A Categorical Approach to Secure Compilation

Stelios Tsampas, Andreas Nuyts, Dominique Devriese, Frank Piessens

▶ **To cite this version:**

Stelios Tsampas, Andreas Nuyts, Dominique Devriese, Frank Piessens. A Categorical Approach to Secure Compilation. 15th International Workshop on Coalgebraic Methods in Computer Science (CMCS), Apr 2020, Dublin, Ireland. pp.155-179, 10.1007/978-3-030-57201-3_9 . hal-03232345

HAL Id: hal-03232345
https://inria.hal.science/hal-03232345

Submitted on 21 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A categorical approach to secure compilation

Stelios Tsampas[1], Andreas Nuyts[1], Dominique Devriese (Corresponding)[2], and Frank Piessens[1]

[1] KU Leuven, Leuven, Belgium name.surname@cs.kuleuven.be
[2] Vrije Universiteit Brussel, Brussels, Belgium dominique.devriese@vub.be

Abstract. We introduce a novel approach to secure compilation based on maps of distributive laws. We demonstrate through four examples that the coherence criterion for maps of distributive laws can potentially be a viable alternative for compiler security instead of full abstraction, which is the preservation and reflection of contextual equivalence. To that end, we also make use of the well-behavedness properties of distributive laws to construct a categorical argument for the contextual connotations of bisimilarity.

Keywords: Secure compilation · Distributive laws · Structural Operational Semantics

1 Introduction

As a field, secure compilation is the study of compilers that formally preserve abstractions across languages. Its roots can be tracked back to the seminal observation made by Abadi [1], namely that compilers which do not protect high-level abstractions against low-level contexts might introduce security vulnerabilities. But it was the advent of secure architectures like the Intel SGX [15] and an ever-increasing need for computer security that motivated researchers to eventually work on formally proving compiler security.

The most prominent [16, 18, 49, 35, 37, 32, 45] formal criterion for compiler security is *full abstraction*: A compiler is fully abstract if it preserves and reflects Morris-style contextual equivalence [31], i.e. indistinguishability under all program contexts, which are usually defined as programs with a hole. The intuition is that contexts represent the ways an attacker can interact with programs and so full abstraction ensures that such interactions are consistent between languages.

Full abstraction is arguably a strong and useful property but it is also notoriously hard to prove for realistic compilers, mainly due to the inherent challenge of having to reason directly about program contexts [37, 18, 9, 24]. There is thus a need for better formal methods, a view shared in the scientific community [10, 33]. While recent work has proposed generalizing from full abstraction towards the so-called *robust* properties [36, 2], the main challenge of quantifying over program contexts remains, which manifests when directly translating target contexts to the source (*back-translation*). Other techniques, such as trace semantics [35] or logical relations [17], require complex correctness and completeness proofs w.r.t. contextual equivalence in order to be applicable.

In this paper we introduce a novel, categorical approach to secure compilation. The approach has two main components: the elegant representation of Structural Operational Semantics (SOS) [38] using category-theoretic *distributive laws* [48] [3] and also *maps of distributive laws* [40, 50, 27] as secure compilers that preserve bisimilarity. Our method aims to be unifying, in that there is a general, shared formalism for operational semantics, and simplifying, in that the formal criterion for compiler security, the *coherence criterion* for maps of distributive laws, is straightforward and relatively easy to prove.

The starting point of our contributions is an abstract proof on how coalgebraic bisimilarity under distributive laws holds *contextual* meaning in a manner similar to contextual equivalence (Section 4.3). We argue that this justifies the use of the coherence criterion for testing compiler security as long as bisimilarity adequately captures the underlying threat model. We then demonstrate the effectiveness of our approach by appeal to four examples of compiler (in)security. The examples model classic, non-trivial problems in secure compilation:

- An example of an extra processor register in the target language that conveys additional information about computations (Section 5).
- A datatype mismatch between the type of variable (Section 6).
- The introduction of illicit control flow in the target language (Section 7)
- A case of incorrect local state encapsulation (Section 8).

For each of these examples we present an insecure compiler that fails the coherence criterion, then introduce *security primitives* in the target language and construct a secure compiler that respects it. We also examine how bisimilarity can be both a blessing and a curse as its strictness and rigidity sometimes lead to relatively contrived solutions. Finally, in Section 9, we discuss related work and point out potential avenues for further development of the underlying theory.

On the structure and style of the paper This work is presented mainly in the style of programming language semantics but its ideas are deeply rooted in category theory. We follow an "on-demand" approach when it comes to important categorical concepts: we begin the first example by introducing the base language used throughout the paper, *While*, and gradually present distributive laws when required. From the second example in Section 6 and on, we relax the categorical notation and mostly remain within the style of PL semantics.

2 The basic *While* language

2.1 Syntax and operational semantics

We begin by defining the set of arithmetic expressions.

$\langle expr \rangle ::= \texttt{lit}\ \mathbb{N} \mid \texttt{var}\ \mathbb{N} \mid \langle expr \rangle\ \langle bin \rangle\ \langle expr \rangle \mid \langle un \rangle\ \langle expr \rangle$

[3] The authors use the term "Mathematical Operational Semantics". The term "Bialgebraic Semantics" is also used in the literature.

The constructors are respectively literals, a dereference operator `var`, binary arithmetic operations as well as unary operations. We let S be the set of lists of natural numbers. The role of S is that of a run-time store whose entries are referred by their index on the list using constructor `var`. We define function $\texttt{eval} : S \times E \to \mathbb{N}$ inductively on the structure of expressions.

Definition 1 (Evaluation of expressions in *While*).

$$\begin{aligned}
&\texttt{eval}\ \mathit{store}\ (\texttt{lit}\ n) = n\\
&\texttt{eval}\ \mathit{store}\ (\texttt{var}\ l) = \texttt{get}\ \mathit{store}\ l\\
&\texttt{eval}\ \mathit{store}\ (e_1\ b\ e_2) = (\texttt{eval}\ \mathit{store}\ e_1)\ [[b]]\ (\texttt{eval}\ \mathit{store}\ e_2)\\
&\texttt{eval}\ \mathit{store}\ (u\ e) = [[u]]\ (\texttt{eval}\ \mathit{store}\ e)
\end{aligned}$$

Programs in *While* language are generated by the following grammar:

$$\langle \mathit{prog} \rangle ::= \texttt{skip} \mid \mathbb{N}\ \texttt{:=}\ \langle \mathit{expr} \rangle \mid \langle \mathit{prog} \rangle\ ;\ \langle \mathit{prog} \rangle \mid \texttt{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{prog} \rangle$$

The operational semantics of our *While* language are introduced in Figure 1. We are using the notation $s, x \Downarrow s'$ to denote that program x, when supplied with $s : S$, terminates producing store s'. Similarly, $s, x \to s', x'$ means that program x, supplied with s, evaluates to x' and produces new store s'.

$$\frac{}{s, \texttt{skip} \Downarrow s} \qquad \frac{}{s, l\ \texttt{:=}\ e \Downarrow \texttt{update}\ s\ l\ (\texttt{eval}\ s\ e)} \qquad \frac{s, p \Downarrow s'}{s, p\,;q \to s', q}$$

$$\frac{s, p \to s', p'}{s, p\,;q \to s', p'\,;q} \qquad \frac{\texttt{eval}\ s\ e = 0}{s, \texttt{while}\ e\ p \to s, \texttt{skip}} \qquad \frac{\texttt{eval}\ s\ e \neq 0}{s, \texttt{while}\ e\ p \to s, p; \texttt{while}\ e\ p}$$

Fig. 1. Semantics of the *While* language.

2.2 *While*, categorically

The categorical representation of operational semantics has various forms of incremental complexity but for our purposes we only need to use the most important one, that of *GSOS laws* [48].

Definition 2. *Given a syntax functor Σ and a behavior functor B, a GSOS law of Σ over B is a natural transformation $\rho : \Sigma(\mathrm{Id} \times B) \Longrightarrow B\Sigma^*$, where (Σ^*, η, μ) is the monad freely generated by Σ.*

Example 1. Let E be the set of expressions of the *While*-language. Then the syntax functor $\Sigma : \mathbf{Set} \to \mathbf{Set}$ for *While* is given by $\Sigma X = \top \uplus (\mathbb{N} \times E) \uplus (X \times X) \uplus (E \times X)$ where $\uplus$ denotes a disjoint (tagged) union. The elements could be denoted as `skip`, $l := e$, $x_1; x_2$ and `while` $e\ x$ respectively. The free monad

Σ^* satisfies $\Sigma^* X \cong X \uplus \Sigma\Sigma^* X$, i.e. its elements are programs featuring program variables from X. Since *While*-programs run in interaction with a store and can terminate, the behavior functor is $BX = S \to (S \times \text{Maybe } X)$, where S is the set of lists of natural numbers and $X \to Y$ denotes the exponential object (internal Hom) Y^X.

