Моделирование и анализ информационных систем. Т. 25, № 5 (2018), с. 534–548
Modeling and Analysis of Information Systems. Vol. 25, No 5 (2018), pp. 534–548

Синтез и преобразования программ
Program Synthesis and Transformations

DOI: 10.18255/1818-1015-2018-5-534-548

UDC 519.681.3

Polyprograms and Polyprogram Bisimulation

Grechanik S. A.

Received 10 September 2018

Abstract. A *polyprogram* is a generalization of a program which admits multiple definitions of a single function. Such objects arise in different transformation systems, such as the Burstall–Darlington framework or equality saturation. In this paper, we introduce the notion of a polyprogram in a non-strict first-order functional language.

We define denotational semantics for polyprograms and describe some possible transformations of polyprograms, namely we present several main transformations in two different styles: in the style of the Burstall–Darlington framework and in the style of equality saturation. Transformations in the style of equality saturation are performed on polyprograms in decomposed form, where the difference between functions and expressions is blurred, and so is the difference between substitution and unfolding. Decomposed polyprograms are well suited for implementation and reasoning, although they are not very human-readable.

We also introduce the notion of *polyprogram bisimulation* which enables a powerful transformation called *merging by bisimulation*, corresponding to proving equivalence of functions by induction or coinduction. Polyprogram bisimulation is a concept inspired by bisimulation of labelled transition systems, but yet it is quite different, because polyprogram bisimulation treats every definition as self-sufficient, that is a function is considered to be defined by any of its definitions, whereas in an LTS the behaviour of a state is defined by all transitions from this state.

We present an algorithm for enumerating polyprogram bisimulations of a certain form. The algorithm consists of two phases: enumerating prebisimulations and converting them to proper bisimulations. This separation is required because polyprogram bisimulations take into account the possibility of parameter permutation. We prove correctness of this algorithm and formulate a certain weak form of its completeness.

The article is published in the author's wording.

Keywords: polyprograms, program transformation, equality saturation, bisimulation

For citation: Grechanik S. A., "Polyprograms and Polyprogram Bisimulation", *Modeling and Analysis of Information Systems*, **25**:5 (2018), 534–548.

On the authors:
Sergei A. Grechanik, orcid.org/0000-0001-8575-9689, PhD,
Keldysh Institute of Applied Mathematics,
4 Miusskaya sq., Moscow 125047, Russia, e-mail: sergei.grechanik@gmail.com

Acknowledgments:
This work was supported by Russian Foundation for Basic Research, grant No. 18-31-00412.

Introduction

Many program transformation methods can be seen as special cases of the Burstall-Darlington framework [3]. The idea behind this framework consists in viewing a program as a set of equations and then transforming this set by inferring new equations. Such a set of equations is essentially a program without the uniqueness constraint on the definitions of its functions (i.e. each function may have several definitions), thus we propose to call it a *polyprogram* (short for "polyvariant program", a term coined by Mikhail Bulyonkov).

Equality saturation [8] may also be considered an instance of the Burstall-Darlington framework if we restrict ourselves to so-called *decomposed polyprograms*. In decomposed polyprograms every definition has a very simple form containing only one nontrivial language construct, thus decomposed polyprograms are essentially closer to ASTs and E-PEGs (E-PEG is a Program Expression Graph with an equivalence relation on nodes [8]), definitions of polyprograms corresponding to nodes and outgoing edges of E-PEGs, and functions corresponding to classes of node equivalence. Decomposed polyprograms can be represented as graphs, or, more precisely, directed hypergraphs, whose nodes correspond to functions and hyperedges to definitions. Thus decomposed polyprograms are better for implementation and formulation of transformations. Every complex definition can be split into several simple definitions by introducing intermediate functions, so every polyprogram can be transformed into a decomposed one.

One of the transformation rules of the Burstall-Darlington framework is called *redefinition*. It allows replacing one function for another if they have isomorphic recursive definitions. In this paper we show how this rule can be formulated using the notion of *polyprogram bisimulation*, which has the benefit of dealing with situations when the definitions are not exactly isomorphic (e.g. the functions are equal only up to argument permutation). Thus, we prefer to call this transformation rule *merging by bisimulation*.

This paper is a continuation of the work on equality saturation for functional languages [5]. In that previous paper the theory behind the implementation wasn't described thoroughly enough, in particular, the semantics wasn't discussed at all, the notion of polyprogram bisimulation wasn't presented, and thus there was no proof of correctness of the bisimulation enumeration algorithm. The present paper discusses these topics in more detail, and its contributions are as follows:

- Articulation of the notion of a polyprogram.
- A polyprogram-based formulation of equality saturation which shows the connection between equality saturation and the Burstall-Darlington framework.
- The notion of polyprogram bisimulation and an algorithm for enumerating polyprogram bisimulations together with a proof of its correctness. This is the main contribution, and most of the paper is devoted to this topic.

The paper is structured as follows: first of all, we describe the language we use throughout the paper and give the definitions of a polyprogram and a decomposed polyprogram in this language (Section 1.), then we show some basic transformation rules (Section 2.), and after that we introduce the notion of polyprogram bisimulation and present an algorithm for enumerating bisimulations (Section 3.).

1. Polyprograms

In this paper we use a simple first-order language. We denote variables with letters x, x_i (from $\mathcal{X}$), functions names with f, f_i (from a set of functional symbols $\mathcal{F}$), and constructors with C, C_i. A set of functional symbols is just a set F equipped with an arity function $arity : F \to \mathbb{N}$.

