Journal of Logical and Algebraic Methods in Programming 85 (2016) 847–859

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Product programs and relational program logics [☆]

Gilles Barthe [a], Juan Manuel Crespo [b], César Kunz [b]

[a] IMDEA Software Institute, Spain
[b] FireEye, Germany

ARTICLE INFO

Article history:
Received 15 June 2015
Received in revised form 19 April 2016
Accepted 24 May 2016
Available online 16 June 2016

ABSTRACT

A common theme in program verification is to relate two programs, for instance to show that they are equivalent, or that one refines the other. Such relationships can be formally established using relational program logics, which are tailored to reason about relations between two programs, or product constructions which allow to build from two programs a product program that emulates the behavior of both input programs. Similarly, product programs and relational program logics can be used to reason about 2-safety properties, an important class of properties that reason about two executions of the same program, and includes as instances non-interference, continuity, and determinism. In this paper, we consider several notions of product programs and explore their relationship with different relational program logics. Moreover, we present applications of product programs to program robustness, non-interference, translation validation, and differential privacy.

1. Introduction

Trace properties have been a prominent focus of program verification. However, many interesting properties of programs cannot be expressed naturally as trace properties. 2-safety properties [38] are a generalization of safety properties that predicates over two traces of execution of a program. Further generalized 2-safety properties allows to reason about the executions of different programs. Prominent examples of 2-safety include non-interference, continuity and determinism, whereas prominent examples of generalized 2-safety include various notions of equivalence that have been considered in the literature, for instance on compiler optimization.

In this paper, we consider the problem of using deductive program verification to prove generalized 2-safety properties. Since it traditionally focuses on trace properties rather than (generalized) 2-safety properties, deductive program verification cannot be used for this purpose in an amenable and practical way.

However, researchers have developed two approaches for applying deductive program verification to generalized 2-safety. The first approach is based on developing relational program logics for reasoning about generalized 2-safety. An example of this approach is Benton's Relational Hoare Logic (RHL) [13]. RHL considers judgments of the form $\{\varphi\}\, c_1 \sim c_2\, \{\psi\}$, where the pre- and post-conditions φ and ψ are relations over states: informally, the judgment above is valid if all terminating runs of c_1 and c_2 starting from pairs of states satisfying the pre-condition φ end in pairs of states satisfying the post-condition ψ. Relational program logics are appealing for their intuitive nature but they are seldom implemented and are inherently in-

[☆] Part of this work was developed while the second and third author were affiliated with IMDEA Software Institute. This article is partially based on [5] and [6].

E-mail addresses: gjbarthe@gmail.com (G. Barthe), juanmanuel.crespo@fireeye.com (J.M. Crespo), cesar.kunz@fireeye.com (C. Kunz).

http://dx.doi.org/10.1016/j.jlamp.2016.05.004
2352-2208/© 2016 Elsevier Inc. All rights reserved.

complete when considering structurally different programs—or in the case of a single program, two executions that follow a different control flow. The second approach reduces the problem of verifying a generalized 2-safety property into a standard verification problem, which can then be tackled using standard deductive program verification. This approach is based on product programs [5,7,40], a general class of constructions which informally transform two programs c_1 and c_2 into a single program c that captures the behavior of c_1 and c_2, so that the verification of a generalized 2-safety property for c_1 and c_2 can be reduced to the verification of a safety property for c. Product programs are appealing because they provide a means to rely on existing tools, but only the simplest product constructions have been implemented, and the scope of the different approaches is not well-understood.

This paper is constituted of two parts. In the first part, we carry a detailed comparison of the different forms of product programs and relational program logics for a core imperative language. In the second part, we sketch applications of product programs to continuity, a 2-safety property which informally states that small modifications in a program's inputs only have a small impact on its outputs, non-interference, a baseline confidentiality policy, translation validation, and differential privacy [21], a mathematically rigorous policy for privacy-preserving computations. We also discuss other applications briefly.

Discussion Note that in theoretical terms, deductive program verification can always be used for establishing (generalized) 2-safety. For instance, non-interference states that a program is constant in one of its arguments, which can be modeled using Hoare logic; for instance, the fact that the output z a program c with variables x and y only depends on y can be deduced from the validity of the Hoare triple $\vdash \{\top\}\, c\, \{z = f(y^\star)\}$, where f is a function that captures the behavior of the program, and $y^\star$ denotes the value of y in the initial memory, and $\top$ is the trivially valid assertion. Similarly, the equivalence between two programs c_1 and c_2 with input x and output z can be deduced from the validity of the Hoare triples $\vdash \{\top\}\, c_1\, \{z = f(x^\star)\}$ and $\vdash \{\top\}\, c_2\, \{z = f(x^\star)\}$, where f is a function that captures the behavior of the program. However, verifying the two programs in isolation is not always practical. An extreme example is to show that an arbitrarily complex program behaves similarly to itself, given equal input values. In this case, full functional verification can be complex whereas relational verification is trivial.

2. Syntax and semantics of programs

We assume given disjoint sets $\mathcal{V}$ of scalar variables, $\mathcal{A}$ of array variables, IExp of arithmetic expressions and BExp of boolean expressions and let x, a, e and b range over $\mathcal{V}$, $\mathcal{A}$, IExp and BExp respectively.

Definition 1 *(Statements).* The set Stmt of statements or commands is given by the grammar:

$$c \ ::= \mathsf{skip} \mid x := e \mid a[e] := e \mid \mathsf{assert}(b) \mid c; c \mid \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ b\ \mathsf{do}\ c$$

We let $\mathsf{var}(c)$ denote the set of (read and write) variables of a command c.

Programs states are partial functions from scalar variables and array variables with indices to integer values, i.e. $\mathcal{S} = (\mathcal{V} + (\mathcal{A} \times \mathbb{N})) \rightharpoonup \mathbb{Z}$; we let $\uplus$ denote disjoint union over states, when viewed as partial maps. We denote $\sigma\langle v \mapsto n\rangle$ and $\sigma\langle a[k] \mapsto n\rangle$ the standard scalar variable and array variable state update, respectively. The interpretation of arithmetic and boolean expressions is given by two functions $([\![e]\!])_{e \in \mathsf{IExp}} : \mathcal{S} \rightharpoonup \mathbb{Z}$, and $([\![b]\!])_{b \in \mathsf{BExp}} : \mathcal{S} \rightharpoonup \mathbb{B}$. Commands are interpreted as functions from states to states with errors; the function $([\![c]\!])_{c \in \mathsf{Stmt}} : \mathcal{S} \rightharpoonup \mathcal{S}_{\mathsf{err}}$ is defined by the clauses:

$$[\![\mathsf{skip}]\!]\,\sigma = \sigma$$

$$[\![x := e]\!]\,\sigma = \sigma\langle x \mapsto [\![e]\!]\,\sigma\rangle$$

$$[\![a[e_0] := e]\!]\,\sigma = \sigma\langle a[[\![e_0]\!]\,\sigma] \mapsto [\![e]\!]\,\sigma\rangle$$

$$[\![\mathsf{assert}(b)]\!]\,\sigma = \begin{cases} \sigma & \text{if } [\![b]\!]\,\sigma = \mathsf{true} \\ \mathsf{err} & \text{if } [\![b]\!]\,\sigma = \mathsf{false} \end{cases}$$

$$[\![c_1; c_2]\!]\,\sigma = [\![c_2]\!]_{\mathsf{err}}\,([\![c_1]\!]\,\sigma)$$

$$[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\,\sigma = \begin{cases} [\![c_1]\!]\,\sigma & \text{if } [\![b]\!]\,\sigma = \mathsf{true} \\ [\![c_2]\!]\,\sigma & \text{if } [\![b]\!]\,\sigma = \mathsf{false} \end{cases}$$

$$[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] = \mathsf{fix}\ F$$

$$\text{where:} \quad F\,f\,\sigma = \begin{cases} f_{\mathsf{err}}\,([\![c]\!]\,\sigma) & \text{if } [\![b]\!]\,\sigma = \mathsf{true} \\ \sigma & \text{if } [\![b]\!]\,\sigma = \mathsf{false} \end{cases}$$

Where fix denotes the least fixpoint in the domain $S \rightharpoonup S_{\mathsf{err}}$, and for all $f : A \rightharpoonup B$ we denote $f_{\mathsf{err}} : A_{\mathsf{err}} \rightharpoonup B$ the natural lifting to A_{err}: $f_{\mathsf{err}}\,\mathsf{err} = \mathsf{err}$ and for all $x \neq \mathsf{err}$ $f_{\mathsf{err}}\,x = f\,x$. Note that $[\![c]\!]\ \sigma$ is either equal to a state σ', to the error value err (we then say that program c is blocking for initial state σ), or undefined.

