subsequent steps.

Despite the broad range of compilation and refactoring tools available [1,8,6], no existing tool fitted the needs of the project, so we decided to implement our own transformation framework, including a domain specific language for the definition of semantically sound code transformation rules (STML), and a transformation engine working

[4] The full example code can be found at `http://goo.gl/LWRNOy`.

at AST level (`http://goo.gl/yuOFiE`). Declarative languages are used in different ways in this project. First, the rewriting engine itself is implemented in Haskell; second, STML rules have a declarative flavor as they are rewriting rules whose application should not change the semantics of the program being rewritten; and last, the rules themselves are translated into Haskell code. Rules are written using a C-like syntax, which makes it easy for C programmers to understand their meaning and to define them, while the rules can transparently access core functionality provided by the Haskell rewriting engine, and be accessed by it.

The engine selects rules and blocks of code where these rules can be safely applied. There may be several possibilities, and the engine is able to return all of them in a list. In the final tool we plan to use heuristics to select the most promising transformation chain (Section 4) and to have available an interactive mode which can interplay with the guided search when it is not possible to automatically determine whether some rule can / should be applied or when the programmer so desires it. At the moment, only the interactive mode is implemented.[5]

2 Tool Description

The main two functionalities of our tool are: 1) to parse transformation rules written in our domain-specific language (*Semantic Transformation Meta-Language*, STML), and to translate them into Haskell; and 2) to perform source-to-source C code transformation based on these rules, possibly making use of information provided in code annotations (*pragmas*). Occasionally, new pragmas can be injected in the transformed code.

The transformation tool is written in Haskell. The `Language.C` library [**?**] is used to parse the input C code and build its *abstract syntax tree* (AST), which is then manipulated using the *Scrap Your Boilerplate* (SYB) [**?**] Haskell library: functions like `everything` and `everywhere` allow us to easily extract information from the AST or modify it with a generic traversal of the whole structure. Both libraries are used to perform the transformation, but also for the translation of STML rules into Haskell: the STML rules themselves are expressed in a subset of C and are parsed using the same `Language.C` library. The tool is composed of four main modules:

- **`Main.hs`:** Implements the tool's workflow: calls the parser on the input C code which builds the AST, links the pragmas to the AST, executes the transformation sequence (interactive or automatically) and outputs the transformed code.
- **`PragmaPolcaLib.hs`:** Reads pragmas and links them to their corresponding AST. Restore / injects pragmas in transformed code.
- **`Rul2Has.hs`:** Translates STML rules (stored in an external file) into Haskell functions which actually perform the AST manipulation. Reads and loads STML rules as an AST and generates the corresponding Haskell code in a **`Rules.hs`** file.
- **`RulesLib.hs`:** Supports `Rules.hs` to identify STML rule applicability (matching, preconditions, etc.) and execution (AST traversal and mutation).

[5] As an temporary step, useful for validation, random selection of rules and locations up to a certain number of transformations, is also available.

2.1 The Rule Language and its Translation

The rule language used by our tool is inspired by CML [3], which is in turn an evolution of CTT [2]. We named it *Semantic Transformation Meta-Language* (STML), to highlight the use of information beyond the syntax of the language to transform (inferred or provided in code annotations). STML is syntactically simpler than CML and closer to C, but it features additional functionality, such as richer conditions or the ability to express only once antecedents common to several transformation rules.

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

Fig. 2: Rule template.

Fig. 2 shows a rule template: whenever a piece of code matching the code in the `pattern` section is found which meets a series of conditions stated in the `condition` section, the matched code is replaced by the code in the `generate` section. The symbols in `pattern` are meta-variables which are substituted for the actual symbols in the code before performing the translation. The `conditions` can refer to both syntactic and semantic properties. The generated code can have additional (semantic) properties which can be explicitly stated in the `assert` section to make the application of other rules possible.

```
undo_distributive {
   pattern: {
      (cexpr(b) * cexpr(a))
      + (cexpr(c) * cexpr(a)); }
   condition: {
      pure(cexpr(a));
      pure(cexpr(b));
      pure(cexpr(c)); }
   generate: {
      cexpr(a) *
      (cexpr(b) + cexpr(c)); }
}
```

Fig. 3: Distributive property.

