Generalized Homogeneous Polynomials for Efficient Template-Based Nonlinear Invariant Synthesis

Kensuke Kojima, Minoru Kinoshita, Kohei Suenaga

Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University, Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan.

Abstract

The *template-based* method is one of the most successful approaches to algebraic invariant synthesis. In this method, an algorithm designates a *template polynomial* p over program variables, generates constraints for $p = 0$ to be an invariant, and solves the generated constraints. However, this approach often suffers from an increasing template size if the degree of a template polynomial is too high.

We propose a technique to make template-based methods more efficient. Our technique is based on the following finding: If $p = 0$ is an algebraic invariant, then p can be decomposed into the sum of specific polynomials that we call *generalized homogeneous* polynomials, that are often smaller. This finding justifies using only a smaller template that corresponds to a generalized homogeneous polynomials.

Concretely, we state and prove our finding above formally. Then, we modify the template-based algorithm proposed by Cachera et al. so that it generates only generalized homogeneous polynomials. This modification is proved to be sound. Furthermore, we also empirically demonstrate the merit of the restriction to generalized homogeneous polynomials. Our implementation outperforms that of Cachera et al. for programs that require a higher-degree template.

Keywords: static analysis, polynomial invariants, homogeneous polynomial

1. Introduction

We consider the following *postcondition problem*: Given a program c, discover a fact that holds at the end of c regardless of the initial state. This article focuses on a postcondition written as an *algebraic condition* $p_1 = 0 \wedge \cdots \wedge p_n = 0$, where $p_1, \ldots, p_n$ are polynomials over program variables; this problem is a basis for static verification of functional correctness.

One approach to this problem is *invariant synthesis*, in which we are to compute a family of predicates P_l indexed by program locations l such that P_l

```
1: x := x_0; v := v_0; t := t_0;
2: while t − a ≠ 0 do
3:     (x, v, t) := (x + v dt, v − g dt, t + dt);
4: end while
5:
```

Figure 1: Program c_{fall}, which models a falling mass point. The symbols in the program represent the following quantities: x is the position of the point, v is its speed, t is time, x_0 is the initial position, v_0 is the initial speed, t_0 is the initial value of the clock t, g is the acceleration rate, and dt is the discretization interval. The simultaneous substitution in the loop body numerically updates the values of x, v, and t. The values of x, v, and t are numerical solutions of the differential equations $\frac{dx}{dt} = v$ and $\frac{dv}{dt} = -g$.

holds whenever the execution of c reaches l. The invariant associated with the end of c is a solution to the postcondition problem.

Because of its importance in static program verification, algebraic invariant synthesis has been intensively studied [1, 2, 3, 4]. Among these proposed techniques, one successful approach is the constraint-based method in which invariant synthesis is reduced to a constraint-solving problem. During constraint generation, this method designates *templates*, which are polynomials over the program variables with unknown parameters at the coefficient positions [1]. The algorithm generates constraints that ensure that the templates are invariants, and obtains the invariants by solving the constraints.[1]

Example 1. The program c_{fall} in Figure 1 models the behavior of a mass point with weight 1 and with a constant acceleration rate.[2] For this program, the postcondition $-gt + gt_0 - v + v_0 = 0$ holds regardless of the initial state.

We describe how a template-based method computes the postcondition in Example 1. The method described here differs from the one we explore in this article; this explanation is intended to suggest the flavor of a template method.

A template-based method generates a *template polynomial* over the program variables that represent a postcondition. For example, the template polynomial of degree 2 is

$$p(x_0, v_0, t_0, x, v, t, a, dt, g) := a_1 + a_{x_0}x_0 + a_{v_0}v_0 + \cdots + a_{dtg}dtg,$$

where $a_1, a_{x_0}, a_{v_0}, \ldots, a_{dtg}$ are unknown parameters representing coefficients of $1, x_0, v_0, \ldots, dtg$, respectively. The procedure then generates constraints under which $p(x_0, v_0, \ldots, g) = 0$ is indeed a postcondition of c_{fall}. The method proposed by Sankaranarayanan et al. [1] based on the Gröbner basis [5] generates the constraints as equations over the parameters; in this case, a solution to the constraints gives $-gt + gt_0 - v + v_0 = 0$, which indeed holds at the end of c_{fall}.

[1] The constraint-based method by Cachera et al. [4], which is the basis of the current article, uses a template also for other purposes. See Section 5 for details.

[2] Although the guard condition $t - a \neq 0$ should be $t - a < 0$ in a real-world numerical program, we use the current example for presentation purposes.

One of the drawbacks of the template-based method is excessive growth of the size of a template. Blindly generating a template of degree d for a degree parameter d makes the algorithm less scalable for higher-degree postconditions. For example, the program in Example 1 has a postcondition $-gt^2 + gt_0^2 - 2tv + 2t_0v_0 + 2x - 2x_0 = 0$ at Line 4. This invariant requires a degree-3 template, which has $\binom{9+3}{3} = 220$ monomials in this case.

We propose a hack to alleviate this drawback in the template-based methods. Our method is inspired by a rule of thumb in physics called the *principle of quantity dimension*: A physical law should not add two quantities with different *quantity dimensions* [6]. If we accept this principle, then, at least for a physically meaningful program such as c_{fall}, its postcondition (and therefore a template) should consist of monomials with the same quantity dimensions.

Indeed, the polynomial $-gt + gt_0 - v + v_0$ calculated in Example 1 consists only of quantities that represent velocities. The polynomial $-gt^2 + gt_0^2 - 2tv + 2t_0v_0 + 2x - 2x_0$ above consists only of quantities corresponding to the length. If we use L and T to represent quantity dimensions for lengths and times (the notation in physics), the first and second polynomials consist only of monomials with the quantity dimension LT^{-1} and L, respectively.

By leveraging the quantity dimension principle in the template synthesis phase, we can reduce the size of a template. For example, we could use a template that consists only of monomials for, say, velocity quantities

$$a_{v_0}v_0 + a_v v + +a_{tg}tg + a_{dtg}dtg + a_{ag}ag$$

instead of the general degree-2 polynomial, which consists of 55 monomials.

The idea of the quantity dimension principle can be nicely captured by generalizing the notion of *homogeneous polynomials*. A polynomial is said to be *homogeneous* if it consists of monomials of the same degree; for example, the polynomial $x^3 + x^2y + xy^2 + y^3$ is a homogeneous polynomial of degree 3. We generalize this notion of homogeneity so that (1) a *degree* is an expression corresponding to a quantity dimension (e.g., LT^{-1}) and (2) each variable has its own degree in degree computation.

Let us describe our idea using an example, deferring formal definitions. Suppose we have the following *degree assignment* for each program variable:

$$\Gamma := \left\{ \begin{array}{l} x_0 \mapsto L, t_0 \mapsto T, g \mapsto LT^{-2}, t \mapsto T, dt \mapsto T, \\ x \mapsto L, v \mapsto LT^{-1}, v_0 \mapsto LT^{-1}, a \mapsto T \end{array} \right\}.$$

This degree assignment intuitively corresponds to the assignment of the quantity dimension to each variable. With this degree assignment Γ, all of the monomials in $-gt + gt_0 - v + v_0$ have the same degree; for example, the monomial $-gt$ has degree $\Gamma(g)\Gamma(t) = (LT^{-2})T = LT^{-1}$. Hence, $-gt+gt_0-v+v_0$ is a homogeneous polynomial in the generalized sense. We call such a polynomial a *generalized homogeneous (GH) polynomial*. We call an algebraic postcondition with a GH polynomial a *generalized homogeneous algebraic (GHA) postcondition*.

The main theoretical result of this article is a formalization of this idea. We can prove that, if there is an algebraic postcondition $p = 0$ of a program c,

then p is actually written as the sum of GH polynomials $p = p_1 + \cdots + p_n$, and $p_i = 0$ are all GHA postconditions of c (Theorem 11). In this sense, there exist enough GHA postconditions, and this fact justifies the use of a template that corresponds to a GH polynomial in the template method. We also empirically show that the algorithm by Cachera et al. [4] can be made more efficient using this idea.

As we saw above, the definition of GH polynomials is parameterized over a degree assignment Γ. The problem of finding an appropriate degree assignment can be solved by applying the type inference algorithm for the *dimension type system* proposed by Kennedy [7, 8]; Γ above is inferred using this algorithm. The dimension type system was originally proposed for detecting a violation of the quantity-dimension principle in a numerical program. Our work gives an application of the dimension type system to postcondition synthesis.

Although the method is inspired by the principle of quantity dimensions, it can be applied to a program that does not model a physical phenomenon because we abstract the notion of a quantity dimension using that of generalized homogeneity. All the programs used in our experiments (Section 6) are indeed physically nonsensical programs.