3. Hoare logic

Program correctness is expressed by triples $\{\varphi\}\,c\,\{\psi\}$, where c is a command, and φ and ψ are assertions; formally, assertions are first-order formulae over program variables and array expressions. We let $[\![\phi]\!]$ denote the set of states satisfying ϕ.

Definition 2 *(Valid triple).* The judgment $\{\varphi\}\,c\,\{\psi\}$ is valid, written $\vDash \{\varphi\}\,c\,\{\psi\}$, iff for all states $\sigma, \sigma' \in \mathcal{S}$, we have that $\sigma \in [\![\varphi]\!]$ implies that $[\![c]\!]\ \sigma \neq \mathsf{err}$ and if moreover $[\![c]\!]\ \sigma = \sigma'$ then $\sigma' \in [\![\psi]\!]$.

Triples can be proved valid using the following rules:

$$\frac{}{\vdash \{\phi\}\,\mathsf{skip}\,\{\phi\}}\ (\textsc{Skip})$$

$$\frac{}{\vdash \{\phi[e/x]\}\,x := e\,\{\phi\}}\ (\textsc{Assign})$$

$$\frac{}{\vdash \{\phi[e/a[e_0]]\}\,a[e_0] := e\,\{\phi\}}\ (\textsc{Update})$$

$$\frac{\phi \Rightarrow b}{\vdash \{\phi\}\,\mathsf{assert}(b)\,\{\phi\}}\ (\textsc{Assert})$$

$$\frac{\vdash \{\varphi\}\,c\,\{\phi\} \qquad \vdash \{\phi\}\,d\,\{\psi\}}{\vdash \{\varphi\}\,c; d\,\{\psi\}}\ (\textsc{Seq})$$

$$\frac{\vdash \{\varphi \wedge b\}\,c\,\{\psi\} \qquad \vdash \{\varphi \wedge \neg b\}\,d\,\{\psi\}}{\vdash \{\varphi\}\,\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ d\,\{\psi\}}\ (\textsc{If})$$

$$\frac{\vdash \{\phi \wedge b\}\,c\,\{\phi\}}{\vdash \{\phi\}\,\mathsf{while}\ b\ \mathsf{do}\ c\,\{\phi \wedge \neg b\}}\ (\textsc{While})$$

$$\frac{\vdash \{\varphi\}\,c\,\{\psi\} \qquad \varphi' \Rightarrow \varphi \qquad \psi \Rightarrow \psi'}{\vdash \{\varphi'\}\,c\,\{\psi'\}}\ (\textsc{Sub})$$

Hoare logic is sound.

Lemma 1. *If* $\vdash \{\varphi\}\,c\,\{\psi\}$ *is derivable, then* $\vDash \{\varphi\}\,c\,\{\psi\}$.

The logic enforces partial correctness, i.e. it does not ensure that the execution terminate.

4. Relational Hoare logic

Relational Hoare logic is a program logic for reasoning about two programs. Its judgments are of the form $\{\varphi\}\,c_1 \sim c_2\,\{\psi\}$, where c_1 and c_2 are commands, and φ and ψ are assertions. For the sake of simplicity, we assume that c_1 and c_2 are separable commands, i.e. they do not have variables in common: $\mathsf{var}(c_1) \cap \mathsf{var}(c_2) = \emptyset$. Under this assumption, relational assertions φ and ψ are first-order formulae over the variables and array expressions of c_1 and c_2. Note, however, that Benton's formulation of relational Hoare logic does not require that the two programs are separable, but uses instead tags $\langle 1\rangle$ and $\langle 2\rangle$ in assertions to distinguish between the interpretations of a variable in the left or right program states.

Definition 3 *(Valid quadruple).* The judgment $\{\varphi\}\,c_1 \sim c_2\,\{\psi\}$ is valid, written $\vDash \{\varphi\}\,c_1 \sim c_2\,\{\psi\}$, iff for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2' \in \mathcal{S}$, we have that $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ implies that $[\![c_1]\!]\ \sigma_1 \neq \mathsf{err}$ and $[\![c_2]\!]\ \sigma_2 \neq \mathsf{err}$ and if moreover $[\![c_1]\!]\ \sigma_1 = \sigma_1'$ and $[\![c_2]\!]\ \sigma_2 = \sigma_2'$ then $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.

Quadruples can be proved valid using the following rules:

$$\frac{}{\vdash_{\min} \{\phi\}\, \mathsf{skip} \sim \mathsf{skip}\, \{\phi\}}\ \text{(R-Skip)}$$

$$\frac{}{\vdash_{\min} \{\phi[e_1/x_1, e_2/x_2]\}\, x_1 := e_1 \sim x_2 := e_2\, \{\phi\}}\ \text{(R-Assign)}$$

$$\frac{}{\vdash_{\min} \{\phi[e_1/a_1[e'_1], e_2/a_2[e'_2]]\}\, a_1[e'_1] := e_1 \sim a_2[e'_2] := e_2\, \{\phi\}}\ \text{(R-Update)}$$

$$\frac{\phi \Rightarrow b_1 \wedge b_2}{\vdash_{\min} \{\phi\}\, \mathsf{assert}(b_1) \sim \mathsf{assert}(b_2)\, \{\phi\}}\ \text{(R-Assert)}$$

$$\frac{\vdash_{\min} \{\varphi\}\, c_1 \sim c_2\, \{\phi\} \qquad \vdash_{\min} \{\phi\}\, d_1 \sim d_2\, \{\psi\}}{\vdash_{\min} \{\varphi\}\, c_1; d_1 \sim c_2; d_2\, \{\psi\}}\ \text{(R-Seq)}$$

$$\frac{\vdash_{\min} \{\varphi \wedge b_1\}\, c_1 \sim c_2\, \{\psi\} \qquad \vdash_{\min} \{\varphi \wedge \neg b_1\}\, d_1 \sim d_2\, \{\psi\} \qquad \varphi \Rightarrow b_1 = b_2}{\vdash_{\min} \{\varphi\}\, \mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ d_1 \sim \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ d_2\, \{\psi\}}\ \text{(R-If)}$$

$$\frac{\vdash_{\min} \{\phi \wedge b_1\}\, c_1 \sim c_2\, \{\phi\} \qquad \phi \Rightarrow b_1 = b_2}{\vdash_{\min} \{\phi\}\, \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ b_2\ \mathsf{do}\ c_2\, \{\phi \wedge \neg b_1\}}\ \text{(R-While)}$$

$$\frac{\vdash_{\min} \{\varphi\}\, c_1 \sim c_2\, \{\psi\} \qquad \varphi' \Rightarrow \varphi \qquad \psi \Rightarrow \psi'}{\vdash_{\min} \{\varphi'\}\, c_1 \sim c_2\, \{\psi'\}}\ \text{(R-Sub)}$$

(Minimal) relational Hoare logic is sound.

