UDK 004.4'418

SSA Algebras

Sokolov P.P. (Higher School of Economics)

SSA (static single-assignment) form is an intermediate representation for compiling imperative programs where every variable is assigned to only once. Properly defined, SSA form gives rise to the family of purely syntactic categories with some nice properties. We hope this lays out the groundwork for categorical approach to compiler optimization.

Keywords: formalisms for program semantics, category theory, program synthesis and transformation

1. Introduction

In compilers of imperative languages, a program is usually represented as a control flow graph (CFG) where every block is a sequence of assignments and, notably, reassignments. SSA (static single-assignment) is a special form of CFG where every variable is assigned to exactly once. SSA form is useful because it opens up many possibilities for optimization of programs: pruning, constant folding, register allocation, language-dependent optimizations etc.

Here we argue that it is also a suitable basis for a categorical framework to judge about compiler optimizations, too. First, we define a base category SSA; then we try to define a product in SSA and arrive at an important equivalence relation for programs. Finally, we provide categorical definitions of optimization and compilation. **Main results** are presented as theorems in bold.

2. Related work

SSA is a hot and thoroughly studied research topic; while there already is a comprehensive book on SSA-based compiler design [1], new papers keep coming out [2, 3]. This paper, however, is not in the tradition of previous SSA works because we focus on the formal, mathematical, categorical side of things.

Applying category theory to programming languages has a long history; it is extensively used to model execution of programs either inside Kleisli category [4] or via Curry-Howard-Lambek correspondence [5]. Unfortunately, it is impossible to express the notion of optimization in these categories; **our approach allows one to judge about optimization inside a categorical framework**.

3. The categories of SSA programs

For the language of expressions of an SSA program, let us use the set of expressions $E = \Sigma[\mathbb{N}]$ in arbitrary algebraic signature Σ where variables are taken from the set of natural numbers starting from 0. The following definition is useful:

Definition 1. *An expression $e \in E$ is said to be **bound by** n iff $n \in \mathbb{N}$ and, for all variables $x \in \mathbb{N}$ mentioned in e, $x < n$.*

Now, we consider SSA programs to be a sequence of assignments where, for new binding x_i, one can refer to one of m *inputs* and to one of i previous assignments using variables $\{0, \ldots, m-1\}$ and $\{m, \ldots, m+i-1\}$, respectively.

Definition 2. *Let $P_{m\to n} \subseteq E^* \times \mathbb{N}^n$ be the set of valid SSA programs for language Σ with m inputs and n outputs, that is, a set of pairs (O, R) where, for every $r \in R$, $r < m + |O|$ and, for every $i < |O|$, O_i is bound by $m + i$.*

Lists of operations O can be concatenated as lists with $(+\!\!+)$; same goes for output tuples R. To turn concatenated lists back into valid programs, we use *substitution* which is applied to natural numbers and substitutes them according to the first matching predicate. An example of how substitution works on an expression in the language with binary $(+)$:

$$(x_1 + x_3 + x_0)[i < 2 \mapsto i + 1 \mid\mid i \mapsto i - 1] = x_2 + x_2 + x_1$$

Notable examples of SSA programs include:

Definition 3. *For every natural n, there is the **identity** SSA program with no operations:*

$$\mathrm{id}_n = (\varepsilon, (0, \ldots, n-1)) \in P_{n\to n}$$

Definition 4. *For every permutation $\sigma \in S_n$, there is an SSA program which reorders the inputs according to σ:*

$$p_\sigma = (\varepsilon, (\sigma_0, \ldots, \sigma_{n-1})) \in P_{n\to n}$$

Definition 5. *Given SSA programs $f : P_{m\to n}$ and $g : P_{k\to m}$, **composition** $f \circ g : P_{k\to n}$ is a program which feeds outputs of g into f, that is,*

$$O^{f\circ g} = O^g +\!\!+ O^f[i < m \mapsto R^g_i \mid\mid i \mapsto i - m + k + |O^g|] \tag{1}$$

$$R^{f\circ g} = R^f[i < m \mapsto R^g_i \mid\mid i \mapsto i - m + k + |O^g|] \tag{2}$$

Our **first main result** is that this turns out to be a category:

Theorem 1. *There is a category* $\mathrm{SSA}(\Sigma)$ *of SSA programs in language* Σ *with* $\mathrm{Ob} = \mathbb{N}$ *and* $\mathrm{Hom}(m, n) = P_{m\to n}$.

The proof goes as follows: id and $f \circ g$ are given above. Their properties are proven by obvious induction on the lengths of programs.

Now, the problem with this category is that SSA programs might contain expressions they never even reuse. It might be useful if the compiled language has side-effects; but we can model effectful programs by threading some extra variables through IO operations, as is done in languages with linear typing like Clean [6]. To get rid of truly unused expressions, we (informally) define

Definition 6. *Let* $\mathrm{prune} : P_{m\to n} \to P_{m\to n}$ *be a function which, for every SSA program, removes operations not reachable from the output and patches indices accordingly.*

Statement 1. *There is a category* PrunedSSA *of pruned SSA programs with* $\mathrm{Hom}(m, n) = \{\mathrm{prune}(p) \mid p \in P_{m\to n}\}$.

