Unleashing relational program logics

Gilles Barthe Juan Manuel Crespo César Kunz

IMDEA Software
Madrid, Spain

Abstract

Relational program logics allow reasoning about two programs or two runs of the same program. They provide an elegant means to prove correctness of compiler optimizations or equivalence between two implementations of an abstract data type, and a unifying formalism for specifying and verifying properties like non-interference or determinism. Yet the current technology for relational verification remains underdeveloped.

We provide an abstract theory of product programs that supports a direct embedding of relational verification into standard verification. The embedding lays the theoretical foundations for leveraging the benefits of program verification to the relational setting, and overcomes the limitations of previous methods such as self-composition and Benton's relational Hoare logic. We use translation validation of loop optimizations as our main vector for illustrating the flexibility and expressivity of products.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors - Compilers; Optimization

General Terms Languages, Theory, Verification

Keywords Product program, relational Hoare logic, correctness, refinement, compiler optimization, translation validation

1. Introduction

Establishing formal relationships between programs is perhaps the most natural means to understand their behavior. In particular, notions of equivalence play a fundamental role in reasoning about programs: they allow capturing that two programs behave similarly, or that the same program behaves similarly on two different runs. Program equivalences are also instrumental in formulating notions of compiler correctness or representation independence [20]. Yet, program equivalences are often contextual. For instance, the correctness of compiler transformations is conditioned by the context in which they are performed: e.g. one can remove an assignment $x := e$ if x is not used anymore, or replace it by $x := n$ if the fragment of code that precedes the assignment ensures that e evaluates to n at the program point where the transformation is done. Such contextual equivalences are conveniently captured using relational judgments of the form

$$\{\varphi\}c_1 \sim c_2\{\psi\}$$

where the pre- and post-conditions φ and ψ are used to model the context: informally, the relational judgment is valid if terminating runs of c_1 and c_2 starting from states satisfying the pre-condition φ end in states satisfying the post-condition ψ.

Relational program logics [8, 10, 35] are general purpose logics that support reasoning about relational judgments. They have been used to validate instances of program optimizations [10] and to prove the correctness of a program by showing its equivalence with a simpler one; for example, Yang [35] uses relational separation logic to show the equivalence between depth-first search and the Schorr-Waite algorithm. More generally, relational logics provide an adequate framework to carry verifications that remain out of reach of traditional methods. For instance, Barthe, Grégoire, and Zanella [8] rely on probabilistic relational Hoare logic to formalize (sequences of) equivalences between probabilistic programs, as they arise in cryptographic proofs.

Fig. 1 presents a relational logic for a core imperative language. The logic is closely related to Benton's relational Hoare logic [10]; an obvious but inessential difference is that we follow Yang's convention of restricting judgments to pairs of commands manipulating disjoint sets of variables—we defer to Section 2.2 for a precise comparison between the two logics. The proof rules of Fig. 1 are simple, intuitive and sufficient for many purposes. On the other hand, they confine reasoning to structurally equal programs, because there is a single and symmetric rule per language construct. Although this restriction can be mitigated by adding one-sided rules, see Fig. 2, the logic still cannot reason about programs with very different structures. In particular, it cannot be used to validate the loop optimizations considered in translation validation [6, 27], or to prove representation independence for complex ADTs.

One can by-pass the limitations of the relational logic of Fig. 1 using self-composition [7, 16], which transforms every valid quadruple $\models \{\varphi\}c_1 \sim c_2\{\psi\}$ into a valid triple $\models \{\varphi\}\, c_1; c_2\, \{\psi\}$. Self-composition does not require programs to be structurally equivalent, and is (relatively) complete. Unfortunately, its theoretical appeal is not matched by its practicality [34], as it is often difficult to establish the validity of the triple $\{\varphi\}\, c_1; c_2\, \{\psi\}$.

In effect, current technology for relational program verification is underdeveloped: existing methods suffer from significant limitations, and are seldom implemented. The challenge addressed in this paper is to lay the foundations for bringing relational verification on par with traditional verification, for which many methods and tools are readily available. Concretely, we elaborate a theory of product programs, which allows transforming relational verification tasks into standard ones, without any of the setbacks of relational Hoare logics or self-composition.

[Copyright notice will appear here once 'preprint' option is removed.]

Commands and basic blocks: x ranges over a set $\mathcal{V}$ of variables, e and b range over integer and boolean expressions

$$\begin{array}{rl} \text{commands} & c ::= x := e \mid \mathsf{skip} \mid c; c \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \\ \text{basic blocks} & i ::= x := e \mid \mathsf{skip} \mid i; i \end{array}$$

Judgments: φ, ψ are first-order formulae and c_1, c_2 are commands whose sets of variables $\mathsf{var}(c_1)$ and $\mathsf{var}(c_2)$ are disjoint

$$\{\varphi\} c_1 \sim c_2 \{\psi\}$$

Rules:

$$\frac{i_1, i_2 \text{ basic blocks} \qquad \vdash \{\varphi\}\, i_1; i_2\, \{\psi\}}{\vdash \{\varphi\} i_1 \sim i_2 \{\psi\}} \text{ (R-INSTR)} \qquad \frac{\vdash \{\phi\} c_1 \sim c_2 \{\psi\}}{\vdash \{\phi\} c_2 \sim c_1 \{\psi\}} \text{ (R-TRANSP)}$$

$$\frac{\vdash \{\varphi\} c_1 \sim c_2 \{\phi\} \qquad \vdash \{\phi\} d_1 \sim d_2 \{\psi\}}{\vdash \{\varphi\} c_1; d_1 \sim c_2; d_2 \{\psi\}} \text{ (R-SEQ)} \qquad \frac{\vdash \{\varphi \wedge b_1 \wedge b_2\} c_1 \sim c_2 \{\psi\} \qquad \vdash \{\varphi \wedge \neg b_1 \wedge \neg b_2\} d_1 \sim d_2 \{\psi\}}{\vdash \{\varphi \wedge (b_1 = b_2)\} \mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ d_1 \sim \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ d_2 \{\psi\}} \text{ (R-IF)}$$

$$\frac{\vdash \{\varphi\} c_1 \sim c_2 \{\psi\} \qquad \varphi' \Rightarrow \varphi \qquad \psi \Rightarrow \psi'}{\vdash \{\varphi'\} c_1 \sim c_2 \{\psi'\}} \text{ (R-SUB)} \qquad \frac{\vdash \{\phi \wedge b_1 \wedge b_2\} c_1 \sim c_2 \{\phi \wedge b_1 = b_2\}}{\vdash \{\phi \wedge b_1 = b_2\} \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ b_2\ \mathsf{do}\ c_2 \{\phi \wedge \neg b_1 \wedge \neg b_2\}} \text{ (R-WHILE)}$$

$$\frac{\vdash \{\phi\} c_1 \sim c_2 \{\psi\} \qquad \vdash \{\phi'\} c_2 \sim c_3 \{\psi'\} \qquad c_2 \text{ terminates} \qquad \mathsf{var}(c_2) \subseteq X_2}{\vdash \{\exists X_2.\, \phi \wedge \phi'\} c_1 \sim c_3 \{\exists X_2.\, \psi \wedge \psi'\}} \text{ (R-COMPOSE)}$$

Fig. 1. Termination-insensitive relational Hoare logic

$$\frac{\vdash \{\varphi \wedge b_1\} c_1 \sim e_2 \{\psi\} \qquad \vdash \{\varphi \wedge \neg b_1\} d_1 \sim e_2 \{\psi\}}{\vdash \{\varphi\} \mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ d_1 \sim e_2 \{\psi\}} \text{ (R-IF-LEFT)} \qquad \frac{\vdash \{\varphi\} \mathsf{if}\ b_1\ \mathsf{then}\ (c_1; \mathsf{while}\ b_1\ \mathsf{do}\ c_1)\ \mathsf{else}\ \mathsf{skip} \sim c_2 \{\psi\}}{\vdash \{\varphi\} \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim c_2 \{\psi\}} \text{ (R-UNR)}$$

Fig. 2. Selected 1-sided rules

Our starting point (Section 2) is the striking similarity between the proof systems of Hoare logic and basic relational Hoare logic. We exploit the 1-1 correspondence between their rules to map derivations of Hoare quadruples $\vdash \{\varphi\} c_1 \sim c_2 \{\psi\}$ into derivations of Hoare triples $\vdash \{\varphi\}\, c\, \{\psi\}$, where the program c, called the synchronized product of c_1 and c_2, behaves as their sequential composition but executes instructions of the two programs in lockstep. The ability to perform synchronous steps, i.e. steps that execute instructions from c_1 and c_2 simultaneously, is the key to making the verification of c significantly easier than the verification of the program $c_1; c_2$ obtained by self-composition.

Synchronized products are an instance of a more general product construction. Section 3 defines the product between two programs c_1 and c_2 as a program c which combines synchronous steps, in which instructions from c_1 and c_2 are executed in lockstep, with asynchronous steps, in which instructions from c_1 or c_2 are executed separately. Thanks to the possibility of performing asynchronous steps, the product construction recovers the flexibility and generality of self-composition, and is applicable to programs with different control structures. We show that, under some fullness conditions, product programs correctly capture the semantics of their constituents, and enjoy closure properties that support compositional reasoning.

Section 4 characterizes the fullness of a product logically, and shows that relational program verification can be reduced to finding a product program that satisfies the specification. In particular, our main result (Theorem 1) provides a method to verify a relational judgment by computing a set of proof obligations and sending them to a prover. Remarkably, a mild modification of the method allows to prove refinement properties by similar means (Theorem 2).

Section 5 shows how product programs support relational translation validation of loop optimizations [6], including static caching [23], a challenging optimization used for incremental computation e.g. in image processing or computational geometry.

In summary, the main contributions of the paper are: i) a general theory of product programs; ii) a method to reduce relational program verification to program verification; iii) an illustration of our methods for translation validation.

2. Relational logics

There is a striking similarity between the proof rules of Hoare logic and those of relational Hoare logic. Yet it appears that no formal connection has been established between their derivations. This section presents a remarkably simple embedding from derivations in relational Hoare logics to derivations in Hoare logics. The embedding relies on the ability to construct for any two programs c_1 and c_2 s.t. $\vdash \{\varphi\} c_1 \sim c_2 \{\psi\}$ their synchronized product program c which performs the execution steps of c_1 and c_2 in lockstep and verifies $\vdash \{\varphi\}\, c\, \{\psi\}$. In preparation for the general definition of product, we further provide a more liberal construction allowing products to perform execution steps of the programs c_1 and c_2 asynchronously, and extend the definition of products to programs with non-deterministic choice operator. Finally, we discuss some applications of products to information flow and briefly comment on the connections with relational separation logic [35].

2.1 Hoare logic

Commands are written in the language of Fig. 1; states are partial functions from variables to integer values, i.e. $\mathcal{S} = \mathcal{V} \rightharpoonup \mathbb{Z}$. The semantics of a command c is modeled by a relation $[\![c]\!] \subseteq \mathcal{S} \times \mathcal{S}$, defined in the usual way from the semantics of expressions, as given by indexed relations $([\![a]\!])_{a \in \mathsf{AExp}} \subseteq \mathcal{S} \times \mathbb{Z}$, and $([\![b]\!])_{b \in \mathsf{BExp}} \subseteq \mathcal{S}$. Program correctness is expressed by triples of the form $\{\varphi\}\, c\, \{\psi\}$, where c is a command, and φ and ψ first-order formulae with variables in $\mathcal{V}$. We let $[\![\phi]\!]$ denote the set of states satisfying ϕ.

Definition 1 (Valid triple). $\{\varphi\} c \{\psi\}$ is valid, written $\vDash \{\varphi\}\, c\, \{\psi\}$, iff for all states $\sigma, \sigma' \in \mathcal{S}$, $\sigma \in [\![\varphi]\!]$ and $(\sigma, \sigma') \in [\![c]\!]$ imply $\sigma' \in [\![\psi]\!]$.

Triples can be proved valid using the rules of Hoare logic (Fig. 3).

$$\frac{}{\vdash \{\phi\}\, \mathsf{skip}\, \{\phi\}}\ (\text{SKIP}) \qquad \frac{}{\vdash \{\phi[e/x]\}\, x := e\, \{\phi\}}\ (\text{ASSIGN})$$

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

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

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

$$\frac{\vdash \{\varphi\}\, c\, \{\psi\} \qquad \varphi' \Rightarrow \varphi \qquad \psi \Rightarrow \psi'}{\vdash \{\varphi'\}\, c\, \{\psi'\}}\ (\text{SUB})$$

Fig. 3. Hoare logic

2.2 Relational Hoare logic

Relational Hoare logic is a variant of Hoare logic to reason about two programs. Its judgments are quadruples $\{\varphi\}c_1 \sim c_2\{\psi\}$, where c_1 and c_2 are separable commands, i.e. commands that do not have variables in common, and φ and ψ are assertions. In contrast to [10, 35], our definition of validity is termination-insensitive, i.e. does not require that the commands co-terminate.

Definition 2 (Valid quadruple). $\{\varphi\}c_1 \sim c_2\{\psi\}$ is valid, written $\models \{\varphi\}c_1 \sim c_2\{\psi\}$, iff for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2' \in \mathcal{S}$, $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$, and $(\sigma_1, \sigma_1') \in [\![c_1]\!]$, and $(\sigma_2, \sigma_2') \in [\![c_2]\!]$ imply $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.

The rules of relational Hoare logic appear in Fig. 1. All rules, except (R-INSTR) and (R-COMPOSE), are standard and self-explanatory. The rule (R-INSTR) allows relating basic blocks of different length, and by extension commands that have the same control flow but differ in their basic blocks, and subsumes the rules for assignment (R-ASSIGN) and skip (R-SKIP)

$$\frac{}{\vdash \{\phi\}\mathsf{skip} \sim \mathsf{skip}\{\phi\}}\ (\text{R-SKIP})$$

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

The rule (R-COMPOSE) allows intermediate commands in reasonings. The assertion $\exists X_2.\ \phi \wedge \phi'$ existentially quantifies over the variables of c_2, and simulates the composition of relations in the setting of assertions. Note that soundness of (R-COMPOSE) requires that c_2 terminates as $\models \{\varphi\}c_1 \sim \mathsf{while\ true\ do\ skip}\{\psi\}$ is valid for every command c_1 and assertions φ and ψ. Relational Hoare logic is sound.

Lemma 1. If $\vdash \{\varphi\}c_1 {\sim} c_2\{\psi\}$ is derivable, then $\models \{\varphi\}c_1 {\sim} c_2\{\psi\}$.

Our variant of relational Hoare logic is closely related to Benton's relational Hoare logic [10]. The most obvious, but inessential difference, is that Benton considers 2-assertions, and relies on tagged expressions $e\langle 1\rangle$ and $e\langle 2\rangle$ to denote the interpretation of e in the first and second memory respectively. One can define a variant of our logic that uses 2-assertions, and give derivability-preserving mappings between the two logics. A more fundamental difference is that Benton's notion of valid judgment requires co-termination between c_1 and c_2. Since none of the two logics involve a sophisticated termination analysis, there is a trade-off between requiring commands to be structurally equal, or abandoning co-termination. While Benton takes the first alternative, we opt for the second. This is reflected in structural rules: Benton's transitivity rule, which specializes our composition rule to partial equivalence relations, does not impose any termination check on the intermediate command.

2.3 Synchronized and loose products

The embedding of relational Hoare logic into Hoare logic is based on the notion of synchronized product of two commands. Synchronized products are closely related to cross-products, introduced by Zacks and Pnueli [37] for translation validation of structure-preserving optimizations.

Definition 3 (Synchronized product). The synchronized product $c_1 \otimes c_2$ of two commands c_1 and c_2 is, if it exists, any command c s.t. $c_1 \sim c_2 \longrightarrow c$ is derivable using the rules of Fig. 4.

Synchronized products are unique up to reordering of instructions in basic blocks. Moreover, the synchronized product of two provably related commands is always defined.

Proposition 1. If $\vdash \{\varphi\}c_1 \sim c_2\{\psi\}$ then $c_1 \otimes c_2$ is defined and moreover $\vdash \{\varphi\}\, c_1 \otimes c_2\, \{\psi\}$.

Example 1 (Loop reversal). Loop reversal is a transformation that reverses the order of iterations in a loop. Consider the commands c_1, its loop reversal c_2, and their synchronized product c

$$\begin{aligned} c_1 &\doteq \mathsf{while}\ (x \le N)\ \mathsf{do}\ y \mathrel{+}= x; x\text{++} \\ c_2 &\doteq \mathsf{while}\ (0 < x')\ \mathsf{do}\ y' \mathrel{+}= x'; x'\text{--} \\ c &\doteq \mathsf{while}\ (x \le N)\ \mathsf{do}\ y \mathrel{+}= x;\ y' \mathrel{+}= x';\ x\text{++};\ x'\text{--} \end{aligned}$$

We have $c_1 \sim c_2 \longrightarrow c$. Moreover, $\vdash \{\varphi\}c_1 \sim c_2\{y = y'\}$, and hence $\vdash \{\varphi\}\, c\, \{y = y'\}$, where φ is $x = 1 \wedge x' = N \wedge y = y'$. The proof uses the non-linear invariant

$$0 \le x' \wedge x + x' = N + 1 \wedge y' - y = x(N + 2 - x) - (N + 1)$$

The embedding is blatantly incomplete, since the (R-WHILE) rule enforces that related while loops perform the same number of iterations. A possible solution to accommodate commands that are not structurally equal is to extend the construction of the synchronized product with a self-composition rule that allows transforming c_1 and c_2 into $c_1; c_2$. The resulting proof system is shown in Fig. 5. Note that, in absence of a general criterion for deciding on the application of the self-composition rule, it is more effective to interleave program verification and product construction—in some cases, such as information flow, one can perform the product construction and verify the program a posteriori, see Section 2.5.