Definition 1. *A* polyprogram *(in our language) is a set of definitions of the form* $f(x_1, \ldots, x_n) \equiv e$ *where* e *has the following form*[1]*:*

$$e ::= x \mid f(e_1, \ldots, e_m) \mid C(e_1, \ldots, e_m) \mid \textbf{case } e_0 \textbf{ of } \{ \overline{C_i(\overline{x_{ij}}) \ \to \ e_i;} \}$$

where there is no variable duplication in case patterns and in left hand sides of definitions.

In a polyprogram each function is allowed to have any number of definitions. The intention is that such definitions should be semantically equal, but may be different performance-wise. Here is an example of a polyprogram:

$$\begin{aligned} \mathrm{not}(t) &\equiv \textbf{case } t \textbf{ of } \{F \to T; T \to F\} \\ \mathrm{even}(x) &\equiv \textbf{case } x \textbf{ of } \{Z \to T; S(y) \to \mathrm{odd}(y)\} \\ \mathrm{even}(x) &\equiv \mathrm{not}(\mathrm{odd}(x)) \\ \mathrm{odd}(x) &\equiv \textbf{case } x \textbf{ of } \{Z \to F; S(y) \to \mathrm{even}(y)\} \\ \mathrm{odd}(x) &\equiv \mathrm{not}(\mathrm{even}(x)) \end{aligned}$$

Note that the functions even and odd have two definitions each.

Definition 2. *A polyprogram in decomposed form, or just* decomposed polyprogram*, is a polyprogram such that all right hand sides of its definitions have the following form:*

$$\begin{aligned} &e ::= r \mid x \mid f(r_1, \ldots, r_m) \mid C(r_1, \ldots, r_m) \mid \textbf{case } r_0 \textbf{ of } \{ \overline{C_i(\overline{x_{ij}}) \ \to \ r_i;} \} \\ &r ::= f(x_1, \ldots, x_l), \textit{ where all } x_j \textit{ differ from each other} \end{aligned}$$

We call expressions of the form $f(x_1, \ldots, x_l)$, where variables are different, *elementary calls*. That is, every right hand side of a decomposed polyprogram is either an elementary call or an expression such that each of its maximal proper subexpressions is an elementary call.

Decomposed form is not very human-readable, but it is better suited for reasoning and implementation. Consider the following polyprogram consisting of one definition:

$$f(x, z) \equiv \textbf{case } x \textbf{ of } \{Z \to Z; S(y) \to f(y, y)\}$$

To transform it into a decomposed polyprogram, we have to factor out subexpressions x, Z, y, and $f(y, y)$ (the last one because it has a duplicated variable). This gives us the following decomposed polyprogram:

$$\begin{aligned} &f(x, z) \equiv \textbf{case } id(x) \textbf{ of } \{Z \to g(); S(y) \to h(y)\} \\ &id(x) \equiv x \\ &g() \equiv Z \\ &h(y) \equiv f(id(y), id(y)) \end{aligned}$$

[1] $\overline{E\langle e_i \rangle}$ expands to $E\langle e_1 \rangle, \ldots, E\langle e_n \rangle$ for some $n = \max i$

1.1. Polyprogram semantics

It is straightforward to define denotational semantics for polyprograms. However, instead of considering only the least fixed point, we consider all fixed points, or *models*, because this makes semantics compositional, i.e. we can replace polyprogram fragments with semantically equivalent fragments without changing the meaning of the whole polyprogram. This property would not hold if we considered only the least fixed point, because the equivalence based on the least fixed point semantics is too coarse.

Our polyprograms operate first-order values from the set A which is the greatest solution of the following equation (i.e. it includes both finite and infinite data built out of constructors):

$$A = \{C(a_1, \ldots, a_n) \mid a_i \in A,\ C \text{ is a constructor}\} \cup \{\bot\}.$$

Let D be the set of continuous functions [10] over A of arbitrary arity, i.e. $D = \bigcup_n [A^n \to A]$. Now let's define the notion of an interpretation.

Definition 3. *Let P be a polyprogram with the set of function names F. Then an* interpretation *of P is a function $\eta : F \to D$ such that $arity(\eta(f)) = arity(f)$.*

Now let's define the valuation of a term t given an interpretation η and a valuation of variables $\nu : \mathcal{X} \to A$, written $[\![t]\!]_{\eta,\nu}$:

$$\begin{aligned}
&[\![x]\!]_{\eta,\nu} = \nu(x)\\
&[\![f(e_1, \ldots, e_n)]\!]_{\eta,\nu} = \eta(f)([\![e_1]\!]_{\eta,\nu}, \ldots, [\![e_n]\!]_{\eta,\nu})\\
&[\![C(e_1, \ldots, e_n)]\!]_{\eta,\nu} = C([\![e_1]\!]_{\eta,\nu}, \ldots, [\![e_n]\!]_{\eta,\nu})\\
&[\![\textbf{case } e_0 \textbf{ of} \{\overline{C_i(y_1, \ldots, y_m) \to e_i}\}]\!]_{\eta,\nu} = [\![e_k]\!]_{\eta,\nu\{y_i \to a_i\}}\\
&\qquad \text{where } [\![e_0]\!]_{\eta,\nu} = C_k(a_1, \ldots, a_m) \text{ for some } k \in \{1, \ldots, \max i\}
\end{aligned}$$

Given a definition d, its valuation $[\![d]\!]_\eta$ is a Boolean value defined as follows:

$$[\![e_1 \equiv e_2]\!]_\eta = (\forall \nu. [\![e_1]\!]_{\eta,\nu} = [\![e_2]\!]_{\eta,\nu})$$

Definition 4. *An interpretation μ is called a* model *of a polyprogram P if for every definition $d \in P$, $[\![d]\!]_\mu$ is true.*

2. Polyprogram transformation rules

According to both the Burstall-Darlington framework and equality saturation, polyprograms should be transformed with some rules which add new function definitions to a polyprogram. After that a new program may be extracted from the polyprogram by choosing a single definition for each function.