To summarize, our main contributions are (1) theoretically, to prove that there exist enough GHA postconditions, and (2) empirically, to demonstrate that the notion of GH polynomials is indeed useful for efficient template-based invariant synthesis by reducing the size of templates.

The rest of this article is organized as follows. Section 2 defines the notion of generalized homogeneity. Section 3 introduces the target language, and describe degree-assignment inference algorithm. Section 4 states and proves the main theorem. Section 5 gives a template-based invariant-synthesis algorithm. Section 6 reports the experimental results. Section 7 discusses related work, and Section 8 presents the conclusions.

This article is a revised and reorganized version of the authors' previous work [9]. The main difference from the previous version is that the main theorem (Theorem 11) is proved with respect to the standard operational semantics; a similar result was previously proved with respect to an abstract semantics. We believe that the current result better justifies the claim of the existence of enough GHA postconditions. In addition, as a result of this change, abstract semantics is not needed when stating the theoretical results, and this significantly simplifies the presentation.

2. Generalized Homogeneous Polynomials

Throughout the article, we use $\mathbb{R}$, $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}[x_1, \ldots, x_n]$ for the set of all real numbers, nonnegative integers, integers, and polynomials in variables $x_1, \ldots, x_n$ over $\mathbb{R}$, respectively.

A polynomial p is said to be homogeneous of degree d if the degree of every monomial in p is d [5]. As we mentioned in Section 1, we generalize this notion of homogeneity to obtain a mathematical formulation of the consistency of quantity dimensions.

We first generalize the notion of the degree of a polynomial.

Definition 2. Let B be a set. We define a *generalized degree (g-degree)* over B as an element of the free Abelian group generated by B. More concretely, a generalized degree over B is an expression of the form $b_1^{n_1} \cdots b_m^{n_m}$ where $b_1, \ldots, b_m \in B$ and $n_1, \ldots, n_m \in \mathbb{Z}$ (we allow $m = 0$, in which case the expression is denoted by 1). Two expressions are identified if they are equal up to permutation and the laws of exponents. For example, $b_1^2 b_2$, $b_2 b_1^2$, $b_1 b_2 b_1$, $b_2^2 b_1^2 b_2^{-1}$, and $b_1^2 b_2 b_3^0$ all denote the same g-degree. We call elements of B *base degrees*.

We denote the set of all generalized degrees by $\mathbf{GDeg}_B$. The set $\mathbf{GDeg}_B$ is equipped with multiplication defined by

$$(b_1^{n_1} \cdots b_m^{n_m}) \cdot (b_1^{r_1} \cdots b_m^{r_m}) := b_1^{n_1+r_1} \cdots b_m^{n_m+r_m}.$$

Then, 1 is the unit element of this multiplication, and the inverse is given by $(b_1^{n_1} \cdots b_m^{n_m})^{-1} = b_1^{-n_1} \cdots b_m^{-n_m}$. We often omit B in $\mathbf{GDeg}_B$ if it is clear from the context.

The ordinary degree can be identified with the generalized degree over a singleton set.

In the analogy of quantity dimensions, the set B corresponds to the base quantity dimensions (e.g., L for lengths and T for times); the set $\mathbf{GDeg}_B$ corresponds to the derived quantity dimensions (e.g., LT^{-1} for velocities and LT^{-2} for acceleration rates.); multiplication expresses the relationship among quantity dimensions (e.g., $LT^{-1} \cdot T = L$ for velocity $\times$ time = distance.)

Definition 3. A *g-degree assignment* is a finite map from a set of variables $\mathbf{Var}$ to $\mathbf{GDeg}$. For a g-degree assignment Γ and a monomial $p = a x_1^{d_1} \cdots x_n^{d_n}$, we write $\mathbf{gdeg}_\Gamma(p)$ for $\Gamma(x_1)^{d_1} \cdots \Gamma(x_n)^{d_n}$ and call it the *g-degree of p under Γ* (or simply *g-degree* of p if Γ is clear from the context).

For example, if $\Gamma = \{\, t \mapsto T, v \mapsto LT^{-1} \,\}$, then $\mathbf{gdeg}_\Gamma(2vt) = L$. In terms of the analogy with quantity dimensions, this means that the expression $2vt$ represents a length.

Definition 4. We say p is a *generalized homogeneous (GH) polynomial of g-degree τ under Γ* if p is the sum of monomials of g-degree τ under Γ. For a polynomial $p \neq 0$, we write $\mathbf{gdeg}_\Gamma(p)$ for its g-degree if p is a GH polynomial under Γ; otherwise, $\mathbf{gdeg}_\Gamma(p)$ is undefined.

Remark 1. By definition, 0 is a GH polynomial of g-degree τ for an arbitrary τ. Therefore we leave $\mathbf{gdeg}_\Gamma(0)$ undefined.

Example 5. The ordinary degree is obtained by letting $B = \{\, b \,\}$ and $\Gamma(x_i) = b$ for every variable x_i. We have $\mathbf{gdeg}_\Gamma(p) = b^d$ for any homogeneous polynomial of degree d.

Example 6. Consider polynomials

$$p_1 := -gt + gt_0 - v + v_0,$$

$$p_2 := -gt^2 + gt_0^2 - 2tv + 2t_0v_0 + 2x - 2x_0,$$

which have appeared in Section 1 as postconditions of c_{fall} given in Figure 1. They are both GH-polynomials under

$$\Gamma := \left\{ g \mapsto LT^{-2}, t \mapsto T, v \mapsto LT^{-1}, x \mapsto L, x_0 \mapsto L, v_0 \mapsto LT^{-1}, t_0 \mapsto T \right\}.$$

It is straightforward to check that p_1 and p_2 are GH polynomials of degree L and LT^{-1}, respectively, under Γ. For example, $\mathbf{gdeg}_\Gamma(-gt^2) = \Gamma(g)\Gamma(t)^2 = (LT^{-2})T^2 = L$, and $\mathbf{gdeg}_\Gamma(-2tv) = \Gamma(t)\Gamma(v) = T(LT^{-1}) = L$.

It is easy to see that any $p \in \mathbb{R}[x_1, \ldots, x_n]$ can be uniquely written as the finite sum of GH polynomials as $p_{\Gamma,\tau_1} + \cdots + p_{\Gamma,\tau_m}$, where $\tau_1, \ldots, \tau_m$ are pairwise distinct; p_{Γ,τ_i} is the sum of all monomials of g-degree τ_i occurring in p. We call $p_{\Gamma,\tau}$ the *homogeneous component of p with g-degree τ under Γ*, or simply *a homogeneous component of p*. We omit Γ if it is clear from the context.

Example 7. Let $\Gamma = \{ t \mapsto T, v \mapsto LT^{-1}, x \mapsto L \}$ and $p = x + tv - v$. Then p is written as $p = p_L + p_{LT^{-1}}$, where $p_L = x + tv$ and $p_{LT^{-1}} = -v$.

Remark 2. The term "generalized homogeneity" appears in various areas; according to Hankey et al. [10], a function $f(x_1, \ldots, x_n)$ is said to be generalized homogeneous if there are $a_1, \ldots, a_n$ and a_f such that, for any positive λ, $f(\lambda^{a_1}x_1, \ldots, \lambda^{a_n}x_n) = \lambda^{a_f} f(x_1, \ldots, x_n)$. Our definition of GH polynomials appears to generalize theirs: if f is generalized homogeneous in their sense, then we can use a g-degree assignment $x_1 \mapsto b^{a_1}, \ldots, x_n \mapsto b^{a_n}$ so that f is homogeneous of degree b^{a_f}. Barenblatt [6] points out that the essence of the quantity dimension principle is generalized homogeneity.

3. A Type System for Generalized Homogeneity

This section introduces the target language and devise a type system that guarantees generalized homogeneity of programs. This amounts to adapting *dimension type system* proposed by Kennedy [7, 8] to our language.