Proposition 2. If $\{\varphi\}\, c_1 \sim c_2 \longrightarrow c\, \{\psi\}$ then $\vdash \{\varphi\}\, c\, \{\psi\}$ and $\models \{\varphi\}c_1 \sim c_2\{\psi\}$.

Section 4 reduces relational verification to program verification, by providing a converse to Propositions 1 and 2.

2.4 Non-determinism

Non-deterministic operators introduce an intermediate layer between specifications and implementations, and allow developers to build their programs by successive refinements, see e.g. [24]. This section sketches how product constructions can be extended to accommodate refinement and non-deterministic operators.

We focus on non-deterministic choice and possibilistic equivalence; however our results can be adapted to non-deterministic assignments and refinement. The non-deterministic choice operator $\oplus$ takes two commands c_1 and c_2 and executes one of them, i.e. $[\![c_1 \oplus c_2]\!] = [\![c_1]\!] \cup [\![c_2]\!]$. Moreover, we say that a program c_2 refines c_1 w.r.t. pre- and post-conditions φ and ψ, written $\{\varphi\}c_1 \mapsto c_2\{\psi\}$ iff for every $\sigma_1, \sigma_2, \sigma_1' \in \mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and $(\sigma_1, \sigma_1') \in [\![c_1]\!]$, there exists $\sigma_2' \in \mathcal{S}$ s.t. $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$ and $(\sigma_2, \sigma_2') \in [\![c_2]\!]$. Then, c_1 and c_2 are possibilistically equivalent w.r.t. pre- and post-conditions φ and ψ, written $\{\varphi\}c_1 \simeq c_2\{\psi\}$, iff $\{\varphi\}c_1 \mapsto c_2\{\psi\}$ and $\{\varphi\}c_2 \mapsto c_1\{\psi\}$. Finally, in order to reason about possibilistic equivalence, we introduce some definitions about program termination: a command c terminates (resp. strongly terminates) from

initial state σ, written $(c,\sigma)\Downarrow$ (resp. $(c,\sigma)\Downarrow^\star$), iff there exists $\sigma'\in\mathcal{S}$ s.t. $(\sigma,\sigma')\in[\![c]\!]$ (resp. if every partial execution of c can be completed to termination; a formal definition using small-step operational semantics is given in Section 3.1). Moreover, we say that c_1 and c_2 co-terminate w.r.t. φ iff i) for every $\sigma_1\in\mathcal{S}$ s.t. $(c_1,\sigma_1)\Downarrow$, there exists $\sigma_2\in\mathcal{S}$ s.t. $\sigma_1\uplus\sigma_2\in[\![\varphi]\!]$ and $(c_2,\sigma_2)\Downarrow^\star$; ii) for every $\sigma_2\in\mathcal{S}$ s.t. $(c_2,\sigma_2)\Downarrow$, there exists $\sigma_1\in\mathcal{S}$ s.t. $\sigma_1\uplus\sigma_2\in[\![\varphi]\!]$ and $(c_1,\sigma_1)\Downarrow^\star$.

Proposition 3. If $\{\varphi\}\,c_1\sim c_2\longrightarrow c\,\{\psi\}$ is derivable using the rules of Fig. 5 and

$$\frac{\{\phi\}\,c_1\sim c_2\longrightarrow c\,\{\psi\}\qquad\{\phi\}\,c_1'\sim c_2'\longrightarrow c'\,\{\psi\}}{\{\phi\}\,c_1\oplus c_1'\sim c_2\oplus c_2'\longrightarrow c\oplus c'\,\{\psi\}}\ \text{(LP-NONDET)}$$

and c_1 and c_2 co-terminate w.r.t. φ then $\{\varphi\}c_1\simeq c_2\{\psi\}$.

As expected, the set of executions of a product does not include all interleavings of its constituents. As a simplistic example, take

$$c_1\doteq x:=1\oplus x:=2\qquad c_2\doteq y:=1\oplus y:=2$$
$$c\doteq(x:=1;y:=1)\oplus(x:=2;y:=2)$$

The command c verifies $\{\mathsf{true}\}\,c_1\sim c_2\longrightarrow c\,\{x=y\}$; in particular, not all executions of c_1 and c_2 are modeled by executions of c. For instance, no run of c terminates with $x=1$ and $y=2$.

2.5 Information flow

Information flow policies [18, 29] are end-to-end confidentiality policies that guarantee absence of illicit information leakage, and prime instances of 2-properties, i.e. properties over two runs of a program [14, 34]. A large body of work on relational methods originates from information flow, or uses it as an application. It is therefore instructive to revisit the results of this section from the perspective of information flow. For our purpose, it is sufficient to focus on (termination-insensitive) non-interference [18], a baseline information flow policy ensuring that terminating runs of a program preserve equivalence between low parts of the memory. A policy is formalized by assigning to each variable a security level, secret H or public L; each policy yields an assertion LowEq

$$x_1=x_1'\wedge\ldots\wedge x_n=x_n'$$

where $x_1\ldots x_n$ are the public variables and $x_1'\ldots x_n'$ are a renaming of $x_1\ldots x_n$. Informally, LowEq states that that a memory coincides with its renaming in their low part. Let c be a command and let c' be a renaming of c obtained by replacing $x_1\ldots x_n$ by $x_1'\ldots x_n'$. c is non-interfering iff for every states $\sigma,\sigma'\in\mathcal{S}$, if $\sigma\in[\![\mathsf{LowEq}]\!]$ and $(\sigma,\sigma')\in[\![c]\!]$, then $\sigma'\in[\![\mathsf{LowEq}]\!]$.

A common means to enforce non-interference is using a type system, see Fig. 6. The type system is parametrized by a mapping Γ from variables to security levels, and manipulates judgments of the form $\vdash c:\tau\ \mathbf{cmd}$ stating that c is non-interfering and only writes on variables with security level greater or equal to τ, and relies on the auxiliary judgment $\vdash e:\tau$, stating that all variables in e have security level smaller or equal to τ. However, it has been observed that information flow type systems are overly conservative, and that one can alternatively characterize non-interference using relational Hoare logic, Hoare logic, and loose products. For example, non-interference is formalized by the relational judgment

$$\{\mathsf{LowEq}\}c\sim c'\{\mathsf{LowEq}\}$$

However, relational Hoare logic is too weak for proving such judgments valid. Specifically, it cannot handle commands that branch on high expressions; in order to embed a (termination-sensitive) information flow type system [31], Benton adds a rule for dead code

$$\frac{\mathsf{mod}(c)\cap\mathsf{var}(\phi)=\emptyset}{\vdash\{\phi\}c\sim\mathsf{skip}\{\phi\}}\ \text{(R-DEADCODE)}$$

where $\mathsf{mod}(c)$ denotes the set of variables modified by c and $\mathsf{var}(\phi)$ denotes the set of free variables of ϕ. Self-composition [7, 16] provides a sound and complete means to capture non-interference in the traditional setting of program verification, as the judgment

$$\{\mathsf{LowEq}\}\,c;c'\,\{\mathsf{LowEq}\}$$

Terauchi and Aiken [34] observe that the triples induced by self-composition are hard to verify by automatic tools. Finally, one can formalize non-interference using loose products: c is non-interfering iff there exists a command c'' s.t.

$$\{\mathsf{LowEq}\}\,c\sim c'\longrightarrow c''\,\{\mathsf{LowEq}\}$$

This characterization relies on the loose product construction being, up to minor differences, a generalization of the type-directed transformation introduced by Terauchi and Aiken [34], and further developed by Naumann [25], to overcome the limits of self-composition. In a nutshell, their transformation relies on a simple static analysis of guards to decide whether or not to apply self-composition: in case the guard is high, it uses self-composition; otherwise, it uses the synchronized product. Essentially, their transformation consists in applying the loose product construction with LowEq used as the sole assertion. We illustrate this claim through a simple example. Let c be the non-interfering program ($\mathsf{while}\ x\leq N\ \mathsf{do}\ x$++)$;y:=1$, and let c' be its renaming. If x is high, then the guard of the loop is high, and the transformation yields $c;c'$. Similarly, the rule for while in Fig. 5 does not apply, so the only product we can build is $c;c'$. If, on the other hand, x is low, then the guard is low, and the transformation yields ($\mathsf{while}\ x\leq N\ \mathsf{do}\ x$++$;x'$++)$;y:=1;y':=1$. Similarly, the rule for while in Fig. 5 now applies and yields the same result.

To conclude, we briefly comment on extending the characterizations of non-interference to non-determinism: a command c is possibilistically non-interfering if for every states $\sigma_1,\sigma_1',\sigma_2\in\mathcal{S}$ s.t. $(\sigma_1,\sigma_1')\in[\![\mathsf{LowEq}]\!]$ and $(\sigma_1,\sigma_1')\in[\![c]\!]$, there exists $\sigma_2'\in\mathcal{S}$ s.t. $\sigma_1'\uplus\sigma_2'\in[\![\mathsf{LowEq}]\!]$ and $(\sigma_2,\sigma_2')\in[\![c']\!]$. Neither Hoare triples nor relational Hoare quadruples can be used to model possibilistic non-interference. On the other hand, the product construction allows to capture possibilistic non-interference. Proposition 3 implies that if c and c' co-terminate w.r.t. LowEq (recall that c' is a renaming of c) and $\{\mathsf{LowEq}\}\,c\sim c'\longrightarrow c''\,\{\mathsf{LowEq}\}$ for some c'', then c is possibilistically non-interfering. While the method is incomplete, it is sufficient to embed the information flow type system in Fig. 6—note that one can enforce co-termination by restricting the typing rule to low loops.

Proposition 4. If $\vdash c:\tau\ \mathbf{cmd}$, then there exists c'' s.t.

$$\{\mathsf{LowEq}\}\,c\sim c'\longrightarrow c''\,\{\mathsf{LowEq}\}$$

2.6 Remarks on relational separation logic

One challenging application of relational logics is to show the equivalence between implementations of algorithms or abstract data types. Yang [35] makes a first step in this direction by developing a relational separation logic, and showing that the Schorr-Waite algorithm and a depth-first algorithm are equivalent. Applications of products to separation logic and to proving representation independence of ADTs is left for future work. It is however instructive to highlight some issues in applying products to separation logic.

The product constructions extend to the programming model of separation logic, extending mutation, lookup, (non-deterministic) allocation and deallocation. Propositions 1 and 2 remain valid provided the relational derivation and loose product construction only use assertions of separation logic.

However, relational separation logic features an assertion Same, stating equality of heaps, and without direct translation into separation logic. The embeddings do not extend to specifications and

$$\frac{c_1 \sim c_2 \longrightarrow c}{\mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ b_2\ \mathsf{do}\ c_2 \longrightarrow \mathsf{while}\ b_1\ \mathsf{do}\ c}\ \text{(SP-WHILE)} \qquad \frac{c_1 \sim c_2 \longrightarrow c \qquad c_1' \sim c_2' \longrightarrow c'}{c_1; c_1' \sim c_2; c_2' \longrightarrow c; c'}\ \text{(SP-SEQ)}$$

$$\frac{c_1 \sim c_2 \longrightarrow c \qquad d_1 \sim d_2 \longrightarrow d}{\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ d_1 \sim \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ d_2 \longrightarrow \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ d}\ \text{(SP-IF)} \qquad \frac{I_1, I_2 \text{ basic blocks}}{I_1 \sim I_2 \longrightarrow I_1; I_2}\ \text{(SP-INSTR)}$$

Fig. 4. Synchronized product

$$\frac{\phi \Rightarrow b_1 = b_2 \qquad \{\phi \wedge b_1\}\, c_1 \sim c_2 \longrightarrow c\, \{\phi\}}{\{\phi\}\ \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ b_2\ \mathsf{do}\ c_2 \longrightarrow \mathsf{while}\ b_1\ \mathsf{do}\ c\, \{\phi \wedge \neg b_1\}}\ \text{(LP-WHILE)} \qquad \frac{\vdash \{\varphi\}\, c_1; c_2\, \{\psi\}}{\{\varphi\}\, c_1 \sim c_2 \longrightarrow c_1; c_2\, \{\psi\}}\ \text{(LP-SELFCOMP)}$$

$$\frac{\{\phi\}\, c_1 \sim c_2 \longrightarrow c\, \{\varphi\} \qquad \{\varphi\}\, c_1' \sim c_2' \longrightarrow c'\, \{\psi\}}{\{\phi\}\, c_1; c_1' \sim c_2; c_2' \longrightarrow c; c'\, \{\psi\}}\ \text{(LP-SEQ)} \qquad \frac{\{\phi'\}\, c_1 \sim c_2 \longrightarrow c\, \{\psi'\} \qquad \phi \Rightarrow \phi' \qquad \psi' \Rightarrow \psi}{\{\phi\}\, c_1 \sim c_2 \longrightarrow c\, \{\psi\}}\ \text{(LP-SUB)}$$

$$\frac{\phi \Rightarrow b_1 = b_2 \qquad \{\phi \wedge b_1\}\, c_1 \sim c_2 \longrightarrow c\, \{\psi\} \qquad \{\phi \wedge \neg b_1\}\, d_1 \sim d_2 \longrightarrow d\, \{\psi\}}{\{\phi\}\ \mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ d_1 \sim \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ d_2 \longrightarrow \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ d\, \{\psi\}}\ \text{(LP-IF)}$$

Fig. 5. Loose products

$$\text{(NI-ASSG)}\ \frac{\vdash e : \tau \qquad \tau \leq \Gamma(x)}{\vdash x := e : \tau\ \mathbf{cmd}} \qquad \text{(NI-SEQ)}\ \frac{\vdash c_1 : \tau\ \mathbf{cmd} \qquad \vdash c_2 : \tau\ \mathbf{cmd}}{\vdash c_1; c_2 : \tau\ \mathbf{cmd}}$$

$$\text{(NI-IF)}\ \frac{\vdash c_1 : \tau\ \mathbf{cmd} \qquad \vdash c_2 : \tau\ \mathbf{cmd} \qquad \vdash b : \tau}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \tau\ \mathbf{cmd}} \qquad \text{(NI-WHILE)}\ \frac{\vdash c : \tau\ \mathbf{cmd} \qquad \vdash b : \tau}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c : \tau\ \mathbf{cmd}}$$

$$\text{(NI-NONDET)}\ \frac{\vdash c_1 : \tau\ \mathbf{cmd} \qquad \vdash c_2 : \tau\ \mathbf{cmd}}{\vdash c_1 \oplus c_2 : \tau\ \mathbf{cmd}} \qquad \text{(NI-SUB)}\ \frac{\vdash c : \tau\ \mathbf{cmd} \qquad \vdash \tau' \leq \tau}{\vdash c : \tau'\ \mathbf{cmd}}$$

Fig. 6. Information flow type system

proofs that use Same, but for specific examples it may be possible to carry proofs using specialized, inductively defined instances of Same that have an interpretation in separation logic. Moreover, allocation raises many issues. First, its relational proof rule (derived from basic blocks) is too weak e.g. for information flow. Besides, it breaks separability, which is central in our development. One partial—and inelegant—solution is to constrain the behavior of allocation to recover separability, and to check that the validity of the judgment at hand remains unaffected.

3. An abstract theory of product programs

The constructions of Section 2 are instances of an abstract theory of products. We define a general notion of product program that it is independent from any programming language and does not require programs to be structurally equal. Then, we prove that under mild conditions product programs faithfully emulate the behavior of their constituents. In particular, we prove that the traces of the product of two programs are interleavings of their own traces.

3.1 Programs

We adopt a graph-based representation of programs. The representation confers additional flexibility to the theory of products, enabling the description of partially synchronized loops, i.e., loops in which only a strict subset of the iterations can executed synchronously, and also allowing to define products that have no structured counterpart—as for the left product for loop range splitting in Fig. 8(d). Formally, programs are modeled as labeled directed graphs. Nodes correspond to program points, and include an initial and a final nodes; for simplicity, we assume their unicity. Edges model the execution flow of the program, and are labeled with statements drawn from a set Stmt. The semantics of statements is given by a function $[\![.]\!] : \mathsf{Stmt} \to \mathcal{P}(\mathcal{S} \times \mathcal{S})$, where $\mathcal{S}$ is a set of states. In contrast to the previous section, both the sets of states and statements are left unspecified. Likewise, we assume given a set Assn of assertions closed under implication and disjunction.

Definition 4 (Program). A program P is a tuple $\langle \mathcal{N}, \mathcal{E}, G\rangle$, where $\langle \mathcal{N}, \mathcal{E}\rangle$ is a directed graph with unique source $\mathsf{in} \in \mathcal{N}$ and sink $\mathsf{out} \in \mathcal{N}$, and $G : \mathcal{E} \to \mathsf{Stmt}$ maps edges to statements.

Example 2 (Compiling commands). Commands can be compiled to a graph-based representation using as set Stmt of statements

$$x := e \mid \mathsf{skip} \mid c; c \mid \{b\}$$

where $\{b\}$ asserts b, i.e. $(\sigma, \sigma') \in [\![\{b\}]\!]$ iff $\sigma = \sigma'$ and $\sigma \in [\![b]\!]$. Fig. 8(a) provides a graph-based representation of the program in Fig. 7(a). Note that basic blocks are conflated into a single edge.

