Type Checking Cryptography Implementations

Manuel Barbosa[1], Andrew Moss[2], Dan Page[3],

Nuno F. Rodrigues[1,4], Paulo F. Silva[1]

[1]Departamento de Informática, Universidade do Minho

[2]School of Computing, Blekinge Institute of Technology

[3]Department of Computer Science, University of Bristol

[4]DIGARC, Instituto Politécnico do Cávado e do Ave

Techn. Report DI-CCTC-11-01
2011, February

Computer Science and Technology Center
Departamento de Informática da Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://cctc.di.uminho.pt/

DI-CCTC-11-01

Type Checking Cryptography Implementations

by Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues and Paulo F. Silva

Abstract

Cryptographic software development is a challenging field: high performance must be achieved, while ensuring correctness and compliance with low-level security policies. CAO is a domain specific language designed to assist development of cryptographic software. An important feature of this language is the design of a novel type system introducing native types such as predefined sized vectors, matrices and bit strings, residue classes modulo an integer, finite fields and finite field extensions, allowing for extensive static validation of source code. We present the formalisation, validation and implementation of this type system.

1 Introduction

The development of cryptographic software is clearly distinct from other areas of software engineering. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. Also, since security is difficult to sell as a feature in software products, cryptography needs to be as close to invisible as possible in terms of computational and communication load. As a result, cryptographic software must be optimised aggressively, without altering the security semantics. Finally, cryptographic software is implemented on a very wide range of devices, from embedded processors with very limited computational power and memory, to high-end servers, which demand high-performance and low-latency. Therefore, the implementation of cryptographic kernels imposes a specific set of challenges that do not apply to other system components. For example, direct implementation in assembly language is common, not only to guarantee a more efficient implementation, but also to ensure that low-level security policies are satisfied by the machine code.

The CAO language. The CAO language aims to change this state of affairs, allowing natural description of cryptographic software implementations, which can be analysed by a compiler that performs security-aware analysis, transformation and optimisation. The driving principle behind the design of CAO is that the language should support cryptographic concepts as first-class language features. Unlike the languages used in mathematical software packages such as Magma or Maple, which allow the description of high-level mathematical constructions in their full generality, CAO is restricted to enabling the implementation of cryptographic components such as block ciphers, hash functions and sequences of finite field arithmetic for Elliptic Curve Cryptography (ECC).

CAO preserves some higher-level features to be familiar to an imperative programmer, whilst focusing on the implementation aspects that are most critical for security and efficiency. The memory model of CAO is, by design, extremely simple to prevent memory management errors (there is no dynamic memory allocation and it has call-by-value semantics). Furthermore, the language does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries. In fact, a typical CAO program comprises only the definition of a global state and a set of functions that permit performing cryptographic operations over that state. Conversely, the native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. In short, the design of CAO allowed trading off the generality of a language such as C or Java, for a richer type system that permits expressing cryptographic software implementations in a more natural way.

CAO introduces as first-class features pure incarnations of mathematical types commonly used in cryptography (arbitrary precision integers, ring of residue classes modulo an integer, finite field of residue classes modulo a prime, finite field extensions and matrices of these mathematical types) and also bit strings of known finite size. A more expressive type system would be expected from any domain-specific language. However, in the case of CAO, the design of the type system was taken a step further in order not only to allow an elegant formalisation of the type checking rules, but also to allow the efficient implementa-

tion of a type checking system that performs extensive preliminary validation of the code, and extracts a very rich body of information from it. This fact makes the CAO type checker a critical building block in the implementation of compilation and formal verification tools supporting the language.

Contributions. This paper presents the formalisation, validation and implementation of the CAO type system. Our main contribution is to show that the trade-offs in language features that were introduced in the design of CAO – specifically for cryptographic software implementation – enabled us to tame the complexity of formalising and validating a surprisingly powerful type system. We also show, resorting to practical examples, how this type system enforces strong typing rules and how these rules detect several common run-time errors. To support this claim, we outline our proof of soundness of the CAO type system.

More in detail, we describe a formalisation of the CAO type system and the corresponding implementation of a type checker[1] as a front-end of the CAO tool chain. One of the main achievements of our system is the enforcement of strong typing rules that are aware of type parameters in the data types of the language. The type checking rules permit determining concrete values for these parameters and, furthermore, resolving the consistency of these parameters inside CAO programs. Concretely, the CAO type system explicitly includes as type parameters the sizes of containers such as vectors, matrices and bit strings. In other words, CAO is dependently typed. Furthermore, typing of complex operations over these containers, including concatenation and extensional assignment, statically checks the compatibility of these parameters.