We write transformation rules as $P_1 \mapsto P_2$. Application of such a rule to a polyprogram consists in replacing the subpolyprogram corresponding to the left hand side up to function and variable renaming with the right hand side. We assume that rules may add new definitions and functions, and also remove some definitions

We will consider only a couple of basic rules in two different styles: in the style of the Burstall-Darlington framework and in the style of equality saturation. The latter one assumes that polyprograms are in decomposed form.

The main difference between the two styles is that rules of equality saturation are more local and fine-grained, because equality saturation was originally designed to perform mostly intraprocedural optimizations (inside function bodies) by rewriting expressions. However, for decomposed polyprograms the difference between expressions and functions becomes blurred, so interprocedural transformations become as naturally expressible in equality saturation as intraprocedural.

2.1. Rules in the style of the Burstall-Darlington framework

Unfolding

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \\ f(\overline{x_i}) \equiv E\langle H\{\overline{y_j \mapsto e_j}\}\rangle \end{array}\right\}$$

This rule substitutes a function call with the function body.

Folding

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto z_j}\}\rangle \\ g(\overline{y_j}) \equiv E \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto z_j}\}\rangle \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv H\langle g(\overline{z_j})\rangle \end{array}\right\}$$

This rule does the inverse: it replaces an instance of some function's body with a call of this function. Note that it is less powerful than the corresponding rule from the paper by Burstall and Darlington [3] because it allows replacing only renamings of g's body, not arbitrary special cases.

2.2. Rules in the style of equality saturation

Decomposed polyprograms are very similar to E-PEGs from the work of Tate et al. on equality saturation [8]. They enable more effective sharing of subexpressions, which is crucial for big polyprograms, especially when a simple heuristic-free rule application strategy is used, as in equality saturation.

However, rules in the form from the previous subsection cannot be applied to decomposed polyprograms because their left hand sides are not in decomposed form. One of the solutions to this problem is to rewrite the rules in such a way that both their sides are in decomposed form. In this case the rules will not only be applicable to decomposed polyprograms but also will preserve them in decomposed form.

Transitivity with symmetry

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv g(\overline{y_j}) \end{array}\right\}$$

This rule is analogous to folding. It infers function equivalence from their having coinciding definitions. The correctness of this rule follows from the transitivity and symmetry of equality, hence the name.

Congruence

$$\left\{\begin{array}{l} g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \\ D\langle g(\overline{e_j})\rangle \end{array}\right\} \mapsto \left\{\begin{array}{l} g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \\ D\langle h(\overline{e_{\theta(j)}})\rangle \end{array}\right\}$$

This rule allows propagating information about function equivalence by replacing one function with another. This rule is best applied to every call site of the function being replaced at once: in this case we can simply remove the old function from the polyprogram. We call such a procedure *merging by congruence* since the functions are effectively merged.

Deduplication

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ f(\overline{x_i}) \equiv E \end{array}\right\} \mapsto \left\{\, f(\overline{x_i}) \equiv E \right\}$$

This rule is used after merging by congruence to remove a coinciding definition of a function. Its only purpose is to reduce memory consumption. The three aforementioned rules together implement *congruence closure* [6] which lies at the heart of equality saturation.

Unfolding of a function consisting of a function call

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k(\overline{y_j})}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k(\overline{y_j})}) \\ f(\overline{x_i}) \equiv g(\overline{q_k(\overline{x_i})}) \\ \overline{q_k(\overline{x_i}) \equiv d_k(\overline{e_j(\overline{x_i})})} \end{array}\right\}$$

The unfolding rule breaks apart into several simpler rules. We show only one of these rules for brevity. The most interesting thing is that we don't need a separate operation for substitution into an expression ($E\{x \mapsto e\}$) since non-elementary calls play the role of explicit substitutions.

3. Polyprogram bisimulation

Merging by bisimulation is a generalization of the redefinition rule from the Burstall-Darlington framework. Consider the following polyprogram:

$$\begin{array}{l} f() \equiv S(f()) \\ g() \equiv S(h()) \\ h() \equiv S(g()) \end{array}$$

The goal is to infer $f() \equiv g()$. Turns out, this cannot be done directly with the simple rules mentioned above, so we need something more powerful. In this case the definitions

of these functions are not even isomorphic, so we need a more general relation than isomorphism, which we call a bisimulation because it remotely resembles the notion of bisimulation for labeled transition systems.

3.1. The notion of polyprogram bisimulation

First of all, let's give some auxiliary definitions. If θ is a function from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$ (which we will write just as $\theta : m \to n$) then it can be applied to any functional symbol to permute, omit and duplicate its parameters. We write this application simply as θf and use the following reduction rule:

$$(\theta e_0)(e_1, \ldots, e_n) \rightsquigarrow e_0(e_{\theta(1)}, \ldots, e_{\theta(m)})$$

Definition 5. *A* morphism of functional symbols *from F_1 to F_2 is a function ϕ that maps each functional symbol $f \in F_1$ to a pair $\phi(f) = (\theta, h)$ where $h \in F_2$ and $\theta : arity(h) \to arity(f)$.*

We will often write $\phi(f) = \theta h$ instead of $\phi(f) = (\theta, h)$.

A morphism of functional symbols maps functional symbols, possibly permuting and dropping their parameters. Morphisms of functional symbols can be applied to definitions and polyprograms. If d is a definition then $\phi(d)$ is obtained by replacing all function names f in d with θh, where $\phi(f) = \theta h$, with subsequent normalization with respect to $\rightsquigarrow$. If P is a polyprogram then $\phi(P) = \{\phi(d) \mid d \in P\}$. For example, if $\phi(g) = idg$ and $\phi(f) = \theta h$ where $\theta(1) = 2$ and $\theta(2) = 1$, then $\phi(f(f(x, y), g(x))) = h(g(x), h(y, x))$.