3.1. Language and semantics

Syntax. The language we consider is a simple imperative language, in which all expressions are polynomials. Its concrete syntax is as follows:

$$c ::= \mathbf{skip} \mid x{:=}p \mid c_1; c_2 \mid \mathbf{if}\, p = 0\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \mid \mathbf{while}\, p \bowtie 0\, \mathbf{do}\, c.$$

Here, $\bowtie$ is either $=$ or $\neq$, and p ranges over polynomials over program variables. We restrict the guard to a single-polynomial algebraic condition (i.e., $p = 0$) or its negation.

$$\frac{}{\mathbf{skip}, \sigma \to \sigma} \qquad \frac{}{x{:=}p, \sigma \to \sigma[x \mapsto \sigma(p)]} \qquad \frac{c_1, \sigma \to \sigma' \qquad c_2, \sigma' \to \sigma''}{c_1; c_2, \sigma \to \sigma''}$$

$$\frac{\sigma(p) \bowtie 0 \qquad c_1, \sigma \to \sigma'}{\mathbf{if}\ p \bowtie 0\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \to \sigma'} \qquad \frac{\sigma(p) \not\bowtie 0 \qquad c_2, \sigma \to \sigma'}{\mathbf{if}\ p \bowtie 0\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \to \sigma'}$$

$$\frac{\sigma(p) \not\bowtie 0}{\mathbf{while}\ p \bowtie 0\ \mathbf{do}\ c, \sigma \to \sigma} \qquad \frac{\sigma(p) \bowtie 0 \qquad c, \sigma \to \sigma' \qquad \mathbf{while}\ p \bowtie 0\ \mathbf{do}\ c, \sigma' \to \sigma''}{\mathbf{while}\ p \bowtie 0\ \mathbf{do}\ c, \sigma \to \sigma''}$$

Figure 2: Standard Big-Step Operational Semantics.

Operational Semantics. A *state*, ranged over by σ, is a map from the set of variables to $\mathbb{R}$. We write $c, \sigma \to \sigma'$ if the execution of a program c starting from a state σ terminates with the final state σ'. This execution relation is formally defined by the standard set of rules summarized in Figure 2 (detailed discussion can be found in standard textbooks, such as [11]). It is easy to see that this semantics is deterministic, that is, given c and σ, there exists at most one σ' such that $c, \sigma \to \sigma'$.

By using the execution relation, we can formally define a postcondition as follows.

Definition 8. A *postcondition* of a program c is a set of states S such that, for any state σ, if there exists a (necessarily unique) state σ' such that $c, \sigma \to \sigma'$, then $\sigma' \in S$. In particular, if p is a polynomial and S is the set of states under which $p = 0$ holds, then we say that S (or "$p = 0$") is an *algebraic postcondition.*

3.2. Type system

We introduce a type system that guarantees the generalized homogeneity of programs. A type judgment has the form $\Gamma \vdash c$, where Γ is a g-degree assignment. Intuitively, this judgment means that the g-degrees assigned to program variables by Γ are consistent with the usage of variables in c. We say Γ is *consistent* with the program c if $\Gamma \vdash c$ holds.

Typing rules are listed in Figure 3. They are obtained by adapting the dimension type system [7, 8] to our language. Checking the consistency amounts to checking that all the expressions (which are polynomials) appearing in c is generalized homogeneous under Γ, and both sides of each assignment have the same g-degree (as in rule T-Assign). Rules T-If and T-While require that p is generalized homogeneous, but its g-degree τ can be arbitrary (because τ does not appear in the conclusion).

3.3. Automated inference of the g-degree assignment

Kennedy also proposed a constraint-based automated type inference algorithm of his type system [7, 8]. We adapt his algorithm so that, given a command c, it infers a g-degree assignment Γ such that $\Gamma \vdash c$. The algorithm is in three steps: (1) designating a template of the g-degree assignment, (2) generating constraints over g-degrees, and (3) solving the constraints. In order to make the current article self-contained, we explain each step below.

$$\Gamma \vdash \mathbf{skip} \quad \text{(T-Skip)} \qquad \frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \quad \text{(T-Seq)}$$

$$\frac{\Gamma(x) = \mathbf{gdeg}_\Gamma(p)}{\Gamma \vdash x{:=}p} \quad \text{(T-Assign)}$$

$$\frac{\mathbf{gdeg}_\Gamma(p) = \tau \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \mathbf{if}\, p = 0\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2} \quad \text{(T-If)}$$

$$\frac{\mathbf{gdeg}_\Gamma(p) = \tau \qquad \Gamma \vdash c}{\Gamma \vdash \mathbf{while}\ p \bowtie 0\ \mathbf{do}\ c} \quad \text{(T-While)}$$

Figure 3: Typing rules

Step 1: Designating a template of the g-degree assignment. Let $x_1, \ldots, x_n$ be the list of the variables occurring in the given program c. Then, the algorithm first designates a template g-degree assignment $\Gamma_c := \{\, x_1 \mapsto \alpha_{x_1}, \ldots, x_n \mapsto \alpha_{x_n} \,\}$ where $\alpha_{x_1}, \ldots, \alpha_{x_n}$ are fresh type variables. For example, given the program c_{fall} in Figure 1, the algorithm designates

$$\Gamma_{c_{fall}} := \left\{ \begin{array}{l} g \mapsto \alpha_g, t \mapsto \alpha_t, dt \mapsto \alpha_{dt}, v \mapsto \alpha_v, x \mapsto \alpha_x, \\ x_0 \mapsto \alpha_{x_0}, v_0 \mapsto \alpha_{v_0}, t_0 \mapsto \alpha_{t_0}, a \mapsto \alpha_a \end{array} \right\}$$

where $\alpha_g, \alpha_t, \ldots$ are distinct type variables for the g-degrees of the variables $g, t, \ldots$ that are to be inferred.

Step 2: Generating constraints over g-degrees. The algorithm then generates the constraints over the g-degrees. We first define the set of constraints. Let B be a set (of type variables) that contains all type variables introduced in the previous step.

A *g-degree constraint* is an equation $\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in \mathbf{GDeg}_B$. We use a metavariable θ for maps from B to $\mathbf{GDeg}_B$. This map can be regarded as a substitution of types (g-degrees) into type variables, and therefore naturally extends to a map from $\mathbf{GDeg}_B$ to itself. We also define, for a given g-degree assignment Γ, a g-degree assignment $\theta(\Gamma)$ by $(\theta(\Gamma))(x) = \theta(\Gamma(x))$. We say that θ is a *solution* of a set of g-degree constraints C if it satisfies all the constraints in C. For example, the map $\theta := \{\, \alpha_v \mapsto LT^{-1}, \alpha_x \mapsto L, \alpha_t \mapsto T \,\}$ is a solution of the constraint set $\{\, \alpha_v = \alpha_x \alpha_t^{-1} \,\}$, since both $\theta(\alpha_v)$ and $\theta(\alpha_x \alpha_t^{-1})$ equal LT^{-1}.

For a polynomial $p = m_1 + \cdots + m_n$, where $m_1, \ldots, m_n$ are nonzero monomials and $n \geq 1$, we define[3]

$$\mathbf{gdeg}'_\Gamma(p) := \mathbf{gdeg}_\Gamma(m_1),$$

[3]Strictly speaking, this definition is not well-defined, because it depends on the choice of the representation of p. For example, if $p = x + y$, then the right-hand side of the definition becomes $(\alpha_x, \{\, \alpha_x = \alpha_y \,\})$, but we could write the same polynomial as $y + x$, and if we chose this representation, then we would get $(\alpha_y, \{\, \alpha_y = \alpha_x \,\})$. This ambiguity does not matter, because all of $\mathbf{gdeg}_\Gamma(m_i)$ are equated by any solution of the constraints.

$$\mathbf{constr}_\Gamma(p) := \{\, \mathbf{gdeg}_\Gamma(m_i) = \mathbf{gdeg}_\Gamma(m_{i+1}) \mid 1 \le i < n \,\}.$$

For any solution θ of $\mathbf{constr}_\Gamma(p)$, the polynomial p is generalized homogeneous under $\theta(\Gamma)$, and its g-degree is given by $\mathbf{gdeg}_{\theta(\Gamma)}(p) = \theta(\mathbf{gdeg}'_\Gamma(p))$. We also define $\mathbf{gdeg}'_\Gamma(0)$ to be a fresh type variable, and $\mathbf{constr}_\Gamma(0) = \emptyset$. This definition reflects the fact that 0 can be regarded as a GH polynomial of an arbitrary g-degree.

For example, let $\Gamma := \{\, v \mapsto \alpha_v, g \mapsto \alpha_g, x \mapsto \alpha_x \,\}$ and $p := 2v^2 + gx$; then, $\mathbf{gdeg}'_\Gamma(p) = \alpha_v^2$ and $\mathbf{constr}_\Gamma(p) = \{\, \alpha_v^2 = \alpha_g\alpha_x \,\}$. This constraint set has a solution $\theta := \{\, \alpha_v \mapsto LT^{-1}, \alpha_g \mapsto LT^{-2}, \alpha_x \mapsto L \,\}$, and for this θ we have $\theta(\Gamma) = \{\, v \mapsto LT^{-1}, g \mapsto LT^{-2}, x \mapsto L \,\}$. The polynomial p is indeed generalized homogeneous under $\theta(\Gamma)$ since $\theta(\Gamma)(v^2) = \theta(\Gamma)(gx) = L^2T^{-2}$.