Next, we extend the notion of separable commands to programs; our conditions are inspired from self-composition [7], and are reminiscent of the monotonicity and frame properties of separation logic [36]. Assume given two functions $\pi_1, \pi_2 : \mathcal{S} \to \mathcal{S}$ s.t. for all $\sigma, \sigma' \in \mathcal{S}$, $\sigma = \sigma'$ iff $\pi_1(\sigma) = \pi_1(\sigma')$ and $\pi_2(\sigma) = \pi_2(\sigma')$. Given two states $\sigma_1, \sigma_2 \in \mathcal{S}$, we define $\sigma_1 \uplus \sigma_2 \in \mathcal{S}$ to be the unique, if it exists, state σ s.t. $\pi_1(\sigma) = \sigma_1$ and $\pi_2(\sigma) = \sigma_2$.

Definition 5 (Separable statements). A statement c is a left statement iff for all $\sigma_1, \sigma_2 \in \mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2$ is defined:

1. for all $\sigma_1' \in \mathcal{S}$, if $(\sigma_1, \sigma_1') \in [\![c]\!]$, then $(\sigma_1 \uplus \sigma_2, \sigma_1' \uplus \sigma_2) \in [\![c]\!]$ and in particular $\sigma_1' \uplus \sigma_2$ is defined;
2. for all $\sigma' \in \mathcal{S}$, if $(\sigma_1 \uplus \sigma_2, \sigma') \in [\![c]\!]$, then there exists $\sigma_1' \in \mathcal{S}$ s.t. $(\sigma_1, \sigma_1') \in [\![c]\!]$ and $\sigma_1' \uplus \sigma_2 = \sigma'$.

Right statements are defined symmetrically. Two statements c_1 and c_2 are separable iff c_1 is a left statement and c_2 is a right statement. Finally, two programs P_1 and P_2 are separable iff P_1 is a left program, i.e. it only contains left statements, and P_2 is a right program, i.e. it only contains right statements.

We next introduce some convenient notation and terminology; all definitions implicitly depend on a program $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$. We write $[\![\langle l, l' \rangle]\!]$ instead of $[\![G\langle l, l' \rangle]\!]$, and $\langle l, \sigma \rangle \rightsquigarrow \langle l', \sigma' \rangle$ instead of $(\sigma, \sigma') \in [\![\langle l, l' \rangle]\!]$; we define $\rightsquigarrow^\star$ as the reflexive transitive closure of $\rightsquigarrow$. Moreover, we write $\langle l, \sigma \rangle \rightsquigarrow l'$ if there exists a state σ' s.t. $\langle l, \sigma \rangle \rightsquigarrow \langle l', \sigma' \rangle$, and $\langle l, \sigma \rangle \rightsquigarrow^\star l'$ if there exists a state σ' s.t. $\langle l, \sigma \rangle \rightsquigarrow^\star \langle l', \sigma' \rangle$. A trace is a sequence of configurations $\langle l, \sigma \rangle \in \mathcal{N} \times \mathcal{S}$ s.t. $(\sigma, \sigma') \in [\![\langle l, l' \rangle]\!]$ for any two consecutive elements $\langle l, \sigma \rangle$ and $\langle l', \sigma' \rangle$ of the sequence. Then, we define a run as a trace starting in an initial configuration $\langle \mathsf{in}, \sigma \rangle$ and ending in a final configuration $\langle \mathsf{out}, \sigma' \rangle$, and a φ-run as a run starting in a configuration $\langle \mathsf{in}, \sigma \rangle$ s.t. $\sigma \in [\![\varphi]\!]$. Finally, some of our results make assumptions on the termination of programs. We say that P terminates from an initial state $\langle l, \sigma \rangle$, written $\langle l, \sigma \rangle \Downarrow$, if there exists a run starting from $\langle l, \sigma \rangle$, i.e. $\langle l, \sigma \rangle \rightsquigarrow^\star \mathsf{out}$, and that P strongly terminates from an initial state $\langle l, \sigma \rangle$, written $\langle l, \sigma \rangle \Downarrow^\star$, if every trace starting from $\langle l, \sigma \rangle$ can be completed into a run, i.e. $\langle l', \sigma' \rangle \Downarrow$ for all $l' \in \mathcal{N}$ and $\sigma' \in \mathcal{S}$ s.t. $\langle l', \sigma \rangle \rightsquigarrow^\star \langle l', \sigma' \rangle$.

3.2 Left products

Informally, a product of two programs is a program that combines their effects. We begin with a weaker definition (Definition 6) which only guarantees that the behavior of products is included in the behavior of their constituents. Then, we provide a sufficient condition (Definition 7) for the behavior of products to coincide with the behavior of its constituents. For generality, the latter definition is asymmetric. Throughout Section 3, we let $P_1 = \langle \mathcal{N}_1, \mathcal{E}_1, G_1 \rangle$ and $P_2 = \langle \mathcal{N}_2, \mathcal{E}_2, G_2 \rangle$ be separable programs.

Definition 6 (Product). The program $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$ is a product of P_1 and P_2, written $P \in P_1 \times P_2$, iff $\mathcal{N} \subseteq \mathcal{N}_1 \times \mathcal{N}_2$, and $(\mathsf{in}_1, \mathsf{in}_2) \in \mathcal{N}$ and the following properties hold:

Semantic coherence: every edge $e \in \mathcal{E}$ is of one of the forms

- *left:* $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2)$, with $\langle l_1, l_1' \rangle \in \mathcal{E}_1$, and $[\![e]\!] = [\![\langle l_1, l_1' \rangle]\!]$;
- *synchronous:* $(l_1, l_2) \Mapsto (l_1', l_2')$, with $\langle l_1, l_1' \rangle \in \mathcal{E}_1$, $\langle l_2, l_2' \rangle \in \mathcal{E}_2$, and $[\![e]\!] = [\![\langle l_1, l_1' \rangle]\!] \circ [\![\langle l_2, l_2' \rangle]\!]$;
- *right:* $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l_2')$, with $\langle l_2, l_2' \rangle \in \mathcal{E}_2$, and $[\![e]\!] = [\![\langle l_2, l_2' \rangle]\!]$;

Repleteness: $(\mathsf{out}_1, \mathsf{out}_2) \in \mathcal{N}$, and for all $(l_1, l_2) \in \mathcal{N}$

- if $l_1 = \mathsf{out}_1$ and $\langle l_2, l_2' \rangle \in \mathcal{E}_2$ then $(\mathsf{out}_1, l_2) \overset{\mathsf{r}}{\mapsto} (\mathsf{out}_1, l_2') \in \mathcal{E}$;
- if $l_2 = \mathsf{out}_2$ and $\langle l_1, l_1' \rangle \in \mathcal{E}_1$ then $(l_1, \mathsf{out}_2) \overset{\mathsf{l}}{\mapsto} (l_1', \mathsf{out}_2) \in \mathcal{E}$.

Moreover, P is minimal iff for all edges $(l_1, l_2) \Mapsto (l_1', l_2') \in \mathcal{E}$, $(l_1, l_2) \overset{\mathsf{r}}{\not\mapsto} (l_1', l_2)$ and $(l_1, l_2) \overset{\mathsf{r}}{\not\mapsto} (l_1, l_2')$, and for all nodes $l_1'' \in \mathcal{N}_2$ and $l_2'' \in \mathcal{N}_2$, $(l_1, l_2) \not\Mapsto (l_1', l_2'')$ and $(l_1, l_2) \not\Mapsto (l_1'', l_2')$.

Semantic coherence ensures that every edge in $\mathcal{E}$ represents either an execution step of program P_1, an execution step of program P_2, or a pair of simultaneous steps of both programs P_1 and P_2. Repleteness ensures that P lets P_1 terminate its execution when P_2 has terminated, and conversely. Products often satisfy the stronger property that for all $(l_1, l_2) \in \mathcal{N}$, $l_1 = \mathsf{out}_1$ iff $l_2 = \mathsf{out}_2$; however, self-composed programs do not satisfy this stronger property. Finally, note that minimality, albeit not built in the definition of products, holds for synchronized products and all our examples; we use minimality to define the composition of products. The synchronized product of two commands, as defined in Section 2, corresponds to a product in which all edges are synchronous.

```
la : while (i<N) do          l1 : while (j<M) do
       x+=i;                        y+=j;
       i++                          j++;
lb : ...                     l2 : while (j<N) do
                                    y+=j;
                                    j++
                             l3 : ...
```

(a) Original program (b) Transformed program

Fig. 7. Range splitting

Example 3 (Product programs). Fig. 8(b) and Fig. 8(c) shows two different products constructions of the programs in Fig. 7. The graph in Fig. 8(b) corresponds to sequential composition, all edges are either left or right and are thus represented by a single line. The graph in Fig. 8(c) pairs the left program loop with the second loop of the right program. Synchronized edges are represented by a double line.

Products underapproximate the behavior of their constituents, in the sense that every trace of $P \in P_1 \times P_2$ is a combination of a P_1-trace and a P_2-trace. We formalize this fact using left and right projections of traces. Formally, the left projection of a P-step is defined by case analysis:

- if $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \rightsquigarrow \langle (l_1', l_2), \sigma_1' \uplus \sigma_2 \rangle$ and $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2)$, or $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \rightsquigarrow \langle (l_1', l_2'), \sigma_1' \uplus \sigma_2' \rangle$ and $(l_1, l_2) \Mapsto (l_1', l_2')$, then the left projection is defined as $\langle l_1, \sigma_1 \rangle \rightsquigarrow \langle l_1', \sigma_1' \rangle$;
- otherwise, the left projection is undefined.

The left projection $\pi_1(t)$ of a trace t is then defined as the concatenation of the left projections of its steps. The right projection $\pi_2(t)$ of a trace t is defined in a similar way.

Lemma 2. Let $P \in P_1 \times P_2$. For all P-runs t, $\pi_1(t)$ is a P_1-run and $\pi_2(t)$ is a P_2-run.

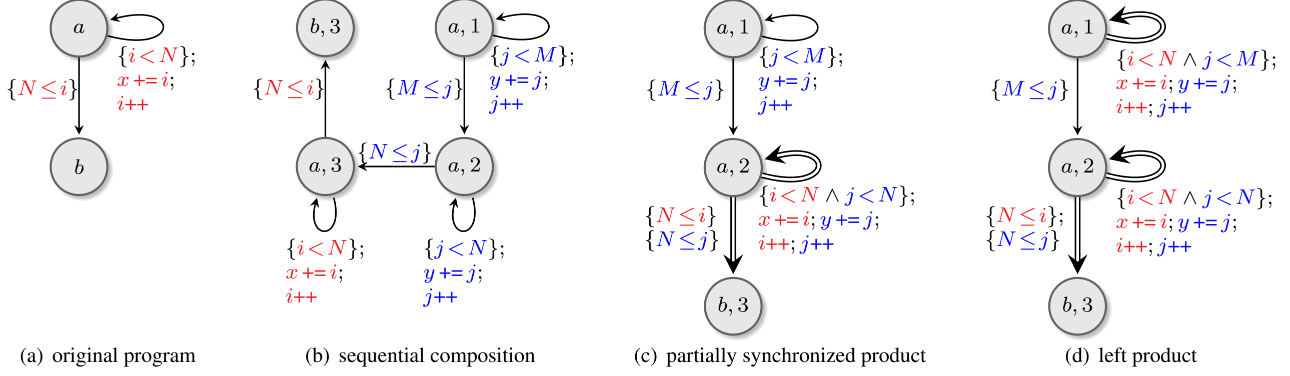
The converse of Lemma 2 holds trivially for products with sufficiently many left and right edges. For practical purposes, we need to rely on a weaker assumption that can accommodate synchronized products. To this end, we introduce the notion of left product. Its intuition is better understood in terms of strategies. At each node (l_1, l_2), P_1 may decide to progress along an edge of its choice, i.e. P_1 fixes l_1' and P progresses along $\langle l_1, l_1' \rangle$, or P_1 may decide to leave P_2 to progress along an edge of its choice, i.e. P_2 fixes l_2' and P progresses along $\langle l_2, l_2' \rangle$, or P_1 may decide to progress synchronously with P_2 along an edge of its choice, i.e. P_1 fixes l_1' and l_2', and P progresses simultaneously along $\langle l_1, l_1' \rangle$ and $\langle l_2, l_2' \rangle$. In the latter case, we need a semantic condition to ensure that P_2 can progress along the edge l_2' fixed by P_1. Since it would be clearly too strong to require this coherence property for arbitrary states, the definition is parametrized by a precondition.

Definition 7 (Left product). A product $P \in P_1 \times P_2$ is a left product w.r.t. a precondition φ, written $P \in P_1 \ltimes_\varphi P_2$, iff for every φ-trace ending in $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle$ and every $l_1' \in \mathcal{N}_1$ and $l_2' \in \mathcal{N}_2$ s.t. $\langle l_1, \sigma_1 \rangle \rightsquigarrow l_1'$ and $\langle l_2, \sigma_2 \rangle \rightsquigarrow l_2'$, one of the following holds:

1. $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2)$ belongs to P;
2. there exists an edge $(l_1, l_2) \Mapsto (l_1', l_2'')$ in P s.t. $\langle l_2, \sigma_2 \rangle \rightsquigarrow l_2''$;
3. $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l_2')$ belongs to P.

P is a full product w.r.t. a precondition φ, written $P \in P_1 \bowtie_\varphi P_2$, iff it satisfies the stronger condition $(l_1, l_2) \Mapsto (l_1', l_2')$ in P instead of condition 2 above.

For deterministic programs, left products and full products coincide. Indeed, assume that P_2 is deterministic, i.e. for all steps



(a) original program (b) sequential composition (c) partially synchronized product (d) left product

Fig. 8. Program representation and products

$\langle l,\sigma\rangle \rightsquigarrow \langle l',\sigma'\rangle$ and $\langle l,\sigma\rangle \rightsquigarrow \langle l'',\sigma''\rangle$, $l' = l''$; note that we do not require $\sigma' = \sigma''$. Then $P \in P_1 \ltimes_\varphi P_2$ iff $P \in P_1 \bowtie_\varphi P_2$. On the other hand, for non-deterministic programs, full products are stronger that left and right products—the notion of right product is dual to left product. Consider the statements $x := 1 \oplus x := 2$ and $y := 1 \oplus y := 2$, where $\oplus$ stands for non-deterministic choice. Then the program $(x := 1; y := 1) \oplus (x := 2; y := 2)$ is both a left and right product, but not a full product—its graph-based representation contains two synchronized edges from the initial to the final node[1], resp. labelled with statements $x := 1; y := 1$ and $x := 2; y := 2$.

Example 4 (Left products). It is straightforward to check that the sequential composition (Fig. 8(b)) is a left product of the programs in Fig. 7. On the other hand, the program in Fig. 8(c) is not a left product w.r.t. the precondition $i{=}j$: indeed, the product may get stuck at node $(a,2)$, since a step from $(a,1)$ to itself may invalidate $i{=}j$, and hence one can subsequently have at $(a,2)$ that $i < N$ but not $j < N$, so that none of the synchronous edges can be taken. The program in Fig. 8(d) is a left product w.r.t. $i{=}j$; see Section 5.2.

Full products allow proving a converse to Lemma 2.

Proposition 5 (Lifting–full products). Assume $P \in P_1 \bowtie_\varphi P_2$. Let t_1 be a P_1-run with initial state σ_1 and t_2 be P_2-run with initial state σ_2. Suppose $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$. Then there exists a P-run t s.t. $\pi_1(t) = t_1$ and $\pi_2(t) = t_2$.

Left products allow giving an asymmetric variant of the theorem.

Proposition 6 (Lifting–left products). Assume $P \in P_1 \ltimes_\varphi P_2$. Let t_1 be a P_1-run with initial state σ_1, and let $\sigma_2 \in \mathcal{S}$ s.t. $\langle \mathsf{in}_2, \sigma_2\rangle \Downarrow^\star$ and $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$. Then there exists a P-run t with initial configuration $\langle(\mathsf{in}_1, \mathsf{in}_2), \sigma_1 \uplus \sigma_2\rangle$ s.t. $\pi_1(t) = t_1$.

3.3 Composing left products

Left products enjoy several closure properties; in this section, we show that they are closed under composition. The justification for closure properties is three-fold: first, they are essential to extend embedding results to structural properties. Second, they establish the compositionality of our methods. Third, they show that proofs of program equivalence that involve several intermediate programs, and hence several intermediate products, can be collapsed in a proof with a single product.

We first dispose of a minor difficulty in defining the composition of two products. Informally, we aim to build from $P_a \in P_1 \times P_2$ and $P_b \in P_2 \times P_3$ a new product $P \in P_1 \times P_3$. However, the definition of product requires P_2 to be a right program in $P_a \in P_1 \times P_2$, and a left program in $P_b \in P_2 \times P_3$. The issue is resolved by introducing the notion of mirror program. Informally, a mirror program of P_2 is a program with the same nodes and edges, and such that the semantics of each mirror edge is a renaming of the semantics of the original edge. For readability, we elide the distinction between the program P_2 and its mirror, and simply refer to P_2.