More interestingly, we are able to handle parameters in mathematical types in a similar way. Our type system maintains information for the concrete values of integer moduli and polynomial moduli, so that it is possible to validate the consistency of complex mathematical expressions, including group and finite field operations, the conversion between a finite field element and its polynomial representation, and other type conversions. Finally, the CAO type system also deals with language usability issues that include implicit (automatic) type conversions between bit strings and the integer value that they represent, and also between values within the same finite field extension hierarchy.

Paper organisation. In Section 2 we expand on the relevant features of CAO. We then build some intuition for the subsequent formal presentation of the type system by introducing real-world examples of CAO code in Section 3. In Section 4 we present the CAO type system, including a detailed example of its operation. In Section 5 we describe our implementation. We conclude with a discussion of soundness and related work in Sections 6 and 7.

2 A closer look at CAO

Real world examples of the most relevant CAO language features are presented in Section 3. We now provide an intuitive description of the CAO type system.

[1] An implementation of a CAO interpreter (including the type system and semantics) is available via `http://www.cace-project.eu`.

Bit strings. The bits type represents a string of n bits (labelled $0 \ldots n-1$, where the 0-th is the least-significant bit). This should not be seen as the "bit vector" type, as the get operator a[i] actually returns type bits[1]. The distinction between ubits and sbits concerns only the conversion convention to the integer type, which can be unsigned or two's complement respectively. The bits type is equipped with a set of C-like bit-wise operators, including the usual Boolean, shift and rotate operators, which are closed over the bit-length. The range selection/assignment (or slicing) operator (..), combined with the concatenation operator @ can be used to (de)construct bit strings of different sizes using a very concise syntax. For example, the following is a valid CAO statement over bit strings:

```
a[3..8] := b[0..2] @ c[2..4];
```

Integers and the mod type. Operations modulo some prime or composite integer are used extensively in cryptography [5]; for example, the ring[2] $\mathbb{Z}_n$ underlies the pervasively used RSA function [3], and the finite field[3] $\mathbb{F}_p$ is widely used in ECC. Therefore, CAO includes not only arbitrary precision integers as a native type (int), but also a mod[n] type. For example, the mod[7] type is an instance of mod with modulus 7. In this case the modulus is prime, and hence inhabitants of this type are actually elements of a finite field. More generally, the modulus can be prime or composite, provided it is fixed at compile-time. Algebraic operations over the mod type are closed over the modulus parameter.

Internal representation and Casts. The internal representation of mathematical types is deliberately undefined. The CAO semantics ensures that arithmetic with such values is valid, but makes no guarantee about (and hence disallows access to) their physical representation. Nevertheless, the CAO type system includes the necessary functionality to access the conceptually natural representation of algebraic types, by supporting appropriate cast operators. For example, to obtain the representation of a finite field element in mod[p] as an integer value of the appropriate range, one simply casts it into the int type. To obtain the representation of an arbitrary precision integer, one can cast it into a bit string of a predetermined size, and so on. Hence, compared to C, a CAO cast is more explicitly a conversion. Aside from this nuance, the syntax of casts is similar to C: one specifies the target type in parenthesis, e.g. y := (int) x.

General moduli. An alternative form of the mod type allows defining finite field extensions, as shown below:

```
typedef a := mod[ 2 ];
typedef b := mod[ a<X> / X**8 + X**4 + X**3 + X + 1 ];
```

The type synonym a represents a mod type whose modulus is 2; this is simply the field $\mathbb{F}_2$. This is used as the base type for a second type synonym b which represents the field $\mathbb{F}_{2^8}$. In addition to the base type one also specifies an indeterminate symbol (in this case

[2] The ring of residue classes modulo an integer n can be seen as the set of numbers in the range 0 to n-1 with addition and multiplication modulo n.

[3] The ring of residue classes modulo an integer p is actually a field when p is prime: all non-zero elements have a multiplicative inverse.

X), and an irreducible polynomial in the ring of polynomials with coefficients in the base type (in this case $P(X) = X^8 + X^4 + X^3 + X + 1$). Intuitively, this declaration defines an implementation of the field based on the referred polynomial ring, with arithmetic defined via standard polynomial algebra with reductions modulo $P(X)$. To access the coefficients in this representation, one can cast the value into a vector of elements in the base type.