The GSOS specification of *While* determines ρ. A premise $s, p \to s', p'$ denotes an element $(p, b) \in (\text{Id} \times B)X$ where $b(s) = (s', \text{just } p')$, and a premise $s, p \Downarrow s'$ denotes an element (p, b) where $b(s) = (s', \text{nothing})$. A conclusion $s, p \to s', p'$ (where $p \in \Sigma X$ is further decorated above the line to $\bar{p} \in \Sigma(\text{Id} \times B)X$) specifies that $\rho(\bar{p}) \in B\Sigma^* X$ sends s to $(s', \text{just } p')$, whereas a conclusion $s, p \Downarrow s'$ specifies that s is sent to $(s', \text{nothing})$. Concretely, $\rho_X : \Sigma(X \times BX) \to B\Sigma^* X$ is the function (partially from [47]):

$$
\begin{aligned}
&\texttt{skip} && \mapsto \lambda\, s.(s, \text{nothing}) \\
&l := e && \mapsto \lambda\, s.(\texttt{update}\ s\ l\ (\texttt{eval}\ s\ e), \text{nothing}) \\
&\texttt{while}\ e\ (x, f) && \mapsto \lambda\, s. \begin{cases} (s, \text{just } (x\ ;\ \texttt{while}\ e\ x)) & \text{if } \texttt{eval}\ s\ e \neq 0 \\ (s, \text{just } (\texttt{skip})) & \text{if } \texttt{eval}\ s\ e = 0 \end{cases} \\
&(x, f)\ ;\ (y, g) && \mapsto \lambda\, s. \begin{cases} (s', \text{just } (x'\ ;\ y)) & \text{if } f(s) = (s', \text{just } x') \\ (s', \text{just } y) & \text{if } f(s) = (s', \text{nothing}) \end{cases}
\end{aligned}
$$

⌟

It has been shown by Lenisa et al. [28] that there is a one-to-one correspondence between GSOS laws of Σ over B and *distributive laws* of the free monad Σ^* over the cofree copointed endofunctor [28] $\text{Id} \times B$.[4]

Definition 3 (In [26]). *A distributive law of a monad (T, η, μ) over a copointed functor (H, ϵ) is a natural transformation $\lambda : TH \Longrightarrow HT$ subject to the following laws: $\lambda \circ \eta = H\eta$, $\epsilon \circ \lambda = T\epsilon$ and $\lambda \circ \mu = H\mu \circ \lambda \circ T\lambda$.*

Given any GSOS law, it is straightforward to obtain the corresponding distributive law via structural induction (In [50], prop. 2.7 and 2.8). By convention, we shall be using the notation ρ for GSOS laws and ρ^* for the equivalent distributive laws unless stated otherwise.

A distributive law λ based on a GSOS law ρ gives a category λ-Bialg of λ-bialgebras [48], which are pairs $\Sigma X \xrightarrow{h} X \xrightarrow{k} BX$ subject to the pentagonal law $k \circ h = Bh^* \circ \rho_X \circ \Sigma[id, k]$, where h^* is the inductive extension of h. Morphisms in λ-Bialg are arrows $X \to Y$ that are both algebra and coalgebra homomorphisms at the same time. The trivial initial B-coalgebra $\bot \to B\bot$ lifts uniquely to the initial λ-bialgebra $\Sigma\Sigma^*\bot \xrightarrow{a} \Sigma^*\bot \xrightarrow{h_\lambda} B\Sigma^*\bot$, while the trivial final Σ-algebra $\Sigma\top \to \top$ lifts uniquely to the final λ-bialgebra $\Sigma B^\infty \top \xrightarrow{g_\lambda} B^\infty \top \xrightarrow{z} BB^\infty \top$ [5]. Since $\Sigma^*\bot$ is the set of programs generated by Σ and $B^\infty\top$ the set of behaviors cofreely generated by B, the unique bialgebra morphism $f : \Sigma^*\bot \to B^\infty\top$ is the *interpretation function* induced by ρ.

[4] A copointed endofunctor is an endofunctor F equipped with a natural transformation $F \Longrightarrow \text{Id}$.

[5] We write B^∞ for the cofree comonad over B, which satisfies $B^\infty X \cong X \times BB^\infty X$.

Remark 1. We write A for $\Sigma^*\bot$ and Z for $B^\infty\top$, and refer to $h_\lambda : A \to BA$ as the *operational model* for λ and to $g_\lambda : \Sigma Z \to Z$ as the *denotational model* [48]. Note also that $a : \Sigma A \cong A$ and $z : Z \cong BZ$ are invertible.

Example 2. Continuing Example 1, the initial bialgebra A is just the set of all *While*-programs. Meanwhile, the final bialgebra Z, which has the meaning of the set of behaviors, satisfies $Z \cong (S \to S \times \text{Maybe } Z)$. In other words, our attacker model is that of an attacker who can count execution steps and moreover, between any two steps, read out and modify the state. In Section 9, we discuss how we hope to consider weaker attackers in the future.

3 An extra register (Part I)

Let us consider the scenario where a malicious party can observe *more* information about the execution state of a program, either because information is being leaked to the environment or the programs are run by a more powerful machine. A typical example is the presence of an extra *flags* register that logs the result of a computation [34, 8, 35]. This is the intuition behind the augmented version of *While* with additional observational capabilities, $\mathit{While}_{\text{🔔}}$.

The main difference is in the behavior so the notation for transitions has to slightly change. The two main transition types, $s, \mathtt{x} \Downarrow_v s'$ and $s, x \to_v s', x'$ work similarly to *While* except for the label $v : \mathbb{N}$ produced when evaluating expressions. We also allow language terms to interact with the labels by introducing the constructor $\mathtt{obs}\ \mathbb{N}\ \langle \mathit{prog} \rangle$. When terms evaluate inside an $\mathtt{obs}$ block, the labels are sequentially placed in the run-time store. The rest of the constructors are identical but the distinction between the two languages should be clear.

While the expressions are the same as before, the syntax functor is now $\Sigma_{\text{🔔}} X = \Sigma X \uplus \mathbb{N} \times X$, and the behavior functor is $B_{\text{🔔}} = S \to \mathbb{N} \times S \times \text{Maybe } X$. The full semantics can be found in Figure 2. As for *While*, they specify a GSOS law $\rho_{\text{🔔}} : \Sigma_{\text{🔔}}(\text{Id} \times B_{\text{🔔}}) \Longrightarrow B_{\text{🔔}}\Sigma^*_{\text{🔔}}$.

$$\frac{}{s, \mathtt{skip} \Downarrow_0 s} \qquad \frac{v = \mathtt{eval}\ s\ e}{s, l\ \mathtt{:=}\ e \Downarrow_v \mathtt{update}\ s\ l\ v} \qquad \frac{v = \mathtt{eval}\ s\ e \quad v \neq 0}{s, \mathtt{while}\ e\ p \to_v s, \mathtt{skip}}$$

$$\frac{s, p \Downarrow_v s'}{s, p; q \to_v s', q} \qquad \frac{s, p \Downarrow_v s' \quad s'' = \mathtt{update}\ s'\ n\ v}{s, \mathtt{obs}\ n\ p \to_v s'', \mathtt{skip}} \qquad \frac{s, p \to_v s', p'}{s, p; q \to_v s', p'; q}$$

$$\frac{s, p \to_v s', p' \quad s'' = \mathtt{update}\ s'\ n\ v}{s, \mathtt{obs}\ n\ p \to_v s'', \mathtt{obs}\ (n+1)\ p'} \qquad \frac{v = \mathtt{eval}\ s\ e \quad v = 0}{s, \mathtt{while}\ e\ p \to_v s, p; \mathtt{while}\ e\ p}$$

Fig. 2. Semantics of $\mathit{While}_{\text{🔔}}$.

Traditionally, the (in)security of a compiler has been a matter of *full abstraction*; a compiler is fully abstract if it preserves and reflects Morris-style [31]

contextual equivalence. For our threat model, where the attacker can directly observe labels, it makes sense to define contextual equivalence in $While_{🔔}$ as:

Definition 4. $p \cong_{🔔} q \iff \forall c : C_{🔔}.\ c[\![p]\!] \Downarrow \iff c[\![q]\!] \Downarrow$

Where C is the set of one-hole contexts, $[\![\_]\!] : C_{🔔} \times A_{🔔} \to A_{🔔}$ denotes the plugging function and we write $p \Downarrow$ when p eventually terminates. Contextual equivalence for *While* is defined analogously. It is easy to show that the simple "embedding" compiler from *While* to $While_{🔔}$ is not fully abstract by examining terms $a \triangleq \texttt{while}\ (\texttt{var}[0])\ (0\ \texttt{:=}\ 0)$ and $b \triangleq \texttt{while}\ (\texttt{var}[0] * 2)\ (0\ \texttt{:=}\ 0)$, for which $a \cong b$ but $a_{🔔} \ncong_{🔔} b_{🔔}$. A context $c \triangleq (\texttt{obs}\ 1\ \_); \texttt{while}\ (\texttt{var}[1] - 1)\ \texttt{skip}$ will log the result of the `while` condition in $a_{🔔}$ and $b_{🔔}$ in $\texttt{var}[1]$ and then either diverge or terminate depending on the value of $\texttt{var}[1]$. An initial $\texttt{var}[0]$ value of 1 will cause $c[\![a]\!]$ to terminate but $c[\![b]\!]$ to diverge.

Securely extending *While*$_{🔔}$ To deter malicious contexts from exploiting the extra information, we introduce *sandboxing* primitives to help hide it. We add an additional constructor in $While_{🔔}$, $\wr\langle progr\rangle\int$, and the following inference rules to form the secure version $While_{🔔\!\!\!/}$ of $While_{🔔}$.

$$\frac{s, p \Downarrow_v s'}{s, \wr p\int \Downarrow_0 s'} \qquad \frac{s, p \to_v s', p'}{s, \wr p\int \to_0 s', \wr p'\int}$$

We now consider the compiler from *While* to $While_{🔔\!\!\!/}$ which, along with the obvious embedding, wraps the the translated terms in sandboxes. This looks to be effective as programs a and b are now contextually equivalent and the extra information is adequately hidden. We will show that this compiler is indeed a *map of distributive laws* between *While* and $While_{🔔\!\!\!/}$ but to do so we need a brief introduction on the underlying theory.

4 Secure compilers, categorically

4.1 Maps of distributive laws

Assume two GSOS laws $\rho_1 : \Sigma_1(\mathrm{Id} \times B_1) \Longrightarrow B_1\Sigma_1^*$ and $\rho_2 : \Sigma_2(\mathrm{Id} \times B_2) \Longrightarrow B_2\Sigma_2^*$, where $(\Sigma_1^*, \eta_1, \mu_1)$ and $(\Sigma_2^*, \eta_2, \mu_2)$ are the monads freely generated by Σ_1 and Σ_2 respectively. We shall regard pairs of natural transformations $(\sigma : \Sigma_1^* \Longrightarrow \Sigma_2^*, b : B_1 \Longrightarrow B_2)$ as compilers between the two semantics, where σ acts as a syntactic translation and b as a translation between behaviors.

Remark 2. If A_1 and A_2 are the sets of terms freely generated by Σ_1 and Σ_2, we can get the compiler $c : A_1 \to A_2$ from σ. On the other hand, b generates a function $d : Z_1 \to Z_2$ between behaviors via finality.