The definition of composition requires that P_a and P_b are compatible w.r.t. $\mathcal{N}$, i.e. for all nodes $(l_1, l_3), (l_1', l_3') \in \mathcal{N}$ and edges $\langle l_1, l_1'\rangle \in \mathcal{E}_1$, $\langle l_2, l_2'\rangle \in \mathcal{E}_2$, $\langle l_3, l_3'\rangle \in \mathcal{E}_3$, $(l_1, l_2) \Mapsto (l_1', l_2') \in \mathcal{E}_a$ iff $(l_2, l_3) \Mapsto (l_2', l_3') \in \mathcal{E}_b$—we use the expected subscripts to refer to the nodes, edges and labeling of each program. Note that one can always replace synchronous edges by two asynchronous ones, so it is always possible to transform P_a and P_b so as to make them compatible.

Definition 8 (Composition of products). The product P is the composition of P_a and P_b, written $P = P_a \circ P_b$, iff P_a and P_b are compatible and minimal, and $\mathcal{N} = \mathcal{N}_a \circ \mathcal{N}_b$, and moreover

- $(l_1, l_3) \overset{\mathsf{l}}{\mapsto} (l_1', l_3) \in \mathcal{E}$ iff $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2) \in \mathcal{E}_a$ for an $l_2 \in \mathcal{N}_2$;
- $(l_1, l_3) \overset{\mathsf{r}}{\mapsto} (l_1, l_3') \in \mathcal{E}$ iff $(l_2, l_3) \overset{\mathsf{r}}{\mapsto} (l_2, l_3') \in \mathcal{E}_b$ for an $l_2 \in \mathcal{N}_2$;
- $(l_1, l_3) \Mapsto (l_1', l_3') \in \mathcal{E}$ iff both $(l_1, l_2) \Mapsto (l_1', l_2') \in \mathcal{E}_a$ and $(l_2, l_3) \Mapsto (l_2', l_3') \in \mathcal{E}_b$ for an edge $\langle l_2, l_2'\rangle \in \mathcal{E}_2$.

The composition of two left products exists and is a left product.

Proposition 7. Assume that $P_a \in P_1 \ltimes_{\varphi_a} P_2$ and $P_b \in P_2 \ltimes_{\varphi_b} P_3$ are compatible products. Then $P_a \circ P_b \in P_1 \ltimes_{\varphi_a \circ \varphi_b} P_3$, provided for every $\sigma_1 \in \mathcal{S}$ s.t. $\langle \mathsf{in}_1, \sigma_1\rangle \Downarrow$, there exists $\sigma_2 \in \mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2 \in [\![\varphi_a]\!]$ and $\langle \mathsf{in}_2, \sigma_2\rangle \Downarrow^\star$.

Section 5.2 provides an application of product composition.

4. Relational verification by program verification

Using the results of the previous section, we prove that relational correctness and refinement can be verified by standard means. More precisely, we show how to compute from two programs P_1 and P_2 and a partial specification Φ a set of verification conditions that can be discharged using standard provers, and that show the correctness of (P_1, P_2) w.r.t. Φ; or that P_2 refines P_1 w.r.t. Φ.

4.1 Program verification

Program correctness is expressed by a triple of the form $\{\varphi\}\ P\ \{\psi\}$, where $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ is a program, and φ, ψ are assertions. A triple $\{\varphi\}\ P\ \{\psi\}$ is valid, written $\vDash \{\varphi\}\ P\ \{\psi\}$, iff for every states $\sigma, \sigma' \in \mathcal{S}$, $\sigma \in [\![\varphi]\!]$ and $\langle \mathsf{in}, \sigma\rangle \rightsquigarrow^\star \langle \mathsf{out}, \sigma'\rangle$ imply $\sigma' \in [\![\psi]\!]$. The standard means to prove correctness is to show that P is correct

[1] Strictly speaking, the program is represented by a multigraph. One can turn a multigraph into a graph by introducing intermediate nodes.

w.r.t. a partial specification Φ that extends φ and ψ, i.e. to exhibit $\Phi : \mathcal{N} \rightharpoonup \mathsf{Assn}$ s.t. i) all cycles in the graph of P go through an annotated node, i.e. a node in $\mathsf{dom}(\Phi)$; ii) $\mathsf{in}, \mathsf{out} \in \mathsf{dom}(\Phi)$ with $\varphi = \Phi(\mathsf{in})$ and $\psi = \Phi(\mathsf{out})$; iii) for every states $\sigma, \sigma' \in \mathcal{S}$, $\sigma \in [\![\Phi(l)]\!]$ and $\langle l, \sigma\rangle \rightsquigarrow^\star \langle l', \sigma'\rangle$ imply $\sigma' \in [\![\Phi(l')]\!]$. In general, one can compute from P and Φ a set of verification conditions.

Example 5 (Verification conditions). One can define for each statement c of Example 2 and assertion ϕ a weakest precondition

$$\begin{array}{ll} \mathsf{wp}(\mathsf{skip}, \phi) \doteq \phi & \mathsf{wp}(x := e, \phi) \doteq \phi[e/x] \\ \mathsf{wp}(\{b\}, \phi) \doteq b \Rightarrow \phi & \mathsf{wp}(c_1; c_2, \phi) \doteq \mathsf{wp}(c_1, (\mathsf{wp}(c_2, \phi))) \end{array}$$

One can use the weakest preconditions to generate a specification $\Phi^\natural$ that extends Φ to all nodes. Using the well-founded induction principle attached to partial specifications, see e.g. [9], we set

$$\Phi^\natural(l) \doteq \bigwedge_{\langle l, l'\rangle \in \mathcal{E}} \mathsf{wp}(G\langle l, l'\rangle, \Phi^\natural(l')) \qquad \text{for all } l \notin \mathsf{dom}(\Phi)$$

The set of verification conditions of P includes all assertions $\Phi(l) \Rightarrow \mathsf{wp}(G\langle l, l'\rangle, \Phi^\natural(l'))$ where $\langle l, l'\rangle \in \mathcal{E}$ and $l \in \mathsf{dom}(\Phi)$.

Example 6 (Proof obligations for loop reversal). The verification conditions for the graph-based representation of the program c from Example 1 can be discharged using simple arithmetic reasoning. E.g. the verification condition for the loop invariant is of the form

$$\begin{array}{c} 0 \leq x' \wedge x + x' = N+1 \wedge y' - y = x(N+2-x) - (N+1) \wedge x \leq N \\ \Rightarrow \\ 0 \leq x' - 1 \;\wedge\; x+1+x'-1 = N+1 \;\wedge \\ y' + x' - y - x = (x+1)(N+2-x-1) - (N+1) \end{array}$$

4.2 Relational verification

Relational program correctness is formalized by quadruples of the form $\{\varphi\} P_1 \sim P_2 \{\psi\}$, where P_1, P_2 are separable programs, and φ, ψ are assertions. A relational judgment $\{\varphi\} P_1 \sim P_2 \{\psi\}$ is valid, written $\vDash \{\varphi\} P_1 \sim P_2 \{\psi\}$, iff for all states $\sigma_1, \sigma_1', \sigma_1, \sigma_2' \in \mathcal{S}$, $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and $\langle \mathsf{in}_1, \sigma_1\rangle \rightsquigarrow^\star \langle \mathsf{out}, \sigma_1'\rangle$ and $\langle \mathsf{in}_2, \sigma_2\rangle \rightsquigarrow^\star \langle \mathsf{out}_2, \sigma_2'\rangle$ imply $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$. Relational verification can be reduced to program verification of a product. For simplicity, we assume that the programs are deterministic.

Proposition 8. Let P_1, P_2 be deterministic separable programs and let φ, ψ be assertions. Then $\vDash \{\varphi\} P_1 \sim P_2 \{\psi\}$, provided $\vDash \{\varphi\}\ P\ \{\psi\}$, and $P \in P_1 \ltimes_\varphi P_2$.

Since the programs are deterministic, it is equivalent to require $P \in P_1 \bowtie_\varphi P_2$ instead of $P \in P_1 \ltimes_\varphi P_2$. We show that the leftness of a product can also be checked by logical means. We use a simple form of path condition, which we call edge condition, to express leftness. Formally, the edge condition $\mathsf{ec}(c)$ for a statement c is, if it exists, the unique (up to logical equivalence) formula ϕ s.t. for all states $\sigma \in \mathcal{S}$, $\sigma \in [\![\phi]\!]$ iff there exists a state $\sigma' \in \mathcal{S}$ s.t. $(\sigma, \sigma') \in [\![c]\!]$. In the sequel, we assume edge conditions exist.

Example 7 (Edge conditions). Weakest preconditions can be used to compute edge conditions. For each statement c of Example 2, one defines $\mathsf{ec}(c)$ by the clauses

$$\begin{array}{ll} \mathsf{ec}(\mathsf{skip}) \doteq \mathsf{true} & \mathsf{ec}(x := e) \doteq \mathsf{true} \\ \mathsf{ec}(\{b\}) \doteq b & \mathsf{ec}(c_1; c_2) \doteq \mathsf{ec}(c_1) \wedge \mathsf{wp}(c_1, \mathsf{ec}(c_2)) \end{array}$$

We define for every node $(l_1, l_2) \in \mathcal{N}$ and edges $\langle l_1, l_1'\rangle \in \mathcal{E}_1$ and $\langle l_2, l_2'\rangle \in \mathcal{E}_2$ s.t. $(l_1, l_2) \overset{\mathsf{l}}{\not\mapsto} (l_1', l_2)$ and $(l_1, l_2) \overset{\mathsf{r}}{\not\mapsto} (l_1, l_2')$ the Φ-leftness condition as

$$\Phi(l_1, l_2) \wedge \mathsf{ec}(l_1, l_1') \wedge \mathsf{ec}(l_2, l_2') \Rightarrow \bigvee_{l_2'' : (l_1, l_2) \mapsto (l_1', l_2'')} \mathsf{ec}(l_2, l_2'')$$

and say that P is Φ-left iff all its Φ-leftness conditions are valid.

Theorem 1. Let P_1, P_2 be deterministic separable programs and let φ, ψ be assertions. Then $\vDash \{\varphi\} P_1 \sim P_2 \{\psi\}$, provided there exists a partial specification Φ and a product program $P \in P_1 \times P_2$ s.t. $\varphi = \Phi(\mathsf{in}_1, \mathsf{in}_2)$, $\psi = \Phi(\mathsf{out}_1, \mathsf{out}_2)$, and P is Φ-left and correct w.r.t. Φ.

Theorem 1 puts relational program verification on par with standard verification, modulo the construction of the product program P: to prove relational correctness of two programs, one must provide a partial specification and discharge the proof obligations generated by weakest preconditions. The construction of the specification can rely on standard tools for inferring annotations, and the proof obligations can be proved with off-the-shelf tools.

Our next objective is to develop methods that automate or assist the construction of products. We intend to rely on a combination of standard methods, such as abstract interpretation [15], to infer invariants that drive the construction of the product, and user-given heuristics that specify relations between elements of the two programs: typically, the relations will relate control structures, as in translation validation, or variables, as in information flow logics. Since product construction may be seen as an instance of program synthesis, it may also be beneficial to explore recent developments in the field, e.g. [32].

4.3 Refinement

We briefly sketch an asymmetric variant of Theorem 1 that supports reasoning about refinement—and does not assume programs are deterministic. The definition of refinement from Section 2.4 extends to programs immediately: we say that P_2 refines P_1 w.r.t. a pre-condition φ and a post-condition ψ, written $\models \{\varphi\} P_1 \mapsto P_2 \{\psi\}$, iff for all states $\sigma_1, \sigma_1', \sigma_2 \in \mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and $\langle \mathsf{in}_1, \sigma_1\rangle \rightsquigarrow^\star \langle \mathsf{out}_1, \sigma_1'\rangle$, there exists a state $\sigma_2' \in \mathcal{S}$ s.t. $\langle \mathsf{in}_2, \sigma_2\rangle \rightsquigarrow^\star \langle \mathsf{out}_2, \sigma_2'\rangle$ and $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$. Moreover, we say that P_1 implies termination of P_2 w.r.t. φ iff for every states $\sigma_1, \sigma_2 \in \mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2 \in [\![\Phi(\mathsf{in}_1, \mathsf{in}_2)]\!]$, $\langle \mathsf{in}_1, \sigma_1\rangle \Downarrow$ implies $\langle \mathsf{in}_2, \sigma_2\rangle \Downarrow^*$.

Theorem 2. Let P_1, P_2 be separable programs and let φ, ψ be assertions. Assume that P_1 implies termination of P_2 w.r.t. φ. Then $\models \{\varphi\} P_1 \mapsto P_2 \{\psi\}$, provided there exists a partial specification Φ s.t. $\varphi = \Phi(\mathsf{in}_1, \mathsf{in}_2)$ and $\psi = \Phi(\mathsf{out}_1, \mathsf{out}_2)$, and a product program $P \in P_1 \times P_2$ that is Φ-left and correct w.r.t. Φ.

Theorem 2 provides direct proofs of correctness for many common refinement steps, e.g. replacing a non-deterministic assignment by an assignment (or a non-deterministic choice by one of its substatements), or inserting an `assert` in a statement. Developing more complete examples—as done e.g. in [11]—is left for future work.

5. Application: relational translation validation

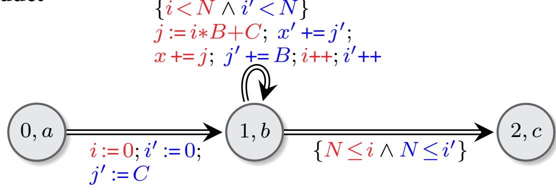
Translation validation [6, 27] is a general method for ensuring the correctness of optimizing compilation by means of a validator which checks after each run of the compiler that the source and target programs are semantically equivalent. Translation validation considers three main classes of transformations: instruction replacement, reordering of loops iterations, and elimination of loop iterations, and proposes for each of them a proof rule, respectively: (VALIDATE), (PERMUTE), and (REDUCE). This section revisits translation validation from the perspective of products, and shows how examples of application of each rule can be treated using products. In particular, we show that our method can be used to justify static caching, an advanced loop optimization for incremental computation [23].

For concreteness, our examples are developed in an extension of the imperative language of Section 2 with arrays in 1 and 2 dimensions. Array accesses are modeled by expressions $a[e]$ and $a[e, e']$, and array writes are modeled by instructions $a[e] := e_0$ and

Source and optimized code

```
0: i := 0;                 a: i' := 0; j' := C;
1: while (i < N) do        b: while (i' < N) do
     j := i*B+C;                x' += j';
     x += j; i++                j' += B; i'++
2: ...                     c: ...
```

Product



Specification

$\Phi(0,a) \doteq \Phi(2,c) \doteq x = x'$
$\Phi(1,b) \doteq x = x' \wedge i = i' \wedge j' = i'B + C$

Fig. 9. Strength reduction

$a[e,e'] := e_0$. For simplicity, each array a has a fixed size, denoted $|a|$, and the semantics does not model array-out-of-bounds, i.e. programs with array-out-of-bounds accesses get stuck. Under this assumption, one can extend the precondition and edge condition functions to arrays by letting $\mathsf{wp}(a[e] := e', \phi) \doteq \phi[[a\colon e \mapsto e']/a]$ and $\mathsf{ec}(a[e] := e') \doteq 0 \leq e < |a|$. The expression $[a\colon e \mapsto e']$ denotes the array a' such that $a'[x] = e'$ if $x = e$, and $a'[x] = a[x]$ otherwise.

5.1 Structure preserving optimizations

Common program optimizations such as constant propagation or common subexpression elimination do not modify the control-flow of programs, and simply affect their basic blocks. Synchronized products provide a convenient means to validate the correctness of such optimizations. We illustrate this approach with strength reduction, a standard transformation that reduces the cost of arithmetic operations occurring inside a loop body. Fig. 9 shows the application of the optimization to a small fragment of code. In the figure, j is a derived induction variable, defined as a linear function on the induction variable i. The optimization replaces the assignment $j := i{*}B{+}C$ by the equivalent and less costly $j' \mathrel{+}= B$, swaps the assignments to x and j, and adds an initialization $j' := C$ immediately before the loop header.

Correctness of the product w.r.t. its specification can be established by simple arithmetic reasoning. Likewise, the leftness conditions are trivial to derive from the specification. Note that the loop invariant at node $(1,b)$ requires $j' = i'B + C$, which can be inferred using certifying analyzers [9].

5.2 Loop optimizations

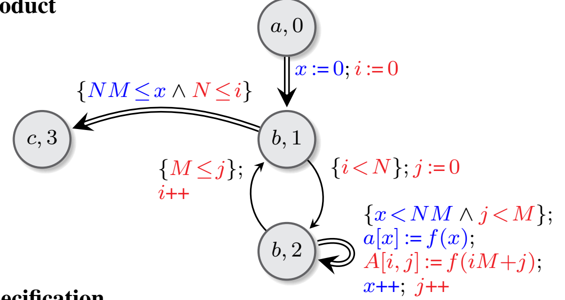
One particular strength of translation validation is its ability to prove correct advanced loop optimizations [6]. We show by examples that products can be used for the same purpose.

Loop range splitting. Loop range splitting extracts from a main loop a number of initial loop iterations. It helps infer stronger invariants separately for each resulting loop, enabling more optimization opportunities. Figure 7 shows an example of this transformation; the product construction used to verify the optimization correct is given in Fig. 8(d). Note that there is no structured representation of this product graph, as the two loop edges at nodes $(a,1)$ and $(a,2)$ correspond to the loop body of the original program. At node $(a,1)$ both programs iterate synchronously while the right program is in the first loop. The right edge $(a,1) \overset{\mathsf{r}}{\mapsto} (a,2)$ corresponds to the right program transition that exits the first loop. From the node $(a,2)$ the execution is completely synchronized.