MATRICES. The matrix type represents a 2-dimensional algebraic matrix over which one can perform addition and multiplication. For this reason, there are some restrictions on what the base type can be. The matrix type also has an undefined representation; its size must be fixed at compile-time, but the ordering of elements in memory (e.g. row-major or column-major order) is a choice that can be made by the compiler. The matrix type also supports get and range selection/assignment operations that permit easily (de)constructing matrices of different sizes.

VECTORS. The vector type represents a 1-dimensional generic container of elements of homogeneous type, where each element is referred to by a single index in the range $0 \ldots n-1$, offering selection/assignment, concatenation and rotate operations similar to the bits type.

3 CAO Type System in Action

In this section we present some examples of CAO code taken from the implementation of the NaCl cryptographic library[4] that illustrate the validation capacity of the type checker over real world examples.

The following program fragment was taken from the implementation of the poly1305 one-time message authentication mechanism [1]. The function receives two vectors ciu and ru of content type byte, which is an alias for type unsigned bits[8], and an integer q. It returns a value of type mod1305, an alias for type mod[2**130-5].

```
def polyStep(ciu:vector[17] of byte, ru:vector[16] of byte, q:int) : mod1305 {
    def r : unsigned bits[16*8]; def ci : unsigned bits[17*8];

    r := ru[0]@ru[1]@ru[2]@ru[3]@ru[4]@ru[5]@ru[6]@ru[7]@ru[8]@ru[9]@ru[10]@
         ru[11]@ru[12]@ru[13]@ru[14]@ru[15];

    ci:= ciu[0]@ciu[1]@ciu[2]@ciu[3]@ciu[4]@ciu[5]@ciu[6]@ciu[7]@ciu[8]@
         ciu[9]@ciu[10]@ciu[11]@ciu[12]@ciu[13]@ciu[14]@ciu[15]@ciu[16];

    return ((mod1305)ci * (mod1305)r**q); }
```

The type system must solve the following problems to type the function body. Firstly, the concatenation of several bit strings must be typed to a single bit string of the appropriate type and size (and fail if these do not match in assignment). Secondly, the type checker must recognise that the cast to type mod1305 requires the expression on the right to be coerced to type int.

The next program fragment is from the NaCl implementation of hsalsa20 [2].

[4] http://nacl.cr.yp.to

```
seq i := 0 to 3 {
    x[i+1]  :=  from_littleendian( k[i*4..i*4+3]);
    x[i+6]  :=  from_littleendian(in[i*4..i*4+3]);
    x[i+11] :=  from_littleendian( k[i*4+16..i*4+19]); }
...
seq i := 0 to 3 {
    out[i*4..i*4+3]     := to_littleendian(x[5*i]);
    out[i*4+16..i*4+19] := to_littleendian(x[i+6]); }
```

This is a good example of how CAO was fine tuned to provide assistance to the programmer in what, at first sight, might seem like a surprisingly powerful validation procedure. Range selection and assignment operators in bit strings, vectors and matrices may depend on the value of integer expressions, which can only be formed by literals, constants and basic arithmetic operations that can be evaluated at compile-time. This might seem just like a pre-processing step of compilation, were it not for the fact that we are also able to include in these expressions locally defined constants. Our type system is able to validate that all range selections (resp. assignments) result in vectors that are compatible with calls to function from_littleendian (resp. return type of function to_littleendian).

Finally, the following code snippet is extracted from a CAO implementation of AES. It shows how our type system is capable of dealing with the complex mathematical types that arise in cryptographic implementations. In this case we have a matrix multiplication operation mix * s[0..3,i], where the contents of the matrices are elements of a finite field extension GF2N.

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X> / X**8 + X**4 + X**3 + X + 1 ];
typedef S    := matrix[4,4] of GF2N;

def mix : matrix[4,4] of GF2N  :=
  {[X],[X+1],[1],[1],[1],[X],[X+1],[1],[1],[1],[X],[X+1],[X+1],[1],[1],[X]};