The function PT for the constraint generation is defined as follows:

$$\begin{aligned}
PT(\Gamma, \mathbf{skip}) &:= \emptyset \\
PT(\Gamma, c_1; c_2) &:= PT(\Gamma, c_1) \cup PT(\Gamma, c_2) \\
PT(\Gamma, x{:=}p) &:= \{\, \Gamma(x) = \mathbf{gdeg}'_\Gamma(p) \,\} \cup \mathbf{constr}_\Gamma(p) \\
PT(\Gamma, \mathbf{if}\, p = 0\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2) &:= \mathbf{constr}_\Gamma(p) \cup PT(\Gamma, c_1) \cup PT(\Gamma, c_2) \\
PT(\Gamma, \mathbf{while}\ p \bowtie 0\ \mathbf{do}\ c) &:= \mathbf{constr}_\Gamma(p) \cup PT(\Gamma, c).
\end{aligned}$$

The constraint set $PT(\Gamma, c)$ is defined so that any of its solutions θ satisfies $\theta(\Gamma) \vdash c$. The definition essentially constructs the derivation tree of $\Gamma \vdash c$ following the rules in Figure 3 and collects the constraints appearing in the tree.

Example 9. $PT(\Gamma_{c_{fall}}, c_{fall})$ generates the constraints

$$\begin{aligned}
&\{\, \alpha_x = \alpha_{x_0}, \alpha_v = \alpha_{v_0}, \alpha_t = \alpha_{t_0} \,\}, \quad \{\, \alpha_t = \alpha_a \,\}, \\
&\{\, \alpha_x = \alpha_v\alpha_{dt}, \alpha_v = \alpha_g\alpha_{dt}, \alpha_t = \alpha_{dt} \,\}
\end{aligned}$$

from Lines 1, 2, and 3, respectively. They ensure the generalized homogeneity of polynomials appearing in the program. From Line 3, additional constraints

$$\{\, \alpha_x = \alpha_x, \alpha_v = \alpha_v, \alpha_t = \alpha_t \,\},$$

are generated to ensure that the g-degrees of both sides of the assignment are identical. In this case, these constraints happen to be trivial, but if we wrote $x{:=}vdt + x$ instead of $x{:=}x + vdt$, then $\alpha_x = \alpha_v\alpha_{dt}$ would have been generated instead of $\alpha_x = \alpha_x$.

Step 3: Solving the constraints. The algorithm then calculates a solution of the generated constraints. The constraint-solving procedure is almost the same as that by Kennedy [7, Section 5.2], which is based on Lankford's unification algorithm [12]. The procedure obtains a solution θ from the given constraint set C by applying the following rewriting rules successively:

$$(\emptyset, \theta) \to \theta$$

$$(\{\, \alpha'^k \vec{\alpha}^{\vec{n}} = 1 \,\} \cup C, \theta) \to (\{\, \alpha' \mapsto \vec{\alpha}^{-\vec{n}/k} \,\}(C), \{\, \alpha' \mapsto \vec{\alpha}^{-\vec{n}/k} \,\} \circ \theta)$$
where k is the exponent with the least absolute value, and
k divides all the integers in $\vec{n}$

$$(\{\, \alpha'^k \vec{\alpha}^{\vec{n}} = 1 \,\} \cup C, \theta) \to (\{\, \omega^k \vec{\alpha}^{\vec{n} \bmod k} = 1 \,\} \cup \theta'(C), \theta' \circ \theta)$$
where k is the exponent with the least absolute value,
there is an integer in $\vec{n}$ that is not divisible by k,
$\theta' = \{\, \alpha' \mapsto \omega \vec{\alpha}^{-\lfloor \vec{n}/k \rfloor} \,\}$, and ω is a fresh type variable

$$(\{\, 1 = 1 \,\} \cup C, \theta) \to (C, \theta)$$
$$(\{\, \tau_1 = \tau_2 \,\} \cup C, \theta) \to (\{\, \tau_1 \tau_2^{-1} = 1 \,\} \cup C, \theta)$$
$$C \to (C, \emptyset).$$

The idea of the procedure is to construct a solution by iteratively converting a constraint $\alpha'^k \vec{\alpha}^{\vec{n}} = 1$ to a simpler one. If k divides all the integers in $\vec{n}$ (i.e., the second case), the procedure simply takes k-th root of the equation to obtain $\{\, \alpha' \mapsto \vec{\alpha}^{-\vec{n}/k} \,\}$. Otherwise (i.e., the third case),[4] the procedure (1) splits $\vec{n}/k$ to the quotient $\lfloor \vec{n}/k \rfloor$ and the remainder $\vec{n} \bmod k$, (2) introduces a fresh type variable ω representing $\vec{\alpha}^{-(\vec{n} \bmod k)/k}$, and (3) sets α' in the solution to $\omega \vec{\alpha}^{-\lfloor \vec{n}/k \rfloor}$ which represents $\vec{\alpha}^{-\vec{n}/k}$.

The minimality of $|k|$ in the second and third rules guarantees the termination of the procedure; by choosing such k, the least absolute value of the nonzero exponents decreases [7, Section 5.2].

After obtaining a solution with the procedure above, the inference algorithm assigns different base degree to each surviving g-degree variable.

Example 10. Consider the following constraint set:

$$\left\{ \begin{array}{l} \alpha_x \alpha_{x_0}^{-1} = 1, \alpha_v \alpha_{v_0}^{-1} = 1, \alpha_t \alpha_{t_0}^{-1} = 1, \alpha_t \alpha_{dt}^{-1} = 1, \\ \alpha_x \alpha_v^{-1} \alpha_{dt}^{-1} = 1, \alpha_v \alpha_g^{-1} \alpha_{dt}^{-1} = 1 \end{array} \right\}$$

which is equivalent to that of Example 9. After applying the second rule to constraints $\alpha_x \alpha_{x_0}^{-1} = 1$, $\alpha_v \alpha_{v_0}^{-1} = 1$, $\alpha_t \alpha_{t_0}^{-1} = 1$, and $\alpha_t \alpha_{dt}^{-1} = 1$ in this order, the procedure obtains

$$\left(\left\{ \begin{array}{l} \alpha_{x_0} \alpha_{v_0}^{-1} \alpha_{dt}^{-1} = 1, \\ \alpha_{v_0} \alpha_g^{-1} \alpha_{dt}^{-1} = 1 \end{array} \right\}, \left\{ \begin{array}{l} \alpha_x \mapsto \alpha_{x_0}, \alpha_v \mapsto \alpha_{v_0}, \\ \alpha_t \mapsto \alpha_{dt}, \alpha_{t_0} \mapsto \alpha_{dt} \end{array} \right\} \right)$$

At the next step, suppose that the procedure picks up the constraint $\alpha_{x_0} \alpha_{v_0}^{-1} \alpha_{dt}^{-1} = 1$. By applying the second rule, the procedure generates

$$\left(\left\{\, \alpha_{v_0} \alpha_g^{-1} \alpha_{dt}^{-1} = 1 \,\right\}, \left\{ \begin{array}{l} \alpha_x \mapsto \alpha_{v_0} \alpha_{dt}, \alpha_v \mapsto \alpha_g \alpha_{dt}, \alpha_t \mapsto \alpha_{dt}, \\ \alpha_{t_0} \mapsto \alpha_{dt}, \alpha_{x_0} \mapsto \alpha_{v_0} \alpha_{dt} \end{array} \right\} \right)$$

[4] We do not use this case in the rest of this article.

Then, with the second and first rules, the procedure obtains the following solution:

$$\left\{ \begin{array}{l} \alpha_x \mapsto \alpha_g \alpha_{dt}^2, \alpha_v \mapsto \alpha_g \alpha_{dt}, \alpha_t \mapsto \alpha_{dt}, \alpha_{t_0} \mapsto \alpha_{dt}, \\ \alpha_{x_0} \mapsto \alpha_g \alpha_{dt}^2, \alpha_{v_0} \mapsto \alpha_g \alpha_{dt} \end{array} \right\}.$$

By assigning the base degree A to α_g and T to α_{dt}, we obtain

$$\left\{ \alpha_x \mapsto AT^2, \alpha_v \mapsto AT, \alpha_t \mapsto T, \alpha_{t_0} \mapsto T, \alpha_{x_0} \mapsto AT^2, \alpha_{v_0} \mapsto AT \right\}.$$

Notice the set of base degrees is different from that we used in Example 6; in this example, the g-degree for the acceleration rates (A) is used as a base degree, whereas that for lengths (L) is used in Example 6. This happens because the order of the constraints chosen in an execution of the inference algorithm is nondeterministic. Our results in the rest of this article do not depend on a specific choice of base degrees.