Remark 3. We shall be writing B^c for the cofree copointed endofunctor $\mathrm{Id} \times B$ over B and $b^c : B_1^c \Longrightarrow B_2^c$ for $\mathrm{id} \times b$.

Definition 5 (Adapted from [50]). *A map of GSOS laws from ρ_1 to ρ_2 consists of a natural transformation $\sigma : \Sigma_1^* \Longrightarrow \Sigma_2^*$ subject to the monad laws $\sigma \circ \eta_1 = \eta_2$ and $\sigma \circ \mu_1 = \mu_2 \circ \Sigma_2^* \sigma \circ \sigma$ paired with a natural transformation $b : B_1 \Longrightarrow B_2$ that satisfies the following* coherence criterion*:*

$$\begin{array}{ccc} \Sigma_1^* B_1^c & \xRightarrow[\rho_1^*]{} & B_1^c \Sigma_1^* \\ {\scriptstyle \sigma \,\circ\, \Sigma_1^* b^c}\Downarrow & & \Downarrow{\scriptstyle b^c \,\circ\, B_1^c \sigma} \\ \Sigma_2^* B_2^c & \xRightarrow{\rho_2^*} & B_2^c \Sigma_2^* \end{array}$$

⌟

Remark 4. A natural transformation $\sigma : \Sigma_1^* \Longrightarrow \Sigma_2^*$ subject to the monad laws is equivalent to a natural transformation $t : \Sigma_1 \Longrightarrow \Sigma_2^*$.

Theorem 1. *If σ and b constitute a map of GSOS laws, then we get a compiler $c : A_1 \to A_2$ and behavior transformation $d : Z_1 \to Z_2$ satisfying $d \circ f_1 = f_2 \circ c : A_1 \to Z_2$. As bisimilarity is exactly equality in the final coalgebra (i.e. equality under $f_i : A_i \to Z_i$), c preserves bisimilarity [50]. If d is a monomorphism (which, under mild conditions, is the case in* **Set** *if every component of b is a monomorphism), then c also reflects bisimilarity.*

What is very important though, is that the well-behavedness properties of the two GSOS laws bestow *contextual* meaning to bisimilarity. Recall that the gold standard for secure compilation is contextual equivalence (Definition 4), which is precisely what is observable through program contexts. Bisimilarity is generally not the same as contextual equivalence, but we can instead show that in the case of GSOS laws or other forms of distributive laws, bisimilarity defines the *upper bound* (most fine-grained distinction) of observability up to program contexts. We shall do so abstractly in the next subsections.

4.2 Abstract program contexts

The informal notion of a *context* in a programming language is that of a program with a hole [31]. Thus contexts are a syntactic construct that models external interactions with a program: a single context is an experiment whose outcome is the evaluation of the subject program *plugged* in the context.

Naïvely, one may hope to represent contexts by a functor H sending a set of variables X to the set HX of terms in ΣX that may have holes in them. A complication is that contexts may have holes at any depth (i.e. any number of operators may have been applied to a hole), whereas ΣX is the set of terms that have exactly one operator in them, immediately applied to variables. One solution is to think of Y in HY as a set of variables that do not stand for terms, but for contexts. This approach is fine for multi-hole contexts, but if we also want to consider single-hole contexts and a given single-hole context c is not the hole itself, then precisely one variable in c should stand for a single-hole context, and all other variables should stand for terms. Thus, in order to support both single- and multi-hole contexts, we make H a two-argument functor, where $H_X Y$ is the set of contexts with term variables from X and context variables from Y.

Definition 6. *Let $\mathbb{C}$ be a distributive category [14] with products $\times$, coproducts $\uplus$, initial object $\bot$ and terminal object $\top$, as is the case for* **Set**. *A* context functor *for a syntax functor $\Sigma : \mathbb{C} \to \mathbb{C}$, is a functor $H : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ (with application to (X, Y) denoted as $H_X Y$) such that there exist natural transformations* hole $: \forall (X, Y).\top \to H_X Y$ *and* con $: \forall X.X \times H_X X \to X \uplus \Sigma X$ *making the following diagram commute for all X:*

$$\begin{array}{ccc} X \times \top & \xrightarrow[\cong]{\pi_1} & X \\ {\scriptstyle \mathrm{id}_X \times \mathrm{hole}_{(X,X)}}\downarrow & & \downarrow{\scriptstyle i_1} \\ X \times H_X X & \xrightarrow{\mathrm{con}_X} & X \uplus \Sigma X \end{array}$$

⌟

The idea of the transformation con is the following: it takes as input a variable $x \in X$ to be plugged into the hole, and a context $c \in H_X X$ with one layer of syntax. The functor H_X is applied again to X rather than Y because x is assumed to have been recursively plugged into the context placeholders $y \in Y$ already. We then make a case distinction: if c is the hole itself, then $i_1\ x$ is returned. Otherwise, $i_2\ c$ is returned.

Definition 7. *Let $\mathbb{C}$ be a category as in Definition 6 and assume a syntax functor Σ with context functor H. If Σ has an initial algebra (A, q_A) (the set of programs) and H_A has a strong initial algebra (C_A, q_{C_A}) [23] (the set of contexts), then we define the plugging function $[\![\ ]\!] : A \times C_A \to A$ as the "strong inductive extension" [23] of the algebra structure $[\mathrm{id}_A, q_A] \circ \mathrm{con}_A : A \times H_A A \to A$ on A, i.e. as the unique morphism that makes the following diagram commute:*

$$\begin{array}{ccccc} A \times H_A C_A & & \xrightarrow[\cong]{id \times q_{C_A}} & & A \times C_A \\ \downarrow{\scriptstyle (\pi, st)} & & & & \\ A \times H_A (A \times C_A) & & & & \downarrow{\scriptstyle [\![\ ]\!]} \\ \downarrow{\scriptstyle id \times H_A [\![\ ]\!]} & & & & \\ A \times H_A A & \xrightarrow{\mathrm{con}_A} & A \uplus \Sigma A & \xrightarrow{[\mathrm{id}, q_A]} & A \end{array}$$

⌟

The above definition of contextual functors is satisfied by both single-hole and multi-hole contexts, the construction of which we discuss below.

Multi-hole contexts Given a syntax functor Σ, its multi-hole context functor is simply $H_X Y = \top \uplus \Sigma Y$. The contextual natural transformation con is the obvious map that returns the pluggee if the given context is a hole, and otherwise the context itself (which is then a program):

$$\begin{aligned} &\mathrm{con} : \forall X.X \times (\top \uplus \Sigma X) \to X \uplus \Sigma X \\ &\mathrm{con} \circ (\mathrm{id} \times i_1) = i_1 \circ \pi_1 : \forall X.X \times \top \to X \uplus \Sigma X \\ &\mathrm{con} \circ (\mathrm{id} \times i_2) = i_2 \circ \pi_2 : \forall X.X \times \Sigma X \to X \uplus \Sigma X \end{aligned}$$

The 'pattern matching' is justified by distributivity of $\mathbb{C}$. For hole $= i_1 : \top \to \top \uplus \Sigma X$, we can see that $\mathrm{con} \circ (\mathrm{id} \times \mathrm{hole}) = i_1 \circ \pi_1$ as required by the definition of a context functor.

Single-hole contexts It was observed by McBride [29] that for inductive types, i.e. least fixpoints / initial algebras μF of certain endofunctors F called *containers* [4] or simply *polynomials*, their single-hole contexts are lists of $\partial F(\mu F)$ where ∂F is the *derivative* of F [6]. Derivatives for containers, which were developed by Abbott et al. in [5], enable us to give a categorical interpretation of single-hole contexts as long as the syntax functor Σ is a container.

It would be cumbersome to lay down the entire theory of containers and their derivatives, so we shall instead focus on the more restricted set of *Simple Polynomial Functors* [22] (or SPF), used to model both syntax and behavior. Crucially, SPF's are differentiable and hence compatible with McBride's construction.

Definition 8 (Simple Polynomial Functors). *The collection of SPF is the least set of functors $\mathbb{C} \to \mathbb{C}$ satisfying the following rules:*

$$id\frac{}{\mathrm{Id} \in \mathrm{SPF}} \qquad const\frac{J \in \mathrm{Obj}(\mathbb{C})}{K_J \in \mathrm{SPF}} \qquad prod\frac{F, G \in \mathrm{SPF}}{F \times G \in \mathrm{SPF}}$$

$$coprod\frac{F, G \in \mathrm{SPF}}{F \uplus G \in \mathrm{SPF}} \qquad comp\frac{F, G \in \mathrm{SPF}}{F \circ G \in \mathrm{SPF}}$$

⌟

We can now define the differentiation action $\partial : \mathrm{SPF} \to \mathrm{SPF}$ by structural induction. Interestingly, it resembles simple derivatives for polynomial functions.

Definition 9 (SPF derivation rules).

$$\partial \mathrm{Id} = \top, \quad \partial K_J = \bot, \quad \partial(G \uplus H) = \partial G \uplus \partial H,$$

$$\partial(G \times H) = (\partial G \times H) \uplus (G \times \partial H), \quad \partial(G \circ H) = (\partial G \circ H) \times \partial H.$$

⌟

Example 3. The definition of con for single-hole contexts might look a bit cryptic at first sight so we shall use a small example from [29] to shed some light. In the case of binary trees, locating a hole in a context can be thought of as traversing through a series of nodes, choosing left or right according to the placement of the hole until it is found. At the same time a record of the trees at the non-chosen branches must be kept so that in the end the entire structure can be reproduced.

Now, considering that the set of binary trees is the least fixed point of functor $\top \uplus (\mathrm{Id} \times \mathrm{Id})$, then the type of "abstract" choice at each intersection is the functor $K_{\mathrm{Bool}} \times \mathrm{Id}$, where K_{Bool} stands for a choice of left or right and the Id part represents the passed structure. Lists of $(K_{\mathrm{Bool}} \times \mathrm{Id})$ BinTree are exactly the sort of record we need to keep, i.e. they contain the same information as a tree with a single hole. And indeed $K_{\mathrm{Bool}} \times \mathrm{Id}$ is (up to natural isomorphism) the derivative of $\top \uplus (\mathrm{Id} \times \mathrm{Id})$! ⌟

[6] The *list* operator itself arises from the derivative of the free monad operator.