def MixColumns( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { r[0..3,i] := mix * s[0..3,i]; }
  return r; }
```

In addition to resolving the matrix size restrictions imposed by the matrix multiplication operation, our type system is able to individually type the finite field literals in the matrix initialisation, and check that these types are compatible with the type of the matrix contents. Note that this implies recognising that a literal of type mod[2] is coercible to GF2N.

4 Formalisation of the CAO Type System

In this Section, we will overview our formalisation of the CAO type system. Since CAO is a relatively large language, only the most interesting features will be covered. A full description of the CAO type system can be found in Appendix A.

CAO Syntax. The formal syntax of CAO is presented in Figure 1. To simplify presentation we use † to represent a set of traditional binary operators, namely

$$\dagger \in \{+, -, *, /, \%, **, \&, \hat{}, |, \gg, \ll, @, ==, !=, <, >, <=, >=, ||, \&\&, \hat{}\hat{}\}$$

n : **Num**	Numerals	pg : **Progs**	Programs
x : $\mathbf{Id}_V$	Variable Identifiers	e : **Exp**	Expressions
fp : $\mathbf{Id}_{FP}$	Function and Procedure Identifiers	c : **Stm**	Statements
dv : $\mathbf{Dec}_V$	Variable declarations	l : **Lv**	LValues
dfp : $\mathbf{Dec}_{FP}$	Function and Procedure declarations	pol : **Poly**	Polynomials
ds : $\mathbf{Dec}_S$	Struct declarations	t : **Types**	Types

$$
\begin{aligned}
e &::= n \mid \mathsf{true} \mid \mathsf{false} \mid x \mid -e \mid e_1 \dagger e_2 \mid e.x \mid e_1[e_2] \mid e_1[e_2..e_3] \mid \\
&\quad e_1[e_2, e_3] \mid e_1[e_2..e_3, e_4..e_5] \mid \sim e \mid (t)\ e \mid fp(e_1, ..., e_n) \mid !\ e \\
l &::= x \mid l.x \mid l[e] \mid l[e_1..e_2] \mid l[e_1, e_2] \mid l[e_1..e_2, e_3..e_4] \\
c &::= dv \mid l_1, ..., l_i := e_1, ..., e_j \mid c_1; c_2 \mid \mathsf{if}\ (e)\ \{\ c_1\ \} \mid \mathsf{if}\ (e)\ \{\ c_1\ \}\ \mathsf{else}\ \{\ c_2\ \} \mid \\
&\quad \mathsf{while}\ (e)\ \{\ c\ \} \mid \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} \mid \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} \mid \\
&\quad \mathsf{return}\ e_1, ..., e_n \mid fp(e_1, ..., e_n) \\
dv &::= \mathsf{def}\ x_1, ..., x_n : t_1, ..., t_n \mid \mathsf{def}\ x_1, ..., x_n : t_1, ..., t_n := e_1, ..., e_n \\
ds &::= \mathsf{typedef}\ x := t; \mid \mathsf{typedef}\ x_1 := \mathsf{struct}\ [\ \mathsf{def}\ x_2 : t_1; ...; \mathsf{def}\ x_n : t_n\ ]; \\
dfp &::= \mathsf{def}\ fp\ (x_1 : t_1, ..., x_n : t_n) : rt\ \{\ c\ \} \\
rt &::= \mathsf{void} \mid t_1, \ldots, t_n \\
t &::= x \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{signed\ bits}\ [e] \mid \mathsf{unsigned\ bits}\ [e] \mid \mathsf{mod}\ [e] \mid \mathsf{mod}\ [\ t\ x\ /\ pol\ ] \mid \\
&\quad \mathsf{vector}\ [n]\ \mathsf{of}\ t \mid \mathsf{matrix}\ [n_1, n_2]\ \mathsf{of}\ t \\
pg &::= dv; \mid ds \mid dfp \mid pg_1\ pg_2
\end{aligned}
$$

Fig. 1: Formal syntax of CAO

Most of the binary operators are the same as their C equivalents, although they are overloaded for multiple types. Worth mentioning are the multiplicative exponentiation operator for integers, residue class groups and fields ($**$); the bit-wise conjunction (AND), inclusive- (IOR) and exclusive-disjunction (XOR) operators (&, | and ^ respectively); the shift operators for bit strings and vectors ($\gg$ and $\ll$); the concatenation operator for bit strings and vectors @; and the boolean logic exclusive-disjunction (XOR) operator (^^).

Most of the language syntactic entities, and the accompanying syntax rules, are also similar to C. Additional domains have been added to this basic set: some for the sake of a clearer presentation, and others because they are part of CAO's domain specific character for cryptography.

4.1 CAO Type System

Function Classification. The type checker is able to automatically classify CAO functions with respect to their interaction with global variables. The type checking rules classify functions as either of the following three types:

Pure functions Do not depend on global variables in any way and can only call other pure functions. These functions are, not only side-effect free, but also return the same result in every invocation with the same input. This property is often called referential transparency.

Read-only functions Can read values from global variables, but they cannot assign values to them. They can call pure functions and other read-only functions, but not procedures. These functions are side-effect free.