Lemma 2. *If* $\vdash_{\min} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ *is derivable, then* $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.

Minimal Relational Hoare Logic requires that both programs execute in lockstep. As a consequence, one cannot relate a program c with the program $\mathsf{skip}; c$ or two sequences of assignments with different length. One can conveniently extend Minimal Relational Hoare Logic into Core Relational Hoare Logic which features rules for reasoning about two programs that do not execute basic instructions and conditional statements in lockstep. Contrary to the core rules above, which require that the left and right programs have the same shape, these rules analyze only one of the two programs. Derivability in Core Relational Hoare Logic is denoted by $\vdash_{\text{core}}$. The rules for reasoning about left programs are:

$$\frac{\vdash_{\text{core}} \{\varphi[e/x]\}\, \mathsf{skip} \sim c\, \{\psi\}}{\vdash_{\text{core}} \{\varphi\}\, x := e \sim c\, \{\psi\}}\ \text{(R-Assign-L)}$$

$$\frac{\vdash_{\text{core}} \{\varphi[e/a[e']]\}\, \mathsf{skip} \sim c\, \{\psi\}}{\vdash_{\text{core}} \{\varphi\}\, a[e'] := e \sim c\, \{\psi\}}\ \text{(R-Update-L)}$$

$$\frac{\vdash_{\text{core}} \{\varphi\}\, \mathsf{skip} \sim c\, \{\psi\} \qquad \varphi \Rightarrow b}{\vdash_{\text{core}} \{\varphi\}\, \mathsf{assert}(b) \sim c\, \{\psi\}}\ \text{(R-Assert-L)}$$

$$\frac{\vdash_{\text{core}} \{\varphi \wedge b\}\, c_1 \sim c\, \{\psi\} \qquad \vdash_{\text{core}} \{\varphi \wedge \neg b\}\, c_2 \sim c\, \{\psi\}}{\vdash_{\text{core}} \{\varphi\}\, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \sim c\, \{\psi\}}\ \text{(R-If-L)}$$

Similar rules exist to reason about right programs.

Although these rules significantly enhance its power, the extended proof system remains incomplete. In particular, the extended proof system is unable to reason about programs which have different loop structures, or when the left and right program coincide but the precondition is too weak to ensure that loops execute in lockstep. This incompleteness can be resolved by adding a self-composition rule:

$$\frac{\vdash \{\varphi\}\, c_1; c_2\, \{\psi\}}{\vdash_{\min} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}}\ \text{(R-SelfComp)}$$

Relative completeness of (core or) minimal relational Hoare logic extended with self-composition follows immediately from relative completeness of self-composition [7]. However, many examples of interest can be handled using an unfolding-based rule, without requiring the full power of self-composition. Informally, we can associate to every program c a set of unfoldings; a program c' is an unfolding of c, written $\vdash [c] \succcurlyeq [c']$, iff for every states σ, σ' and σ'', we have that: i. $[\![c]\!]\ \sigma = \sigma'$

and $[\![c']\!]\ \sigma = \sigma''$ implies that $\sigma' = \sigma''$; ii. $[\![c]\!]\ \sigma = \mathsf{err}$ implies $[\![c']\!]\ \sigma = \mathsf{err}$. We define extended relational Hoare logic by adding to core relational Hoare logic the rule for unfolding:

$$\frac{\vdash_{\mathsf{ext}} \{\varphi\}\, c_1' \sim c_2' \,\{\psi\} \qquad \vdash [c_1] \succcurlyeq [c_1'] \qquad \vdash [c_2] \succcurlyeq [c_2']}{\vdash_{\mathsf{ext}} \{\varphi\}\, c_1 \sim c_2 \,\{\psi\}}\ (\text{R-Unfold})$$

The rule is obviously sound. One can then introduce a proof system for deriving valid judgments of the form $\vdash [c] \succcurlyeq [c']$. Some typical rules are:

$$\frac{}{\vdash [\text{if } b \text{ then } c_1 \text{ else } c_2] \succcurlyeq [\text{assert}(b); c_1]}$$

$$\frac{}{\vdash [\text{if } b \text{ then } c_1 \text{ else } c_2] \succcurlyeq [\text{assert}(\neg b); c_2]}$$

$$\frac{\vdash [c_1] \succcurlyeq [c_1'] \qquad \vdash [c_2] \succcurlyeq [c_2']}{\vdash [\text{if } b \text{ then } c_1 \text{ else } c_2] \succcurlyeq [\text{if } b \text{ then } c_1' \text{ else } c_2']}$$

$$\frac{}{\vdash [\text{while } b \text{ do } c] \succcurlyeq [\text{assert}(b); c; \text{while } b \text{ do } c]}$$

$$\frac{}{\vdash [\text{while } b \text{ do } c] \succcurlyeq [\text{while } b \wedge b' \text{ do } c; \text{while } b \text{ do } c]}$$

$$\frac{}{\vdash [\text{while } b \text{ do } c] \succcurlyeq [\text{assert}(b); c; \text{assert}(\neg b)]}$$

$$\frac{\vdash [c] \succcurlyeq [c']}{\vdash [\text{while } b \text{ do } c] \succcurlyeq [\text{while } b \text{ do } c']}$$

$$\frac{\vdash [c] \succcurlyeq [c'] \qquad \vdash [c'] \succcurlyeq [c'']}{\vdash [c] \succcurlyeq [c'']}$$

$$\frac{}{\vdash [c] \succcurlyeq [c]}$$

$$\frac{\vdash [c_1] \succcurlyeq [c_1'] \qquad \vdash [c_2] \succcurlyeq [c_2']}{\vdash [c_1; c_2] \succcurlyeq [c_1'; c_2']}$$

Notice, however, that the rules above still assume commands have some structural similarity. Therefore, the self-composition rule is still required to achieve completeness.

5. Product programs

Recall that a product program for c_1 and c_2 is a program c that emulates the behaviors of c_1 and c_2. In this section, we define different notions of product programs and put them in correspondence with different relational logics.

5.1. Minimal products

We first define minimal products, and show they are equivalent to minimal relational Hoare logic.

Definition 4 *(Minimal product).* The product of two commands c_1 and c_2 is, if it exists, the command c s.t. $c_1 \times c_2 \longrightarrow_{\min} c$ is derivable using the rules:

$$\frac{}{\text{skip} \times \text{skip} \longrightarrow_{\min} \text{skip}}\ (\text{P-Skip})$$

$$\frac{}{(x_1 := e_1) \times (x_2 := e_2) \longrightarrow_{\min} x_1 := e_1; x_2 := e_2}\ (\text{P-Assign})$$

$$\frac{}{(a_1[e_1'] := e_1) \times (a_2[e_2'] := e_2) \longrightarrow_{\min} a_1[e_1'] := e_1; a_2[e_2'] := e_2}\ (\text{P-Update})$$

$$\frac{}{(\mathsf{assert}(b_1)) \times (\mathsf{assert}(b_2)) \longrightarrow_{\min} \mathsf{assert}(b_1); \mathsf{assert}(b_2)}\ (\text{P-ASSERT})$$

$$\frac{c_1 \times c_2 \longrightarrow_{\min} c \qquad c_1' \times c_2' \longrightarrow_{\min} c'}{c_1; c_1' \times c_2; c_2' \longrightarrow_{\min} c; c'}\ (\text{P-SEQ})$$

$$\frac{c_1 \times c_2 \longrightarrow_{\min} c \qquad c_1' \times c_2' \longrightarrow_{\min} c'}{(\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1') \times (\mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2') \longrightarrow_{\min} \begin{array}{l}\mathsf{assert}(b_1 = b_2);\\ \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ c'\end{array}}\ (\text{P-IF})$$

$$\frac{c_1 \times c_2 \longrightarrow_{\min} c}{(\mathsf{while}\ b_1\ \mathsf{do}\ c_1) \times (\mathsf{while}\ b_2\ \mathsf{do}\ c_2) \longrightarrow_{\min} \begin{array}{l}\mathsf{assert}(b_1 = b_2);\\ \mathsf{while}\ b_1\ \mathsf{do}\ (c; \mathsf{assert}(b_1 = b_2))\end{array}}\ (\text{P-WHILE})$$

Minimal products are unique—if they exist. Moreover, the minimal product of two commands that are provably related by a relational Hoare specification is always defined, and verifies this same specification—viewed as a Hoare specification. The converse also holds.