Note that even if P is a polyprogram, $\phi(P)$ is not necessarily a polyprogram, because ϕ may introduce variable duplication in left hand sides of definitions. We call such objects *quasipolyprograms*. However, if ϕ maps every f into a pair (θ, f') such that θ is injective then $\phi(P)$ will be a polyprogram if P is a polyprogram.

By definition, morphism application affects both elementary and non-elementary calls. This actually considerably complicates the theory of polyprogram bisimulations, because the ability of morphisms to rearrange parameters in non-elementary calls requires considering all possible permutations in the bisimulation enumeration algorithm. To simplify things, we use the following semantically equivalent construct instead of a non-elementary call $f(e_1, \ldots, e_n)$:

$$\textbf{case } C(e_1, \ldots, e_n) \textbf{ of } \{\ C(x_1, \ldots, x_n) \to f(x_1, \ldots, x_n)\ \}$$

Since it is not human-readable, $f(e_1, \ldots, e_n)$ will still be used as a syntactic sugar. Note that now morphisms of functional symbols only affect the order of bound variables. Moreover, in subsequent proofs and definitions we will need to consider only three language constructs.

We call two definitions *α-equivalent*, written $d_1 \approx d_2$, if they are equal up to variable renaming. We use the notation $P_1 \lesssim P_2$ if P_1 is a subpolyprogram of P_2 up to α-equivalence of its definitions.

Definition 6. *A* polyprogram bisimulation *over a polyprogram P is a polyprogram B with two morphisms of functional symbols ϕ and ψ such that $\phi(B) \lesssim P$, $\psi(B) \lesssim P$, and if a function $f \in B$ has no definitions then $\phi(f) = \psi(f)$.*

Polyprogram bisimulations are useful for proving equivalence of functions (and subsequently merging them, the transformation we call *merging by bisimulation*) using the following theorem.

Theorem 1. *Let B with ϕ and ψ be a polyprogram bisimulation over P. If for every interpretation ν of B's functions without definitions there is only one model μ that coincides with ν on B's functions without definitions then it is possible to add definitions of the following form into P for each function $f \in B$ without changing the set of P's models:*

$$g(x_{\theta(1)}, \ldots, x_{\theta(m)}) \equiv h(x_{\xi(1)}, \ldots, x_{\xi(n)})$$

where $(\theta, g) = \phi(f)$, $(\xi, h) = \psi(f)$, $m = arity(g)$, $n = arity(h)$.

Proof. Let μ be a model of P. Then there are two models of B: $\mu_\phi(f) = \theta\mu(g)$ where $(\theta, g) = \phi(f)$, and $\mu_\psi(f) = \xi\mu(h)$ where $(\xi, h) = \psi(f)$. But for every function $f \in B$ with no definitions $\phi(f) = \psi(f)$, hence $\mu_\phi(f) = \mu_\psi(f)$. Since for every interpretation ν of B's functions without definitions there is only one model of B, models μ_ϕ and μ_ψ must be equal. This means that for every function $f \in B$, $\theta\mu(g) = \xi\mu(h)$, i.e. $\mu(g)(x_{\theta(1)}, \ldots, x_{\theta(m)}) = \mu(h)(x_{\xi(1)}, \ldots, x_{\xi(n)})$. This is exactly the semantics of the new definitions, so they can be safely added to P, and μ will still be a model of the augmented quasipolyprogram.

Since adding definitions cannot expand the set of models, the assertion of the theorem is proved. □

Not every bisimulation is good enough for merging by bisimulation, because it must also satisfy the model uniqueness property. To test this property some decidable sufficient conditions may be used, like structural and guarded recursion [1], or the presence of ticks [7]. This topic is out of scope of this paper.

3.2. Enumerating bisimulations

In this section we assume that polyprograms are in decomposed form and non-elementary calls are encoded using case expressions. There is an infinite number of polyprogram bisimulations, so we are going to present an algorithm that produces an infinite stream polyprogram bisimulations, but only of some specific form. The algorithm can be informally outlined in the following way: first enumerate *prebisimulations*, quasipolyprograms that are precursors of bisimulations, and then transform each prebisimulation into a polyprogram bisimulation if possible.

Prebisimulations will be quasipolyprograms consisting of *products of definitions* of the original polyprogram. Its idea of a product of two definitions *def-product*(d_1, d_2) is to combine every pair of corresponding functions into a single one with their parameter lists concatenated, e.g.:

$$\begin{aligned}
&\textit{def-product}((f(x) \equiv x), (g(y) \equiv y)) = (\langle f, g\rangle(x, x) \equiv x)\\
&\textit{def-product}((f(x, z) \equiv \mathbf{case}\ h()\ \mathbf{of}\{S(y) \to g(x, y)\}),\\
&\qquad\qquad (f'(x) \equiv \mathbf{case}\ h'(x)\ \mathbf{of}\{S(y) \to g'(y, x)\})) =\\
&\qquad = (\langle f, f'\rangle(x, z, x') \equiv \mathbf{case}\ \langle h, h'\rangle(x')\ \mathbf{of}\{S(y) \to \langle g, g'\rangle(x, y, y, x')\})
\end{aligned}$$