Source and intermediate code

```
0: i := 0;                          a: x := 0;
1: while (i ≤ N) do                 b: while (x ≤ MN) do
     j := 0;                             a[x] := f(x);
2:   while (j ≤ M) do                    x++
       A[i,j] := f(iM+j); j++;      c: ...
     i++
3: ...
```

Product



Specification

$\Phi(a,0) \doteq$ true
$\Phi(b,2) \doteq x = iM + j \wedge i < N \wedge j \leq M \wedge \varphi(i) \wedge \psi$
$\Phi(c,3) \doteq \varphi(N)$
where $\varphi(i) \doteq \forall l, r.\ 0 \leq l < i \wedge 0 \leq r < M \Rightarrow A[l,r] = a[lM + r]$
$\psi \doteq \forall r.\ 0 \leq r < j \Rightarrow A[i,r] = a[iM + r]$

Fig. 10. Loop interchange

The specification maps each program node to the assertion $x = y \wedge i = j$ holds at every program node. The correctness of the program against this specification can be verified by simple reasoning. Likewise, Φ-rightness conditions are easy to establish. E.g. to prove the rigthness condition for the node $(a,1) \in \mathcal{N}$ and for the edges $\langle a,b\rangle \in \mathcal{N}_1$ and $\langle 1,1\rangle \in \mathcal{N}_2$—since neither $(a,1) \overset{\mathsf{l}}{\mapsto} (a,1) \notin \mathcal{E}$ nor $(a,1) \overset{\mathsf{r}}{\mapsto} (a,1) \notin \mathcal{E}$—one must discharge a formula of the form $\Phi(a,1) \wedge \mathsf{ec}(\langle a,b\rangle) \wedge \mathsf{ec}(\langle 1,1\rangle) \Rightarrow \ldots$, which holds trivially since the antecedent is false.

Loop interchange. Loop interchange is a compiler optimization that maximizes the contiguity of memory accesses and improves cache performance by transposing row and column accesses in accordance to the underlying array model. Interchanging the nested loops for the original program at Fig. 10 is defined as swapping the boolean guards $i \leq N$ with $j \leq M$, and the statements $i := 0$ and $i{+}{+}$ with $j := 0$ and $j{+}{+}$. For simplicity, we consider the transformation of the original program to an intermediate program as shown in Fig. 10, together with the symmetric step. The correctness of loop interchange follows by composition (i.e., by Proposition 7.)

The product merges the loop bodies (i.e., edges $\langle 2,2\rangle$ and $\langle b,b\rangle$) in a single synchronized edge. In order to ensure the invariant $x = iM + j$, when the value of j becomes equal to M, left edges increment by 1 the value of i and set j equal to 0 (the symmetric transformation step requires $x = jM + i$ instead.)

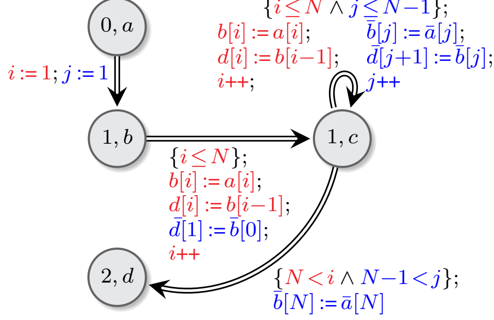
It is immediate to verify the correctness of the product against its specification, and the Φ-leftness conditions. E.g. Φ-leftness at the node $(b,2)$ and for the edges $\langle b,b\rangle$ and $\langle 2,2\rangle$ reduces to showing that $\Phi(b,2) \wedge \mathsf{ec}(2,2) \wedge \mathsf{ec}(b,b)$ implies $\mathsf{ec}((b,2) \mapsto (b,2))$, which holds trivially.

Loop alignment. Loop alignment improves cache performance by increasing the proximity of the memory locations accessed by a loop iteration. Consider the optimization example shown in Fig. 11, and assume $1 \leq N$. The array b is accessed twice in the loop body: a write access to the position i, and a read access to the position $i{-}1$. The optimization aligns the accesses of the array b in the loop body to a same index i. Notice that if the values stored by the array

Source and optimized code

```
0:  i := 1;                     a:  j := 1;
1:  while (i ≤ N) do            b:  d̄[1] := b̄[0];
      b[i] := a[i];             c:  while (j ≤ N−1) do
      d[i] := b[i−1];                 b̄[j] := ā[j];
      i++                             d̄[j+1] := b̄[j];
2: ...                                j++;
                                    b̄[N] := ā[N]
                                d: ...
```

Product



Specification

$\Phi(0,a) \doteq a = \bar{a} \wedge b[0] = \bar{b}[0]$
$\Phi(1,c) \doteq d[1,i) = \bar{d}[1,i) \wedge b[j] = a[j] \wedge \bar{b}[i] = b[i] \wedge i = j+1$
$\Phi(2,d) \doteq d[1,N] = \bar{d}[1,N]$

Fig. 11. Loop alignment

b are not needed after the loop execution, the occurrence of $b[i]$ in the loop body is replaced by a fresh scalar variable.

The product is shown in the same figure. The construction of the product synchronizes the initialization statement on the right $\bar{d}[1] := \bar{b}[0]$ with the first loop iteration on the left, and then synchronizes the remaining $N-1$ loop iterations.

The specification ensures that given equal input arrays a and b, the values stored in the array d coincide after the execution of both programs. The expression $d[1,i) = \hat{d}[1,i)$ means that the values of the arrays d and $\hat{d}$ coincide in the index range $[1,i)$—and similarly with the range $[1,N]$.

We briefly comment on proving leftness w.r.t. the specification. First, note that the product contains only synchronized edges and the only missing edge at node $(1,b) \in \mathcal{N}$ is $\langle 1,2 \rangle$, corresponding to the execution branch exiting the loop execution. However, this edge is never taken since at node $(1,b)$ the condition $i = 0$ holds and we have $1 \leq N$ as hypothesis. Since all the remaining edges are in the graph, it is sufficient to ensure that the left and right edge conditions of each synchronous edge are equivalent. For the edge $(1,b) \mapsto (1,c)$ one must verify that $i \leq N$ is equivalent to $\mathtt{true}$, which holds by hypothesis. For the loop edge $(1,c) \mapsto (1,c)$, one must check the equivalence of $i \leq N$ and $j \leq N-1$, which holds from the invariant $i = j+1$.

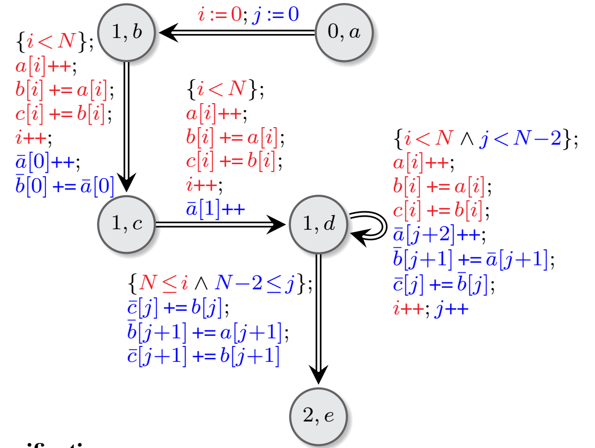
Loop pipelining. The effect of loop pipelining is the opposite to loop alignment, as it reduces the proximity of memory references inside a loop, in order to introduce parallelization opportunities. Consider the simple example shown in Fig. 12 (drawn from [21].) Assume a, b and c are arrays of size N, with $2 \leq N$.

The program product shown in Fig. 12 pairs the initialization statement at nodes l_b and l_c with the first loop iteration of the original program. Similarly, the final statements affecting $\bar{c}[N-2]$, $\bar{c}[N-2]$ and $\bar{c}[N-1]$ are executed synchronously with the final loop iteration of the original loop. The remaining $N-2$ loop iterations are synchronized together. In order to verify that $a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}$ is a valid pre and post condition, we require a specification that establishes the equalities in b and $\bar{b}$ and c and $\bar{c}$, except

Source and optimized code

```
0: i := 0;                a: j := 0;
1: while (i < N) do       b: ā[0]++;
     a[i]++;                 b̄[0] += ā[0];
     b[i] += a[i];        c: ā[1]++;
     c[i] += b[i];        d: while (j < N−2) do
     i++                       ā[j+2]++;
2: ...                         b̄[j+1] += ā[j+1];
                               c̄[j] += b̄[j];
                               j++
                             c̄[j] += b̄[j];
                             b̄[j+1] += ā[j+1];
                             c̄[j+1] += b̄[j+1]
                          e: ...
```

Product



Specification

$\Phi(0,a) \doteq a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}$
$\Phi(1,d) \doteq a = \bar{a} \wedge i = j+2 \;\wedge b = [\bar{b}\colon j+1 \mapsto \bar{b}[j+1] + a[j+1]] \wedge$
$\quad c = [\bar{c}\colon (j, j+1) \mapsto (\bar{c}[j] + b[j], \bar{c}[j+1] + b[j+1])]$
$\Phi(2,e) \doteq a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}$

Fig. 12. Loop pipelining

for the indices j and $j+1$. In particular, the invariant at the node $(1,d)$ states that $b[j+1] = \bar{b}[j+1] + a[j+1]$, $c[j] = \bar{c}[j] + b[j]$, and $c[j+1] = \bar{c}[j+1] + b[j+1]$, and $b[i'] = \bar{b}[i']$ and $c[i'] = \bar{c}[i']$ for any other index i'.

Checking that the program is a full product requires proving for instance at node $(1,b)$ and for the edges $\langle 1,2 \rangle$ and $\langle b,c \rangle$ a formula of the form $\Phi(1,b) \wedge \mathsf{ec}(1,2) \wedge \mathsf{ec}(b,c) \Rightarrow \ldots$, which holds since the hypothesis $2 \leq N$ contradicts the antecedent $i = 0 \wedge \mathsf{ec}(1,2)$. The rest of the edges are in the graph in a synchronized form, so it only remains to verify that the corresponding edge conditions are equivalent. From the invariant $i = j+2$ the guard conditions $i < N$ and $j < N-2$ are equivalent.

5.3 Static caching

Static caching [23] is an optimization that removes redundant computations by exploiting memoized intermediate results. One of its applications is the row summation algorithm in Fig. 13. The algorithm takes as input an $N \times L$ matrix a and returns an array s of length $N-M+1$ (assume $M \leq N$) such that $s[i] = \sum_{i',j'=i,0}^{M,L} A[i',j']$, for all $i \in [0, N-M]$. The original program performs a significant amount of redundant computation. Let $b[i]$ stand for $\sum_{j=0}^{N} A[i,j]$. One can see that for all i, $s[i]$ differs from $s[i+1]$ on the value $b[i+M] - b[i]$. The computations of the summations $b[i']$ for $i' \in [i+1, i+M-1]$ are thus redundant and can be removed. In the optimized version of the algorithm, the ar-

```
a: i1 := 0;
b: while (i1 ≤ N−M) do
     s[i1] := 0;
     k1 := 0;
c:   while (k1 ≤ M−1) do
       l1 := 0;
d:     while (l1 ≤ L−1) do
         s[i1] += a[i1+k1, l1];
         l1++;
       k1++;
     i1++
e: ...
```

```
0: t[0] := 0;
   k2 := 0;
1: while (k2 ≤ M−1) do
     b[k2] := 0;
     l2 := 0;
2:   while (l2 ≤ L−1) do
       b[k2] += a[k2, l2];
       l2++;
     t[0] += b[k2];
     k2++;
   i2 := 1;
3: while (i2 ≤ N−M) do
     b[i2+M−1] := 0;
     l2 := 0;
4:   while (l2 ≤ L−1) do
       b[i2+M−1] += a[i2+M−1, l2];
       l2++;
     z := b[i2+M−1] − b[i2−1];
     t[i2] := t[i2−1] + z;
5:   i2++
6: ...
```

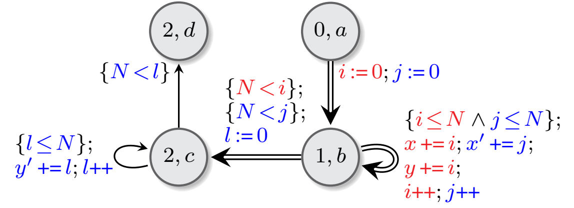
Fig. 13. Static caching: source and optimized code

Source and optimized code

```
0: i := 0;
1: while (i ≤ N) do
     x += i; y += i; i++
2: ...
```

```
a: j := 0;
b: while (j ≤ N) do
     x' += j; j++;
   l := 0;
c: while (l < N) do
     y' += l; l++
d: ...
```

Product



Specification

$\Phi(0,a) \doteq x = x' \wedge y = 0 = y'$
$\Phi(1,b) \doteq i = j \wedge x = x' \wedge 2y = i(i-1) \wedge i \leq N+1$
$\Phi(2,c) \doteq x = x' \wedge 2y = N(N+1) \wedge 2y' = l(l-1) \wedge l \leq N+1$
$\Phi(2,d) \doteq x = x' \wedge y = y'$

Fig. 15. Loop fission/fusion

ray b of size N is used to store the intermediate computation of row summations. The matrix summations are computed using the computations saved in the array b, and then stored in the array t. As a result, the transformed algorithm has a quadratic complexity, whereas the complexity of the original algorithm is cubic.

Figure 14 shows the product of the original row-summation algorithm and of its optimized version. In the product construction, the first loop iteration of the original program is synchronized with the first half of the optimized program in which the M first positions of the array b are initialized. From the node $(b, 3)$ the remaining $N-M$ iterations are synchronized. The asynchronous nested loop at the nodes $(c, 5)$ and $(d, 5)$ represents the redundant computation in the original program.

The specification states that the output arrays s and t coincide in the range $[0, N-M]$ after executing the original and optimized program. The correctness and leftness of the product w.r.t. its specification can be verified by simple arithmetic reasoning.

5.4 Discussion

Despite its ability to handle complex loop reordering, we have found that our framework is sometimes constrained by its simple-minded representation of programs. Specifically, there are examples of transformations for which a representation that exhibits implicit parallelism in programs would be beneficial. Consider for instance the loop fission transformation shown in Fig. 15. The product construction synchronizes the loop in the original program with the first loop of the final program: the second loop of the final program executes asynchronously. From the product specification one can see that equality $x = x'$ is easily enforced. However, proving that $y = y'$ requires stronger invariants, since y and y' are affected in separated program points. This particular example would clearly benefit from a more relaxed definition of product, in which both loops of the final program are synchronized with the original loop. In order to handle such examples, it would be desirable to extend the theory of products to a representation of programs that combines sequential and parallel execution of statements, *à la* alternating automata. Besides loop optimizations of sequential programs, this representation of programs could also be useful for translation validation of parallelizing compilers.

6. Related work

Relational methods play a prominent role in semantic models of programming languages, especially in the context of functional programming, where types are commonly interpreted as logical relations [26] or as PERs [1]. Besides, logical relations allow reasoning about program correctness and representation independence. In particular, Ahmed, Dreyer and Rossberg [2] prove representation independence for generative ADTs, using step-indexed logical relations [4], a variant of logical relations that minimizes the need for domain-theoretical constructions.

Relational logics provide a syntactical counterpart to semantic relational methods, and can be used for similar purposes. To date, relational logics have been applied to prove compiler correctness [10], program equivalence [8, 35], and non-interference [10, 25]; more generally, relational methods are applicable to regression verification [17], and to verification of 2-properties [14, 34], including determinism [12]. Furthermore, quantitative properties such as continuity [13] or indistinguishability [19] appear as a natural generalization of 2-properties. Extending the scope of relational logics and program products so that they can accommodate these properties is an important goal for future work. Dually, a common means to build automated program analyses is to isolate restricted—and possibly decidable—fragments of Hoare logics; it is for example possible to interpret information flow logics [3, 5] in this light. We intend to derive specialized proof systems for specific classes of properties: in particular, it would be interesting to build tools for reasoning about program improvement in the spirit of [30].

Finally, there are many commonalities between translation validation and product constructions. Kundu, Tatlock and Lerner [21] extend translation validation to parametrized programs, and generate automatically proofs of correctness for many optimizations. Moreover, Tatlock and Lerner [33] build *certified* translation validators using Leroy's CompCert infrastructure [22]. A challenge for further work is to develop *certifying* translation validators that handle parametrized programs and produce a product, its specification, and a proof of correctness.