Fig. 3 shows a simple example: a rule which applies the distributive property to optimize code by transforming code like `((a[i] - 1) * v[i]) + (v[i] * f(b,3))` into `v[i] * ((a[i] - 1) + f(b,3))`. Meta-variables are marked to denote their role, i.e. what type of syntactic entity they can match: an expression (`cexpr(·)`), a statement (`cstmt(·)`), or a sequence of statements (`cstmts(·)`). In the example, meta-variables `a`, `b`, and `c` will only match expressions. Tables 1 and 2 briefly describe additional constructions which can be used in the `condition` and `generate` sections to check for properties of the code being matched (e.g., `is_identity(·, ·)`) and to have a more flexible and powerful code generation (e.g., `subs(·,·,·)`). In these tables, `E` represents an expression, `S` represents a statement, and [S] represents a sequence of statements. Additionally, other constructs such as `bin_oper(`$\mathtt{E_{op}}$`,`$\mathtt{E_l}$`,`$\mathtt{E_r}$`)` can be used both to match and generate previously matched syntactic constructs (a binary operand, in this case). The tables are not meant to be exhaustive and, in fact, they can be extended to incorporate whatever property *imported* from external analysis tools.

2.2 Matching and Generating Code through Synthesized Haskell

The actual transformation of the AST is performed by Haskell code automatically synthesized from STML rules, and contained in the file `Rules.hs`. We can classify STML rules among those which operate at expression level (easier to implement) and those which can manipulate both expressions and (sequences of) statements. The latter need to consider sequences of statements (`cstmts`) of unbound size, for which Haskell code that explicitly performs an AST traversal is generated.

When generating Haskell code, the rule sections (`pattern`, `condition`, `generate`, `assert`) generate the corresponding LHS's, guards, and RHS's of a Haskell function.

Function	Description
`is_identity`(E_{op},E)	E is an identity for E_{op}
`no_writes`(E_v,(S|[S]|E))	E_v is not written in (S|[S]|E)
`no_reads`(E_v,(S|[S]|E))	E_v is not read in (S|[S]|E)
`no_rw`(E_v,(S|[S]|E))	E_v is neither read nor written in (S|[S]|E)
`pure((S|[S]|E))`	There is not any assignment in (S|[S]|E)
`is_const(E)`	There is not any variable inside E
`is_block(S)`	S is a block of statements
`not`(E_{cond})	E_{cond} is false

Table 1: Basic functions for the `condition` section of rules.

Function/Construct	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 statement/expression in [(S|[S]|E)] produces a different rule consequent.

Table 2: Language constructs and functions for the `generate` section of the rules.

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. Since several rules can be applied at several locations of the AST, the result of applying the Haskell function implementing an STML rule is a list of tuples, one for each rule and location where the rule can be applied. This list will in a future have a heuristically determined benefit associated, which would make it possible to prioritize them. The transformation stops when either no more rules are applicable, or a stop condition is found – e.g., no applicable rule increase code quality above some threshold or a maximum number of rule applications is reached.

Example 1 (Augmented Addition). The rule in Fig. 4a transforms the augmented assignment += to a simple assignment: `x += f(3)` is transformed into `x = x + f(3)`. Fig. 4b shows its Haskell translation. Note that `v[i++] += 1` can not be transformed because `v[i++]` is not *pure* – one of the conditions required by the rule. When conditions are present in the rule, the transformation express them in the Haskell code. In this case, the purity condition `pure(cexpr(a))` is translated to the Haskell guard `(null (allDefs [(CBlockStmt (CExpr (Just var_a_463) undefNode))]))`, which constructs an *artificial* block of statements containing only the expression `var_a_463` and checks that the list of variables assigned inside it (returned by function `allDefs` from `RulesLib.hs`) is empty. Symbols `CBlockStmt`, `CExpr`, and `undefNode` are defined in `Language.C`.

Example 2 (Undo Distributive). Consider again the STML rule in Fig. 3. This rule is translated into the code in Fig. 5, where some clauses have been omitted: the commutative properties of addition and multiplication, known by the tool, force the generation of eight clauses. Checking the applicability of the rule (either because of the pattern or because of the `conditions`) is again implemented as clause guards. One example of the former is `(exprEqual var_a_79 var_a_77)`, that checks that both expressions

```
aug_addition_assign {
   pattern:   {
      cexpr(a) += cexpr(b); }
   condition: {
      pure(cexpr(a)); }
   generate: {
      cexpr(a) =
        cexpr(a) + cexpr(b); }
}
```

(a) STML Rule.

```
--aug_addition_assign
rule_aug_addition_assign_462
  old@(CAssign CAddAssOp var_a_463 var_b_464 _) | True &&
       (null (allDefs [(CBlockStmt (CExpr (Just var_a_463)
       undefNode))])) =
  [("aug_addition_assign",old,(CAssign CAssignOp var_a_463
       (CBinary CAddOp var_a_463 var_b_464 undefNode)
       undefNode))]
rule_aug_addition_assign_462 _ = []
```

(b) Haskell code.