Proposition 1. *The following are equivalent:*

- $\vdash_{\min} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ *is derivable;*
- *there exists* c *such that* $c_1 \times c_2 \longrightarrow_{\min} c$ *and* $\vdash \{\varphi\}\, c\, \{\psi\}$ *is derivable in Hoare logic.*

5.2. Core products

Extending the equivalence to one-sided rules requires to extend the definition of product programs with the following clauses (we only show clauses for left rules; clauses for right rules are symmetric):

$$\frac{}{(x := e) \times \mathsf{skip} \longrightarrow_{\text{core}} x := e}\ \text{P-ASSIGN-L}$$

$$\frac{}{(a[e'] := e) \times \mathsf{skip} \longrightarrow_{\text{core}} a[e'] := e}\ \text{P-UPDATE-L}$$

$$\frac{}{(\mathsf{assert}(b)) \times \mathsf{skip} \longrightarrow_{\text{core}} \mathsf{assert}(b)}\ \text{P-ASSERT-L}$$

$$\frac{c_1 \times \mathsf{skip} \longrightarrow_{\text{core}} c \qquad c_1' \times c_2' \longrightarrow_{\text{core}} c'}{c_1; c_1' \times c_2' \longrightarrow_{\text{core}} c; c'}\ \text{P-SEQSKIP-L}$$

$$\frac{c_1 \times c \longrightarrow_{\text{core}} c_1' \qquad c_2 \times c \longrightarrow_{\text{core}} c_2'}{(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \times c \longrightarrow_{\text{core}} \mathsf{if}\ b\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'}\ \text{P-IF-L}$$

Proposition 1 extends to this setting.

Proposition 2. *The following are equivalent:*

- $\vdash_{\text{core}} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ *is derivable;*
- $c_1 \times c_2 \longrightarrow_{\text{core}} c$ *and* $\vdash \{\varphi\}\, c\, \{\psi\}$ *is derivable.*

5.3. Extended products

In order to reason about program optimizations, it is useful to extend the core product construction with a clause for unfoldings; we let $c_1 \times c_2 \longrightarrow_{\text{ext}} c$ denote derivability of the product construction with unfoldings. The only new rule is:

$$\frac{\vdash [c_1] \succcurlyeq [c_1'] \qquad \vdash [c_2] \succcurlyeq [c_2'] \qquad c_1' \times c_2' \longrightarrow_{\text{ext}} c}{c_1 \times c_2 \longrightarrow_{\text{ext}} c}$$

Again, Proposition 1 extends to this setting.

Proposition 3. *The following are equivalent:*

- $\vdash_{\text{ext}} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ *is derivable;*
- $c_1 \times c_2 \longrightarrow_{\text{ext}} c$ *and* $\vdash \{\varphi\}\, c\, \{\psi\}$ *is derivable.*

5.4. Summary

We have introduced three notions of product programs: minimal products, core products, and extended products. The first notion is equivalent to a minimal relational logic.

The second notion is equivalent to a core relational logic that is close to Benton's original formulation, and can be used to prove 2-safety properties; in particular, one can prove that the core relational logic is more expressive than baseline information flow type systems for proving non-interference.

The third notion includes a notion of unfolding, and is useful in the context of translation validation to prove the correctness of program optimizations that modify control flow.

6. Combining verification and products

Proposition 1 and its generalizations show that one can prove relational specifications in two steps: first, construct product programs; second, use standard verification. The main benefit of this two-step approach is that one can use off-the-shelf verification tools for the second step. However, it might be challenging to choose which product program makes subsequent verification simple or even possible, especially because in this approach the product construction is not driven by the intended specification. An interesting alternative is to combine product construction with some form of verification, using the intended specification to drive the process. For instance, one can envision an approach where the precondition is propagated forward using symbolic execution, and where products synchronize branching statements in as much as possible. Rather than elaborate of this approach, we present a related approach in which the program construction and the verification is performed simultaneously; this approach can be seen as a straightforward generalization of [38]. Formally, we consider judgments of the form $\vdash \{\phi\}c_1 \times c_2 \to c\{\psi\}$, and use the following proof system (some rules omitted):

$$\frac{\vdash \{\varphi\}\, c_1; c_2\, \{\psi\}}{\vdash \{\varphi\}c_1 \times c_2 \to c_1; c_2\{\psi\}}\ (\text{LP-SelfComp})$$

$$\frac{\vdash \{\phi\}c_1 \times c_2 \to c\{\varphi\} \qquad \vdash \{\varphi\}c_1' \times c_2' \to c'\{\psi\}}{\vdash \{\phi\}c_1; c_1' \times c_2; c_2' \to c; c'\{\psi\}}\ (\text{LP-Seq})$$

$$\frac{\phi \Rightarrow b_1 = b_2 \qquad \vdash \{\phi \wedge b_1\}c_1 \times c_2 \to c\{\psi\} \qquad \vdash \{\phi \wedge \neg b_1\}d_1 \times d_2 \to d\{\psi\}}{\vdash \{\phi\}\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ d_1 \times \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ d_2 \to \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ d\{\psi\}}\ (\text{LP-If})$$

$$\frac{\phi \Rightarrow b_1 = b_2 \qquad \vdash \{\phi \wedge b_1\}c_1 \times c_2 \to c\{\phi\}}{\vdash \{\phi\}\mathsf{while}\ b_1\ \mathsf{do}\ c_1 \times \mathsf{while}\ b_2\ \mathsf{do}\ c_2 \to \mathsf{while}\ b_1\ \mathsf{do}\ c\{\phi \wedge \neg b_1\}}\ (\text{LP-While})$$

$$\frac{\vdash \{\phi'\}c_1 \times c_2 \to c\{\psi'\} \qquad \phi \Rightarrow \phi' \qquad \psi' \Rightarrow \psi}{\vdash \{\phi\}c_1 \times c_2 \to c\{\psi\}}\ (\text{LP-Sub})$$

$$\frac{\vdash [c_1] \succcurlyeq [c_1'] \qquad \vdash [c_2] \succcurlyeq [c_2'] \qquad \vdash \{\phi\}c_1' \times c_2' \to c\{\psi\}}{\vdash \{\phi\}c_1 \times c_2 \to c\{\psi\}}\ (\text{LP-Ref})$$

The approach is sound.

Proposition 4. *If* $\vdash \{\varphi\}c_1 \times c_2 \to c\{\psi\}$ *then* $\vdash \{\varphi\}\, c\, \{\psi\}$ *and* $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.

Conversely, note that if $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$, then a relatively complete strategy to prove $\vdash \{\varphi\}c_1 \times c_2 \to c\{\psi\}$ for some c is to set $c = c_1; c_2$, and to prove $\vdash \{\varphi\}\, c\, \{\psi\}$.