Using derivatives we can define the context functor $H_X Y = \top \uplus ((\partial\Sigma\ X) \times Y)$ for syntax functor Σ. Then the initial algebra C_A of H_A is indeed List $((\partial\Sigma)\ A)$, the set of single-hole contexts for $A \cong \Sigma A$.

Plugging Before defining con, we define an auxiliary function conStep : $\partial\Sigma \times \mathrm{Id} \Longrightarrow \Sigma$. We defer the reader to [29] for the full definition of conStep, which is inductive on the SPF Σ, and shall instead only define the case for coproducts. So, for $\partial(F \uplus G) = \partial F \uplus \partial G$ we have:

$$\begin{aligned}
&\mathrm{conStep}_{F \uplus G} : (\partial F \uplus \partial G) \times \mathrm{Id} \Longrightarrow F \uplus G \\
&\mathrm{conStep}_{F \uplus G} \circ (i_1 \times \mathrm{id}) = i_1 \circ \mathrm{conStep}_F : \partial F \times \mathrm{Id} \Longrightarrow F \uplus G \\
&\mathrm{conStep}_{F \uplus G} \circ (i_2 \times \mathrm{id}) = i_2 \circ \mathrm{conStep}_G : \partial G \times \mathrm{Id} \Longrightarrow F \uplus G
\end{aligned}$$

We may now define $\mathrm{con} : X \times H_X X \to X \uplus \Sigma X$ as follows:

$$\begin{aligned}
&\mathrm{con} : \forall X.X \times (\top \uplus (\partial\Sigma\ X \times X)) \to X \uplus \Sigma X \\
&\mathrm{con} \circ (\mathrm{id} \times i_1) = i_1 \circ \pi_1 : \forall X.X \times \top \to X \uplus \Sigma X \\
&\mathrm{con} \circ (\mathrm{id} \times i_2) = i_2 \circ \mathrm{conStep}_\Sigma \circ \pi_2 : \forall X.X \times (\partial\Sigma\ X \times X) \to X \uplus \Sigma X
\end{aligned}$$

By setting $\mathrm{hole} = i_1 : \top \to \top \uplus (\partial\Sigma\ X \times X)$ we can see that $\mathrm{con} \circ (\mathrm{id} \times \mathrm{hole}) = i_1 \circ \pi_1$ as required by Definition 6.

4.3 Contextual coclosure

Having established a categorical notion of contexts, we can now move towards formulating contextual categorical arguments about bisimilarity. We assume a context functor H for Σ such that H_A has strong initial algebra (C_A, q_{C_A}) (the object containing all contexts).

First, since we prefer to work in more general categories than just **Set**, we will encode relations $R \subseteq X \times Y$ as spans $X \xleftarrow{r_1} R \xrightarrow{r_2} Y$. One may wish to consider only spans for which $(r_1, r_2) : R \to X \times Y$ is a monomorphism, though this is not necessary for our purposes.

We want to reason about contextually closed relations on the set of terms A, which are relations such that $a_1\ R\ a_2$ implies $(c\,[\![a_1]\!])\ R\ (c\,[\![a_2]\!])$ for all contexts $c \in C_A$. Contextual equivalence will typically be defined as the coclosure of equitermination: the greatest contextually closed relation that implies equitermination. For spans, this becomes:

Definition 10. *In a category as in Definition 6, a span $A \xleftarrow{r_1} R \xrightarrow{r_2} A$ is called contextually closed if there is a morphism $[\![\ ]\!] : C_A \times R \to R$ making the following diagram commute:*

$$\begin{array}{ccccc}
C_A \times A & \xleftarrow{\mathrm{id} \times r_1} & C_A \times R & \xrightarrow{\mathrm{id} \times r_2} & C_A \times A \\
\downarrow{\scriptstyle [\![\ ]\!]} & & \downarrow{\scriptstyle [\![\ ]\!]} & & \downarrow{\scriptstyle [\![\ ]\!]} \\
A & \xleftarrow{r_1} & R & \xrightarrow{r_2} & A
\end{array}$$

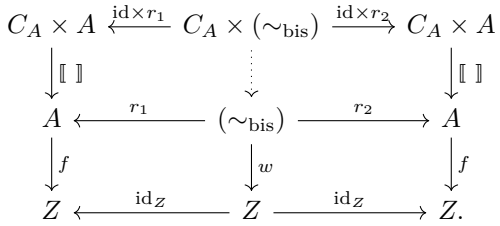
The contextual co-closure $A \xleftarrow{\bar{r}_1} \bar{R} \xrightarrow{\bar{r}_2} A$ of an arbitrary span $A \xleftarrow{r_1} R \xrightarrow{r_2} A$ is the final contextually closed span on A with a span morphism $\bar{R} \to R$. ⌟

We call terms bisimilar if the operational semantics $f : A \to Z$ assigns them equal behaviors:

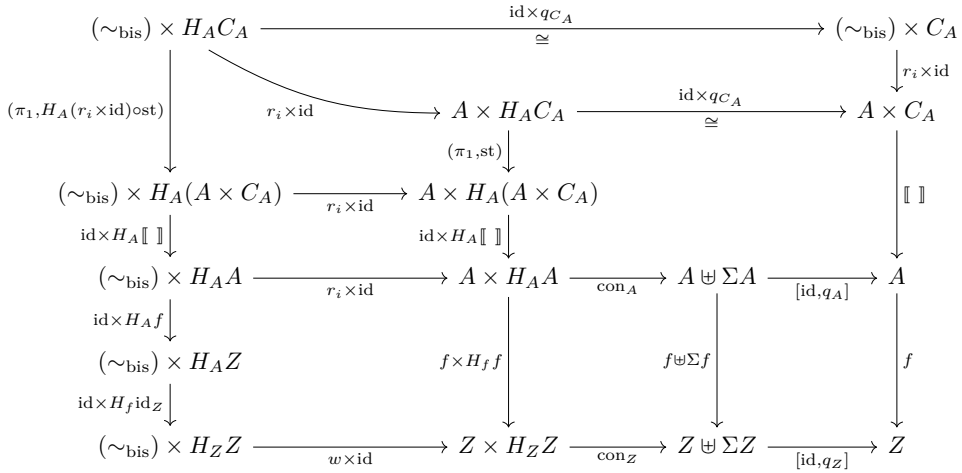
Definition 11. *We define (strong) bisimilarity $\sim_{\mathrm{bis}}$ as the pullback of the equality span $(\mathrm{id}_Z, \mathrm{id}_Z) : Z \to Z \times Z$ along $f \times f : A \times A \to Z \times Z$ (if existent).*

Theorem 2. *Under the assumptions of 7, bisimilarity (if existent) is contextually closed.*

Proof. We need to give a morphism of spans from $C_A \times (\sim_{\mathrm{bis}})$ to $(\sim_{\mathrm{bis}})$:



By definition of $(\sim_{\mathrm{bis}})$, it suffices to give a morphism of spans to the equality span on Z, i.e. to prove that $f \circ [\![\ ]\!] \circ (\mathrm{id} \times r_1) = f \circ [\![\ ]\!] \circ (\mathrm{id} \times r_2)$. To this end, consider the following diagram (parameterized by $i \in \{1, 2\}$), in which every polygon is easily seen to commute:



The bottom-right square stems from the underlying GSOS law: it is the algebra homomorphism part of the bialgebra morphism between the initial and the final bialgebras. Commutativity of the outer diagram reveals that $f \circ [\![\ ]\!] \circ (r_i \times \mathrm{id})$ is, regardless of i, *the* strong inductive extension of $[\mathrm{id}_Z, q_Z] \circ \mathrm{con}_Z \circ (w \times H_f \mathrm{id}_Z) : (\sim_{\mathrm{bis}}) \times H_A Z \to Z$. Thus, it is independent of i. □

Corollary 1. *In* **Set**, *bisimilarity is its own contextual coclosure:* $a \sim_{\text{bis}} b \iff \forall\ c \in C_A.\ c\,[\![a]\!] \sim_{\text{bis}} c\,[\![b]\!]$

Corollary 2. *In* **Set**, *bisimilarity implies contextual equivalence.*[7]

Proof. Bisimilarity implies equitermination. This yields an implication between their coclosures. □

Comparing Corollary 1 to contextual equivalence in Definition 4 reveals their key difference. Contextual equivalence makes minimal assumptions on the underlying observables, which are simply divergence and termination. On the other hand, the contextual coclosure of bisimilarity assumes maximum observability (as dictated by the behavior functor) and in that sense it represents the upper bound of what can be observed through contexts. Consequently, this criterion is useful if the observables adequately capture the threat model, which is true for the examples that follow.

This theorem echoes similar results in the broader study of coalgebraic bisimulation [11, 41]. There are, however, two differences. The first is that our theorem allows for extra flexibility in the definition of contexts as the theorem is parametric on the context functor. Second, by making the context construction explicit we can directly connect (the contextual coclosure of) bisimilarity to contextual equivalence (Corollary 2) and so have a more convincing argument for using maps of distributive laws as secure compilers.

5 An extra register (Part II)

The next step is to define the syntax and behavior natural transformations. The first compiler, $\sigma_{🔔} : \Sigma \Longrightarrow \Sigma_{🔔}$, is a very simple mapping of constructors in *While* to their $While_{🔔}$ counterparts. The second natural transformation, $\sigma_{🔕} : \Sigma \Longrightarrow \Sigma^*_{🔕}$, is more complex as it involves an additional layer of syntax in $While_{🔕}$.