Procedures Can read and assign values from/to global variables. They can call pure functions, read-only functions and other procedures.

For the CAO type checker, the most important distinction is that between procedures and other functions. Procedures are only admitted in restricted contexts, such as simple assignment constructions. This distinction is completely automated in the type-checking rules that associate the following total order of classifiers to CAO constructions: Pure $<$ ReadOnly $<$ Procedure

Put simply, the type checking system enforces the following rules: 1) A construction depending only on local variables is classified as Pure; 2) When reading the value of a global variable, the classifier is set to Read-only; 3) When a global variable is used in an assignment target, the classifier is set to Procedure; 4) Expressions and statements procedures are classified with respect to their sub-elements using the *maximum operator* defined over the total order specified above. Note that this classification system is conservative in the sense that, for example, it will fail to correctly classify a function as pure when it reads a global variable but does not use its value.

Environments, type judgements and conventions. We use symbol τ (possibly with subscripts) to represent an arbitrary (fixed) data type. We write $x :: \tau$ to denote that x has type τ. We use two distinct environments in our type rules: the type environment relation Γ, which collects all the declarations (e.g. variables, function, procedures) together with their associated types; and the constant environment relation Δ, which records the values associated with integer constants. The Γ environment is partitioned into two relations: Γ_G for global definitions and Γ_L for local definitions. This distinction is important to deal with symbol scoping and visibility when typing, for example function declarations. Whenever this distinction is not important we will just write Γ to abbreviate Γ_G, Γ_L. Notation $\Gamma[x :: \tau]$ is used to extend the environment Γ with a new variable x of type τ, providing that x is not in the original environment (i.e., $x \notin dom(\Gamma)$). Similarly, $\Delta[x := n]$ is used to extend the environment Δ with a new constant x with value n, also provided that x is not in the domain of environment Δ. Notation $\Gamma(x)$ and $\Delta(x)$ represent, respectively, the type and the integer value associated with identifier x, assuming that x belongs to the domain of the respective environment. Environments are built by order of declaration in source code, implying that recursive declarations are not possible and that function classifiers are already known when the functions are called.

We use symbol $\vdash$ for type judgement of expressions of the form $\Gamma, \Delta \vdash e :: (\tau, \mathsf{c})$, retrieving type τ and functional classifier c associated to an expression. Operator $\Vdash_\beta$ denotes type judgements of statements that may modify the type environment relation: it retrieves not only a typed statement, but also a new type environment relation. Subscript β (seen as a place-holder) in operator $\Vdash_\beta$ represents the return type of the function in which the statement was defined. This information is particularly useful, allowing the type checker to guarantee that the several return statements that may appear in a function are always in accordance with the return type of the corresponding function declaration.

Evaluation of integer expressions. We define a partial function ϕ_Δ to deal with type parameters such as vector sizes that must be determined at compile time. This function is used in typing rules to compute the integer value of a given expression e in context Δ. If this value cannot be determined, then typing will fail. This function is defined as follows

Table 1: CAO data types.

Bool	Booleans
Int	Arbitrary precision integers
UBits $[i]$	Unsigned bit strings of length i
SBits $[i]$	Signed bit strings of length i
Mod $[n]$	Rings or fields defined by integer n
Mod $[\tau/pol]$	Extension field defined by τ/pol
Vector $[i]$ of τ	Vectors of i elements of type τ
Matrix $[i,j]$ of α	Matrices of $i \times j$ elements of type $\alpha \in \mathcal{A}$

$$\mathcal{A} = \{\mathsf{Int}, \mathsf{Mod}\ [m], \mathsf{Matrix}\ [i,j]\ \mathsf{of}\ \alpha \ \mid \alpha \in \mathcal{A}\}$$

$$\phi_\Delta(n) = n \qquad \phi_\Delta(x) = \Delta(x),\ x \in dom\ \Delta$$