Limitation. A limitation of the current g-degree inference algorithm is that, even if a constant symbol in a program is intended to be of a g-degree other than 1, it has to be of g-degree 1 in the current type system. For example, consider the assignment $v := v - gdt - \rho v dt$, a variant of c_{fall} which takes friction between the mass point and air into account. If we replace g with 9.81 and ρ with 0.24, then the g-degrees of v and dt are inferred to be 1 due to the assignment $v := v - 9.8dt - 0.24vdt$: The constraints for this assignment generated by the inference algorithm is $\{ \alpha_v = \alpha_{dt}, \alpha_{dt} = \alpha_v \alpha_{dt}, \alpha_v = \alpha_v \}$, whose only solution is $\{ \alpha_v \mapsto 1, \alpha_{dt} \mapsto 1 \}$. Such a g-degree assignment is not useful for the template-size reduction; any polynomial is a GH polynomial under this assignment.

As a workaround, our current implementation that will be described in Section 6 uses an extension that can assign a g-degree other than 1 to each occurrence of a constant symbol by treating a constant symbol as a variable. For example, consider the following program $\texttt{sumpower}_d$.

$x := X + 1; y := 0; s := \underline{1}$
while $x \neq 0$ **do**
  if $y = 0$ **then** $(x, y) := (x - 1, x)$ **else** $(s, y) := (s + y^d, y - 1)$
end while

The inference algorithm treats the underlined occurrence of 1 as a variable and assigns T^d to it; the other occurrences of 0 and 1 are given g-degree T. This g-degree assignment indeed produces a smaller template.

4. GHA Postconditions Generate All Algebraic Postconditions

In this section, we prove the main result: well-typed programs have enough GHA postconditions. This is formally stated as follows.

Theorem 11. *If $\Gamma \vdash c$ and $p = 0$ is a postcondition of c (i.e. if $c, \sigma \to \sigma'$ then $\sigma'(p) = 0$ for all σ and σ'), then so are $p_\tau = 0$ for all τ.*

We first prove several lemmas used in the proof of Theorem 11. Hereafter, we assume that c contains at most k variables $y_1, \ldots, y_k$, and is well-typed under

$$\Gamma = \{\, y_1 \mapsto \tau_1, \ldots, y_k \mapsto \tau_k \,\}.$$

Let $b_1, \ldots, b_m$ be the list of base degrees appearing in Γ (m is the number of base degrees).

First, we generalize the following fact about homogeneous polynomials: if $p \in \mathbb{R}[x_1, \ldots, x_n]$ is homogeneous of degree d, then $p(\lambda x) = \lambda^d p(x)$ for all $\lambda \in \mathbb{R}$. To state the lemma, we use the following notations.

Definition 12. Let $\tau = b_1^{n_1} \cdots b_m^{n_m}$ be a g-degree.

1. For $\lambda \in \mathbb{R}^m$, we define $\lambda^\tau := \lambda_1^{n_1} \cdots \lambda_m^{n_m}$. Similarly, for a sequence of variables $\vec{x} = (x_1, \ldots, x_m)$, we define $\vec{x}^\tau := x_1^{n_1} \cdots x_m^{n_m}$.
2. For a state σ, we define another state $\lambda \bullet \sigma$ by $(\lambda \bullet \sigma)(y_i) = \lambda^{\tau_i} \sigma(y_i)$.

Lemma 13. *If p is a GH polynomial with degree τ, then $(\lambda \bullet \sigma)(p) = \lambda^\tau \sigma(p)$.*

Lemma 14. *If $c, \sigma \to \sigma'$ and $\lambda \in (\mathbb{R} \setminus \{\, 0 \,\})^m$, then $c, \lambda \bullet \sigma \to \lambda \bullet \sigma'$.*

Proof. By induction on the derivation of $c, \sigma \to \sigma'$, using the fact that c contains only GH polynomials. The assumption that every component of λ is nonzero is needed to ensure that $\sigma(p) \bowtie 0$ and $(\lambda \bullet \sigma)(p) \bowtie 0$ are equivalent. □

Lemma 15. *Let q be a polynomial in m variables. Suppose that $q(\lambda) = 0$ for all $\lambda \in (\mathbb{R} \setminus \{\, 0 \,\})^m$. Then, we have $q = 0$ as a polynomial.*

Proof. We prove this by induction on m. If $m = 0$, the statement is trivial. Otherwise, let us write

$$q(x_1, \ldots, x_m) = \sum_n q_n(x_2, \ldots, x_m) x_1^n$$

where $q_n(x_2, \ldots, x_m)$ are polynomials in $m - 1$ variables. Fix any $\lambda_2, \ldots, \lambda_m \in \mathbb{R} \setminus \{\, 0 \,\}$. Then $q(x_1, \lambda_2, \ldots, \lambda_m) = \sum_n q_n(\lambda_2, \ldots, \lambda_m) x_1^n$ is a polynomial in one variable, and by assumption any element of $\mathbb{R} \setminus \{\, 0 \,\}$ is its root. However, a nonzero polynomial in one variable can have only finitely many roots, and thus $q(x_1, \lambda_2, \ldots, \lambda_m) = 0$ as polynomials. This means that all the coefficients $q_n(\lambda_2, \ldots, \lambda_m)$ of x_1^n are zero for any $\lambda_2, \ldots, \lambda_m \in \mathbb{R} \setminus \{\, 0 \,\}$, and then by the induction hypothesis we conclude that $q_n = 0$ for all n, that is, $q = 0$. □

Proof of Theorem 11. Suppose $p = 0$ is a postcondition of c. Let σ and σ' be states such that $c, \sigma \to \sigma'$. We shall prove that $\sigma'(p_\tau) = 0$ for all τ. Let $a_\tau = \sigma'(p_\tau)$, and consider a polynomial $q = \sum_\tau a_\tau \vec{x}^\tau$, where $\vec{x}$ consists of m variables. Then, by Lemma 15, it is sufficient to show that $q(\lambda) = 0$ for all $\lambda \in (\mathbb{R} \setminus \{\, 0 \,\})^m$. By Lemma 14, we have $c, \lambda \bullet \sigma \to \lambda \bullet \sigma'$. Then, by the assumption that $p = 0$ is a postcondition of c, we obtain $(\lambda \bullet \sigma')(p) = 0$. Now $q(\lambda) = 0$ follows from the sequence of equalities

$$(\lambda \bullet \sigma')(p) = \sum_\tau (\lambda \bullet \sigma')(p_\tau) = \sum_\tau \lambda^\tau \sigma'(p_\tau) = \sum_\tau \lambda^\tau a_\tau = q(\lambda).$$

We use Lemma 13 in the second equality. □

5. Template-based algorithm

This section applies our idea to template-based algorithm by Cachera et al. [4].

5.1. Constraint generation algorithm

Cachera et al. proposed a sound template-based algorithm for the postcondition problem. Their basic idea is to express a fixed point by constraints on the parameters in a template in order to avoid fixed-point iteration. We shall first recall their constraint-generation algorithm.

Definition 16. We hereafter assume a fixed set of *parameters*, ranged over by a. A parameter represents an unknown value. A *template* is an expression of the form $a_1p_1 + \cdots + a_np_n$ where $a_1, \ldots, a_n$ are parameters and $p_1, \ldots, p_n$ are polynomials. A *GH template* (of g-degree τ under Γ) is a template $a_1p_1 + \cdots + a_mp_m$ where $p_1, \ldots, p_m$ are GH polynomials (of g-degree τ under Γ).

An *equality constraint* is an expression of the form $\langle G \equiv G' \rangle$, where G, G' are (finite) sets of templates. A *constraint set*, ranged over by C, is a set of equality constraints.

A *valuation* is a map from the set of parameters to $\mathbb{R}$. We can extend a valuation v as a map from templates to polynomials by $v(a_1p_1 + \cdots + a_mp_m) := v(a_1)p_1 + \cdots + v(a_m)p_m$. A valuation v *satisfies* an equality constraint $\langle G \equiv G' \rangle$, written $v \models \langle G \equiv G' \rangle$, if $v(G)$ and $v(G')$ are equivalent as algebraic predicates, that is, the following holds for any state σ:

$$\sigma(v(f)) = 0 \text{ for all } f \in G \iff \sigma(v(f)) = 0 \text{ for all } f \in G'.$$

A *solution* of a constraint set C is a valuation that satisfies all constraints in C. If v is a solution of C, we write $v \models C$.

The algorithm is parameterized over a remainder-like operation **Rem**. This is an operation that takes a template f and a polynomial p, and returns a template of the form $f - pg$, where g is some template. We write $\mathbf{Rem}(G, p)$, where G is a set of templates, for $\{\, \mathbf{Rem}(f, p) \mid f \in G \,\}$. A typical example is $\mathbf{Rem}^{\mathtt{par}}$ defined below, which is the remainder operation Cachera et al. used in their algorithm.