Definition 7. *Let d and d' be two decomposed definitions with the same language construct, the same number of function calls and the same number of variables bound by corresponding patterns. Assume also that corresponding variables in corresponding patterns of the two definitions have coinciding names, and also if the right hand sides have the form x then x is the same for both definitions, but there are no more variable collisions between the definitions (these requirements may be satisfied by applying α-conversion). Then the* product of these definitions *is defined as follows:*

$$\begin{aligned}
&\mathit{def\text{-}product}(f(x_1,\ldots,x_n) \equiv x_i, f'(y_1,\ldots,y_m) \equiv y_j) = \\
&\quad = \langle f, f'\rangle(x_1 \ldots x_n, y_1 \ldots y_{j-1}, x_i, y_{j+1} y_m) \equiv x_i \\
&\mathit{def\text{-}product}(f(\overline{x}) \equiv C(g_1(\overline{x}_1),\ldots,g_n(\overline{x}_n)), \\
&\qquad\qquad f'(\overline{y}) \equiv C(g'_1(\overline{y}_1),\ldots,g'_n(\overline{y}_1))) = \\
&\qquad\qquad\quad = \langle f, f'\rangle(\overline{x},\overline{y}) \equiv C(\langle g_1, g'_1\rangle(\overline{x}_1,\overline{y}_1),\ldots) \\
&\mathit{def\text{-}product}(f(\overline{x}) \equiv \mathbf{case}\ g_0(\overline{x}_0)\ \mathbf{of}\{C_1(\overline{z}_1) \to g_1(\overline{x}_1);\ldots\}, \\
&\qquad\qquad f'(\overline{y}) \equiv \mathbf{case}\ g'_0(\overline{y}_0)\ \mathbf{of}\{C_1(\overline{z'}_1) \to g'_1(\overline{x'}_1);\ldots\}) = \\
&\quad = \langle f, f'\rangle(\overline{x},\overline{y}) \equiv \mathbf{case}\ g_0(\overline{x}_0, \overline{x'}_0)\ \mathbf{of} \\
&\qquad\qquad\qquad \{C_1(\overline{z}_1) \to \langle g_1, g'_1\rangle(\overline{w}_1, \overline{w'}_1\{\overline{z'}_1 \mapsto \overline{z}_1\});\ldots\}
\end{aligned}$$

Here $\langle f, g\rangle$ denotes a unique functional symbol corresponding to f and g with arity $\mathit{arity}(f) + \mathit{arity}(g)$.

We enumerate prebisimulations in the form of trees with back edges growing from a pair of functions. The enumeration procedure is shown in Fig. 1 in the form of a function taking a polyprogram and two functional symbols and returning a set of prebisimulations (programmatically this function should be implemented as returning a stream). Its idea is to traverse the polyprogram P in depth-first order simultaneously from the functions f and f'. A branch may be finished if we encounter a pair of coinciding functions (reflexivity) or a pair of functions which we have already visited (folding). If we do not finish the branch then we choose a pair of definitions of these functions such that the product of these definitions is defined, and descend to pairs of corresponding functions in their right hand sides.

Now let's define two morphisms, $\pi_1(\langle f_1, f_2, l\rangle) = (\gamma_1, f_1)$ where $\gamma_1(i) = i$, and $\pi_2(\langle f_1, f_2, l\rangle) = (\gamma_2, f_2)$ where $\gamma_2(i) = i + \mathit{arity}(f_1)$. A prebisimulation with π_1 and π_2 is not yet a bisimulation for two reasons: it is not a polyprogram because of variable duplication, and π_1 and π_2 differ on functions without definitions.

To transform a prebisimulation into a polyprogram bisimulation duplicated variables should be merged. To do so, we propagate the information about variable equivalence, thus making the quasipolyprogram "coarser". If this process succeeds, the resulting quasipolyprogram may be transformed into a polyprogram, moreover, it will be a bisimulation.

Definition 8. *A quasipolyprogram Q_1 is* no more coarse *than Q_2, written $Q_1 \sqsubseteq Q_2$, if their definitions are in one-to-one correspondence, and for every definition $d_1 \in Q_1$ the corresponding definition d_2 is α-equal to $d_1\{x \mapsto y\}$ for some variables x and y.*

$$
\begin{aligned}
&\mathit{prebisimulations}(P, f, f') = B \\
&\qquad \text{where } (\_, B) = \mathit{prebisimulations}'(P, f, f', \{\}) \\
&\mathit{prebisimulations}'(P, f, f', \mathit{history}) \\
&\quad = \{(f_{new}, \{\}) \mid f = f'\} \\
&\quad \cup \{(f_{old}, \{\}) \mid (f, f', f_{old}) \in \mathit{history}\} \\
&\quad \cup \{(f_{new}, \{q'\} \cup B_1 \cup \ldots \cup B_n) \mid \\
&\qquad\qquad d = (f(\ldots) \equiv L(\ldots)) \in P, \\
&\qquad\qquad d' = (f'(\ldots) \equiv L(\ldots)) \in P, \\
&\qquad\qquad \text{such that } \mathit{def\text{-}product}(d, d') \text{ is defined}, \\
&\qquad\qquad q = \mathit{def\text{-}product}(d, d'), \\
&\qquad\qquad (\langle f, f' \rangle(\ldots) \equiv L(\langle g_1, g_1' \rangle(\ldots), \ldots)) = q, \\
&\qquad\qquad \mathit{history}' = \mathit{history} \cup \{(f, f', f_{new})\}, \\
&\qquad\qquad (h_i, B_i) \in \mathit{prebisimulations}'(P, g_i, g_i', \mathit{history}'), \\
&\qquad\qquad q' \text{ is } q \text{ with } \langle f, f' \rangle \text{ replaced with } f_{new} \\
&\qquad\qquad\qquad \text{and } \langle g_i, g_i' \rangle \text{ replaced with } h_i\} \\
&\quad \text{where } f_{new} = \langle f, f', l \rangle \text{ where } l \text{ is a fresh unique label}, \\
&\quad f_{new} \text{ has arity } \mathit{arity}(f) + \mathit{arity}(f')
\end{aligned}
$$

Figure 1. Enumeration of prebisimulations

Information about variable equivalence may be propagated with the following transformation.