7. Concluding remarks

Relational Hoare logics provide a convenient means to enforce a wide range of correctness and security properties, but have lacked

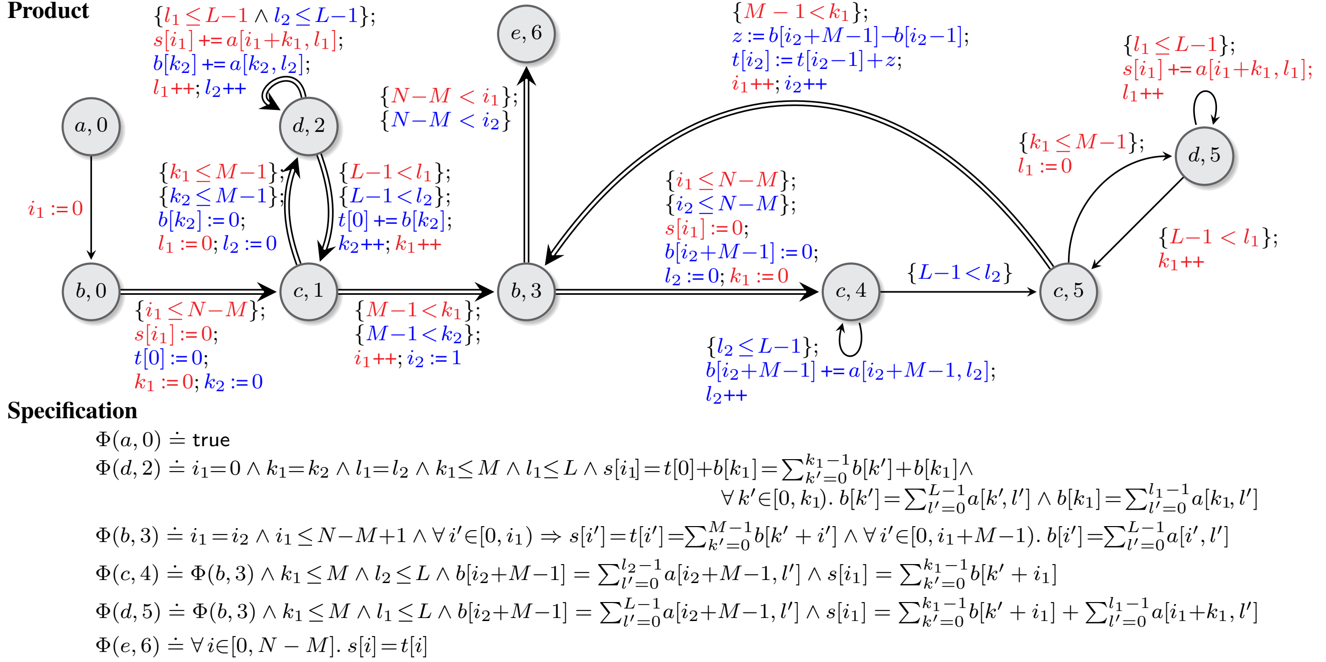


Fig. 14. Static caching

methods and tools that are available for program logics. This paper develops a notion of product between programs and reduces verification of relational properties between two programs to verification of functional properties of their product. The notion of product program is general and flexible, and overcomes the limitations of previous approaches.

The key to practical relational program verification is now to infer products or assist their construction. Our main priority is to develop methods and tools for building products, and to connect them with off-the-shelf tools to provide a tool for relational verification. In a separate line of work, we are developing a machine-checked formalization of products to check representation independence of ADT implementations[2]. Moreover, we are investigating applications of products to probabilistic programs, and intend to apply the resulting formalism to provable security [8] and privacy [28].

References

[1] M. Abadi and G. D. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science*, pages 355–365, 1990.

[2] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages*, pages 340–353, 2009.

[3] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Principles of Programming Languages*, pages 91–102. ACM Press, 2006.

[4] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

[6] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer-Verlag, 2005.

[7] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.

[8] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages*, pages 90–101. ACM Press, 2009.

[9] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382. Springer-Verlag, 2008.

[10] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Principles of Programming Languages*, pages 14–25. ACM Press, 2004.

[11] R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Principles of Programming Languages*, pages 339–352, 2010.

[12] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. *Communications of the ACM*, 53(6):97–105, 2010.

[13] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *Principles of Programming Languages*, pages 57–70, 2010.

[14] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Computer Security Foundations Symposium*, pages 51–65, 2008.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[16] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.

[2] We have already formalized the correctness of the Schorr-Waite algorithm *à la* Yang.

[17] B. Godlin and O. Strichman. Regression verification. In *Design Meets Automation*, pages 466–471, 2009.

[18] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–22. IEEE Press, 1982.

[19] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001-2004.

[20] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *European Symposium on Programming*, pages 187–196, 1986.

[21] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Languages Design and Implementation*, pages 327–337, 2009.

[22] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[23] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

[24] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[25] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *European Symposium On Research In Computer Security*, number 4189 in Lecture Notes in Computer Science, pages 279–296. Springer-Verlag, 2006.

[26] G. D. Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[27] A. Pnueli, E. Singerman, and M. Siegel. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.

[28] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *International Conference on Functional Programming*, 2010. To appear.

[29] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, 2003.

[30] D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4), 1995.

[31] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Principles of Programming Languages*, pages 355–364, 1998.

[32] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Principles of Programming Languages*, pages 313–326, 2010.

[33] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Programming Languages Design and Implementation*, pages 111–121, 2010.

[34] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2005.

[35] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[36] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures*, pages 402–416, 2002.

[37] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Formal Methods*, pages 35–51, 2008.

A. Selected Proofs

Proofs of Section 2

Lemma 3. If $\vdash \{\varphi\}\, c_1 \otimes c_2\, \{\psi\}$ then $c_2 \otimes c_1$ is defined and moreover $\vdash \{\varphi\}\, c_2 \otimes c_1\, \{\psi\}$.

Lemma 4. If $\vdash \{\varphi\}\, c_1 \otimes c_2\, \{\psi\}$, $\vdash \{\varphi'\}\, c_2 \otimes c_3\, \{\psi'\}$, c_2 terminates and let X be a set of variables such that $\mathsf{var}(c_2) \subseteq X$ then $c_1 \otimes c_3$ is defined and moreover:

$$\vdash \{\exists X.\varphi \wedge \varphi'\}\, c_1 \otimes c_3\, \{\exists X.\psi \wedge \psi'\}$$

Proof of Proposition 1. If $\vdash \{\varphi\}c_1 \sim c_2\{\psi\}$ then the statement $c_1 \otimes c_2$ is defined and moreover $\vdash \{\varphi\}\, c_1 \otimes c_2\, \{\psi\}$.

Proof. The proof is done by induction on the derivation of the judgment $\vdash \{\varphi\}c_1 \sim c_2\{\psi\}$ with case analysis on the last rule applied. We only account for selected cases.

- Case (R-INSTR). We have that $c_1 \equiv i_1$ and $c_2 \equiv i_2$ with i_1, i_2 basic instructions. Then $c_1 \otimes c_2 = i_1; i_2$ and we know that:

$$\vdash \{\varphi\}\, i_1; i_2\, \{\psi\}$$

- Case (R-WHILE). We have that $c_1 \equiv \mathsf{while}\ b_1\ \mathsf{do}\ d_1$, $c_2 \equiv \mathsf{while}\ b_2\ \mathsf{do}\ d_2$ and we know that:

$$\vdash \{\varphi \wedge b_1 \wedge b_2\}d_1 \sim d_2\{\varphi \wedge b_1 = b_2\}$$

By Inductive Hypothesis then we know that $d_1 \otimes d_2$ is defined, so we take $c_1 \otimes c_2 = \mathsf{while}\ b_1\ \mathsf{do}\ (d_1 \otimes d_2)$. Moreover, we know:

$$\vdash \{\varphi \wedge b_1 \wedge b_2\}\, d_1 \otimes d_2\, \{\varphi \wedge b_1 = b_2\}$$

Since $\varphi \wedge b_1 = b_2 \wedge b_1 \Rightarrow \varphi \wedge b_1 \wedge b_2$, using (SUB) we obtain:

$$\vdash \{\varphi \wedge b_1 = b_2 \wedge b_1\}\, d_1 \otimes d_2\, \{\varphi \wedge b_1 = b_2\}$$

Applying (WHILE) we get:

$$\vdash \{\varphi \wedge b_1 = b_2\}\, \mathsf{while}\ b_1\ \mathsf{do}\ (d_1 \otimes d_2)\, \{\varphi \wedge b_1 = b_2 \wedge \neg\, b_1\}$$

Also $\varphi \wedge b_1 = b_2 \wedge \neg\, b_1 \Rightarrow \varphi \wedge \neg\, b_1 \wedge \neg\, b_2$ so again, using (SUB) we get:

$$\vdash \{\varphi \wedge b_1 = b_2\}\, \mathsf{while}\ b_1\ \mathsf{do}\ (d_1 \otimes d_2)\, \{\varphi \wedge \neg\, b_1 \wedge \neg\, b_2\}$$

which is the desired result.

- Case (R-SEQ). We have that $c_1 \equiv d_1; e_1$, $c_2 \equiv d_2; e_2$ and we know that:

$$\vdash \{\varphi\}d_1 \sim d_2\{\phi\}$$
$$\vdash \{\phi\}e_1 \sim e_2\{\psi\}$$

By induction hypotheses we know that $d_1 \otimes d_2$ and $e_1 \otimes e_2$ are defined, so we take $c_1 \otimes c_2 = d_1 \otimes d_2; e_1 \otimes e_2$. Moreover, we have:

$$\vdash \{\varphi\}\, d_1 \otimes d_2\, \{\phi\}$$
$$\vdash \{\phi\}\, e_1 \otimes e_2\, \{\psi\}$$

Applying (SEQ) we conclude:

$$\vdash \{\varphi\}\, d_1 \otimes d_2; e_1 \otimes e_2\, \{\psi\}$$

- Case (R-COMPOSE). We have that:

$$\vdash \{\exists X.\varphi \wedge \varphi'\}c_1 \sim c_3\{\exists X.\psi \wedge \psi'\}$$

because:

$$\vdash \{\varphi\}c_1 \sim c_2\{\psi\}$$
$$\vdash \{\varphi'\}c_2 \sim c_3\{\psi'\}$$

with $X \subseteq \mathsf{var}(c_2)$ and c_2 terminates. By induction hypotheses we have:

$$\vdash \{\varphi\}\, c_1 \otimes c_2\, \{\psi\}$$
$$\vdash \{\varphi'\}\, c_2 \otimes c_3\, \{\psi'\}$$

The result follows from application of Lemma 4.

Proof of Proposition 2. If $\{\varphi\}\, c_1 \sim c_2 \longrightarrow c\,\{\psi\}$ then $\vdash \{\varphi\}\, c\,\{\psi\}$ and $\vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$.

Proof. The proof is done by induction on the derivation of the loose product $\{\varphi\}\, c_1 \sim c_2 \longrightarrow c\,\{\psi\}$ with case analysis on the last rule applied.

- Case (LP-SEQ). We have that $c_1 \equiv d_1; e_1$, $c_2 \equiv d_2; e_2$ and $c \equiv d; e$ with:

$$\{\varphi\}\, d_1 \sim d_2 \longrightarrow d\,\{\phi\}$$
$$\{\phi\}\, e_1 \sim e_2 \longrightarrow e\,\{\psi\}$$

  By induction hypotheses, left conjunct, we obtain:

$$\vdash \{\varphi\}\, d\,\{\phi\}$$
$$\vdash \{\phi\}\, e\,\{\psi\}$$

  Applying rule (SEQ) we obtain:

$$\vdash \{\varphi\}\, d; e\,\{\psi\}$$

  By induction hypotheses, right conjunct, we obtain:

$$\vDash \{\varphi\} d_1 \sim d_2 \{\phi\}$$
$$\vDash \{\phi\} e_1 \sim e_2 \{\psi\}$$

  Which by definition means that:

  1. for all $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ such that $(\sigma_1 \uplus \sigma_2) \in [\![\varphi]\!]$, $(\sigma_1, \sigma_1') \in [\![d_1]\!]$ and $(\sigma_2, \sigma_2') \in [\![d_2]\!]$ then it is the case that $(\sigma_1' \uplus \sigma_2') \in [\![\phi]\!]$,
  2. for all $\sigma_1', \sigma_2', \sigma_1'', \sigma_2''$ such that $(\sigma_1' \uplus \sigma_2') \in [\![\phi]\!]$, $(\sigma_1', \sigma_1'') \in [\![e_1]\!]$ and $(\sigma_2', \sigma_2'') \in [\![e_2]\!]$ then it is the case that $(\sigma_1'' \uplus \sigma_2'') \in [\![\psi]\!]$.

  We have to show that, given $\sigma_1, \sigma_2, \sigma_1'', \sigma_2''$ such that $(\sigma_1 \uplus \sigma_2) \in [\![\varphi]\!]$, $(\sigma_1, \sigma_1'') \in [\![d_1; e_1]\!]$ and $(\sigma_2, \sigma_2'') \in [\![d_2; e_2]\!]$ then $(\sigma_1'' \uplus \sigma_2'') \in [\![\psi]\!]$.
  But note that $(\sigma_1, \sigma_1'') \in [\![d_1; e_1]\!]$ implies that there is σ_1' such that $(\sigma_1, \sigma_1') \in [\![d_1]\!]$ and $(\sigma_1', \sigma_1'') \in [\![e_1]\!]$. Reasoning similarly, $(\sigma_2, \sigma_2'') \in [\![d_2; e_2]\!]$ implies that there is σ_2' such that $(\sigma_2, \sigma_2') \in [\![d_2]\!]$ and $(\sigma_2', \sigma_2'') \in [\![e_2]\!]$.
  The result follows by applying the two right inductive hypotheses, i.e. 1 and 2.

- Case (LP-WHILE). We have that $c_1 \equiv \mathsf{while}\ b_1\ \mathsf{do}\ d_2$, $c_2 \equiv \mathsf{while}\ b_2\ \mathsf{do}\ d_2$ and $c \equiv \mathsf{while}\ b_1\ \mathsf{do}\ d$ with:

$$\{\varphi \wedge b_1\}\, d_1 \sim d_2 \longrightarrow d\,\{\varphi\}$$
$$\varphi \Rightarrow b_1 = b_2$$

  By inductive hypothesis, we get:

$$\vdash \{\varphi \wedge b_1\}\, d\,\{\varphi\}$$
$$\vDash \{\varphi \wedge b_1\} d_1 \sim d_2 \{\varphi\}$$

  Using the former and (WHILE) we conclude:

$$\vdash \{\varphi\}\, \mathsf{while}\ b_1\ \mathsf{do}\ d\,\{\varphi \wedge \neg\, b_1\}$$

  By definition of the latter, we have that for all $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ if $\sigma_1 \uplus \sigma_2 \in [\![\varphi \wedge b_1]\!]$, $(\sigma_1, \sigma_1') \in [\![d_1]\!]$ and $(\sigma_2, \sigma_2') \in [\![d_2]\!]$ then $\sigma_1 \uplus \sigma_2 \in [\![\varphi \wedge b_1]\!]$.
  Now, assume we have:

$$\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!] \tag{1}$$
$$(\sigma_1, \sigma_1') \in [\![\mathsf{while}\ b_1\ \mathsf{do}\ d_1]\!] \tag{2}$$
$$(\sigma_2, \sigma_2') \in [\![\mathsf{while}\ b_2\ \mathsf{do}\ d_2]\!] \tag{3}$$

  and we have to show $\sigma_1' \uplus \sigma_2' \in [\![\varphi]\!]$. We will do so by simultaneous induction on 2 and 3. Since $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and $\varphi \Rightarrow b_1 = b_2$ then we have $\sigma_1 \in [\![b_1]\!]$ iff $\sigma_2 \in [\![b_2]\!]$. Hence we only need to account for 2 cases:

  - $\sigma_1 \in [\![\neg\, b_1]\!]$, $\sigma_2 \in [\![\neg\, b_2]\!]$, $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$. Trivially, we conclude $\sigma_1' \uplus \sigma_2' \in [\![\varphi \wedge \neg\, b_1]\!]$.
  - We have the following:

$$\sigma_1 \in [\![b_1]\!] \tag{4}$$
$$\sigma_2 \in [\![b_2]\!] \tag{5}$$
$$(\sigma_1, \sigma_1^*) \in [\![d_1]\!] \tag{6}$$
$$(\sigma_2, \sigma_2^*) \in [\![d_2]\!] \tag{7}$$
$$(\sigma_1^*, \sigma_1') \in [\![\mathsf{while}\ b_1\ \mathsf{do}\ d_1]\!] \tag{8}$$
$$(\sigma_2^*, \sigma_2') \in [\![\mathsf{while}\ b_2\ \mathsf{do}\ d_2]\!] \tag{9}$$

    By Inductive Hypothesis, we can conclude $(\sigma_1' \uplus \sigma_2') \in [\![\varphi \wedge \neg\, b_1]\!]$, provided we show that $(\sigma_1^* \uplus \sigma_2^*) \in [\![\varphi]\!]$. This follows from the second part of the outer inductive hypothesis, 1, 4, 5, 6 and 7.

- Case (LP-SELFCOMP). We have that $c \equiv c_1; c_2$ and we know that:

$$\vdash \{\varphi\}\, c_1; c_2\,\{\psi\} \tag{10}$$

  which concludes the first part.
  For the second part, assume we have:

$$\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!] \tag{11}$$
$$(\sigma_1, \sigma_1') \in [\![c_1]\!] \tag{12}$$
$$(\sigma_2, \sigma_2') \in [\![c_2]\!] \tag{13}$$

  and we have to show $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$. From 10 and soundness of Hoare Logic, we know that for every state σ, σ' such that $\sigma \in [\![\varphi]\!]$ and $(\sigma, \sigma') \in [\![c_1; c_2]\!]$ then $\sigma' \in [\![\psi]\!]$. Now, take $\sigma = \sigma_1 \uplus \sigma_2$ and $\sigma' = \sigma_1' \uplus \sigma_2'$ then it only remains to be shown that:

$$(\sigma_1 \uplus \sigma_2, \sigma_1' \uplus \sigma_2') \in [\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!]$$

  It suffices to show that

$$(\sigma_1 \uplus \sigma_2, \sigma_1' \uplus \sigma_2) \in [\![c_1]\!]$$
$$(\sigma_1' \uplus \sigma_2, \sigma_1' \uplus \sigma_2') \in [\![c_2]\!]$$

  which follows from 12,13 and disjointness of program variables.

