On the Semantics of Loop Transformation Languages

Adilla Susungi
MINES ParisTech
PSL Research University
France

ABSTRACT

Languages for loop transformations have been leveraged for different type of tools and frameworks in different application domains, yet they lack formal semantics. As a step towards formal specification, this works intends to clarify the underlying concepts of such languages using a denotational approach.

CCS CONCEPTS

• **Software and its engineering** → **Semantics**; **Source code generation**;

KEYWORDS

semantics, meta-programming, code generation and optimization

ACM Reference Format:
Adilla Susungi. 2018. On the Semantics of Loop Transformation Languages. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3191697.3213796

Context. Compilers are ill-equipped for optimizing programs targeting parallel architectures with deep memory hierarchies; issues such as portability or finding relevant and powerful transformations become increasingly hard to tackle without proper alternatives.

Fully automatic alternatives include multi-layer compilation chains where different levels of expertise are put together to compose a powerful chain, or empirical autotuning tools in which optimizations are iteratively performed using performance feedback until a program variant, suitable for the target architecture, is found.

As semi-automatic approaches, interactive compilation tools allow the programmer to directly interact with the compiler, thus providing hints to help finding efficient optimizations.

Nevertheless, challenges in certain domains may lack convenient tools. One last resort is then hand-writing optimizations by an expert.

Such different types of approaches exhibit several needs:

- Empirical autotuning systems must be thoughtfully designed to properly address the generation of multiple versions of a program;
- Ergonomic interfaces are necessary to facilitate interactive optimization experience;
- The productivity of experts hand-writing optimizations should be enhanced with languages designed for this purpose;
- Compiler intermediate representations must have the ability to efficiently compose sequences of transformations.

Regardless of which need, *languages for loop transformations* appear to be a groundwork [1, 3–5, 7–9, 13, 14].

They may come in a variety of forms such as scripting languages, intermediate languages, pragma-based or multi-staged. They may also rely on different levels of abstractions of the input program. Yet, they do have one common design principle which is the ability to express loop transformations using language constructs. As these languages mainly target compute-intensive programs generally characterized by deep loop nests (e.g. linear algebra, tensor computations), typical transformations supported are loop fusion, tiling, unrolling or index-splitting just to cite a few.

"Is the transformation legal"[1]? – A fundamental question optimization experts bear in mind when implementing transformations. However, using language-based approaches introduces another level of concern: "Can we guarantee the language to actually do what it says it does?" Language developers may rely on the general knowledge of what each optimization does but formal semantics are almost non-existent. Furthermore, they are often implemented as embedded-DSLs in Python or C++ for instance. Despite the practicality of relying on embedding languages, this adds a level of hardship for semantics definition.

Contribution. This work contributes to one aspect of the semantics of languages for loop transformations:

How exactly does such languages transform the input program? Relying on denotational semantics [12],

- We first define a functional language generalizing features found in such languages, that is, constructs for the specification of arrays, computations and loop transformations.
- We specify the semantics of low level constructs, which is useful to also deduce, through compositions, those of higher-level constructs such as tensor operators or more complex loop transformations.

Applications. Our formalism can serve as a base for more specific semantics with respect to a given language. For instance, domain-specific languages (e.g, TVM [1] for deep learning or Halide [9] for image processing) may require further extensions for more complete semantics. In the context of empirical autotuning, we may also use it to formalize optimization search space exploration.

[1] In other words, does the transformation preserve data dependencies?

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3213796

Related Work. Few languages for loop transformations have a formal definition. Lift [11] is a functional language for optimized and portable GPU code generation. A denotational semantics of its core language is defined in [10]. However, it has a completely different approach for abstracting computations and transformations: computations are expressed using combinators and a set of rewrite rules are used to transform the program. Clay [2], URUK [4], CHiLL [3] and Loo.py [6] rely to some extent on the polyhedral formalism which focuses on the representation, analysis and transformations of loops. Instead, we consider both program representation (at different levels of abstraction) and loop transformations. To the best of our knowledge, this is the first work that provides denotational semantics for tensor operations and classic loop transformations.

REFERENCES

[1] 2017. TVM: An End to End IR Stack for Deploying Deep Learning Workloads on Hardware Platforms. http://tvmlang.org/2017/08/17/tvm-release-announcement.html. (2017).

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[3] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Technical Report 08-897, University of Southern California.

[4] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. 2005. Facilitating the Search for Compositions of Program Transformations. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/1088149.1088169

[5] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. 2006. *A Language for the Compact Representation of Multiple Program Versions.* Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151.

[6] Andreas Klöckner. 2014. Loo.Py: Transformation-based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, Article 82, 6 pages. https://doi.org/10.1145/2627373.2627387

[7] Andreas Klöckner, Lucas C Wilcox, and T Warburton. 2016. Array Program Transformation with Loo.Py by Example: High-order Finite Elements. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2935323.2935325

[8] Ralph Müller-Pfefferkorn, Wolfgang E. Nagel, and Bernd Trenkler. 2004. Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests. In *Euro-Par 2004 Parallel Processing*, Marco Danelutto, Marco Vanneschi, and Domenico Laforenza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–81.

[9] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[10] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

[11] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 74–85. http://dl.acm.org/citation.cfm?id=3049832.3049841

[12] Joe Stoy. 1977. *Denotational Semantics.* MIT Press.

[13] Adilla Susungi, Norman A. Rink, Jerónimo Castrillón, Immo Huismann, Albert Cohen, Claude Tadonki, Jörg Stiller, and Jochen Fröhlich. 2017. Towards Compositional and Generative Tensor Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, New York, NY, USA, 169–175. https://doi.org/10.1145/3136040.3136050

[14] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370637