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 work 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:

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? – 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.

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