Fig. 4: Augmented addition: STML rule and Haskell code.

```
-- undo_distributive
rule_undo_distributive_75
  old@(CBinary CAddOp (CBinary CMulOp var_b_76 var_a_77 _) (CBinary CMulOp var_c_78 var_a_79
       _) _)
  | (exprEqual var_a_79 var_a_77) && True
       && (null (allDefs [(CBlockStmt (CExpr (Just var_a_79) undefNode))]))
       && (null (allDefs [(CBlockStmt (CExpr (Just var_b_76) undefNode))]))
       && (null (allDefs [(CBlockStmt (CExpr (Just var_c_78) undefNode))])) =
  [("undo_distributive",old,(CBinary CMulOp var_a_79 (CBinary CAddOp var_b_76 var_c_78
       undefNode) undefNode))]
...
rule_undo_distributive_75 _ = []
```

Fig. 5: Haskell code compiled from the `undo_distributive` rule.

matching "a" are indeed the same. The code for `pure(cexpr(a))` (cf. `b` and `c`) is the same as in the previous example.

```
useless_assign {
   pattern: {
      cstmts(ini);
      cexpr(lhs) = cexpr(lhs);
      cstmts(fin); }
   condition: { pure(cexpr(lhs)); }
   generate: {
      cstmts(ini);
      cstmts(fin);}
}
```

Fig. 6: Rule to remove useless assignments.

Our final example shows the translation of a rule which transforms a sequence of statements. The code produced for this case is more complex than for the case of expression-transforming rules. Due to space limitations, we will just provide some insight on how the translation is done.

Example 3 (Useless Assignment Removal). The STML rule in Fig. 6 removes an assignment that does not change the expression being evaluated nor the l-value, i.e. it would remove `v[i] = v[i]`, but not `v[i++] = v[i++]` because `v[i++]` is not pure.[6] Fig. 7 shows its Haskell translation. The helper function `rule_useless_assign_503` searches for the rule pattern: the assignment `cexpr(lhs)= cexpr(lhs)` and its surrounding "holes" (cf. `ctstms(_)`). Function `rule_useless_assign_504` builds the consequent of the rule, checking the guard conditions and generating, for each occurrence of the pattern in the block, the corresponding consequent. The rule itself is implemented by function `rule_useless_assign_501` which is, essentially, a Haskell

[6] It is debatable whether that rule can be applied to human-produced code. However, it is useful when several rules are chained (see the *Identity Matrix* example at `http://goo.gl/LWRNOy`).

```
-- useless_assign
rule_useless_assign_501 old@(CCompound ident bs nodeInfo) =
  (concat [rule_useless_assign_504 item ident nodeInfo old | item@(True,_,_) <-
      (rule_useless_assign_503 bs [] True )]) ++ []
rule_useless_assign_501 _ = []

rule_useless_assign_504 (True,[var_ini_507,var_fin_508], [(CBlockStmt (CExpr (Just (CAssign
    CAssignOp var_lhs_505 var_lhs_506 _)) _))]) ident nodeInfo old | (exprEqual var_lhs_506
    var_lhs_505) && (null (allDefs [(CBlockStmt (CExpr (Just var_lhs_506) undefNode))])) =
  concat ([[("useless_assign", old, (CCompound ident (var_ini_507 ++ var_fin_508)
      nodeInfo))] | True] ++ [])
rule_useless_assign_504 _ _ _ _ _ = []

rule_useless_assign_503 [] acc False = [(True,[acc],[])]
rule_useless_assign_503 [] acc _ = [(False,[acc],[])]
rule_useless_assign_503 (stat@(CBlockStmt (CExpr (Just (CAssign CAssignOp var_lhs_509
    var_lhs_510 _)) _)):tail_) accsl True =
  let
    listItems = rule_useless_assign_503 tail_ [] False
  in [(True, (accsl:(inter)),(stat:(pats))) | (True,inter,pats) <- listItems] ++
      (rule_useless_assign_503 tail_ (accsl ++ [stat]) True)
rule_useless_assign_503 (other:tail_) acc bool1 =
  rule_useless_assign_503 tail_ (acc ++ [other]) bool1
```

Fig. 7: Haskell code obtained from rule `useless_assign`.

list comprehension calling `rule_useless_assign_503` in its *generator* expression and `rule_useless_assign_504` in the *construction* expression, to return the list of triples.