Definition 9. *Let Q be a quasipolyprogram and $f(x_1, \ldots, x_n)$ be a term from some of Q's definitions such that x_i and x_j are one and the same variable. The following transformation is called* variable equivalence propagation step*: for some definition of Q containing a term $f(y_1, \ldots, y_n)$ replace y_j with y_i in the whole definition.*

Example 1. *Consider the following polyprogram:*

$$
\begin{aligned}
g(x, y, z) &\equiv C(f(x, y), g(z, x, y)) \\
f(x, x) &\equiv x
\end{aligned}
$$

Since $f(x, x)$ in the second definition has a variable duplication, it can be propagated to the first definition by replacing y with x, leading to the following definition:

$$g(x, x, z) \equiv C(f(x, x), g(z, x, x))$$

Now $g(x, x, z)$ contains variable duplication, so variable equivalence may be propagated further, resulting in the definition $g(x, x, x) \equiv C(f(x, x), g(x, x, x))$.

Sometimes variable equivalence propagation may lead to variable duplication in patterns (like **case** $h(x)$ **of**$\{C(x, x) \to f(x, x)\}$*). Quasipolyprograms containing such definitions will not lead to bisimulations and may be filtered out.*

If Q' is derived from Q by a variable equivalence propagation step then $Q \sqsubseteq Q'$ by the definition of $\sqsubseteq$. In particular, it means that we can fully propagate variable equivalence in a finite number of steps. Another important property of variable equivalence propagation is that it commutes with $\sqsubseteq$ in the following sense.

Proposition 1. *If $Q_1 \sqsubseteq Q_2$ and it is possible to apply a variable equivalence propagation step to Q_1 and get Q_1' then it is possible to apply no more than one variable equivalence propagation step to Q_2 and get Q_2' such that $Q_1' \sqsubseteq Q_2'$.*

This implies confluence of variable equivalence propagation.

Before performing variable equivalence propagation we need to fix another issue of a prebisimulation: we need to equate corresponding variables of functions without definitions. That is, for each call of function without definitions of the form

$$\langle f, f, l\rangle(x_1, \ldots, x_m, y_1, \ldots, y_m),$$

where $m = \mathit{arity}(f)$, replace y_i with x_i in the whole definition containing this call. This transformation is very similar to a variable equivalence propagation step, and the resulting quasipolyprogram is more or equally coarse than the original prebisimulation.

If variable equivalence is fully propagated then for every term $f(x_1, \ldots, x_n)$ from the quasipolyprogram, if variables x_i and x_j coincide then for every other term $f(y_1, \ldots, y_n)$ variables y_i and y_j also coincide. In this case the following proposition may be applied to transform the quasipolyprogram into a polyprogram.

Proposition 2. *Let Q be a quasipolyprogram. Assume that for each function f of arity n, the set $\{1 \ldots n\}$ of its argument positions can be partitioned into m equivalence classes such that if i and j are from the same class then for every term of the form $f(x_1, \ldots, x_n)$ from Q the variables x_i and x_j coincide. Then there is a pair of morphisms, σ and σ' such that $\sigma'(\sigma(Q)) = Q$. Moreover, if for every term of the form $f(y_1, \ldots, y_n)$ such that variables y_i and y_j coincide, i and j are from the same class, then the quasipolyprogram $\sigma(Q)$ is a polyprogram.*

Proof. For each function $f \in Q$ of arity n with m equivalence classes there is a mapping $\xi_f : n \to m$ that maps each position to the corresponding equivalence class index, and a mapping $\xi_f^{-1} : m \to n$, its right inverse, which maps each equivalence class index to some representative. Let's define $\sigma(f) = (\xi_f^{-1}, f')$ where f' is a function with arity m corresponding to the function f, and $\sigma'(f') = (\xi_f, f)$.

Note that σ' is not a left inverse of σ, so we need to prove that $\sigma'(\sigma(Q)) = Q$. For each definition $d \in Q$, $\sigma'(\sigma(d))$ will replace each occurrence of $f(x_1, \ldots, x_n)$ with $(\xi_f^{-1}\xi_f f)(x_1, \ldots, x_n)$ which reduces to to $f(x_{\xi_f^{-1}\xi_f(1)}, \ldots, x_{\xi_f^{-1}\xi_f(n)})$. Now if $\xi_f^{-1}\xi_f(i) = j$ then i and j are from the same class, so $x_j = x_i$ by the hypothesis of the proposition, and we can rewrite this term as $f(x_1, \ldots, x_n)$ which is equal to the corresponding term in d. Therefore $\sigma'(\sigma(d)) \approx d$, and thus $\sigma'(\sigma(Q)) \approx Q$.

Now assume that for every term from Q of the form $f(y_1, \ldots, y_n)$, if variables y_i and y_j coincide then i and j are from the same class. In this case any such term will be mapped by σ into the term $f'(y_{\xi_f^{-1}(1)}, \ldots, y_{\xi_f^{-1}(m)})$ which cannot contain duplicate variables, because if $y_{\xi_f^{-1}(l)}$ and $y_{\xi_f^{-1}(k)}$ $(l \neq k)$ coincide then $\xi_f(\xi_f^{-1}(l)) = \xi_f(\xi_f^{-1}(k))$, and consequently $l = k$. Then in this case $\sigma(Q)$ is a polyprogram. □

Now let's combine everything into a polyprogram bisimulation enumeration algorithm, expressed as a function returning a set.