Definition 12 (Sandboxing natural transformation). *Consider the natural transformation* $e : \Sigma \Longrightarrow \Sigma_{🔕}$ *which embeds* Σ *in* $\Sigma_{🔕}$. *Using PL notation, we define* $\sigma_{🔕} : \Sigma X \to \Sigma^*_{🔕} X : p \mapsto \wr e(p) \int$. *This yields a monad morphism* $\sigma^*_{🔕} : \Sigma^* \to \Sigma^*_{🔕}$ *(Remark 4).*

Defining the natural translation between behaviors is a matter of choosing a designated value for the added observable label. The only constraint is that the chosen value has to coincide with the label that the sandbox produces. $B_{🔔}$ and $B_{🔕}$ are identical so we need a single natural transformation $b : B \Longrightarrow B_{🔔/🔕}$:

$$b : \forall X.\ (S \to S \times \text{Maybe } X) \to S \to \mathbb{N} \times S \times \text{Maybe } X$$
$$b\ f = \lambda(s : S) \to (0, f(s))$$

[7] Note that we can *not* conclude that preservation of bisimilarity would imply preservation of contextual equivalence.

***While* to *While*$_{🔔}$** We now have the natural translation pairs $(\sigma_{🔔}, b)$ and $(\sigma_{🔕}, b)$, which allows us to check the coherence criterion from Section 4.1. We shall be using a graphical notation that provides for a good intuition as to what failure or success of the criterion *means*. For example, Fig. 3 shows failure of the coherence criterion for the first pair.

The horizontal arrows in the diagram represent the two semantics, ρ^* and $\rho^*_{🔔}$, while the vertical arrows are the two horizontal compositions of the natural translation pair. The top-left node holds an element of $\Sigma^*(\mathrm{Id} \times B)$, which in this case is an assignment operation. The two rightmost nodes represent behaviors, so the syntactic element is missing from the left side of the transition arrows.

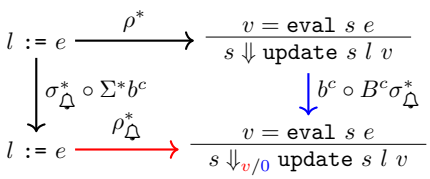


Fig. 3. Failure of the criterion for $(\sigma_{🔔}, b)$.

In the upper path, the term is first applied to the GSOS law ρ^* and the result is then passed to the translation pair, thus producing the designated label 0, typeset in blue for convenience. In the lower path, the term is first applied to the translation and then goes through the target semantics, $\rho^*_{🔔}$, where the label v is produced. It is easy to find such an s so that $v \neq 0$.

***While* to *While*$_{🔕}$** The same example is investigated for the second translation pair $(\sigma_{🔕}, b)$. Figure 4 shows what happens when we test the same case as before. Applying $\rho^*_{🔕}$ to $\wr l$:= $e\int$ is similar to $\rho_{🔕}$ acting twice. The innermost transition is the intermediate step and as it only appears in the bottom path it is typeset in red. This time the diagram commutes as the label produced in the inner layer, v, is effectively erased by the sandboxing rules of *While*$_{🔕}$.

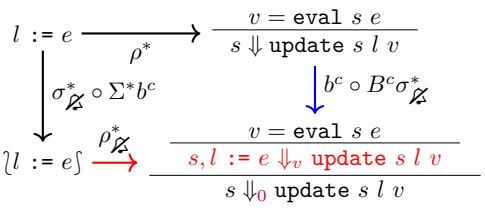


Fig. 4. The coherence criterion for $(\sigma_{🔕}, b)$.

An endo-compiler for *While*$_{🔕}$ If $A_{🔕}$ is the set of closed terms for *While*$_{🔕}$, the compiler $u : A_{🔕} \to A_{🔕}$, which "escapes" *While*$_{🔕}$ terms from their sandboxes can be elegantly modeled using category theory. As before, it is not possible to express it using a simple natural transformation $\Sigma_{🔕} \Longrightarrow \Sigma_{🔕}$. We can, however, use the *free pointed endofunctor* [28] over $\Sigma_{🔕}$, $\mathrm{Id} \uplus \Sigma_{🔕}$. What we want is to map non-sandboxed terms to themselves and lift the extra layer of syntax from sandboxed terms. Intuitively, for a set of variables X, $\Sigma_{🔕}X$ is one layer of syntax "populated" with elements of X. If $X \uplus \Sigma_{🔕}X$ is the union of $\Sigma_{🔕}X$ with the set of variables X, lifting the sandboxing layer is mapping the X in $\wr X\int$ to the left of $X \uplus \Sigma_{🔕}X$ and the rest to themselves at the right.

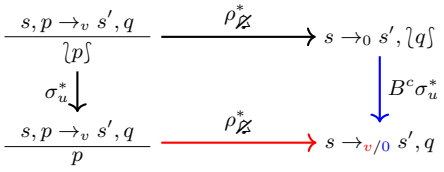


Fig. 5. Failure of the criterion for σ_u.

This is obviously not a secure compiler as it allows discerning previously indistinguishable programs. As we can see in Figure 5, the coherence criterion fails in the expected manner.

6 State mismatch

Having established our categorical foundations, we shall henceforth focus on examples. The first one involves a compiler where the target machine is not necessarily more powerful than the source machine, but the target *value* primitives are not isomorphic to the ones used in the source. This is a well-documented problem [37], which has led to failure of full abstraction before [8, 18, 25].

For example, we can repeat the development of *While* except we substitute natural numbers with integers. We call this new version $While_{\mathbb{Z}}$.

$\langle expr \rangle ::= \texttt{lit}\ \mathbb{Z} \mid \texttt{var}\ \mathbb{N} \mid \langle expr \rangle\ \langle bin \rangle\ \langle expr \rangle \mid \langle un \rangle\ \langle expr \rangle$

The behavior functor also differs in that the store type S is substituted with $S_{\mathbb{Z}}$, the set of lists of integers. We can define the behavioral natural transformation $b_{\mathbb{Z}} : B \Longrightarrow B_{\mathbb{Z}}$ as the best "approximation" between the two behaviors. In **Set**:

$$b_{\mathbb{Z}} : \forall X.\ (S \to S \times (\top \uplus X)) \to S_{\mathbb{Z}} \to S_{\mathbb{Z}} \times (\top \uplus X)$$
$$b_{\mathbb{Z}}\ f = [\mathrm{to}\mathbb{Z}, \mathrm{id}] \circ f \circ \mathrm{to}\mathbb{N}$$

Where to$\mathbb{N}$ replaces all negative numbers in the store with 0 and to$\mathbb{Z}$ typecasts S to $S_{\mathbb{Z}}$. It is easy to see that the identity compiler from *While* to $While_{\mathbb{Z}}$ is not fully abstract. For example, the expressions 0 and `min(var[0],0)` are identical in *While* but can be distinguished in $While_{\mathbb{Z}}$ (if `var[0]` is negative). This is reflected in the coherence criterion diagram for the identity compiler in Figure 6, when initiating the store with a negative integer.

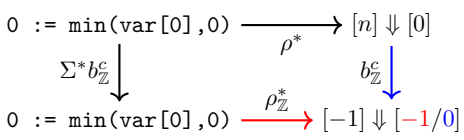


Fig. 6. Failure of the criterion for $(\mathrm{id}, b_{\mathbb{Z}})$.

The solution is to create a special environment where $While_{\mathbb{Z}}$ forgets about negative integers, in essence copying what $b_{\mathbb{Z}}$ does on the store. This is a special kind of sandbox, written $\langle \_ \rangle$, for which we introduce the following rules:

$$\frac{\mathrm{to}\mathbb{N}(s), p \Downarrow s'}{s, \langle p \rangle \Downarrow s'} \qquad \frac{\mathrm{to}\mathbb{N}(s), p \to s', p'}{s, \langle p \rangle \to s', \langle p' \rangle}$$

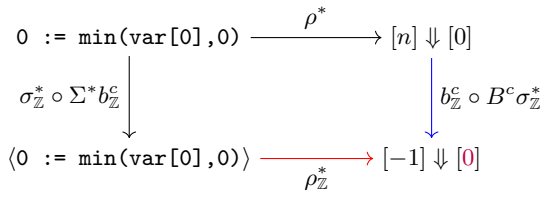


Fig. 7. The coherence criterion for $(\sigma_{\mathbb{Z}}, b_{\mathbb{Z}})$.

We may now repeat the construction from Definition 12 to define the compiler $\sigma_{\mathbb{Z}}$. We can easily verify that the pair $(\sigma_{\mathbb{Z}}, b_{\mathbb{Z}})$ constitutes a map of distributive laws. For instance, Figure 7 demonstrates how the previous failing case now works under $(\sigma_{\mathbb{Z}}, b_{\mathbb{Z}})$.

7 Control Flow

Many low-level languages support unrestricted control flow in the form of jumping or branching to an address. On the other hand, control flow in high-level languages is usually restricted (think if-statements or function calls). A compiler from the high-level to the low-level might be insecure as it exposes source-level programs to illicit control flow. This is another important and well-documented example of failure of full abstraction [8, 37, 3, 33].

$$\frac{}{s, 0, \texttt{stop}\ [;;\ x] \Downarrow s, 0} \qquad \frac{v = \texttt{eval}\ s\ e \quad s' = \texttt{update}\ s\ n\ v}{s, 0, \texttt{assign}\ n\ v\ [;;\ x] \to s', 1, \texttt{assign}\ n\ v\ [;;\ x]}$$

$$\frac{\texttt{PC} \geq 0 \quad s, \texttt{PC}, x \Downarrow s', \texttt{PC}'}{s, \texttt{PC} + 1, i\ ;;\ x \Downarrow s', \texttt{PC}' + 1} \qquad \frac{v = \texttt{eval}\ s\ e \quad v = 0}{s, 0, \texttt{br}\ e\ z\ [;;\ x] \to s, 1, \texttt{br}\ e\ z\ [;;\ x]}$$

$$\frac{v = \texttt{eval}\ s\ e \quad v \neq 0}{s, 0, \texttt{br}\ e\ z\ [;;\ x] \to s, z, \texttt{br}\ e\ z\ [;;\ x]} \qquad \frac{\texttt{PC} < 0}{s, \texttt{PC}, i\ ;;\ x \Downarrow s, \texttt{PC}}$$

$$\frac{p = \texttt{nop}\ [;;\ x]}{s, 0, p \to s, 1, p} \qquad \frac{\texttt{PC} \geq 0 \quad s, \texttt{PC}, x \to s', \texttt{PC}', x'}{s, \texttt{PC} + 1, i\ ;;\ x \to s', \texttt{PC}' + 1, i\ ;;\ x'} \qquad \frac{\texttt{PC} \neq 0}{s, \texttt{PC}, i \Downarrow s, \texttt{PC}}$$

Fig. 8. Semantics of the *Low* language. Elements in square brackets are optional.