Lemma 5. Assume $(c, \sigma) \Downarrow^*$ then

- if $c \equiv \mathsf{while}\ b\ \mathsf{do}\ d$ then $(d, \sigma) \Downarrow^*$.
- if $c \equiv \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$ then $(d, \sigma) \Downarrow^*$ and $(e, \sigma) \Downarrow^*$.
- if $c \equiv d \oplus e$ then $(d, \sigma) \Downarrow^*$ and $(e, \sigma) \Downarrow^*$.

Proof of Proposition 3. Assume $\{\varphi\}\, c_1 \sim c_2 \longrightarrow c\,\{\psi\}$ and suppose given states $\sigma_1, \sigma_2 \in \mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$.

- If $(c_2, \sigma_2) \Downarrow^\star$, then for every $\sigma_1' \in \mathcal{S}$ s.t. $(\sigma_1, \sigma_1') \in [\![c_1]\!]$, there exists $\sigma_2' \in \mathcal{S}$ s.t. $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$ and $(\sigma_2, \sigma_2') \in [\![c_2]\!]$;
- If $(c_1, \sigma_1) \Downarrow^\star$, then for every $\sigma_2' \in \mathcal{S}$ s.t. $(\sigma_2, \sigma_2') \in [\![c_2]\!]$, there exists $\sigma_1' \in \mathcal{S}$ s.t. $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$ and $(\sigma_1, \sigma_1') \in [\![c_1]\!]$

Proof. We will only show the first part since the other one is symmetric. The proof follows by induction on the judgment $\{\varphi\}\, c_1 \sim c_2 \longrightarrow c\,\{\psi\}$ with case analysis on the last applied rule. We only show selected cases.

- Case (LP-NONDET). We have $c_1 \equiv d_1 \oplus e_1$, $c_2 \equiv d_2 \oplus e_2$ and $c \equiv d \oplus e$ where:

$$\{\varphi\}\, d_1 \sim d_2 \longrightarrow d\,\{\psi\} \tag{14}$$
$$\{\varphi\}\, e_1 \sim e_2 \longrightarrow e\,\{\psi\} \tag{15}$$

  Assume we have σ_1, σ_2 such that:

$$\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!] \tag{16}$$
$$(d_2 \oplus e_2, \sigma_2) \Downarrow^* \tag{17}$$

From the latter, by Lemma 5 we have $(d_2, \sigma_2) \Downarrow^*$ and $(e_2, \sigma_2) \Downarrow^*$. Using this and IH's we know that:

- for every σ_1' such that $(\sigma_1, \sigma_1') \in [\![d_1]\!]$ there is σ_2' such that $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$ and $(\sigma_2, \sigma_2') \in [\![d_2]\!]$
- for every σ_1' such that $(\sigma_1, \sigma_1') \in [\![e_1]\!]$ there is σ_2' such that $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$ and $(\sigma_2, \sigma_2') \in [\![e_2]\!]$

Suppose we have σ_1' such that $(\sigma_1, \sigma_1') \in [\![d_1 \oplus e_1]\!]$. Then there are two cases:

- $(\sigma_1, \sigma_1') \in [\![d_1]\!]$. In this case we apply the IH and conclude that there is σ_2' such that $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$ and $(\sigma_2, \sigma_2') \in [\![d_2]\!]$. We still have to show that $(\sigma_2, \sigma_2') \in [\![d_2 \oplus e_2]\!]$, but this is trivial.
- $(\sigma_1, \sigma_1') \in [\![e_1]\!]$ is symmetric.

– Case (LP-WHILE). We have $c_1 \equiv \mathsf{while}\ b_1\ \mathsf{do}\ d_1$, $c_2 \equiv \mathsf{while}\ b_2\ \mathsf{do}\ d_2$ and $c \equiv \mathsf{while}\ b_1\ \mathsf{do}\ d$ where:

$$\{\varphi \wedge b_1\}\, d_1 \sim d_2 \longrightarrow d\, \{\varphi\}$$
$$\varphi \Rightarrow b_1 = b_2$$

We also know that there is σ_2 such that $(\mathsf{while}\ b_2\ \mathsf{do}\ d_2, \sigma_2) \Downarrow^*$. Using this and Lemma 5 we know $(d_2, \sigma_2) \Downarrow^*$.
By induction hypothesis (IH1), we know that for every σ_1 such that $\sigma_1 \uplus \sigma_2 \in [\![\varphi \wedge b_1]\!]$ and for every σ_1' such that $\sigma_1, \sigma_1' \in [\![d_1]\!]$ there is a σ_2' such that $(\sigma_2, \sigma_2') \in [\![d_2]\!]$ and $\sigma_1' \uplus \sigma_2' \in [\![\varphi]\!]$.
Now, assume we have σ_1 such that $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and σ_1' such that $(\sigma_1, \sigma_1') \in [\![\mathsf{while}\ b_1\ \mathsf{do}\ d_1]\!]$. We need to prove that for every σ_2 such that $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$, there exists σ_2' such that $(\sigma_2, \sigma_2') \in [\![\mathsf{while}\ b_2\ \mathsf{do}\ d_2]\!]$. We will do so by induction on the execution $(\sigma_1, \sigma_1') \in [\![\mathsf{while}\ b_1\ \mathsf{do}\ d_1]\!]$.

- We have $\sigma_1 = \sigma_1'$ and $\sigma_1 \uplus \sigma_2 \in [\![\neg b_1]\!]$. In this case there is σ_2. We have to show $\sigma_1 \uplus \sigma_2 \in [\![\varphi \wedge \neg b_1]\!]$, which follows trivially and $(\sigma_2, \sigma_2) \in [\![\mathsf{while}\ b_2\ \mathsf{do}\ d_2]\!]$, which follows from the former and $\varphi \Rightarrow b_1 = b_2$.
- We have:

$$\sigma_1 \uplus \sigma_2 \in [\![b_1]\!]$$
$$(\sigma_1^*, \sigma_1') \in [\![\mathsf{while}\ b_1\ \mathsf{do}\ d_1]\!]$$
$$(\sigma_1', \sigma_2') \in [\![\varphi \wedge \neg\, b_1]\!]$$
$$(\sigma_1, \sigma_1^*) \in [\![d_1]\!]$$

  Also we know by induction hypothesis (IH2), for all σ_2^* such that $\sigma_1^* \uplus \sigma_2^* \in [\![\varphi]\!]$ there is σ_2' such that $\sigma_2^*, \sigma_2' \in [\![\mathsf{while}\ b_2\ \mathsf{do}\ d_2]\!]$.
  We also have that $\sigma_1 \uplus \sigma_2 \in [\![\varphi \wedge b_1]\!]$. By outer induction hypothesis (IH1), we know that there is σ_2^* such that $(\sigma_2, \sigma_2^*) \in [\![d_2]\!]$ and $\sigma_1^* \uplus \sigma_2^* \in [\![\varphi]\!]$. Applying the inner induction hypothesis (IH2) to σ_2^* we get that there is σ_2' such that $(\sigma_2^*, \sigma_2') \in [\![\mathsf{while}\ b_2\ \mathsf{do}\ d_2]\!]$ and $\sigma_1' \uplus \sigma_2' \in [\![\varphi \wedge \neg\, b_1]\!]$. The result follows using σ_2'.

– Case (LP-SELFCOMP). We have $c \equiv c_1; c_2$ and we know that:

$$\{\varphi\}\, c_1; c_2\, \{\psi\}$$

From this, by soundness of Hoare Logic, we know that for any σ_1, σ_2, σ_1' and σ_2' such that $(\sigma_1, \sigma_1') \in [\![c_1]\!]$, $(\sigma_2, \sigma_2') \in [\![c_2]\!]$ and $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ then $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.
Assume we have σ_1, σ_2 such that:

$$\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$$
$$(c_2, \sigma_2) \Downarrow^*$$

From the latter we conclude that there is σ_2' such that $(\sigma_2, \sigma_2') \in [\![c_2]\!]$. It remains to be shown that $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$, which follows from soundness.

Definition 9 (Security Policy). A security policy Γ is a mapping associating labels in $\{L, H\}$ to variables.

Definition 10. Given a security policy Γ we define a state predicate characterizing equality of low variables as follows:

$$\mathsf{LowEq} \stackrel{def}{=} x_1 = x_1' \wedge \ldots \wedge x_n = x_n'$$

where $\Gamma(x_i) = L$ for $i \in [1 \ldots n]$.

Lemma 6. Let x be an identifier, Γ a security policy and τ a label then if $\vdash x : \tau\ var$ then $\Gamma(x) = \tau$.

Proof. Straightforward.

Lemma 7. Let e be an expression, Γ a security policy and τ a label then for every σ, σ' such that $(\sigma_1, \sigma_2) \in [\![\mathsf{LowEq}]\!]$, if $\vdash e : L$ then $(\sigma_1, \sigma_2) \in [\![e = e']\!]$.

Proof. Straightforward induction on the derivation of the judgment.

Lemma 8. Let c be a command, Γ a security policy, if $\vdash c : H\ cmd$ and for all $x \in FV(\varphi), \Gamma(x) = L$ then $\vdash \{\varphi\}\, c\, \{\varphi\}$

Proof. Straightforward induction on the derivation of the judgment.

Proof of Proposition 4. Let c be a command, Γ a security policy and τ a label, if $\vdash c : \tau\ cmd$ then there exists c'' such that the judgment $\{\mathsf{LowEq}\}\, c \sim c' \longrightarrow c''\, \{\mathsf{LowEq}\}$ holds.

Proof. The proof follows by induction on the derivation of $\vdash c : \tau\ cmd$ with case analysis on the last applied rule. We only cover selected cases.

– Case NI-WHILE. We have $c \equiv \mathsf{while}\ b\ \mathsf{do}\ d$, $c' \equiv \mathsf{while}\ b'\ \mathsf{do}\ d'$ with $\vdash b : \tau$ and $\vdash d : \tau\ cmd$. By renaming also $\vdash b' : \tau$ and $\vdash d' : \tau\ cmd$. The proof follows by case analysis on τ.

- Case $\tau \equiv H$. We take $c'' \equiv \mathsf{while}\ b\ \mathsf{do}\ d; \mathsf{while}\ b'\ \mathsf{do}\ d'$. We apply Lemma 8 and we obtain $\vdash \{\mathsf{LowEq}\}\ \mathsf{while}\ b\ \mathsf{do}\ d\, \{\mathsf{LowEq}\}$ and $\vdash \{\mathsf{LowEq}\}\ \mathsf{while}\ b'\ \mathsf{do}\ d'\, \{\mathsf{LowEq}\}$. Using (LP-SELFCOMP) we conclude:

$$\{\mathsf{LowEq}\}\mathsf{while}\ b\ \mathsf{do}\ d \sim \mathsf{while}\ b'\ \mathsf{do}\ d'$$
$$\longrightarrow \mathsf{while}\ b\ \mathsf{do}\ d; \mathsf{while}\ b'\ \mathsf{do}\ d'\{\mathsf{LowEq}\}$$

- Case $\tau \equiv L$. Using Lemma 7, we know $\mathsf{LowEq} \Rightarrow b = b'$. Also, by inductive hypothesis we know there is d'' such that $\{\mathsf{LowEq}\}\, d \sim d' \longrightarrow d''\, \{\mathsf{LowEq}\}$. Take $c'' \equiv \mathsf{while}\ b\ \mathsf{do}\ d''$. But note that $\mathsf{LowEq} \wedge\ b \Rightarrow \mathsf{LowEq}$. From this, using (LP-SUB) we get $\{\mathsf{LowEq} \wedge b\}\, d \sim d' \longrightarrow d''\, \{\mathsf{LowEq}\}$. Using (LP-WHILE) we obtain:

$$\{\mathsf{LowEq}\}\mathsf{while}\ b\ \mathsf{do}\ d \sim \mathsf{while}\ b'\ \mathsf{do}\ d' \longrightarrow$$
$$\mathsf{while}\ b'\ \mathsf{do}\ d''\{\mathsf{LowEq} \wedge \neg\, b\}$$

  and since $\mathsf{LowEq} \wedge \neg\, b \Rightarrow \mathsf{LowEq}$, we obtain

$$\{\mathsf{LowEq}\}\mathsf{while}\ b\ \mathsf{do}\ d \sim \mathsf{while}\ b'\ \mathsf{do}\ d' \longrightarrow$$
$$\mathsf{while}\ b'\ \mathsf{do}\ d''\{\mathsf{LowEq}\}$$

  using (LP-SUB) to conclude.

– Case (NI-ASSG). We have $c \equiv x := e$, $c' \equiv x' := e'$. Take $c'' \equiv x := e; x' := e'$. We can conclude applying (LP-SELFCOMP) provided we show:

$$\vdash \{\mathsf{LowEq}\}\, x := e; x' := e'\, \{\mathsf{LowEq}\}$$

We proceed by case analysis on τ.

- Case $\tau \equiv H$. We conclude using Lemma 8 and the fact that for all $x \in FV(\mathsf{LowEq})$, it is the case that $x : L\ var$.

- Case $\tau \equiv L$. From this we know $x : L\,var$ and $e : L$. Hence $\mathsf{LowEq} \equiv \mathsf{LowEq}' \wedge x = x'$. Also, using Lemma 7 we know $\mathsf{LowEq}' \Rightarrow e = e'$. So we have $\mathsf{LowEq} \Rightarrow \mathsf{LowEq}' \wedge e = e'$. Using (LP-SUB), it suffices to show:

$$\vdash \{\mathsf{LowEq}' \wedge \mathsf{e} = \mathsf{e}'\}\, x := e; x' := e' \,\{\mathsf{LowEq}' \wedge \mathsf{x} = \mathsf{x}'\}$$

By applying (SEQ) we have to show:

$$\vdash \{\mathsf{LowEq}' \wedge \mathsf{e} = \mathsf{e}'\}\, x := e \,\{\mathsf{LowEq}' \wedge \mathsf{x} = \mathsf{e}'\}$$
$$\vdash \{\mathsf{LowEq}' \wedge \mathsf{x} = \mathsf{e}'\}\, x' := e' \,\{\mathsf{LowEq}' \wedge \mathsf{x} = \mathsf{x}'\}$$

both of which follow from (ASSIGN).

– Case (NI-NONDET). We have $c \equiv d \oplus e$ and $c' \equiv d' \oplus e'$ with

$$\vdash d : \tau\ cmd$$
$$\vdash e : \tau\ cmd$$

By inductive hypotheses, we know there are d'' and e'' such that:

$$\{\mathsf{LowEq}\}\, d \sim d' \longrightarrow d'' \,\{\mathsf{LowEq}\}$$
$$\{\mathsf{LowEq}\}\, e \sim e' \longrightarrow e'' \,\{\mathsf{LowEq}\}$$

We take $c'' \equiv d'' \oplus e''$. We obtain:

$$\{\mathsf{LowEq}\}\, d \oplus e \sim d' \oplus e' \longrightarrow d'' \oplus e'' \,\{\mathsf{LowEq}\}$$

by applying (LP-NONDET).

Proofs from Section 3

Left products

We first show that $P \in P_1 \ltimes_\varphi P_2$ can emulate the behavior of one step of P_1.

Lemma 9 (Step lifting for left products). Assume $P \in P_1 \ltimes_\varphi P_2$. Let t be a φ-trace that ends in a configuration $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2\rangle$ and let $t_2 \cdot \langle l_2, \sigma_2\rangle \cdot t_2'$ be a P_2-run. For every P_1-step $\langle l_1, \sigma_1\rangle \rightsquigarrow \langle l_1', \sigma_1'\rangle$, there exists a φ-trace t' extending t s.t. $\pi_1(t') = \pi_1(t) \cdot \langle l_1', \sigma_1'\rangle$. Moreover, if $P \in P_1 \bowtie_\varphi P_2$ then one can choose t' s.t. $t' = t \cdot t''$ with $\pi_2(t'')$ a prefix of t_2'.