7. Applications

In this section, we outline applications of product programs to a representative set of generalized 2-properties, including continuity, non-interference, equivalence, and to a probabilistic 2-property called differential privacy. For the sake of readability, we sometimes omit the insertion of assert statements.

```
$i := 0;$
while $(i < N)$ do
  $j := N-1;$
  while $(j > i)$ do
    if $(a[j-1] > a_1[j])$ then $(x := a[j];\ a[j] := a[j-1];\ a[j-1] := x)$
    $j--$
  $i++$
```

Fig. 1. Bubble-sort algorithm.

```
$i_1 := 0; i_2 := 0;$ assert$((i_1 < N) = (i_2 < N));$
while $(i_1 < N)$ do
  $j_1 := N-1; j_2 := N-1;$
  assert$((j_1 > i_1) = (j_2 > i_2));$
  while $(j_1 > i_1)$ do
    if $(a_1[j_1-1] > a_1[j_1])$ then $x_1 := a_1[j_1];\ a_1[j_1] := a_1[j_1-1];\ a_1[j_1-1] := x_1;$
    if $(a_2[j_2-1] > a_2[j_2])$ then $x_2 := a_2[j_2];\ a_2[j_2] := a_2[j_2-1];\ a_2[j_2-1] := x_2;$
    $j_1--; j_2--;$ assert$((j_1 > i_1) = (j_2 > i_2))$
  $i_1++; i_2++;$ assert$((i_1 < N) = (i_2 < N))$
```

Fig. 2. Bubble-sort: product program.

7.1. Program continuity

In [5], we use product programs for proving program continuity, a.k.a. robustness [16]. Informally, a program is continuous if small variations on its inputs only causes small variations on its output. As for mathematical functions, there exist different notions of program continuity. It is often beneficial for the purpose of verification to prove Lipschitz continuity, which bounds the distance of the program outputs by a multiplicative factor of the program inputs. In this paragraph, we consider Lipschitz continuity for sorting algorithms. Consider arrays a_1 and a_2 of length N and such that $|a_1[i] - a_2[i]| < \epsilon$ for all $i \in \{0, \dots, N-1\}$. Clearly, the variation ϵ on a_1 and a_2 may affect the permutations used to sort the arrays. However, this small variation on the input data can at most cause a small variation in the sorted arrays, which indeed satisfy that $|a_1[i] - a_2[i]| < \epsilon$ for all $i \in \{0, \dots, N-1\}$.

Consider the command c in Fig. 1 that applies the bubble-sort algorithm to an array a. One can verify the validity of the relational judgment $\{\phi\}\, c_1 \sim c_2\, \{\phi\}$, where c_1 and c_2 are renamings of the bubble-sort algorithm described above (all variables are tagged with a subscript 1 and 2 respectively), and ϕ is the assertion

$$\forall i \in \{0, \dots, N-1\}.\, |a_1[i] - a_2[i]| < \epsilon$$

The verification can be performed using the product program c, shown in Fig. 2, that weaves the instructions of c_1 and c_2. The product program c synchronizes the loops iterations of c_1 and c_2, as their loop guards are equivalent and thus perform the same number of iterations. This is not the case with the conditional statements inside the loop body, as the small variations on the contents of the array a_1 w.r.t. a_2 may break the equivalence of the guards $a_1[j_1-1] > a_1[j_1]$ and $a_2[j_2-1] > a_2[j_2]$. One can use Hoare logic to verify the validity of the non-relational judgment $\vdash \{\phi\}\, c\, \{\phi\}$. Since the program product is a correct representation of its components, the validity of the Hoare triple over c is enough to establish the validity of the relational judgment over c_1 and c_2.

7.2. Information flow

Non-interference [23] is an information flow policy which captures that the input/output behavior of a program does not leak any information about the confidential data it manipulates. In the simplest case, non-interference for a program c_0 assumes that its program variables are tagged as secret or public and requires that the final values of its public variables do not depend on the initial values of the secret variables. Non-interference can be modeled as a 2-property. Indeed, let c_1 and c_2 be renamings of the program c_0 by tagging each variable in c_0 with subscript 1 and 2 respectively. If x is the sole public program variable, non-interference of c_0 can be captured by the relational judgment $\{x_1 = x_2\}\, c_1 \sim c_2\, \{x_1 = x_2\}$, and can be verified by constructing a valid product c of c_1 and c_2 such that $\vdash \{x_1 = x_2\}\, c\, \{x_1 = x_2\}$.

We illustrate this approach with an example drawn from [20]. The example merges a table containing personal information with a table containing salary information. The salary information is conditioned by a special field *JoinInd* indicating whether the personal information is private and that it should not be included as the result of the join operation. We model tables as arrays of records. Thus, we assume given two arrays *ps* and *es* that respectively model the *Payroll* and the *Employee* tables. Each element in the *Payroll* table is a record with three fields: *PID*, which stores the personal ID, *salary*, which holds the salary, and *JoinInd*, which stores a boolean which determines whether the record can be included in a join operation. Each element in the employee table is a record with two fields: *Name*, which contains the employee name,

```
$i_1 := 0$;  $i_2 := 0$; assert($(i_1 < N) = (i_2 < N)$);
while ($i_1 < N$) do
  assert($ps_1[i_1].JoinInd = ps_2[i_2].JoinInd$);
  if ($ps_1[i_1].JoinInd$) then
    $j_1 := 0$;  $j_2 := 0$;  assert($(j_1 < M) = (j_2 < M)$);
    while ($j_1 < M$) do
      assert($(ps_1[i_1].PID = es_1[j_1].EID) = (ps_2[i_2].PID = es_2[j_2].EID)$);
      if ($ps_1[i_1].PID = es_1[j_1].EID$) then
        $tab_1[i_1].employee := es_1[j_1]$;  $tab_2[i_2].employee := es_2[j_2]$;
        $tab_1[i_1].payroll := ps_1[i_1]$;  $tab_2[i_2].payroll := ps_2[i_2]$;
      $j_1$++;  $j_2$++; assert($(j_1 < M) = (j_2 < M)$)
  $i_1$++;  $i_2$++; assert($(i_1 < N) = (i_2 < N)$)
```

Fig. 3. Non-interference product.

and EID, which stores the employee ID. The result of the program is a joint table, modeled as an array with two fields *employee* and *payroll*. The code of the program is given below:

```
$i := 0$;
while ($i < N$) do
  if ($ps[i].JoinInd$) then
    $j := 0$;
    while ($j < M$) do
      if ($ps[i].PID = es[j].EID$) then $tab[i].employee := es[j]$;  $tab[i].payroll := ps[i]$
      $j$++
  $i$++
```

Fig. 3 shows the construction of the program product c. Using Hoare logic on the product program, we can prove that the input data marked as private does not interfere with the final result: if the values stored in the input arrays coincide for the public indices (i.e., for i s.t. $ps[i].JoinInd$ is true), then the return data coincides at the public indices. More formally, we define the precondition ϕ:

$$\begin{array}{l} \forall j.\ 0 \le j < M \Longrightarrow es_1[j] = es_2[j] \\ \wedge \forall i.\ 0 \le i < N \Longrightarrow ps_1[i].PID = ps_2[i].PID \\ \wedge \forall j.\ 0 \le j < N \Longrightarrow ps_1[i].JoinInd = ps_2[i].JoinInd \\ \wedge \forall j.\ 0 \le j < N \Longrightarrow ps_1[i].JoinInd \Longrightarrow ps_1[i].salary = ps_2[i].salary \end{array}$$

and the postcondition ψ as

$$\forall i.\ 0 \le i < N \Longrightarrow ps_1[i].JoinInd \Rightarrow tab_1[i] = tab_2[i]$$

and we can prove that $\vdash \{\phi\}\, c\, \{\psi\}$.

7.3. Translation validation

In [5], we use (extended) product programs for proving correctness of loop optimizations, including a static caching based optimization for incremental computation. Many of these optimizations do not preserve the structure of programs, requiring to use extended product programs for their verification.