We introduce low-level language *Low*, the programs of which are non-empty lists of instructions. *Low* differs significantly from *While* and its derivatives in both syntax and semantics. For the syntax, we define the set of instructions $\langle inst \rangle$ and set of programs $\langle asm \rangle$.

$\langle inst \rangle ::= \texttt{nop} \mid \texttt{stop} \mid \texttt{assign}\ \mathbb{N}\ \langle expr \rangle \mid \texttt{br}\ \langle expr \rangle\ \mathbb{Z}$

$\langle asm \rangle ::= \langle inst \rangle \mid \langle inst \rangle\ \texttt{;;}\ \langle asm \rangle$

Instruction `nop` is the no-operation, `stop` halts execution and `assign` is analogous to the assignment operation in *While*. The `br` instruction is what really defines *Low*, as it stands for bidirectional relative branching.

Semantics of *Low* Figure 8 shows the operational semantics of *Low*. The execution state of a running program consists of a run-time store and the program counter register $\texttt{PC} \in \mathbb{Z}$ that points at the instruction being processed. If the program counter is zero, the leftmost instruction is executed. If the program counter is greater than zero, then the current instruction is further to the right. Otherwise, the program counter is out-of-bounds and execution stops. The categorical interpretation suggests a GSOS law ρ_L of syntax functor $\Sigma_L X = \text{inst} \uplus (\text{inst} \times X)$ over behavior functor $B_L X = S \times \mathbb{Z} \to S \times \mathbb{Z} \times \text{Maybe}\ X$.

An insecure compiler This time we start with the behavioral translation, which is less obvious as we have to go from $BX = S \to S \times \text{Maybe}\ X$ to $B_L X = S \times \mathbb{Z} \to S \times \mathbb{Z} \times \text{Maybe}\ X$. The increased arity in B_L poses an interesting question as to what the program counter should mean in *While*. It makes sense to consider the program counter in *While* as zero since a program in *While* is treated uniformly as a single statement.

$$b_L : \forall X.\ (S \to S \times \text{Maybe}\ X) \to S \times \mathbb{Z} \to S \times \mathbb{Z} \times \text{Maybe}\ X$$

$$b_L\ f\ (s, 0) = \begin{cases} (s', 1, \text{nothing}) & \text{if } f\ s = (s', \text{nothing}) \\ (s', 0, \text{just}\ y) & \text{if } f\ s = (s', \text{just}\ y) \end{cases}$$

$$b_L\ f\ (s, n \neq 0) = (s, n, \text{nothing})$$

When it comes to translating terms, a typical compiler from *While* to *Low* would untangle the tree-like structure of *While* and convert it to a list of *Low* instructions. For `while` statements, the compiler would use branching to simulate looping in the low-level.

Example 4. Let us look at a simple case of a loop. The *While* program
`while (var 0 < 2) (1 := var 1 + 1)` is compiled to
`br !(var 0 < 2) 3 ;; assign 1 (var 1 + 1) ;; br (lit 1) -2` ⌟

This compiler, called c_L, cannot be defined in terms of a natural transformation $\Sigma \Longrightarrow \Sigma_L^*$ as per Remark 4, but it is inductive on the terms of the source language. In this case we can directly compare the two operational models $b_A \circ h : A \to B_L A$ (where $h : A \to BA$) and $h_L : A_L \to B_L A_L$ and notice that $c_L : A \to A_L$ is not a coalgebra homomorphism (Figure 9). The key is

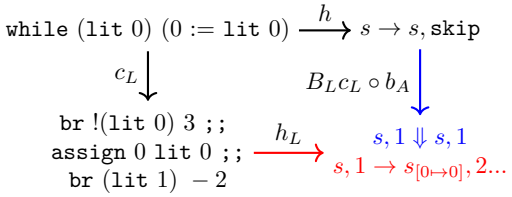


Fig. 9. c_L is not a coalgebra homomorphism.

that the program counter in *Low* allows for finer observations on programs. Take for example the case for `while` (`lit` 0) (0 := `lit` 0), where the loop is always skipped. In *Low*, we can still access the loop body by simply pointing the program counter to it. This is a realistic attack scenario because *Low* allows manipulation of the program counter via the `br` instruction.

Solution By comparing the semantics between *While* in Figure 1 and *Low* in Figure 8 we find major differences. The first one is the reliance of *Low* to a program counter which keeps track of execution, whereas *While* executes statements from left to right. Second, the sequencing rule in *While* dictates that statements are removed from the program state[8] upon completion. On the other hand, *Low* keeps the program state intact at all times. Finally, there is a stark contrast between the two languages in the way they handle `while` loops.

To address the above issues we introduce a new sequencing primitive $;;_c$ and a new looping primitive `loop` for *Low*, which prohibit illicit control flow and properly propagate the internal state. Furthermore, we change the semantics of the singleton `assign` instruction so that it mirrors the peculiarity of its *While* counterpart. The additions can be found in Figure 10.

$$\frac{\texttt{PC} \neq 0}{s, \texttt{PC}, x \;;;_c\; y \Downarrow s', \texttt{PC}} \qquad \frac{s, 0, x \Downarrow s', z}{s, 0, x \;;;_c\; y \to s', 0, y} \qquad \frac{s, 0, x \to s', z, x'}{s, 0, x \;;;_c\; y \to s', 0, x' \;;;_c\; y}$$

$$\frac{\texttt{PC} \neq 0}{s, \texttt{PC}, \texttt{loop}\ e\ x \Downarrow s, \texttt{PC}} \qquad \frac{v = \texttt{eval}\ s\ e \qquad v = 0}{s, 0, \texttt{loop}\ e\ x \to s, 0, \texttt{stop}}$$

$$\frac{v = \texttt{eval}\ s\ e \qquad v \neq 0}{s, 0, \texttt{loop}\ e\ x \to s, 0, x \;;;_c\; \texttt{loop}\ e\ x} \qquad \frac{v = \texttt{eval}\ s\ e \qquad s' = \texttt{update}\ s\ n\ v}{s, 0, \texttt{assign}\ n\ v \Downarrow s', 0}$$

Fig. 10. Secure primitives for the *Low* language.

We may now define the simple "embedding" natural transformation $\sigma_E : \Sigma \Longrightarrow \Sigma_L$, which maps `skip` to `stop`, assignments to `assign`, sequencing to $;;_c$ and `while` to `loop`.

Figure 11 shows success of the coherence criterion for the `while` case. Since the diagram commutes for all cases, (σ_E, b_E) is a map of GSOS laws between *While* and the secure version of *Low*. This guarantees that, remarkably, despite the presence of branching, a low-level attacker cannot illicitly access code that is unreachable on

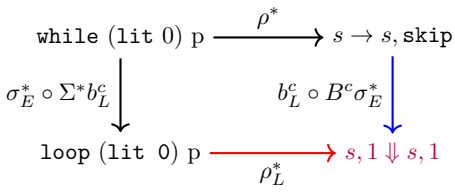


Fig. 11. The coherence criterion for (σ_E, b_L).

[8] We are not referring to the store, but to the internal, algebraic state.

the high-level. Regardless, the solution is a bit contrived in that the new *Low* primitives essentially copy what *While* does. This is partly because the above are complex issues involving radically different languages but also due to the current limitations of the underlying theory. We elaborate further on said limitations, as well as advantages and future improvements, at Section 9.

8 Local state encapsulation

High-level programming language abstractions often involve some sort of private state space that is protected from other objects. Basic examples include functions with local variables and objects with private members. Low-level languages do not offer such abstractions but when it comes to *secure architectures*, there is some type of *hardware sandboxing* [9] to facilitate the need for *local state encapsulation*. Compilation schemes that respect confidentiality properties have been a central subject in secure compilation work [37, 8, 18, 46], dating all the way back to Abadi's seminal paper [1].

In this example we will explore how local state encapsulation fails due to lack of stack clearing [46, 44]. We begin by extending *While* to support blocks which have their own private state, thus introducing $While_B$. More precisely, we add the `frame` and `return` commands that denote the beginning and end of a new block. We also have to modify the original behavior functor B to act on a stack of stores by simply specifying $B_B X = [S] \to [S] \times \text{Maybe } X$, where $[S]$ denotes a list of stores. For reasons that will become apparent later on, we shall henceforth consider stores of a certain length, say L.

$$\frac{}{m, \texttt{skip} \Downarrow m} \qquad \frac{v = \texttt{eval'}\ m\ e \quad m' = \texttt{update'}\ m\ l\ v}{m, l\ \texttt{:=}\ e \Downarrow m'} \qquad \frac{m, p \Downarrow m'}{m, p; q \to m', q}$$

$$\frac{m, p \to m', p'}{m, p; q \to m', p'; q} \qquad \frac{\texttt{eval'}\ m\ e = 0}{m, \texttt{while}\ e\ p \to m, \texttt{skip}}$$

$$\frac{\texttt{eval'}\ m\ e \neq 0}{m, \texttt{while}\ e\ p \to m, p; \texttt{while}\ e\ p} \qquad \frac{s_0 = [0, 0, \ldots, 0]}{m, \texttt{frame} \Downarrow s_0 :: m} \qquad \frac{}{s :: m, \texttt{return} \Downarrow m}$$

Fig. 12. Semantics of the $While_B$ language.

The semantics for $While_B$ can be found in Figure 12. Command `frame` allocates a new private store by appending one to the stack of stores while `return` pops the top frame from the stack. This built-in, automatic (de)allocation of frames guarantees that there are no traces of activity, in the form of stored values, of past blocks. The rest of the semantics are similar to *While*, only now `eval`uating an expression and `updat`ing the state acts on a stack of stores instead of a single, infinite store and `var` expressions act on the active, topmost frame.