Definition 17. We define $\mathbf{Rem}^{\mathtt{par}}(f, p) = f - pg$ where g is the most general template of degree $\mathbf{deg}(f) - \mathbf{deg}(p)$,[5] and all the coefficients of g are fresh parameters.

For example, $\mathbf{Rem}^{\mathtt{par}}(x^2, x + 1) = x^2 - (a_1x + a_2)(x + 1)$, because $a_1x + a_2$ is the most general template of degree $\mathbf{deg}(x^2) - \mathbf{deg}(x + 1) = 1$.

The constraint-generation algorithm is given as a transformation on pairs of the form (G, C) where G is a set of templates, and C is a constraint set.

[5] Here, **deg** represents the ordinary degree, not a g-degree.

Intuitively, (G, C) represents a predicate expressed by the conjuction $\bigwedge_{f \in G}(f = 0)$, where parameters occurring in G are some real numbers subject to the constraints in C. The algorithm is defined by

$$\begin{aligned}
[\![\mathbf{skip}]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C) &= (G, C) \\
[\![x{:=}p]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C) &= (G[x := p], C) \\
[\![c_1; c_2]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C) &= [\![c_1]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}([\![c_2]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C)) \\
[\![\mathbf{if}\, p = 0\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C) &= (p \cdot G_2 \cup \mathbf{Rem}(G_1, p), C_1 \cup C_2) \\
[\![\mathbf{while}\ p \bowtie 0\ \mathbf{do}\ c_1]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C) &= (G, C_1 \cup \{\langle G \equiv G_1 \rangle\}),
\end{aligned}$$

where $(G_i, C_i) := [\![c_i]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G, C)$ in the last two cases.

This algorithm receives a postcondition (G, C) and computes a precondition of c; at the same time it also accumulates the generated constraints to C. A constraint $\langle G \equiv G_1 \rangle$ is added when a **while** statement is encountered. This constraint expresses the requirement that G itself becomes a loop invariant. In this way, the algorithm avoids computing the greatest fixed-point.

The algorithm above is the one proposed by Cachera et al., except that it is parameterized by **Rem**; the original algorithm is obtained by instantiating **Rem** with $\mathbf{Rem}^{\mathsf{par}}$.

Soundness of this algorithm can be formulated as in the theorem below. Intuitively, this theorem states that if $(G, C) = [\![c]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G', C')$, then the Hoare triple $\{(G, C)\}c\{(G', C')\}$ is valid, where (G, C) and (G', C') are regarded as predicates in the manner explained above.

Theorem 18. *Suppose $c, \sigma \to \sigma'$, and let $(G, C) = [\![c]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}(G', C')$. Moreover, suppose that $v \models C$, and σ satisfies $\sigma(v(f)) = 0$ for all $f \in G$. Then, $\sigma'(v(f')) = 0$ for all $f' \in G'$.*

Proof. By induction on the derivation of $c, \sigma \to \sigma'$, using $C' \subseteq C$ (which is easy to verify). If c is a loop, use the fact that "$v(f) = 0$ for all $f \in G$" is a loop invariant.

The highlight of the algorithm is the definition of $[\![\mathbf{if}\, p = 0\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2]\!]^{\sharp \mathsf{c}}_{\mathbf{Rem}}$. In order to explain this definition, we prove this case. Let us assume $v \models C_1 \cup C_2$, and $\sigma(v(f)) = 0$ for all $f \in p \cdot G_2 \cup \mathbf{Rem}(G_1, p)$. There are two cases: $\sigma(p) = 0$, and $\sigma(p) \neq 0$.

If $\sigma(p) = 0$, then c_1 is executed, and therefore $c_1, \sigma \to \sigma'$. We have $\sigma(v(f)) = 0$ for all $f \in \mathbf{Rem}(G_1, p)$. By definition of a remainder operation, for each $f' \in G_1$ there exists g such that $f' - pg \in \mathbf{Rem}(G_1, p)$. Therefore $\sigma(v(f')) - \sigma(v(pg)) = 0$, but by assumption $\sigma(v(p)) = \sigma(p) = 0$ (the first equality holds because p does not contain parameters), and hence $\sigma(v(f')) = 0$. Therefore $\sigma(v(f')) = 0$ for all $f' \in G_1$. Then, by the induction hypothesis (and $C \subseteq C_1$), we obtain $\sigma(v(f)) = 0$ for all $f \in G$.

If $\sigma(p) \neq 0$, then c_2 is executed, and therefore $c_2, \sigma \to \sigma'$. We have $\sigma(v(f)) = 0$ for all $f \in p \cdot G_2$, and thus $\sigma(v(pf')) = 0$ for all $f' \in G_2$. By the assumption we have $\sigma(v(p)) \neq 0$, and hence $\sigma(v(f')) = 0$ for all $f' \in G_2$. The rest of the proof is similar to the previous case. □

Algorithm 1 Inference of algebraic postconditions.

1: **procedure** InvInf(c, d)
2: $g \leftarrow$ the most general template of degree d
3: $(G, C) \leftarrow [\![c]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}^{\mathtt{par}}}(\{\, g \,\}, \emptyset)$
4: **return** $v(g)$ where v is a solution of $C \cup \{\, \langle G \equiv \{\, 0 \,\}\rangle \,\}$
5: **end procedure**

Remark 3. The soundness would still hold even if we omitted the multiplier p and defined

$$[\![\mathbf{if}\, p = 0\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}}(G) = (G_2 \cup \mathbf{Rem}(G_1, p), C_1 \cup C_2).$$

However, this makes the precondition less precise.

5.2. Cachera et al.'s template-based algorithm

The algorithm proposed by Cachera et al. is shown in Algorithm 1. It solves the postcondition problem using the constraint-generating subprocedure introduced above, with **Rem** being instantiated by $\mathbf{Rem}^{\mathtt{par}}$ (Definition 17). This algorithm receives a program c and a degree d, and returns a postcondition that can be expressed as an algebraic condition of degree at most d. The algorithm first generates the most general template g of degree d for the postcondition, and applies $[\![c]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}^{\mathtt{par}}}$ to g. For the returned set of polynomials G and the constraint set C, the algorithm computes a solution of $C \cup \langle G \equiv \{\, 0 \,\}\rangle$; the equality constraint $\langle G \equiv \{\, 0 \,\}\rangle$ forces the computed precondition G to be valid for all initial state.

This algorithm is proved to be sound: If $p \in \mathrm{InvInf}(c, d)$, then $p = 0$ holds at the end of c for any initial states [4]. This is a consequence of Theorem 18 and the fact that all templates in G returned by the constraint-generation algorithm (which expresses a precondition) are forced to be zero by the constraint $\langle G \equiv \{\, 0 \,\}\rangle$.

Completeness is not discussed by Cachera et al. In fact, it does not hold, because the template generation algorithm ignores loop guards. For example, consider $c = \mathbf{while}\ x \neq 0\ \mathbf{do}\ \mathbf{skip}$, and $p = x$. It is easy to check that $p = 0$ is a postcondition of c. Let us calculate constraints generated from c. Because $[\![\mathbf{skip}]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}}$ is an identity function, we have

$$[\![c]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}}(G, \emptyset) = (G, \{\, \langle G \equiv G\rangle \,\}).$$

Therefore $[\![c]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}}(G, \emptyset)$ returns a constraint set

$$\{\, \langle G \equiv G\rangle, \langle G \equiv \{\, 0 \,\}\rangle \,\},$$

where $G = \{\, g \,\}$ for a template g. Its only solution is the valuation that assigns zero to all parameters. Therefore the template-based algorithm cannot produce nontrivial postconditions of c. Notice that this argument does not depend on the specific choice of **Rem** or g, and therefore the template-based algorithm is not complete regardless of the choices of a remainder operation and an initial template.

Algorithm 2 Inference of algebraic postconditions (homogeneous version).

1: **procedure** $\textsc{InvInf}^{\mathtt{H}}(c,\, d,\, \Gamma,\, \tau)$
2: $\quad g \leftarrow$ the most general template of g-degree τ and degree d
3: $\quad (G, C) \leftarrow [\![c]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}_\Gamma^{\mathtt{parH}}}(\{\, g \,\}, \emptyset)$
4: $\quad$ **return** $v(g)$ where v is a solution of $C \cup \{\, \langle G \equiv \{\, 0 \,\}\rangle \,\}$
5: **end procedure**

Remark 4. The algorithm requires a solver for the constraints of the form $\langle G \equiv G' \rangle$. A typical sufficient (but not necessary) condition for v to be its solution is that $v(G)$ and $v(G')$ generate the same ideal: if $G = \{\, f_1, \ldots, f_m \,\}, G' = \{\, g_1, \ldots, g_n \,\}$, then there exist polynomials p_{ij}, q_{ij} such that $v(f_i) = \sum_j p_{ij} v(g_j)$ and $v(g_j) = \sum_i q_{ij} v(f_i)$. To avoid high-cost computation, Cachera et al. proposed heuristics to solve an equality constraint.