Proof. The proof proceeds by induction on the length of t_2'. For the base case, we have $l_2 = \mathsf{out}_2$. By repleteness, semantics coherence, and definition of separable statement $\langle (l_1, \mathsf{out}_2), \sigma_1 \uplus \sigma_2\rangle \rightsquigarrow \langle (l_1', \mathsf{out}_2), \sigma_1' \uplus \sigma_2\rangle$ and hence $t \cdot \langle (l_1', \mathsf{out}_2), \sigma_1' \uplus \sigma_2\rangle$ is a P-trace with the expected properties. For the induction step, let $l_2'' \in \mathcal{N}_2$ and $\sigma_2'' \in \mathcal{S}$ s.t. $t_2' = \langle l_2'', \sigma_2''\rangle \cdot t_2''$. We proceed by a case analysis on the definition of left product:

- $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2)$: then we conclude as in the base case;
- $(l_1, l_2) \Mapsto (l_1', l_2')$ and there exist $l_2' \in \mathcal{N}_2$ and $\sigma_2' \in \mathcal{S}$ s.t. $\langle l_2, \sigma_2\rangle \rightsquigarrow \langle l_2', \sigma_2'\rangle$: then $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2\rangle \rightsquigarrow \langle (l_1', l_2'), \sigma_1' \uplus \sigma_2'\rangle$ by definition of product and separable statements. We set $t' = t \cdot \langle (l_1', l_2'), \sigma_1' \uplus \sigma_2'\rangle$. If $P \in P_1 \bowtie_\varphi P_2$ then $l_2' = l_2''$ and $\pi_2(\langle (l_1', l_2'), \sigma_1' \uplus \sigma_2'\rangle)$ is a prefix of t_2';
- $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l_2'')$: then $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2\rangle \rightsquigarrow \langle (l_1, l_2''), \sigma_1 \uplus \sigma_2''\rangle$ by definition of product and separable statements. Now $t'' = t \cdot \langle (l_1, l_2''), \sigma_1 \uplus \sigma_2''\rangle$ is a φ-trace and we can apply the I.H. to conclude there exists a φ-trace t^{ind} extending t'' and ending in $\langle (l_1', l_2^{ind}), \sigma_1' \uplus \sigma_2^{ind}\rangle$. Moreover, if $P \in P_1 \bowtie_\varphi P_2$ then $t^{ind} = t \cdot \langle (l_1, l_2''), \sigma_1 \uplus \sigma_2''\rangle \cdot t_p$ for some t_p s.t. $\pi(t_p)$ is a prefix of t_2''. We set $t' = t^{ind}$ to conclude.

We now generalize Lemma 9 to show that P can emulate the behavior of many steps of P_1.

Lemma 10 (Trace lifting for left products). Assume $P \in P_1 \ltimes_\varphi P_2$. Let t be a φ-trace that ends in a configuration $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2\rangle$ s.t. $\langle l_2, \sigma_2\rangle \Downarrow$. For every P_1-trace t_1 extending $\pi_1(t)$, there exists a φ-trace t' extending t s.t. $\pi_1(t') = t_1$.

Proof. The proof proceeds by induction on the length of t_1' where $t_0 = t_1 \cdot t_1'$. The base case is immediate. For the induction step, assume $t_1' = t_1'' \cdot \langle l_1', \sigma_1'\rangle$. By I.H. there exists a trace t'' extending t s.t. $\pi_1(t'') = t_1''$. We then conclude by applying Lemma 9.

Proof of Proposition 6. Assume $P \in P_1 \ltimes_\varphi P_2$. Let t_1 be a P_1-run with initial state σ_1, and let $\sigma_2 \in \mathcal{S}$ s.t. $\langle \mathsf{in}_2, \sigma_2\rangle \Downarrow^\star$ and $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$. Then there exists a P-run t with initial configuration $\langle (\mathsf{in}_1, \mathsf{in}_2), \sigma_1 \uplus \sigma_2\rangle$ s.t. $\pi_1(t) = t_1$.

Proof. This proposition follows by application of Lemma 10. Let $t = \langle (\mathsf{in}_1, \mathsf{in}_2), \sigma_1 \uplus \sigma_2\rangle$ a φ-trace. Since t_1 extends $\pi_1(t)$, from 10, there exists a φ-trace t' extending t (i.e., with initial configuration $\langle (\mathsf{in}_1, \mathsf{in}_2), \sigma_1 \uplus \sigma_2\rangle$) such that $\pi_1(t') = t_1$.

Lemma 11 (Step-lifting for full products). Assume $P \in P_1 \bowtie_\varphi P_2$. Let $t_1 \cdot \langle l_1, \sigma_1\rangle \cdot t_1'$ be a P_1-run with initial state $\hat\sigma_1$ and $t_2 \cdot \langle l_2, \sigma_2\rangle \cdot t_2'$ be a P_2-run with initial state $\hat\sigma_2$. Suppose $\hat\sigma_1 \uplus \hat\sigma_2 \in [\![\varphi]\!]$. Then, for all φ-trace t s.t. $\pi_1(t) = t_1$ and $\pi_2(t) = t_2$ there is a trace $t \cdot t'$ s.t.:

- $\pi_1(t') = t_1''$ for some non-empty t_1'' prefix of $\langle l_1, \sigma_1\rangle \cdot t_1'$
- $\pi_2(t') = t_2''$ for some non-empty t_2'' prefix of $\langle l_2, \sigma_2\rangle \cdot t_2'$

Proof. Let $\langle (l_1', l_2'), \sigma_1' \uplus \sigma_2'\rangle$ the final configuration of t, then we have $\langle l_1', \sigma_1'\rangle \rightsquigarrow \langle l_1, \sigma_1\rangle$ and $\langle l_2', \sigma_2'\rangle \rightsquigarrow \langle l_2, \sigma_2\rangle$. Consider the three cases on the definition of full products:

- if $(l_1', l_2') \Mapsto (l_1, l_2)$ then we are done with the proof by setting $t' = t \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2\rangle$;
- if $(l_1', l_2') \overset{\mathsf{l}}{\mapsto} (l_1, l_2')$ then $t \cdot \langle (l_1, l_2'), \sigma_1 \uplus \sigma_2'\rangle$ is a trace extending t. Since P is a right product, from the symmetricity of Lemma 9 there is a trace t'' extending $t \cdot \langle (l_1, l_2'), \sigma_1 \uplus \sigma_2'\rangle$ such that $\pi_2(t'') = t_2 \cdot \langle l_2, \sigma_2\rangle$ and $\pi_1(t'')$ is a prefix of $t_1 \cdot \langle l_1, \sigma_1\rangle \cdot t_1'$;
- if $(l_1', l_2') \overset{\mathsf{r}}{\mapsto} (l_1', l_2)$ then the reasoning is symmetric to the previous case.

Lemma 12 (Trace lifting for full products). Assume $P \in P_1 \bowtie_\varphi P_2$. Let t be a φ-trace that ends in a configuration $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2\rangle$ s.t. $\langle l_1, \sigma_1\rangle \Downarrow$ and $\langle l_2, \sigma_2\rangle \Downarrow$. For every P_1-run t_1 extending $\pi_1(t)$ and every P_2-run t_2 extending $\pi_2(t)$ there is a trace t' extending t s.t. $\pi_1(t')$ extends t_1 and $\pi_2(t')$ extends t_2.

Proof. The proof follows by induction on the length of t_1' plus the length of t_2', where $t_1 = \pi_1(t) \cdot t_1'$ and $t_2 = \pi_2(t) \cdot t_2'$. If both t_1' and t_2' are empty traces the lemma follows trivially. For the inductive case, if one of t_1' and t_2' is an empty trace the proof follows by Lemma 9. Consider now the case in which $t_1' = \langle l_1', \sigma_1'\rangle \cdot t_1''$ and $t_2' = \langle l_2', \sigma_2'\rangle \cdot t_2''$. From Lemma 11, there is a ϕ-trace $t \cdot \hat t$ s.t.:

- $\pi_1(\hat t) = t_1'''$ for some non-empty t_1''' extended by $\langle l_1', \sigma_1'\rangle \cdot t_1''$
- $\pi_2(\hat t) = t_2'''$ for some non-empty t_2''' extended by $\langle l_2', \sigma_2'\rangle \cdot t_2''$

By I.H. applied to t_1'''' and t_2'''' s.t. $t_1''' \cdot t_1'''' = \langle l_1', \sigma_1'\rangle \cdot t_1''$ and $t_2''' \cdot t_2'''' = \langle l_2', \sigma_2'\rangle \cdot t_2''$ we have a P-trace t', s.t. $\pi_1(t')$ extends $t_1 \cdot t_1'''$ (and thus t_1) and $\pi_2(t')$ extends $t_2 \cdot t_2'''$ (and thus t_2).

Proof of Proposition 5. Assume $P \in P_1 \bowtie_\varphi P_2$. Let t_1 be a P_1-run with initial state σ_1 and t_2 be P_2-run with initial state σ_2. Suppose $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$. Then there exists a P-run t s.t. $\pi_1(t) = t_1$ and $\pi_2(t) = t_2$.

Proof. The proposition follows from Lemma 12, by taking t as the φ-trace $\langle (\mathsf{in}_1, \mathsf{in}_2), \sigma_1 \uplus \sigma_2\rangle$.

Composing left products

In the following we prove that the composition of left products $P_a \in P_1 \ltimes_{\varphi_a} P_2$ and $P_b \in P_2 \ltimes_{\varphi_b} P_3$ is a left product. We first prove that traces of the product $P = P_a \circ P_b$ can be projected onto traces of P_a and P_b.

Lemma 13. Let $P = P_a \circ P_b$ and $t \cdot \langle (l_1, l_3), \sigma_1 \uplus \sigma_3 \rangle$ a $\varphi_a \circ \varphi_b$-trace of P. Then, there exists a φ_a-trace $t_a \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle$ of P_a and a φ_b-trace $t_b \cdot \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle$, s.t. $\pi_1(t) = \pi_1(t_a)$, $\pi_2(t_a) = \pi_1(t_b)$, and $\pi_2(t_b) = \pi_2(t)$. Moreover, if $\langle \mathsf{in}_2, \hat{\sigma_2} \rangle \Downarrow^*$ ($\hat{\sigma_2}$ stands for the initial state of t_a and t_b), one can choose l_2 s.t. there is no $l'_2 \in \mathcal{N}_2$ s.t. $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l'_2)$ and $(l_2, l_3) \overset{\mathsf{l}}{\mapsto} (l'_2, l_3)$ hold simultaneously.

Proof. The proof proceeds by induction on the length of the P-trace t. For $t = \epsilon$ is trivial. For the inductive step, consider a trace of the form $t \cdot \langle (l_1, l_3), \sigma_1 \uplus \sigma_3 \rangle \cdot \langle (l'_1, l'_3), \sigma'_1 \uplus \sigma'_3 \rangle$. By I.H., there exists a trace $t_a \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle$ and a trace $t_b \cdot \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle$. We proceed by case analysis on the last edge of the trace:

- If $(l_1, l_3) \Mapsto (l'_1, l'_3)$, then we have $(l_1, l_2) \Mapsto (l'_1, l'_2) \in \mathcal{E}_a$ and $(l_2, l_3) \Mapsto (l'_2, l'_3) \in \mathcal{E}_b$ by definition of $P_a \circ P_b$. Since P_a is left and minimal, we have that $\langle l_2, \sigma_2 \rangle \rightsquigarrow \langle l'_2, \sigma'_2 \rangle$ for some σ'_2. Then, one can build the traces
$$t_a \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \cdot \langle (l'_1, l'_2), \sigma'_1 \uplus \sigma'_2 \rangle$$
and
$$t_b \cdot \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle \cdot \langle (l'_2, l'_3), \sigma'_2 \uplus \sigma'_3 \rangle$$
for the products P_a and P_b, respectively.
- If $(l_1, l_3) \overset{\mathsf{l}}{\mapsto} (l'_1, l_3)$, then by definition of $P_a \circ P_b$ we have $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l'_1, l_2) \in P_a$. The proof follows by choosing the traces
$$t_a \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \cdot \langle (l'_1, l_2), \sigma'_1 \uplus \sigma_2 \rangle$$
and $t_b \cdot \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle$ for the products P_a and P_b, respectively.
- If $(l_1, l_3) \overset{\mathsf{r}}{\mapsto} (l_1, l'_3)$, the proof is symmetric to the previous case.

To conclude, we must show that one can choose l_2 s.t. there is no $l'_2 \in \mathcal{N}_2$ s.t. $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l'_2)$ and $(l_2, l_3) \overset{\mathsf{l}}{\mapsto} (l'_2, l_3)$ hold simultaneously. Given the traces $t_a \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle$ and $t_b \cdot \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle$, from the termination hypothesis, we have that $\langle l_2, \sigma_2 \rangle \rightsquigarrow^\star \mathsf{out}_2$. We show that one can extend t_a and t_b, by induction on $\langle l_2, \sigma_2 \rangle \rightsquigarrow^\star \mathsf{out}_2$, in order to meet the condition required. For the base case, we have $l_2 = \mathsf{out}_2$ and then the requirement holds trivially (out_2 is a sink.) For the inductive case, consider $l'_2 \in \mathcal{N}_2$ and $\sigma_2 \in \mathcal{S}$ s.t. $\langle l_2, \sigma \rangle \rightsquigarrow \langle l'_2, \sigma'_2 \rangle$ and $\langle l'_2, \sigma'_2 \rangle \rightsquigarrow^\star \mathsf{out}_2$. If both $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l'_2)$ and $(l_2, l_3) \overset{\mathsf{l}}{\mapsto} (l'_2, l_3)$ hold, the lemma follows by application of the inductive hypothesis to the traces
$$t_a \cdot \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \cdot \langle (l_1, l'_2), \sigma_1 \uplus \sigma'_2 \rangle$$
and
$$t_b \cdot \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle \cdot \langle (l'_2, l_3), \sigma'_2 \uplus \sigma_3 \rangle$$
Otherwise, the lemma follows trivially.

We now prove that P is a left product w.r.t. $\varphi_a \circ \varphi_b$. Consider a node $(l_1, l_3) \in \mathcal{N}$ and a P-trace of the form $t \cdot \langle (l_1, l_3), \sigma_1 \uplus \sigma_3 \rangle$ s.t. $\langle l_1, \sigma_1 \rangle \rightsquigarrow \sigma'_1$ and $\langle l_3, \sigma_3 \rangle \rightsquigarrow \sigma'_3$. We must show that at least on of the three conditions required for P to be a left product is satisfied. Using the previous lemma, we can project the P-trace to a P_a-trace ending in $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle$ and a P_b trace ending in $\hat{\sigma_2} \uplus \hat{\sigma_3} \downarrow \langle (l_2, l_3), \sigma_2 \uplus \sigma_3 \rangle$, and s.t. $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l'_2)$ and $(l_2, l_3) \overset{\mathsf{l}}{\mapsto} (l'_2, l_3)$ do not hold simultaneously. We proceed by case analysis on the definition of left product applied to these two traces. All cases follow directly from the definitions of left products, minimality and compatibility, except:

1. $(l_1, l_2) \Mapsto (l'_1, l'_2)$, with $\langle l_2, \sigma \rangle \rightsquigarrow l'_2$ and $(l_2, l_3) \Mapsto (l'_2, l'_3)$, with $\langle l_3, \sigma_3 \rangle \rightsquigarrow l'_3$;
2. $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l'_2)$ and $(l_2, l_3) \overset{\mathsf{l}}{\mapsto} (l'_2, l_3)$.

In the first case, we have that $(l_1, l_3) \Mapsto (l'_1, l'_3)$ by definition of $P_a \circ P_b$. Since $\langle l_3, \sigma_3 \rangle \rightsquigarrow l'_3$, we conclude.

Proof from Section 4

Proof of Theorem 1. Let P_1, P_2 be deterministic separable programs and let φ, ψ be assertions. Then $\vDash \{\varphi\} P_1 \sim P_2 \{\psi\}$, provided there exists a partial specification Φ and a product program $P \in P_1 \times P_2$ s.t. $\varphi = \Phi(\mathsf{in}_1, \mathsf{in}_2)$ and $\psi = \Phi(\mathsf{out}_1, \mathsf{out}_2)$, and P is Φ-left and correct w.r.t. Φ.

Proof. P is left and correct w.r.t. Φ implies $P \in P_1 \ltimes_\varphi P_2$. Moreover, since P_2 is deterministic we have $P \in P_1 \bowtie_\varphi P_2$. Let t_1 and t_2 be runs of P_1 and P_2 with initial configurations $\langle \mathsf{in}_1, \sigma_1 \rangle$ and $\langle \mathsf{in}_2, \sigma_2 \rangle$. Assume that $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$. Then, from Proposition 5, we have that there is a P-run t s.t. $\pi_1(t) = t_1$ and $\pi_2(t) = t_2$. Since P is correct w.r.t. to Φ, we have $\sigma'_1 \uplus \sigma'_2 \in [\![\psi]\!]$, where $\langle \mathsf{out}_1, \sigma'_1 \rangle$ and $\langle \mathsf{out}_2, \sigma'_2 \rangle$ are the final configurations of t_1 and t_2.

Proof of Theorem 2. Let P_1, P_2 be separable programs and let φ, ψ be assertions. Assume that P_1 implies termination of P_2 w.r.t φ. Then $\{\varphi\} P_1 \mapsto P_2 \{\psi\}$ provided there exists a partial specification Φ s.t. $\varphi = \Phi(\mathsf{in}_1, \mathsf{in}_2)$ and $\psi = \Phi(\mathsf{out}_1, \mathsf{out}_2)$, and a product program $P \in P_1 \times P_2$ that is Φ-left and correct w.r.t. Φ.

Proof. If P is Φ-left and correct w.r.t. Φ then $P \in P_1 \ltimes_\Phi P_2$. Let $\sigma_1, \sigma_2, \sigma'_1, \sigma'_2$ be s.t. $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and $\langle \mathsf{in}_1, \sigma_1 \rangle \cdot t_1 \cdot \langle \mathsf{out}_1, \sigma'_1 \rangle$ be a P_1-run. From Proposition 6, there exists a P-run t with initial configuration $\langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle$ s.t. $\pi_1(t) = t_1$. From Lemma 2, $\pi_2(t)$ is a P_2-run with final configuration $\langle \mathsf{out}_2, \sigma'_2 \rangle$ where $\langle (\mathsf{out}_1, \mathsf{out}_2), \sigma'_1 \uplus \sigma'_2 \rangle$ is the final configuration of t. Since P is correct w.r.t. Φ we have $\sigma_1 \uplus \sigma'_2 \in [\![\psi]\!]$.