[9] Examples of this are enclaves in Intel SGX [15] and object capabilities in CHERI [51].

$$\frac{m' = \texttt{update}\ m\ (l + L * sp)\ (\texttt{evalSP}\ m\ sp\ e)}{(m, sp), l\ \texttt{:=}\ e \Downarrow (m', sp)} \qquad \frac{sp > 0}{(m, sp), \texttt{return} \Downarrow (m, sp - 1)}$$

$$\frac{(m, sp), p \Downarrow (m', sp)}{(m, sp), p; q \to (m', sp), q} \qquad \frac{(m, sp), p \to (m', sp), p'}{(m, sp), p; q \to (m', sp), p'; q}$$

$$\frac{}{(m, sp), \texttt{skip} \Downarrow (m, sp)} \qquad \frac{\texttt{evalSP}\ m\ sp\ e = 0}{(m, sp), \texttt{while}\ e\ p \to (m, sp), \texttt{skip}}$$

$$\frac{\texttt{evalSP}\ m\ sp\ e \neq 0}{(m, sp), \texttt{while}\ e\ p \to (m, sp), p; \texttt{while}\ e\ p} \qquad \frac{}{(m, sp), \texttt{frame} \Downarrow (m, sp + 1)}$$

Fig. 13. Semantics of the *Stack* language.

Low-level stack In typical low-level instruction sets like the Intel x86 [21] or MIPS [30] there is a single, continuous *memory* which is partitioned in *frames* via processor registers. Figure 13 shows the semantics of *Stack*, a variant of $While_B$ with the same syntax that incorporates a simple *low-level* stack. The difference is that the stack frames are all sized L, the same size as each individual store in $While_B$, so at each `frame` and `return` we need only increment and decrement the *stack pointer*. The presence of the stack pointer, which is essentially a natural number, means that the behavior of *Stack* is $B_S X = S \times \mathbb{N} \to S \times \mathbb{N} \times \text{Maybe}\ X$. The new evaluation function, `evalSP`, works similarly to `eval` in Definition 1, except for `var` l expressions that dereference values at offset $l + L * sp$.

An insecure compiler $While_B$ and *Stack* share the same syntax so we only need a behavioral translation, which is all about relating the two different notions of stack. We thus define natural transformation $b_B : B_B \Longrightarrow B_S$:

$$b_B : \forall X.\ ([S_L] \to [S_L] \times \text{Maybe}\ X) \to S \to \mathbb{N} \to S \times \mathbb{N} \times \text{Maybe}\ X$$
$$b_B\ f\ s\ sp = (\text{override}\ (\text{join}\ m)\ s, \text{len}\ m, y)\ \texttt{where}\ (m, y) = f\ (\text{div}\ s\ sp)$$
$$\text{div}\ s\ sp = (\text{take}\ L\ s) :: (\text{div}\ (\text{drop}\ L\ s)\ (sp - 1))$$
$$\text{override}\ s'\ s = s'\ \texttt{++}\ \text{drop}\ (\text{len}\ s')\ s$$

We "divide" an infinite list by the number of stack frames, feed the result to the behavior function f and join ("flatten") it back together while keeping the original part of the infinite list which extends beyond the *active stack* intact. Note that in the case of the `frame` command f adds a new frame to the list of stores. The problem is that in $While_B$ the new frame is initialized to 0 in contrast to *Stack* where `frame` does not initialize new frames. This leads to a failure of the coherence criterion for (id, b_B) as we can see in Figure 14.

Failure of the criterion is meaningful in that it underlines key problems of this compiler which can be exploited by a low-level attacker. First, the low-level calling convention indirectly allows terms to access expired stack frames. Second, violating the assumption in $While_B$ that new frames are properly initialized breaks behavioral equivalence. For example, programs $a \triangleq$ `frame ; 0 := var`$[0]+1$ and $b \triangleq$ `frame ; 0 :=` 1 behave identically in $While_B$ but not in *Stack*.

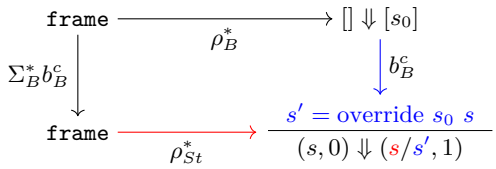


Fig. 14. Failure of the criterion for (id, b_B).

Solution It is clear that the lack of stack frame initialization in *Stack* is the lead cause of failure so we introduce the following fix in the `frame` rule.

$$\frac{m' = (\text{take } (L * sp)\ m)\ \texttt{++}\ s_0\ \texttt{++}\ (\text{drop } ((L+1)` * sp)\ m)}{(m, sp), \texttt{frame} \Downarrow (m', sp+1)}$$

The idea behind the new `frame` rule is that the L-sized block in position sp, which is going to be the new stack frame, has all its values replaced by zeroes. As we can see in Figure 15, the coherence criterion is now satisfied and the example described earlier no longer works.

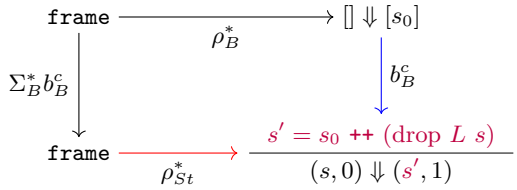


Fig. 15. The coherence criterion for (id, b_B) under the new `frame` rule.

9 Discussion and future work

On Mathematical Operational Semantics The cases we covered in this paper are presented using Plotkin's Structural Operational Semantics [38], yet their foundations are deeply categorical [48]. Consequently, for one to use the methods presented in this paper, the semantics involved must fall within the framework of distributive laws, the generality of which has been explored in the past [47, 50], albeit not exhaustively. To the best of our knowledge, Section 7 and Section 8 show the first instances of distributive laws as low-level machines.

Bialgebraic semantics are well-behaved in that *bisimilarity* is a *congruence* [19]. We used that to show that two bisimilar programs will remain bisimilar irrespective of the context they are plugged into, which is not the same as contextual equivalence. However, full abstraction is but one of a set of proposed characterizations of secure compilation [36, 2] and the key intuition is that our framework

is suitable as long as bisimilarity adequately captures the threat model. While this is the case in the examples, we can imagine situations where the threat model is *weaker* than the one implied by bisimilarity.

For example, language $While_{\Delta}$ in Section 3 includes labels in its transition structure and the underlying model is accurate in that $While_{\Delta}$ terms can manipulate said labels. However, if we were to remove `obs` statements from the syntax, the threat model becomes weaker than the one implied by bisimilarity. Similarly in Section 7 and *Low*, we could remove the implicit assumption that the program counter can be manipulated by a low-level attacker.

This issue can be classified as part of the broader effort towards coalgebraic weak bisimilarity, a hard problem which has been an object of intense, ongoing scientific research [42, 39, 20, 13, 43, 42, 12]. Of particular interest is the work by Abou-Saleh and Pattinson [7, 6] about bialgebraic semantics, where they use techniques introduced in [20] to obtain a more appropriate semantic domain for effectful languages as a final coalgebra in the Kleisli category of a suitable monad. This method is thus a promising avenue towards exploring weaker equivalences in bialgebraic semantics, as long as these can be described by a monad.

On Maps of Distributive Laws Maps of distributive laws were first mentioned by Power and Watanabe [40], then elaborated as *Well-behaved translations* by Watanabe [50] and more recently by Klin and Nachyla [27]. Despite the few examples presented in [50, 27], this paper is the first major attempt towards applying the theory behind maps of distributive laws in a concrete problem, let alone in secure compilation.

From a theoretical standpoint, maps of distributive laws have remained largely the same since their introduction. This comes despite the interesting developments discussed in Section 9 regarding distributive laws, which of course are the subjects of *maps* of distributive laws. We speculate the existence of *Kleisli* maps of distributive laws that guarantee preservation of equivalences weaker than bisimilarity. We plan to develop this notion and explore its applicability in future work.

Conclusion It is evident that the systematic approach presented in this work may markedly streamline proofs for compiler security as it involves a single, simple coherence criterion. Explicit reasoning about program contexts is no longer necessary, but that does not mean that contexts are irrelevant. On the contrary, the guarantees are implicitly *contextual* due to the well-behavedness of the semantics. Finally, while the overall usability and eventual success of our method remains a question mark as it depends on the expressiveness of the threat model, the body of work in coalgebraic weak bisimilarity and distributive laws in Kleisli categories suggests that there are many promising avenues for further progress.

Acknowledgements. This work was partially supported by the Research Fund KU Leuven. Andreas Nuyts holds a PhD fellowship from the Research Foundation - Flanders (FWO).

References

[1] Martín Abadi. "Protection in Programming-Language Translations". In: *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. 1999, pp. 19–34. DOI: 10.1007/3-540-48749-2_2. URL: https://doi.org/10.1007/3-540-48749-2_2.

[2] Carmine Abate et al. *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*. 2018. arXiv: 1807.04603 [cs.PL].

[3] Carmine Abate et al. "When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, 2018, pp. 1351–1368. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243745. URL: https://doi.org/10.1145/3243734.3243745.

[4] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing strictly positive types". In: *Theor. Comput. Sci.* 342.1 (2005), pp. 3–27. DOI: 10.1016/j.tcs.2005.06.002. URL: https://doi.org/10.1016/j.tcs.2005.06.002.

[5] Michael Gordon Abbott et al. "for Data: Differentiating Data Structures". In: *Fundam. Inform.* 65.1-2 (2005), pp. 1–28. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-02.

[6] Faris Abou-Saleh. "A coalgebraic semantics for imperative programming languages". PhD thesis. Imperial College London, UK, 2014. URL: http://hdl.handle.net/10044/1/13693.

[7] Faris Abou-Saleh and Dirk Pattinson. "Towards Effects in Mathematical Operational Semantics". In: *Electr. Notes Theor. Comput. Sci.* 276 (2011), pp. 81–104. DOI: 10.1016/j.entcs.2011.09.016. URL: https://doi.org/10.1016/j.entcs.2011.09.016.