Example 19. We explain how $\textsc{InvInf}(c_{fall}, 3)$ works. The algorithm generates a degree-3 template $f(x, v, t, x_0, v_0, t_0, a, dt, g)$ over $\{\, x, v, t, x_0, v_0, t_0, a, dt, g \,\}$. The algorithm then generates the following constraints by using $[\![c_{fall}]\!]^{\sharp \mathtt{c}}_{\mathbf{Rem}^{\mathtt{par}}}$:

$$\begin{aligned}
&\langle \{\, f(x_0, v_0, t_0, x_0, v_0, t_0, a, dt, g) \,\} \equiv \{\, 0 \,\} \rangle, \\
&\langle \{\, f(x, v, t, x_0, v_0, t_0, a, dt, g) \,\} \equiv \\
&\qquad \{\, f(x + v\,dt, v - g\,dt, t + dt, x_0, v_0, t_0, a, dt, g) \,\} \rangle.
\end{aligned}$$

By solving these constraints, and by substituting the solution to f, we obtain postconditions such as $-gt + gt_0 - v + v_0$ and $-gt^2 + gt_0^2 - 2tv + 2t_0v_0 + 2x - 2x_0$, depending on the choice of solutions.

5.3. Restriction to GH templates

We can restrict Cachera et al.'s algorithm to GH templates. To this end, we need to define a remainder operation that respects generalized homogeneity.

Definition 20. The remainder operation $\mathbf{Rem}_\Gamma^{\mathtt{parH}}(f, p)$ returns $f - pg$ where g is the most general GH template of g-degree $\mathbf{gdeg}(f)\mathbf{gdeg}(p)^{-1}$, and with degree $\mathbf{deg}(f) - \mathbf{deg}(p)$, and all the coefficients of g are fresh parameters.

By instantiating **Rem** with $\mathbf{Rem}_\Gamma^{\mathtt{parH}}$ (and by restricting the initial template to GH one), we obtain Algorithm 2, which is a variant of Algorithm 1.

The algorithm $\textsc{InvInf}^{\mathtt{H}}$ takes the input τ that specifies the g-degree of the postcondition. We have not obtained a theoretical result for τ to be passed to $\textsc{InvInf}^{\mathtt{H}}$ so that it generates a good postcondition. However, during the experiments in Section 6, we found that the following strategy often works: *Pass the g-degree of the monomial of interest.* For example, if we are interested in a property related to x, then pass $\Gamma(x)$ (i.e., L) to $\textsc{InvInf}^{\mathtt{H}}$ for the postcondition $-gt^2 + gt_0^2 - 2tv + 2t_0v_0 + 2x - 2x_0 = 0$. How to help a user to find such "monomial of her interest" is left as an interesting future direction.

6. Experiment

We implemented Algorithm 2 and conducted experiments. Our implementation Fastind$_{dim}$ takes a program c, a maximum degree d of the template g in the algorithm, and a monomial w. It conducts type inference of c to generate Γ and calls $\textsc{InvInf}^{\mathtt{H}}(c, d, \Gamma, \mathbf{gdeg}_{\Gamma}(w))$. The type inference algorithm is implemented with OCaml; the other parts (e.g., a solver for equality constraints) are implemented with Mathematica.

To demonstrate the merit of our approach, we applied this implementation to the benchmark used in the experiment by Cachera et al. [4] and compared our result with that of their implementation, which is called Fastind. The entire experiment was conducted on a MacBook Air 13-inch Mid 2013 model with a 1.7 GHz Intel Core i7 (with two cores, each of which has 256 KB of L2 cache) and 8 GB of RAM (1600 MHz DDR3). The modules written in OCaml were compiled with `ocamlopt`. The version of OCaml is 4.02.1. The version of Mathematica is 10.0.1.0. We refer the reader to [13, 4, 3] for detailed descriptions of each program in the benchmark. Each program contains a nested loop with a conditional branch (e.g., `dijkstra`), a sequential composition of loops (e.g., `divbin`), and nonlinear expressions (e.g., `petter(n)`.) We generated a nonlinear postcondition in each program.

Table 1 shows the result. The column deg shows the degree of the generated polynomial, t_{sol} shows the time spent by the constraint solver (ms), $\#m$ shows the number of monomials in the generated template, t_{inf} shows the time spent by the dimension-type inference algorithm (ms), and $t_{inf} + t_{sol}$ shows the sum of t_{inf} and t_{sol}. By comparing $\#m$ for Fastind with that of Fastind$_{dim}$, we can observe the effect of the use of GH polynomials on the template sizes. Comparison of t_{sol} for Fastind with that of Fastind$_{dim}$ suggests the effect on the constraint reduction phase; comparison of t_{sol} for Fastind with $t_{inf} + t_{sol}$ for Fastind$_{dim}$ suggests the overhead incurred by g-degree inference.

Discussion. The size of the templates, measured as the number of monomials ($\#m$), was reduced in 13 out of 20 programs by using GH polynomials. The value of t_{sol} decreased for these 13 programs; it is almost the same for the other programs. $\#m$ did not decrease for the other seven programs because the extension of the type inference procedure mentioned at the end of Section 3 introduced useless auxiliary variables. We expect that such variables can be eliminated by using a more elaborate program analysis.

By comparing t_{sol} for Fastind and $t_{inf} + t_{sol}$ for Fastind$_{dim}$, we can observe that the inference of the g-degree assignment sometimes incurs an overhead for the entire execution time if the template generated by Fastind is sufficiently small; therefore, Fastind is already efficient. However, this overhead is compensated in the programs for which Fastind requires more computation time.

To summarize, our current approach is especially effective for a program for which (1) the existing algorithm is less efficient owing to the large size of the template and (2) a nontrivial g-degree assignment can be inferred. We expect

Name		Fastind		Fastind$_{dim}$			
	deg	t_{sol}	$\#m$	t_{inf}	t_{sol}	$t_{inf}+t_{sol}$	$\#m$
dijkstra	2	9.29	21	0.456	8.83	9.29	21
divbin	2	0.674	21	0.388	0.362	0.750	8
freire1	2	0.267	10	0.252	0.258	0.510	10
freire2	3	2.51	35	0.463	2.60	3.06	35
cohencu	3	1.74	35	0.434	0.668	1.10	20
fermat	2	0.669	21	0.583	0.669	1.25	21
wensley	2	104	21	0.436	28.5	28.9	9
euclidex	2	1.85	45	1.55	1.39	2.94	36
lcm	2	0.811	28	0.513	0.538	1.05	21
prod4	3	31.6	84	0.149	2.78	2.93	35
knuth	3	137	220	4.59	136	141	220
mannadiv	2	0.749	21	0.515	0.700	1.22	18
petter1	2	0.132	6	0.200	0.132	0.332	6
petter2	3	0.520	20	0.226	0.278	0.504	6
petter3	4	1.56	35	0.226	0.279	0.505	7
petter4	5	7.15	56	0.240	0.441	0.681	8
petter5	6	17.2	84	0.228	0.326	0.554	9
petter10	11	485	364	0.225	0.354	0.579	14
sumpower1	3	2.20	35	0.489	2.31	2.80	35
sumpower5	7	670	330	0.469	89.1	89.6	140

Table 1: Experimental result.

that our approach will be effective for a wider range of programs if we find a more competent g-degree inference algorithm.

7. Related work

The template-based algebraic invariant synthesis proposed to date [1, 4] has focused on reducing the problem to constraint solving and solving the generated constraints efficiently; strategies for generating a template have not been the main issue. A popular strategy for template synthesis is to iteratively increase the degree of a template. This strategy suffers from an increase in the size of a template in the iterations when the degree is high.

Our claim is that a prior analysis of a program can effectively reduce the size of a template; we used the dimension type system for this purpose in this article inspired by the principle of quantity dimensions in the area of physics. Of course, there is a tradeoff between the cost of the analysis and its effect on the template-size reduction; our experiments suggest that the cost of dimension type inference is reasonable.

Semialgebraic invariants (i.e., invariants written using *inequalities* on polynomials) are often useful for program verification. The template-based approach

is also popular in semialgebraic invariant synthesis. One popular strategy in template-based semialgebraic invariant synthesis is to reduce this problem to one of semidefinite programming, for which many efficient solvers are widely available.