A first intuition on the construction of products from structurally dissimilar components is shown in the following basic example, in which we assume that N is a positive number, i.e. $0 \le N$. The left program is:

$$i := 0;\ \text{while}\,(i \le N)\,\text{do}(x \mathrel{+}= i;\ i{+}{+})$$

and the right program is:

$$j := 1;\ \text{while}\,(j \le N)\,\text{do}(y \mathrel{+}= j;\ \ j{+}{+})$$

The left program performs $N + 1$ iterations, whereas the right program performs N iterations, and hence their minimal product program will be blocking. On the other hand, one can unroll the first loop iteration of the left program before synchronizing the loop statements. The product program becomes:

$$\begin{array}{l} i := 0;\ x \mathrel{+}= i;\ i{+}{+};\ j := 1;\ \text{assert}((i \le N) = (j \le N)); \\ \text{while}\,(i \le N)\,\text{do}\ (y \mathrel{+}= j;\ j{+}{+};\ x \mathrel{+}= i;\ i{+}{+};\ \text{assert}((i \le N) = (j \le N))) \end{array}$$

This simple idea maximizes synchronization instead of relying plainly on self-composition, which requires a greater specification and verification effort. Indeed, the sequential composition of the source and transformed program requires providing invariants of the form $x = X + \frac{i(i-1)}{2}$ and $y = Y + \frac{j(j-1)}{2}$, respectively (under the preconditions $x = X$ and $y = Y$). In contrast, by the construction of the product, the trivial loop invariant $i = j \wedge x = y$ is sufficient to verify that the two programs above satisfy the pre and post-condition $x = y$.

```
i := 0;  j := 0;  assert(i < N);
a1[i]++;  b1[i] += a1[i];  c1[i] += b1[i];  i++;  a2[0]++;  b2[0] += a2[0];  assert(i < N);
a1[i]++;  b1[i] += a1[i];  c1[i] += b1[i];  i++;  a2[1]++;  assert((i < N) = (j < N−2));
while (i < N) do (a1[i]++; b1[i] += a1[i]; c1[i] += b1[i]; i++; a2[j+2]++; b2[j+1] += a2[j+1]; c2[j] += b2[j]; j++;
                assert((i < N) = (j < N−2)))
c2[j] += b2[j];  b2[j+1] += a2[j+1];  c2[j+1] += b2[j+1]
```

Fig. 4. Loop pipelining: product program.

Product programs can be used in a similar way to validate instances of loop pipelining, a standard optimization that is used to introduce new opportunities for parallelization. The source program is:

$$i := 0;\ \mathsf{while}\ (i < N)\ \mathsf{do}\ (a_1[i]{+}{+};\ b_1[i] \mathrel{+}= a_1[i];\ c_1[i] \mathrel{+}= b_1[i];\ i{+}{+})$$

where a_1, b_1 and c_1 are arrays of size N, with $2 \leq N$. The transformed program is:

$$\begin{array}{l} j := 0;\ a_2[0]{+}{+};\ b_2[0] \mathrel{+}= a_2[0];\ a_2[1]{+}{+}; \\ \mathsf{while}\ (j < N{-}2)\ \mathsf{do}\ (a_2[j{+}2]{+}{+};\ b_2[j{+}1] \mathrel{+}= a_2[j{+}1];\ c_2[j] \mathrel{+}= b_2[j];\ j{+}{+}) \\ c_2[j] \mathrel{+}= b_2[j];\ b_2[j{+}1] \mathrel{+}= a_2[j{+}1];\ c_2[j{+}1] \mathrel{+}= b_2[j{+}1] \end{array}$$

The source and transformed programs have a similar structure, but one cannot verify them using relational Hoare logic or minimal products, since the number of loop iterations in the source and transformed program do not coincide. However, one can use the product program p shown in Fig. 4 to build a valid product which can be verified using Hoare logic. Formally, one can use Hoare Logic to prove

$$\vdash \{a_1 = a_2 \wedge b_1 = b_2 \wedge c_1 = c_2\}\ p\ \{a_1 = a_2 \wedge b_1 = b_2 \wedge c =_1 c_2\}$$

7.4. Differential privacy

Differential privacy [21] is a privacy policy which provides strong guarantees in the context of privacy-preserving computations by requiring that queries yield probabilistically close results whether or not the data of an individual is included in the database. More formally, a probabilistic program c that takes as input a database d and returns an integer value v is ϵ-differentially private, where $\epsilon \geq 0$, if for every two adjacent databases d_1 and d_2, and for every integer z,

$$\Pr[c(d_1) = z] \leq \exp(\epsilon)\ \Pr[c(d_2) = z]$$

where $\Pr[c(v) = z]$ denotes the probability that the output of the execution of c with initial input v yields z, and adjacency indicates that the databases differ in one of their entries.

There are two basic tools for achieving differential privacy. The first one is to apply differentially private mechanisms for adding probabilistic noise. For instance, one can apply the Laplace mechanism, which is defined by

$$\mathsf{Lap}_\epsilon(x) = x + \frac{\exp(-\epsilon|v|)}{\lambda}$$

If F is k-sensitive with respect to adjacency, i.e. $|F(d_1) - F(d_2)| \leq k$ for all adjacent inputs d_1 and d_2, then the probabilistic function that maps d to $\mathsf{Lap}_\epsilon(F(d))$ is $k\epsilon$-differentially private. The second tool is to use composition theorems for combining two or more differentially private computations into a single private one; for instance, the sequential composition theorem states that sequential composition of an ϵ_1-differentially private computation with an ϵ_2-differentially private computation yields an $\epsilon_1 + \epsilon_2$-differentially private computation.

In [9], we define a product construction to reason about differential privacy of probabilistic computations. In this setting, the product construction takes as input a probabilistic program and outputs a deterministic program that keeps track of the privacy budget of the computation. The basic idea of the product construction is that each call to a mechanism has a privacy cost, that is added from the privacy budget every time the mechanism is called. To keep track of the privacy budget, we use ghost variables ϵ which is incremented after each mechanism is executed, in terms of the distance between their two inputs. In more detail, the product construction conflates the calls to the Laplace mechanism Lap_ϵ in the left and right programs into a single call to an abstract procedure $\mathsf{Lap}^\diamond_\epsilon$, which takes two arguments and returns a pair. The specification of the abstract procedure $\mathsf{Lap}^\diamond_\epsilon$ is:

$$\vdash \{\epsilon_0 = \varepsilon\}\ (v_1, v_2) \leftarrow \mathsf{Lap}^\diamond_\epsilon(e_1, e_2)\ \{v_1 = v_2 \wedge \epsilon_0 = \varepsilon + |e_1 - e_2| \cdot \epsilon\}$$

where ε is an auxiliary variable. Note that our approach requires to extend the programming and assertion languages with real valued variables.

To illustrate our approach, consider the simple program c_0 which takes an array a of length N and computes its noisy partial sums as follows:

```
i := 0;
while i < N do v := Lap (a[i]); b[i] := b[i − 1] + v; i++
return b
```

The product program is:

```
$\epsilon_0 := 0; i_1 := 0; i_2 := 0;$ assert$((i_1 < N) = (i_2 < N));$
while $i_1 < N$ do $(v_1, v_2) := \mathsf{Lap}^{\diamond}_{\epsilon}(a_1[i_1], a_2[i_2]);$
                        $b_1[i_1] := b_1[i_1 - 1] + v_1; b_2[i_2] := b_2[i_2 - 1] + v_2;$
                        $i_1{+}{+}; i_2{+}{+};$ assert$((i_1 < N) = (i_2 < N))$
return $(b_1, b_2)$
```

The main result of [9] states that the original program is $\widehat{\epsilon}$-differentially private provided the product program c satisfies the Hoare triple

$$\vdash \{\mathsf{Adj}(a_1, a_2)\}\, c\, \{b_1 = b_2 \wedge \epsilon_0 \le \widehat{\epsilon}\}$$

where $\mathsf{Adj}(a_1, a_2)$ states that the arrays a_1 and a_2 are adjacent, i.e. differ by at most one on one of their valid index i and coincide on all other valid indices. And indeed, we can use Hoare logic on the product program to prove that the original program is ϵ-differentially private.