[8] Pieter Agten et al. "Secure Compilation to Modern Processors". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 171–185. ISBN: 978-1-4673-1918-8. DOI: 10.1109/CSF.2012.12. URL: https://doi.org/10.1109/CSF.2012.12.

[9] Amal Ahmed and Matthias Blume. "An equivalence-preserving CPS translation via multi-language semantics". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 431–444. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034830. URL: https://doi.org/10.1145/2034773.2034830.

[10] Amal Ahmed et al. "Secure Compilation (Dagstuhl Seminar 18201)". In: *Dagstuhl Reports* 8.5 (2018). Ed. by Amal Ahmed et al., pp. 1–30. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.5.1. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9891.

[11] Falk Bartels. *On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalgebraic Modelling*. 2004.

[12] Filippo Bonchi et al. "Lax Bialgebras and Up-To Techniques for Weak Bisimulations". In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. Ed. by Luca Aceto and David de Frutos-Escrig. Vol. 42. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 240–253. ISBN: 978-3-939897-91-0. DOI: 10.4230/LIPIcs.CONCUR.2015.240. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2015.240.

[13] Tomasz Brengos. "Weak bisimulation for coalgebras over order enriched monads". In: *Logical Methods in Computer Science* 11.2 (2015). DOI: 10.2168/LMCS-11(2:14)2015. URL: https://doi.org/10.2168/LMCS-11(2:14)2015.

[14] J. Robin B. Cockett. "Introduction to Distributive Categories". In: *Mathematical Structures in Computer Science* 3.3 (1993), pp. 277–307. DOI: 10.1017/S0960129500000232. URL: https://doi.org/10.1017/S0960129500000232.

[15] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

[16] Dominique Devriese, Marco Patrignani, and Frank Piessens. "Fully-abstract compilation by approximate back-translation". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 164–177. DOI: 10.1145/2837614.2837618. URL: https://doi.org/10.1145/2837614.2837618.

[17] Derek Dreyer, Amal Ahmed, and Lars Birkedal. "Logical Step-Indexed Logical Relations". In: *Logical Methods in Computer Science* 7.2 (2011). DOI: 10.2168/LMCS-7(2:16)2011. URL: https://doi.org/10.2168/LMCS-7(2:16)2011.

[18] Cédric Fournet et al. "Fully abstract compilation to JavaScript". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 2013, pp. 371–384. DOI: 10.1145/2429069.2429114. URL: https://doi.org/10.1145/2429069.2429114.

[19] Jan Friso Groote and Frits W. Vaandrager. "Structured Operational Semantics and Bisimulation as a Congruence". In: *Inf. Comput.* 100.2 (1992), pp. 202–260. DOI: 10.1016/0890-5401(92)90013-6. URL: https://doi.org/10.1016/0890-5401(92)90013-6.

[20] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. "Generic Trace Semantics via Coinduction". In: *Logical Methods in Computer Science* 3.4 (2007). DOI: 10.2168/LMCS-3(4:11)2007. URL: https://doi.org/10.2168/LMCS-3(4:11)2007.

[21] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. 2016. URL: https://www.intel.com/content/dam/www/public/

us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[22] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Vol. 59. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016. ISBN: 9781316823187. DOI: 10.1017/CBO9781316823187. URL: https://doi.org/10.1017/CBO9781316823187.

[23] Bart Jacobs. "Parameters and Parametrization in Specification, Using Distributive Categories". In: *Fundam. Inform.* 24.3 (1995), pp. 209–250. DOI: 10.3233/FI-1995-2431. URL: https://doi.org/10.3233/FI-1995-2431.

[24] Radha Jagadeesan et al. "Local Memory via Layout Randomization". In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 2011, pp. 161–174. ISBN: 978-1-61284-644-6. DOI: 10.1109/CSF.2011.18. URL: https://doi.org/10.1109/CSF.2011.18.

[25] Andrew Kennedy. "Securing the .NET programming model". In: *Theor. Comput. Sci.* 364.3 (2006), pp. 311–317. DOI: 10.1016/j.tcs.2006.08.014. URL: https://doi.org/10.1016/j.tcs.2006.08.014.

[26] Bartek Klin. "Bialgebras for structural operational semantics: An introduction". In: *Theor. Comput. Sci.* 412.38 (2011), pp. 5043–5069. DOI: 10.1016/j.tcs.2011.03.023. URL: https://doi.org/10.1016/j.tcs.2011.03.023.

[27] Bartek Klin and Beata Nachyla. "Presenting Morphisms of Distributive Laws". In: *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015, June 24-26, 2015, Nijmegen, The Netherlands*. 2015, pp. 190–204. DOI: 10.4230/LIPIcs.CALCO.2015.190. URL: https://doi.org/10.4230/LIPIcs.CALCO.2015.190.

[28] Marina Lenisa, John Power, and Hiroshi Watanabe. "Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads". In: *Electr. Notes Theor. Comput. Sci.* 33 (2000), pp. 230–260. DOI: 10.1016/S1571-0661(05)80350-0. URL: https://doi.org/10.1016/S1571-0661(05)80350-0.

[29] Conor Mcbride. *The Derivative of a Regular Type is its Type of One-Hole Contexts (Extended Abstract)*. 2001.

[30] *MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual*. MIPS Technologies. 2016. URL: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf.

[31] James H. Morris. "Lambda-Calculus Models of Programming Languages". PhD thesis. Massachusetts Institute of Technology, 1968.

[32] Max S. New, William J. Bowman, and Amal Ahmed. "Fully abstract compilation via universal embedding". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 103–116. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951941. URL: https://doi.org/10.1145/2951913.2951941.

[33] Marco Patrignani, Amal Ahmed, and Dave Clarke. "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work". In: *ACM Comput. Surv.* 51.6 (Feb. 2019), 125:1–125:36. ISSN: 0360-0300. DOI: 10.1145/3280984.

[34] Marco Patrignani, Dave Clarke, and Frank Piessens. "Secure Compilation of Object-Oriented Components to Protected Module Architectures". In: *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. Ed. by Chung-chieh Shan. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 176–191. ISBN: 978-3-319-03541-3. DOI: 10.1007/978-3-319-03542-0_13. URL: https://doi.org/10.1007/978-3-319-03542-0_13.

[35] Marco Patrignani, Dominique Devriese, and Frank Piessens. "On Modular and Fully-Abstract Compilation". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 17–30. ISBN: 978-1-5090-2607-4. DOI: 10.1109/CSF.2016.9. URL: https://doi.org/10.1109/CSF.2016.9.

[36] Marco Patrignani and Deepak Garg. "Robustly Safe Compilation". In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. 2019, pp. 469–498. DOI: 10.1007/978-3-030-17184-1_17. URL: https://doi.org/10.1007/978-3-030-17184-1_17.

[37] Marco Patrignani et al. "Secure Compilation to Protected Module Architectures". In: *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 6:1–6:50. DOI: 10.1145/2699503. URL: https://doi.org/10.1145/2699503.

[38] Gordon D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 17–139.

[39] Andrei Popescu. "Weak Bisimilarity Coalgebraically". In: *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*. 2009, pp. 157–172. DOI: 10.1007/978-3-642-03741-2_12. URL: https://doi.org/10.1007/978-3-642-03741-2_12.

[40] John Power and Hiroshi Watanabe. "Distributivity for a monad and a comonad". In: *Electr. Notes Theor. Comput. Sci.* 19 (1999), p. 102. DOI: 10.1016/S1571-0661(05)80271-3. URL: https://doi.org/10.1016/S1571-0661(05)80271-3.

[41] Jurriaan Rot et al. "Enhanced coalgebraic bisimulation". In: *Mathematical Structures in Computer Science* 27.7 (2017), pp. 1236–1264. DOI: 10.1017/S0960129515000523. URL: https://doi.org/10.1017/S0960129515000523.

[42] Jan Rothe and Dragan Masulovic. "Towards Weak Bisimulation For Coalgebras". In: *Electr. Notes Theor. Comput. Sci.* 68.1 (2002), pp. 32–46. DOI: 10.1016/S1571-0661(04)80499-7. URL: https://doi.org/10.1016/S1571-0661(04)80499-7.

[43] Jan J. M. M. Rutten. "A note on coinduction and weak bisimilarity for while programs". In: *ITA* 33.4/5 (1999), pp. 393–400. DOI: 10.1051/ita:1999125. URL: https://doi.org/10.1051/ita:1999125.

[44] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 475–501. ISBN: 978-3-319-89883-4. DOI: 10.1007/978-3-319-89884-1_17. URL: https://doi.org/10.1007/978-3-319-89884-1_17.

[45] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 19:1–19:28. ISSN: 2475-1421. DOI: 10.1145/3290332.

[46] Stelios Tsampas, Dominique Devriese, and Frank Piessens. "Temporal Safety for Stack Allocated Memory on Capability Machines". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 243–255. ISBN: 978-1-7281-1407-1. DOI: 10.1109/CSF.2019.00024. URL: https://doi.org/10.1109/CSF.2019.00024.

[47] Daniele Turi. "Categorical Modelling of Structural Operational Rules: Case Studies". In: *Category Theory and Computer Science, 7th International Conference, CTCS '97, Santa Margherita Ligure, Italy, September 4-6, 1997, Proceedings*. 1997, pp. 127–146. DOI: 10.1007/BFb0026985. URL: https://doi.org/10.1007/BFb0026985.

[48] Daniele Turi and Gordon D. Plotkin. "Towards a Mathematical Operational Semantics". In: *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*. 1997, pp. 280–291. DOI: 10.1109/LICS.1997.614955. URL: https://doi.org/10.1109/LICS.1997.614955.

[49] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. "Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code". In: *Proc. ACM Program. Lang.* ICFP (2019). accepted.

[50] Hiroshi Watanabe. "Well-behaved Translations between Structural Operational Semantics". In: *Electr. Notes Theor. Comput. Sci.* 65.1 (2002), pp. 337–357. DOI: 10.1016/S1571-0661(04)80372-4. URL: https://doi.org/10.1016/S1571-0661(04)80372-4.

[51] Robert N. M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.9. URL: https://doi.org/10.1109/SP.2015.9.