As of this writing, it is an open problem whether our idea regarding GH polynomials also applies to semialgebraic invariant synthesis; for physically meaningful programs, at least, we guess that it is reasonable to use GH polynomials because of the success of the quantity dimension principle in the area of physics. A possible approach to this problem would be to investigate the relationship between GH polynomials and Stengle's Postivstellensatz [14], which is the theoretical foundation of the semidefinite-programming approach mentioned above. There is a homogeneous version of the Stengle's Positivstellensatz [15, Theorem II.2]; because the notion of homogeneity considered there is equivalent to generalized homogeneity introduced in this article, we conjecture that this theorem provides a theoretical foundation of an approach to semialgebraic invariant synthesis using GH polynomials.

Although the application of the quantity dimension principle to program verification is novel, this principle has been a handy tool for discovering hidden knowledge about a physical system. A well-known example in the field of hydrodynamics is the motion of a fluid in a pipe [6]. One fundamental result in this regard is that of Buckingham [16], who stated that *any physically meaningful relationship among n quantities can be rewritten as one among $n - r$ independent dimensionless quantities, where r is the number of the quantities of the base dimension.* Investigating the implications of this theorem in the context of our work is an important direction for future work.

The structure of $\mathbb{R}[x_1, \ldots, x_n]$ resulting from the notion of the generalized degrees is an instance of *graded rings* from ring theory. Concretely, R is said to be *graded* over an Abelian group $\mathbb{G}$ if R is decomposed into the direct sum of a family of additive subgroups $\{ R_g \mid g \in \mathbb{G} \}$ and these subgroups satisfy $R_g \cdot R_h \subseteq R_{gh}$ for all $g, h \in \mathbb{G}$. Then, an element $x \in R$ is said to be *homogeneous of degree g* if $x \in R_g$. We leave an investigation of how our method can be viewed in this abstract setting as future work.

8. Conclusion

We presented a technique to reduce the size of a template used in template-based invariant-synthesis algorithms. Our technique is based on the finding that, if an algebraic postcondition of a program c exists, then it is the sum of GHA postcondition of c; hence, we can reduce the size of a template by synthesizing only GH polynomials. We presented the theoretical development as a modification of the framework proposed by Cachera et al. and empirically confirmed the effect of our approach using the benchmark used by Cachera et al. Although we used the framework of Cachera et al. as a baseline, we believe that we can apply our idea to the other template-based methods [17, 18, 1, 3, 2, 4, 19].

Our motivation behind the current work is safety verification of hybrid systems, in which the template method is a popular strategy. For example, Gulwani

et al. [20] proposed a method of reducing the safety condition of a hybrid system to constraints on the parameters of a template by using Lie derivatives. We expect our idea to be useful for expediting these verification procedures.

In this regard, Suenaga et al. [21, 22, 23] have proposed a framework called *nonstandard static analysis*, in which one models the continuous behavior of a system as an imperative or a stream-processing program using an *infinitesimal* value. An advantage of modeling in this framework is that we can apply program verification tools without an extension for dealing with continuous dynamics. However, their approach requires highly nonlinear invariants for verification. This makes it difficult to apply existing tools, which do not handle nonlinear expressions well. We expect that the current technique will address this difficulty with their framework.

We are also interested in applying our idea to decision procedures and satisfiability modulo theories (SMT) solvers. Support of nonlinear predicates is an emerging trend in many SMT solvers (e.g., Z3 [24]). Dai et al. [25] proposed an algorithm for generating a semialgebraic Craig interpolant using semidefinite programming [25]. Application of our approach to these method is an interesting direction for future work.

Acknowledgment

We appreciate anonymous reviewers of SAS 2016, Toshimitsu Ushio, Naoki Kobayashi and Atsushi Igarashi for their comments. This work is partially supported by JST PRESTO, JST CREST, KAKENHI 70633692, and in collaboration with the Toyota Motor Corporation.

[1] S. Sankaranarayanan, H. Sipma, Z. Manna, Non-linear loop invariant generation using Gröbner bases, in: POPL 2004, 2004, pp. 318–329. doi:10.1145/964001.964028.

[2] M. Müller-Olm, H. Seidl, Computing polynomial program invariants, Inf. Process. Lett. 91 (5) (2004) 233–244. doi:10.1016/j.ipl.2004.05.004.

[3] E. Rodríguez-Carbonell, D. Kapur, Generating all polynomial invariants in simple loops, J. Symb. Comput. 42 (4) (2007) 443–476. doi:10.1016/j.jsc.2007.01.002.

[4] D. Cachera, T. P. Jensen, A. Jobin, F. Kirchner, Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases, Sci. Comput. Program. 93 (2014) 89–109. doi:10.1016/j.scico.2014.02.028.

[5] D. A. Cox, J. Little, D. O'Shea, Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[6] G. I. Barenblatt, Scaling, self-similarity, and intermediate asymptotics: dimensional analysis and intermediate asymptotics, Vol. 14, Cambridge University Press, 1996.

[7] A. Kennedy, Dimension types, in: ESOP'94, 1994, pp. 348–362.

[8] A. Kennedy, Programming languages and dimensions, Ph.D. thesis, St. Catharine's College (Mar. 1996).

[9] K. Kojima, M. Kinoshita, K. Suenaga, Generalized homogeneous polynomials for efficient template-based nonlinear invariant synthesis, in: X. Rival (Ed.), Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, Vol. 9837 of Lecture Notes in Computer Science, Springer, 2016, pp. 278–299. doi:10.1007/978-3-662-53413-7_14.
URL http://dx.doi.org/10.1007/978-3-662-53413-7_14

[10] A. Hankey, H. E. Stanley, Systematic application of generalized homogeneous functions to static scaling, dynamic scaling, and universality, Physical Review B 6 (9) (1972) 3515.

[11] G. Winskel, The Formal Semantics of Programming Languages, The MIT Press, 1993.

[12] D. Lankford, G. Butler, B. Brady, Abelian group unification algorithms for elementary terms, Contemporary Mathematics 29 (1984) 193–199.

[13] E. Rodríguez-Carbonell, Some programs that need polynomial invariants in order to be verified, http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html (Accessed on January 25th, 2016).

[14] G. Stengle, A nullstellensatz and a positivstellensatz in semialgebraic geometry, Mathematische Annalen 207 (2) (1974) 87–97.

[15] L. Gonzalez-Vega, H. Lombardi, Smooth parametrizations for several cases of the Positivstellensatz, Mathematische Zeitschrift 225 (3) (1997) 427–451. doi:10.1007/PL00004620.
URL http://dx.doi.org/10.1007/PL00004620

[16] E. Buckingham, On physically similar systems; illustrations of the use of dimensional equations, Phys. Rev. 4 (1914) 345–376. doi:10.1103/PhysRev.4.345.

[17] P. Garg, C. Löding, P. Madhusudan, D. Neider, ICE: A robust framework for learning invariants, in: A. Biere, R. Bloem (Eds.), CAV 2014, Vol. 8559 of LNCS, Springer, 2014, pp. 69–87. doi:10.1007/978-3-319-08867-9.

[18] F. Somenzi, A. R. Bradley, IC3: where monolithic and incremental meet, in: FMCAD 2011, 2011, pp. 3–8.

[19] A. Adjé, P. Garoche, V. Magron, Property-based polynomial invariant generation using sums-of-squares optimization, in: S. Blazy, T. Jensen (Eds.), SAS 2015, Vol. 9291 of LNCS, Springer, 2015, pp. 235–251. doi:10.1007/978-3-662-48288-9_14.

[20] S. Gulwani, A. Tiwari, Constraint-based approach for analysis of hybrid systems, in: A. Gupta, S. Malik (Eds.), CAV 2008, Vol. 5123 of LNCS, Springer, 2008, pp. 190–203. doi:10.1007/978-3-540-70545-1_18.

[21] K. Suenaga, H. Sekine, I. Hasuo, Hyperstream processing systems: nonstandard modeling of continuous-time signals, in: R. Giacobazzi, R. Cousot (Eds.), POPL 2013, ACM, 2013, pp. 417–430. doi:10.1145/2429069.2429120.

[22] I. Hasuo, K. Suenaga, Exercises in nonstandard static analysis of hybrid systems, in: P. Madhusudan, S. A. Seshia (Eds.), CAV 2012, Vol. 7358 of LNCS, Springer, 2012, pp. 462–478. doi:10.1007/978-3-642-31424-7.

[23] K. Suenaga, I. Hasuo, Programming with infinitesimals: A while-language for hybrid system modeling, in: L. Aceto, M. Henzinger, J. Sgall (Eds.), ICALP 2011, Vol. 6756 of LNCS, Springer, 2011, pp. 392–403. doi:10.1007/978-3-642-22012-8.

[24] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: TACAS 2008, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[25] L. Dai, B. Xia, N. Zhan, Generating non-linear interpolants by semidefinite programming, in: N. Sharygina, H. Veith (Eds.), CAV 2013, Vol. 8044 of LNCS, Springer, 2013, pp. 364–380. doi:10.1007/978-3-642-39799-8_25.