Definition 10.

$$\begin{aligned}
&\mathit{bisimulations}(P, f, f') = \\
&\quad = \{\langle B, \pi'_1, \pi'_2\rangle \mid Q \in \mathit{prebisimulations}(P, f, f'),\\
&\qquad\qquad Q' \text{ is } Q \text{ with corresponding variables of functions}\\
&\qquad\qquad\qquad \text{without definitions equated,}\\
&\qquad\qquad Q'' \text{ is } Q' \text{ with variable equivalence fully propagated,}\\
&\qquad\qquad B = \sigma(Q'') \text{ is converted from } Q'' \text{ by Proposition 2,}\\
&\qquad\qquad \text{let } \pi'_1 = \pi_1 \circ \sigma' \text{ and } \pi'_2 = \pi_2 \circ \sigma',\\
&\qquad\qquad B \text{ with } \pi'_1 \text{ and } \pi'_2 \text{ is a polyprogram bisimulation}\}
\end{aligned}$$

Note that we still need to check if the result is a polyprogram bisimulation, because variable equivalence propagation may equate too much, resulting in $\pi'_i(B)$ not being subpolyprograms of P.

The presented algorithm is not strictly complete, since it enumerate bisimulations only of a certain shape, but it is still possible to prove that if there is a bisimulation of this shape, then an equivalent bisimulation will be found by the algorithm.

Theorem 2. *Let P be a polyprogram and B with ϕ and ψ be a polyprogram bisimulation over it such that:*

- *Every functions of B has no more than one definition.*
- *B has a shape of a tree with back edges (as if it was built by depth-first search).*

Let $\phi(s) = (\_, s_\phi)$ and $\psi(s) = (\_, s_\psi)$. Then there is a polyprogram bisimulation $R \in \mathit{bisimulations}(P, s_\phi, s_\psi)$ with ϕ_R and ψ_R such that $\phi(B) \approx \phi_R(R)$ and $\psi(B) \approx \psi_R(R)$.

The proof is omitted for brevity.

Note that the found bisimulation may differ from the original one despite their images being equal. Consider the following polyprogram:

$$\begin{aligned}
f(x) &\equiv S(f(x))\\
g(x) &\equiv S(g(x))
\end{aligned}$$

We can construct a bisimulation $B = \{h(x) \equiv S(h(x))\}$ with morphisms $\phi = \{h \mapsto (id, f)\}$ and $\psi = \{h \mapsto (id, g)\}$ which will lead to the definition $f(x_1) \equiv g(x_1)$ being added to the polyprogram.

But the function *bisimulations* will not return this bisimulation. Indeed, it will find a prebisimulation $D = \{\langle f, g, l\rangle(x, y) \equiv S(\langle f, g, l\rangle(x, y))\}$, but this prebisimulation has variable equivalence information fully propagated, and it will not be changed during conversion to a polyprogram. So the resulting bisimulation will be exactly D with $\pi_1 = \{\langle f, g, l\rangle \mapsto (\{1 \mapsto 1\}, f)\}$ and $\pi_2 = \{\langle f, g, l\rangle \mapsto (\{1 \mapsto 2\}, g)\}$. This bisimulation will lead to the definition $f(x_1) \equiv g(x_2)$ which is better than $f(x_1) \equiv g(x_1)$ because it indicates that the parameters of the both functions are dummy.

3.3. Performance tricks and limitations

The described algorithm is quite inefficient if implemented directly, so in practice it is important to apply some tricks.

- The algorithm was described in a modular way: first enumerate prebisimulations, then try to convert them to bisimulations. In practice it is much more efficient to filter out classes of prebisimulations which cannot be converted to bisimulations before they are fully constructed. To do this we need to add a parameter to the function *prebisimulations* describing the relationship between variables we have inferred so far from the parent pairs of definitions and check if it does not contradict the information we can infer from the pair of definitions we are currently considering to add to the prebisimulation. This will actually perform partial variable equivalence propagation. Note though that we still need to fully propagate variable equivalence in the end since this trick cannot filter out all prebisimulations not leading to bisimulations.

- Since we are only interested in bisimulations for which we can prove model uniqueness, we can also partially check these model uniqueness conditions in the function *prebisimulations*. They should be usually checked during folding to one of the predecessor pairs of functions (e.g. forbid folding if we haven't passed through a constructor or a pattern matching).

- For many pairs of functions it is immediately obvious that we cannot find a bisimulation with a unique model growing from this pair of functions, because they just cannot be equal. This information can be inferred by analyzing certain definitions of these functions, for example if one function has a definition $f(\ldots) \equiv C(\ldots)$, and the other has $g(\ldots) \equiv D(\ldots)$, then they cannot be equal because of different top-level constructors. Another way is to run the functions on some data to find counterexamples to their equivalence.

- Running functions on test data can also help to infer more information about variable equivalence which can be used in conjunction with the first trick.

- The algorithm enumerates an infinite number of bisimulations, which may be useless in practice. Of course, we can limit the depth of our search with $kn^2m!$, where k is the number of definitions, n is the number of functions and m is the maximal arity: all deeper bisimulations will be just equivalent to some more shallow bisimulations. But this is still a big number, so in practice it is better to limit the number of generated prebisimulations and use some depth-limiting heuristics (like limiting the factor of loop unrolling). Note that if all the aforementioned filtering tricks are used, then usually the first found prebisimulation will be a bisimulation with a unique model.

- Sometimes we visit the same pair of functions several times, so we can memoize sets of prebisimulations. It is especially important in the case when we want to find prebisimulations for all pairs of functions. Note though that the set of prebisimulations depends not only on the pair of functions, but also on the history.

4. Related work

Polyprograms are essentially systems of equations from the Burstall-Darlington framework [3]. Decomposed polyprograms are closer to AST and can be used to implement equality saturation [8] for functional languages, which indicates that equality saturation may be seen as another instance of the Burstall-Darlington framework.

Our definition of polyprogram bisimulation is not relational and instead it is based on the notion of a span, although it can be reformulated in relational form. It should be noted that polyprogram bisimulation and LTS bisimulation are quite different since nondeterminism in polyprograms is not related to nondeterminism in LTS. Polyprogram bisimulation also resembles the notion of term graph bisimilarity [2].