7.5. Other applications

In [6], we adapt the approach of the previous sections to a control-flow graph representation of programs. Moreover, we define an asymmetric notion of product which can be used to reason about properties such as refinements, which involve universal quantification on the traces of the first program and existential quantification on the traces of the second program. Using asymmetric products, we validate abstraction/refinement relations between programs. The constructions of product programs on control-flow graphs is closely related to cross-products, introduced by Zacks and Pnueli [40] for translation validation of compiler optimizations in unstructured languages.

In [4], we propose a new methodology that combines product program construction and program synthesis to automatically vectorize implementations for loops drawn from libraries such as the STL for C++ or the BCL for C#.

8. Related work

Semantics-based relational methods play a prominent role in building denotational models of typed programming languages, and in reasoning about a variety of notions, including correctness, equivalence, or representation independence. Relational logics and product programs provide a more syntactical counterpart to semantic-based relational methods, and they can be used for similar purposes. In this section, we briefly review some relevant related work.

Information flow, 2-safety and hyperproperties Andrews and Reitman [2] were among the first to propose an encoding of information flow policies in Hoare logic; however, their encoding requires extending the set of axioms of Hoare logic in order to account for security properties. Joshi and Leino [27] use weakest precondition calculi to provide a characterization of non-interference for a wide range of programming constructs.

Barthe, D'Argenio and Rezk [7] consider self-composition for several programming constructs and verification techniques, and prove that deductive verification of a self-composed program yields a relatively complete method to prove non-interference of programs. Independently, Darvas, Hähnle and Sands [19] apply self-composition and dynamic logic to verify information flow policies of imperative programs. Related approaches include relational decomposition of Beringer [14] and its predecessor [15].

Terauchi and Aiken [38] use self-composition to formulate a notion of relaxed non-interference, and propose a type-directed transformation, akin to product program construction, to construct a single program that emulates the behavior of the original program and of its copy, and that is easier to verify. In a nutshell, the type-directed transformation of programs does not self-compose branching statements depending on public variables, but rather uses synchronous programs, and performs copy propagation on self-composed assignments with low expressions to variables. Kovacs, Seidl and Finkbeiner [28] further refine this approach, again by providing a product program that is easier to verify.

While self-composition and product programs provide a means to verify 2-safety properties using standard verification on the product program, relational program logics such as [13] or [39] provide direct support for relational verification of two programs, and in particular they can be used to verify information flow policies. Other works propose dedicated verification mechanisms for verifying information flow policies; for instance, Amtoft et al. [1] propose a dedicated logic based on independence assertions for carrying information flow analysis of object-oriented programs.

In their aforementioned work [38], Terauchi and Aiken introduce k-safety properties, and observe that non-interference is a 2-safety property. Later, Clarkson and Schneider [18] introduce hyperproperties, and show that every hyperproperty can be decomposed into a hypersafety and a hyperliveness property. Formal verification of hyperproperties is considered by Milushev and Clarke [33] and by Clarkson et al. [17].

Relational program logics, product programs and differential verification Relational Hoare Logic (RHL) was proposed by Benton [13], who used it to verify non-interference and program equivalence; his formulation of RHL closely corresponds to the logic described in this paper, and is primarily focused on the verification of structurally equal programs. Yang [39] defines relational separation logic (RSL) and uses it for proving the equivalence between the Schorr–Waite graph marking algorithm and a depth-first search algorithm. Like RHL, RSL lacks the ability to cope with structurally dissimilar programs. Relational program logics have recently been extended to probabilistic programs, both in an imperative setting [11,12] and in a higher-order setting [8,10], and applied to reason about cryptographic constructions, differentially private computations, and mechanism design. Regression verification is another approach proposed by Godlin and Strichman [22] for proving equivalence between two programs.

Closely related methods have been used for proving correctness of program optimizations. In particular, Necula [34] develops a translation validation prototype based on GCC. The validation is defined in terms of a simulation relation, i.e., a connection between source and transformed program points, plus a set of constraints characterizing the states reached by the programs at each pair of program points. Establishing the connection between program points requires the compiler to provide additional information. Parametrized equivalence checking [29] extends Necula's approach to consonant optimizations by combining it with the Permute rule used by Pnueli and his co-workers for translation validation of loops [3,24,37,41].

The notion of product is widely used in labeled transition systems; however, there has been comparatively little work to define program products. Pnueli and Zack [40] introduce the notion of cross-products to reduce the relational verification of the original and transformed programs to the analysis of a single program. Product programs are also used in SymDiff [30,31], a relational verification tool which has been used for many purposes, such as cross-version compiler validation [26], verification modulo versions [32], and security validation of applications [25]. Other works such as [36] and [35] explore similar approaches based on symbolic execution or abstract interpretations, respectively.

9. Conclusion

We have carried a detailed comparison of the strengths of product programs and relational program logics, and outlined several of their applications, in the setting of a core imperative (or probabilistic) language. In the future, it would be interesting to extend the study of product programs and relational program logics for advanced language features, including procedures, exceptions, concurrency or higher-order functions, and for mainstream programming languages.

References

[1] Torben Amtoft, Sruthi Bandhakavi, Anindya Banerjee, A logic for information flow in object-oriented programs, in: J. Gregory Morrisett, Simon L. Peyton Jones (Eds.), Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06, ACM Press, 2006, pp. 91–102.

[2] Gregory R. Andrews, Richard P. Reitman, An axiomatic approach to information flow in programs, ACM Trans. Program. Lang. Syst. 2 (1) (1980) 56–76.

[3] Clark W. Barrett, Yi Fang, Benjmain Goldberg, Ying Hu, Amir Pnueli, Lenore D. Zuck, Tvoc: a translation validator for optimizing compilers, in: Kousha Etessami, Sriram K. Rajamani (Eds.), Computer Aided Verification, in: Lecture Notes in Computer Science, vol. 3576, Springer-Verlag, 2005, pp. 291–295.

[4] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, Mark Marron, From relational verification to SIMD loop synthesis, in: Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, Richard W. Vuduc (Eds.), Proceedings of 2013 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, ACM, 2013, pp. 123–134.

[5] Gilles Barthe, Juan Manuel Crespo, Cesar Kunz, Relational verification using product programs, in: Michael J. Butler, Wolfram Schulte (Eds.), Formal Methods, in: Lecture Notes in Computer Science, vol. 6664, Springer, 2011, pp. 200–214.

[6] Gilles Barthe, Juan Manuel Crespo, César Kunz, Beyond 2-safety: asymmetric product programs for relational program verification, in: Sergei N. Artëmov, Anil Nerode (Eds.), Logical Foundations of Computer Science, International Symposium, LFCS 2013, in: Lecture Notes in Computer Science, vol. 7734, Springer, 2013, pp. 29–43.

[7] Gilles Barthe, Pedro D'Argenio, Tamara Rezk, Secure information flow by self-composition, in: R. Foccardi (Ed.), Proceedings of 17th IEEE Computer Security Foundations Workshop, CSFW'04, IEEE Press, 2004, pp. 100–114.

[8] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, Santiago Zanella Béguelin, Probabilistic relational verification for cryptographic implementations, in: Suresh Jagannathan, Peter Sewell (Eds.), Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14, ACM, 2014, pp. 193–206.