3 Experimental Evaluation

We have implemented (among others) the transformation rules mentioned in Fig. 1 and checked that the tool can successfully apply them in sequence. We have also carried out a preliminary performance test. The code in Fig. 1, which implements the C equivalent of the linear algebra $\mathbf{c} = a\mathbf{v} + b\mathbf{v}$, was compiled using `gcc -O3` in all cases and cache misses, number of floating point (FP) operations, and execution time were measured. We performed the evaluation with `gcc 4.47` in a Linux CentOS 6.5 with kernel 2.6.32 running in an Intel i7 3770, using PAPI 5.4.0 to profile the execution. The number of cache misses is of interest for CPU-based (multicore) architectures, and therefore also relevant for parallel platforms using the OpenMP and MPI programming models. The number of FP operations is interesting for CPU-based architectures and also for systems with scarce computational resources (like FPGAs and SoC platforms).

Table 3 shows, cumulatively, the effect of the transformation sequence in Fig. 1 on cache misses, FP operations, and execution time as percentages of the initial values (which is, for execution time, 46 ms.). These values are significantly reduced: ~50% for execution time and L1 misses and ~33% for FP operations. These results were obtained averaging values for 30 runs. For all parameters, the standard deviation was lower than 0.46% during the runs.

4 Conclusions and Future Work

We have briefly presented the goals and internal design of a tool to perform rule-based refactoring of procedural programs. The tool is written in Haskell, and the rules it exe-

Metric	original	For-LoopFusion	AugAdditionAssign	JoinAssignments	UndoDistribute	LoopInvCodeMotion
TIME	100,00	50,23	50,22	49,83	49,85	49,32
FP_OPS	100,00	99,88	99,89	99,95	33,34	33,34
L1_MISS	100,00	49,62	49,62	49,62	49,62	49,62

Table 3: Impact of successive code transformations in Fig. 1 on several metrics.

cutes are of a declarative nature. The use of Haskell (instead of, for example, the infrastructure provided by LLVM) has proven to simplify and accelerate the development of the tool without compromising its speed / scalability so far. An experimental validation of a simple but relevant case, which uses algebraic properties of the code under transformation, has shown substantial improvements. The tool (`http://goo.gl/yuOFiE`) is being applied to a series of examples [7] elicited within the POLCA project, and to other examples (`http://goo.gl/LWRNOy`) where e.g., complexity reductions have been achieved for some cases.

As future work, we plan to implement metrics-based heuristics to perform an automatic (guided) search through the space of transformations. While we have already defined some metrics to determine the *adequacy* of transformations for different architectures, this is ongoing research within the consortium. We plan also to improve the interface to external analysis tools, of which we have identified those performing dependency analysis (e.g., polytope-based compilation) and reasoning over heap pointers (e.g., separation logic) as immediately applicable. A (more) formal definition of STML is now on the works.

References

1. Bagge, O.S., Kalleberg, K.T., Visser, E., Haveraaen, M.: 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. pp. 65–75. IEEE (2003)
2. Boekhold, M., Karkowski, I., Corporaal, H.: Transforming and parallelizing ANSI C programs using pattern recognition. In: High-Performance Computing and Networking. pp. 673–682. Springer (1999)
3. Brown, A., Luk, W., Kelly, P.: Optimising Transformations for Hardware Compilation. Tech. rep., Imperial College London (2005)
4. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 63–74. ACM (2010)
5. Lindtjorn, O., Clapp, R.G., Pell, O., Fu, H., Flynn, M.J., Mencer, O.: Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. Ieee Micro 31(2), 41–49 (2011)
6. Schupp, S., Gregor, D., Musser, D., Liu, S.M.: Semantic and behavioral library transformations. Information and Software Technology 44(13), 797–810 (2002)
7. The POLCA Consortium: Specific Use Case Requirements. POLCA Project (#610686) Deliverable D-5.1 (February 2014)
8. Visser, E.: Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Consel, C., Odersk, M. (eds.) Domain-Specific Program Generation, Lecture Notes in Computer Science, vol. 3016, pp. 216–238. Springer-Verlag (June 2004)