Polyprogram bisimulation is used to implement merging by bisimulation, a generalization of the redefinition rule from the Burstall-Darlington framework. Correctness of merging by bisimulation has to be established for each bisimulation by checking additional conditions which are not discussed in this paper. These conditions may be based on termination and productivity conditions [1] as in our previous work, or on the theory of improvement [7] as in the supercompiler of Ilya Klyuchnikov [11].

Our bisimulation enumeration algorithm is related to finding intersection of two languages of term equalities [4]. It is also structurally very similar to supercompilation [9].

5. Conclusion

In this paper we have introduced the notions of a polyprogram and a decomposed polyprogram. A polyprogram is a generalization of a program which admits multiple definitions of a single function. Decomposed polyprograms are polyprograms whose definitions are decomposed into definitions of the simplest form by introducing intermediate functions. Decomposed polyprograms are better suited for reasoning and implementation.

We have presented several main transformation rules in two styles: in the style of the Burstall-Darlington framework for ordinary polyprograms and in the style of equality saturation for decomposed polyprograms. This shows the connection between the two program transformation methods.

We have also introduced the notion of polyprogram bisimulation, on which merging by bisimulation is based. We have presented a bisimulation enumeration algorithm, which enumerates polyprogram bisimulations of a specific form, and proved its correctness.

References

[1] Abel A., Altenkrich T., “A predicative analysis of structural recursion”, *Journal of Functional Programming*, **12**:1 (2002), 1–41.

[2] Ariola Z. M., Klop J. W., Plump D., “Bisimilarity in Term Graph Rewriting”, *Information and Computation*, **156** (2000), 2–24.

[3] Burstall R. M., Darlington J., “A transformation system for developing recursive programs”, *Journal of the ACM*, **24**:1 (1977), 44–67.

[4] Emelyanov P., “Analysis of equality relationships for imperative programs”, *CoRR*, 2006, https://arxiv.org/abs/cs/0609092.

[5] Grechanik S., "Inductive prover based on equality saturation (extended version)", *Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation*, 2014, 26–53.

[6] Nelson G., Oppen D.C., "Fast decision procedures based on congruence closure", *Journal of the ACM*, **27**:2 (1980), 356–364.

[7] Sands D., "Total correctness by local improvement in the transformation of functional programs", *ACM TOPLAS*, **18**:2 (1996), 175–234.

[8] Tate R., Stepp M., Tatlock Z., Lerner S., "Equality saturation: a new approach to optimization", *SIGPLAN Not.*, **44**:1 (2009), 264–276.

[9] Turchin V., "The concept of a supercompiler", *ACM TOPLAS*, **8**:3 (1986), 292–325.

[10] Scott D.S., "Domains for Denotational Semantics", *Automata, Languages, and Programming: 9th Colloquium Aarhus, Denmark*, Lecture Notes in Computer Science, **140**, 1982, 577–610.

[11] Klyuchnikov I., Romanenko S., "Towards Higher-Level Supercompilation", *Proceedings of the Second International Workshop on Metacomputation in Russia*, 2010, 82–101.

Гречаник С. А., "Полипрограммы и бисимуляция полипрограмм", *Моделирование и анализ информационных систем*, **25**:5 (2018), 534–548.

DOI: 10.18255/1818-1015-2018-5-534-548

Аннотация. Полипрограмма — это обобщение программы, допускающее множественность определений одной и той же функции. Подобные объекты возникают в различных системах преобразования программ, таких как система Бёрстолла–Дарлингтона и насыщение равенствами. В данной работе мы вводим понятие полипрограммы на нестрогом функциональном языке первого порядка. Мы определяем денотационную семантику полипрограмм и описываем некоторые преобразования полипрограмм в двух разных стилях: в стиле системы Бёрстолла–Дарлингтона и в стиле насыщения равенствами. Преобразования в стиле насыщения равенствами осуществляются над полипрограммами в расчленённой форме, в которой стирается грань между функциями и выражениями и между подстановкой и раскрытием вызова функции. Расчленённые полипрограммы хорошо подходят для реализации и проведения рассуждений, но трудны для человеческого восприятия. Мы также вводим понятие бисимуляции полипрограмм, на котором основано преобразование — слияние по бисимуляции, соответствующее доказательству эквивалентности функций по индукции или коиндукции. Бисимуляция полипрограмм — понятие, вдохновлённое понятием бисимуляции размеченных систем переходов, но несколько от него отличающееся, поскольку бисимуляция полипрограмм рассматривает каждое определение как самодостаточное, т.е. функция полипрограммы задаётся любым своим определением, в то время как в размеченной системе переходов поведение системы в состоянии определяется всей совокупностью переходов, которые можно осуществить из этого состояния. Мы предлагаем алгоритм перечисления бисимуляций некоторого определённого вида. Алгоритм состоит из двух фаз: перечисление пребисимуляций и преобразование их в бисимуляции. Такое разделение требуется из-за того, что бисимуляции полипрограмм учитывают возможность перестановки параметров функций. Мы доказываем корректность данного алгоритма, а также формулируем некоторую слабую форму его полноты. Статья публикуется в авторской редакции.

Ключевые слова: полипрограммы, преобразование программ, насыщение равенствами, бисимуляция

Об авторах:
Гречаник Сергей Александрович, orcid.org/0000-0001-8575-9689, канд. физ.-мат. наук,
Институт прикладной математики им. М.В. Келдыша РАН,
Миусская пл., 4, г. Москва, 125047 Россия, e-mail: sergei.grechanik@gmail.com

Благодарности:
Работа выполнена при поддержке гранта РФФИ №18-31-00412