[9] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, Pierre-Yves Strub, Proving differential privacy in hoare logic, in: Proceedings of 27th IEEE Computer Security Foundations Symposium, CSF 2014, IEEE, 2014, pp. 411–424.

[10] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, Pierre-Yves Strub, Higher-order approximate relational refinement types for mechanism design and differential privacy, in: Sriram K. Rajamani, David Walker (Eds.), Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, ACM, 2015, pp. 55–68.

[11] Gilles Barthe, Benjamin Grégoire, Santiafo Zanella Béguelin, Formal certification of code-based cryptographic proofs, in: Zhong Shao, Benjamin C. Pierce (Eds.), Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'09, ACM Press, 2009, pp. 90–101.

[12] Gilles Barthe, Boris Köpf, Federico Olmedo, Santiago Zanella Béguelin, Probabilistic relational reasoning for differential privacy, in: John Field, Michael Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'12, ACM, 2012, pp. 97–110.

[13] Nick Benton, Simple relational correctness proofs for static analyses and program transformations, in: Neil D. Jones, Xavier Leroy (Eds.), Proceedings of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'04, ACM Press, 2004, pp. 14–25.

[14] Lennart Beringer, Relational decomposition, in: Marko C.J.D. van Eekelen, Herman Geuvers, Julien Schmaltz, Freek Wiedijk (Eds.), Interactive Theorem Proving – Second International Conference, ITP'11, in: Lecture Notes in Computer Science, vol. 6898, Springer, 2011, pp. 39–54.

[15] Lennart Beringer, Martin Hofmann, Secure information flow and program logics, in: Proceedings of 20th IEEE Computer Security Foundations Symposium, CSF'07, IEEE Computer Society, 2007, pp. 233–248.

[16] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, Continuity analysis of programs, in: Manuel V. Hermenegildo, Jens Palsberg (Eds.), Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'10, 2010, pp. 57–70.

[17] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, César Sánchez, Temporal logics for hyperproperties, in: Martín Abadi, Steve Kremer (Eds.), Principles of Security and Trust – Third International Conference, POST 2014, in: Lecture Notes in Computer Science, vol. 8414, Springer, 2014, pp. 265–284.

[18] Michael R. Clarkson, Fred B. Schneider, Hyperproperties, in: Proceedings of CSF'08, 2008, pp. 51–65.
[19] Adam Darvas, Reiner Hähnle, David Sands, A theorem proving approach to analysis of secure information flow, in: D. Hutter, M. Ullmann (Eds.), Security in Pervasive Computing, in: Lecture Notes in Computer Science, vol. 3450, Springer, 2005, pp. 193–209, Preliminary version in the informal proceedings of WITS'03.
[20] Guillaume Dufay, Amy P. Felty, Stan Matwin, Privacy-sensitive information flow with JML, in: Robert Nieuwenhuis (Ed.), Automated Deduction – CADE-20, 20th International Conference on Automated Deduction, in: Lecture Notes in Computer Science, vol. 3632, Springer, 2005, pp. 116–130.
[21] Cynthia Dwork, Differential privacy, in: Michele Bugliesi, Bart Preneel, Vladimiro Sassone, Ingo Wegener (Eds.), Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, in: Lecture Notes in Computer Science, vol. 4052, Springer, 2006, pp. 1–12.
[22] Benny Godlin, Ofer Strichman, Regression verification: proving the equivalence of similar programs, Softw. Test. Verif. Reliab. 23 (3) (2013) 241–258.
[23] Joseph Goguen, Jose Meseguer, Security policies and security models, in: Proceedings of IEEE Symposium on Security and Privacy, IEEE Press, 1982, pp. 11–22.
[24] Benjamin Goldberg, Lenore D. Zuck, Clark W. Barrett, Into the loops: practical issues in translation validation for optimizing compilers, Electron. Notes Theor. Comput. Sci. 132 (1) (2005) 53–71.
[25] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, Brian Zill, Ironclad apps: end-to-end security via automated full-system verification, in: Jason Flinn, Hank Levy (Eds.), Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI'14, USENIX Association, 2014, pp. 165–181.
[26] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, Scott Wadsworth, Will you still compile me tomorrow? Static cross-version compiler validation, in: Bertrand Meyer, Luciano Baresi, Mira Mezini (Eds.), Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, ACM, 2013, pp. 191–201.
[27] Rajeev Joshi, K. Rustan, M. Leino, A semantic approach to secure information flow, Sci. Comput. Program. 37 (1–3) (2000) 113–138.
[28] Máté Kovács, Helmut Seidl, Bernd Finkbeiner, Relational abstract interpretation for the verification of 2-hypersafety properties, in: Ahmad-Reza Sadeghi, Virgil D. Gligor, Moti Yung (Eds.), Proceedings of 2013 ACM Conference on Computer and Communications Security, CCS'13, ACM, 2013, pp. 211–222.
[29] Sudipta Kundu, Zachary Tatlock, Sorin Lerner, Proving optimizations correct using parameterized program equivalence, in: Michael Hind, Amer Diwan (Eds.), Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'09, 2009, pp. 327–337.
[30] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, Henrique Rebêlo, SYMDIFF: a language-agnostic semantic diff tool for imperative programs, in: P. Madhusudan, Sanjit A. Seshia (Eds.), Computer Aided Verification – 24th International Conference, CAV 2012, in: Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 712–717.
[31] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, Chris Hawblitzel, Differential assertion checking, in: Bertrand Meyer, Luciano Baresi, Mira Mezini (Eds.), Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, ACM, 2013, pp. 345–355.
[32] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, Sam Blackshear, Verification modulo versions: towards usable verification, in: Michael F.P. O'Boyle, Keshav Pingali (Eds.), Proceedings of 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, ACM, 2014, p. 32.
[33] Dimiter Milushev, Dave Clarke, Incremental hyperproperty model checking via games, in: Hanne Riis Nielson, Dieter Gollmann (Eds.), Secure IT Systems – 18th Nordic Conference, NordSec 2013, in: Lecture Notes in Computer Science, vol. 8208, Springer, 2013, pp. 247–262.
[34] George C. Necula, Translation validation for an optimizing compiler, in: Monica S. Lam (Ed.), Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 2000, pp. 83–94.
[35] Nimrod Partush, Eran Yahav, Abstract semantic differencing via speculative correlation, in: Andrew P. Black, Todd D. Millstein (Eds.), Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'14, ACM, 2014, pp. 811–828.
[36] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, Corina S. Păsăreanu, Differential symbolic execution, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'08, ACM, 2008, pp. 226–237.
[37] Amir Pnueli, Michael Siegel, Eli Singerman, Translation validation, in: Bernhard Steffen (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 1384, Springer-Verlag, 1998, pp. 151–166.
[38] Tachio Terauchi, Alex Aiken, Secure information flow as a safety problem, in: Chris Hankin, Igor Siveroni (Eds.), Static Analysis Symposium, in: Lecture Notes in Computer Science, vol. 3672, Springer, 2005, pp. 352–367.
[39] Hongseok Yang, Relational separation logic, Theor. Comput. Sci. 375 (1–3) (2007) 308–334.
[40] Anna Zaks, Amir Pnueli, CoVaC: compiler validation by program analysis of the cross-product, in: Jorge Cuéllar, T.S.E. Maibaum, Kaisa Sere (Eds.), Formal Methods, in: Lecture Notes in Computer Science, vol. 5014, Springer, 2008, pp. 35–51.
[41] Lenore D. Zuck, Amir Pnueli, Benjamin Goldberg, VOC: a methodology for the translation validation of optimizing compilers, J. Univers. Comput. Sci. 9 (3) (2003) 223–247.