The proof goes as follows: let $\mathrm{id}(n) = \mathrm{id}_{\mathrm{SSA}}(n)$ and $f \circ g = \mathrm{prune}(f \circ_{\mathrm{SSA}} g)$. The following are obvious:

$$\mathrm{prune}(\mathrm{id}_{\mathrm{SSA}}(n)) = \mathrm{id}_{\mathrm{SSA}}(n); \tag{3}$$

$$\mathrm{prune}(f \circ \mathrm{prune}(g \circ h)) = \mathrm{prune}(f \circ g \circ h) = \mathrm{prune}(\mathrm{prune}(f \circ g) \circ h). \tag{4}$$

Statement 2. *In* PrunedSSA*, 0 is a terminal object with* $! = (\varepsilon, ())$.

To get into products, we first have to define another operation on programs.

Definition 7. *Given SSA programs* $f : k \to m$ *and* $g : k \to n$*, their* ***parallel composition*** $f|g : k \to (m + n)$ *is a program which computes both* f *and* g *from the same inputs, that is,*

$$O^{f|g} = O^f +\!\!+ O^g[i < k \mapsto i \mid\mid i \mapsto i + |O^f|] \tag{5}$$

$$R^{f|g} = R^f +\!\!+ R^g[i < k \mapsto i \mid\mid i \mapsto i + |O^f|] \tag{6}$$

Statement 3. *In* PrunedSSA*, for every* $f : k \to m$ *and* $g : k \to n$*,* $f|g$ *commutes with* f*,* g*,* $\pi_1 = (\varepsilon, (0, \ldots, m-1))$ *and* $\pi_2 = (\varepsilon, (m, \ldots, m+n-1))$ *as a product.*

Note that $f|g$ is not unique as a product arrow: operations from both f and g can be freely intertwined and the result would still satisfy the product property. Furthermore, both f and g might contain the same expressions which can be reused in a product arrow, which, actually, is a well-known CSE (common subexpression elimination) problem [7]. Finally, composing effectful programs like this is slightly wrong. The following equivalence helps with the first problem:

Definition 8. *Two SSA programs $p, q : P_{m\to n}$ are called equivalent (or $p \sim q$) iff $l = |O^p| = |O^q|$ and there exists a permutation $\sigma \in S_l$ such that the following holds:*

$$O^p_j = O^q_{\sigma(j)}[i < m \mapsto i \mid\mid i \mapsto \sigma(i-m)+m] \tag{7}$$

$$R^p = (\sigma(R^q_1), \ldots, \sigma(R^q_n)) \tag{8}$$

Theorem 2. *$p \sim q$ is an equivalence relation; $f \circ g$ and $f|g$ respect it.*

The proof goes as follows: equivalence follows from properties of permutations; respect of $f \circ g$ and $f|g$ can be proven by taking permutations that act on parts of $f \circ g$ and $f|g$.

4. Optimization, compilation and semantic functors

In this framework, optimization and compilation procedures become functors of SSA which act on morphisms in the semantic-preserving way. Reserving study of semantics of effectful programs for further work, here we outline the definitions and properties of such functors via **semantic functors**:

Definition 9. *Given a structure S of signature Σ, semantic category $\mathrm{Sem}(S)$ is a category with $\mathrm{Ob} = \mathbb{N}$ and $\mathrm{Hom}(m,n) = (S^m \to S)^n$.*

Definition 10. *A semantic functor $F_S : \mathrm{SSA}(\Sigma) \to \mathrm{Sem}(S)$ is a functor with $F_S(n) = n$ which interprets SSA programs as functions on vectors over S.*

In terms of semantic functors, optimization and compilation can be defined like this:

Definition 11. *An endofunctor $Q : \mathrm{SSA}(\Sigma) \to \mathrm{SSA}(\Sigma)$ is an optimization with respect to S iff $F_S \circ Q \simeq F_S$.*

Definition 12. *Given a functor $G : \mathrm{Sem}(S) \to \mathrm{Sem}(T)$, where S is a structure of Σ and T is a structure of Ξ, a functor $C : \mathrm{SSA}(\Sigma) \to \mathrm{SSA}(\Xi)$ is a compilation with respect to G iff $F_T \circ C \simeq G \circ F_S$.*

Note that, in the case of compilation, G (and, consequently, C) might not keep the objects (sizes of inputs/outputs) intact. This is reserved for the case where a single $s \in S$ is represented by a vector of values from T.

Study of the properties of optimization and compilation functors is reserved for further work.

References

1. Rastello F. SSA-Based Compiler Design. Springer Publishing Company, Incorporated, 2016.
2. Buchwald S., Lohner D., Ullrich S. Verified Construction of Static Single Assignment Form // Proceedings of the 25th International Conference on Compiler Construction. Barcelona, Spain, 2016. P. 67–76.
3. Bhat S., Grosser T. Lambda the Ultimate SSA: Optimizing Functional Programs in SSA. [2022]. URL: https://arxiv.org/abs/2201.07272 (access date: 28.10.2023).
4. Moggi E. Notions of computation and monads // Information and Computation. 1991. Vol. 93, № 1. P. 55–92.
5. Lambek J., Scott P.J. Introduction to higher-order categorical logic. Cambridge University Press, 1988.
6. Plasmeijer R., Eekelen M. Keep It Clean: A Unique Approach to Functional Programming // SIGPLAN Not. New York, NY, USA: Association for Computing Machinery, June 1999. Vol. 34, № 6. P. 23–31.
7. Muchnick S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997. P. 378–396