$$\phi_\Delta(-e) = -\phi_\Delta(e) \qquad \phi_\Delta(e_1 \dagger e_2) = \phi_\Delta(e_1) \dagger \phi_\Delta(e_2)$$
$$\phi_\Delta(e_1 ** e_2) = (\phi_\Delta(e_1))^{(\phi_\Delta(e_2))} \qquad \phi_\Delta(e_1 \,\%\, e_2) = \phi_\Delta(e_1) \mod \phi_\Delta(e_2)$$

for $\dagger \in \{+, -, *, /\}$. When evaluating integer expressions in typing rules, we write

$$\frac{\ldots \quad \phi_\Delta(e) = n \quad \ldots}{\Gamma, \Delta \vdash \ldots} \text{ to mean } \frac{\ldots \quad \Gamma, \Delta \vdash e :: (\mathsf{Int}, \mathsf{Pure}) \quad \phi_\Delta(e) = n \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

which implicitly implies that expression e is of integer type.

Data types. In Section 2, types were informally described using CAO syntax for type declarations. Here we will distinguish between a type declaration and the type it refers to in our formalisation. We use upper case to indicate the CAO data types shown in Table 1. An important difference is that the CAO grammar allows any expression as a parameter of a type declaration, while CAO types must have parameters of the correct type and with a fully determined value, e.g., sizes must be integer values. In Table 1, $\mathcal{A}$ denotes the set of algebraic types, which are the only ones that can be used to construct matrices. These are types for which addition, multiplication and symmetric operators are closed. In order to emphasise occurrences where the type must be algebraic, we will use α (possibly with subscripts) instead of τ.

Type translation. To deal with the type parameters informally described in Section 1, we introduce a new judgement that makes the translation between type declaration in the CAO syntax and types used in the type checking process. This judgement, of the form $\Delta \vdash_t t \rightsquigarrow \tau$, depends only on the environment Δ, which can in turn be used to determine the values of expressions that only depend on constants. This accounts for the fact that, during type checking, types must have their parameters fully determined, while type declarations in CAO can depend on arithmetic expressions using constants stored in the environment Δ. Hence the translation judgement uses evaluation function ϕ_Δ to compute parameter expressions in the declaration of bit string, vector and matrix sizes, ensuring that no negative or zero sizes are used. The evaluation function is also used in modular types with integer modulus to determine its value and ensure that it is meaningful (i.e., greater than 1). We present only part of this definition below.

Table 2: Type coercion relation, $\vdash_{\leq} t_1 \leq t_2$

$\mathbf{t_1}$	$\mathbf{t_2}$	**Condition**
UBits$[n]$	Int	
SBits$[n]$	Int	
τ	Mod$[\tau'/pol]$	$\vdash_{\leq} \tau \leq \tau'$
Vector$[n]$ of τ_1	Vector$[n]$ of τ_2	$\vdash_{\leq} \tau_1 \leq \tau_2$
Matrix $[i,j]$ of α_1	Matrix $[i,j]$ of α_2	$\vdash_{\leq} \alpha_1 \leq \alpha_2$ and $\alpha_1, \alpha_2 \in \mathcal{A}$

$$\frac{\phi_{\Delta}(e) = n}{\Delta \vdash_t \text{unsigned bits } [e] \rightsquigarrow \text{UBits}[n]} n \geq 1 \qquad \frac{\phi_{\Delta}(e) = n}{\Delta \vdash_t \text{mod } [e] \rightsquigarrow \text{Mod}[n]} n \geq 2$$

$$\frac{\phi_{\Delta}(e) = n \quad \Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \vdash_t \text{vector } [e] \text{ of } t \rightsquigarrow \text{Vector } [n] \text{ of } \tau} n \geq 1$$

$$\frac{\phi_{\Delta}(e_1) = n \quad \phi_{\Delta}(e_2) = m \quad \Delta \vdash_t t \rightsquigarrow \alpha}{\Delta \vdash_t \text{matrix } [e_1, e_2] \text{ of } t \rightsquigarrow \text{Matrix } [n, m] \text{ of } \alpha} \quad \alpha \in \mathcal{A}, n \geq 1, m \geq 1$$

Type coercions. Type coercions are essentially implicit (typically data preserving) type conversions, whereby the programmer is allowed to use terms of some type whenever another type is expected. In CAO, this mechanism is remarkably useful, for example when dealing with field extensions (cf. the third rule in Table 2), since a field can be seen as a subtype of all its field extensions. In general, when a CAO type τ_1 is coercible to another type τ_2, then the set of values in τ_1 can be seen as a subset of the values in τ_2. For example, all bit-strings of a given size can be coerced to the integer type. We define a coercion relation $\leq$, associated with a new kind of judgement $\vdash_{\leq}$. Coercions are naturally reflexive, and Table 2 summarises the other possible coercions.

Often the arguments of an operation have different types but are coercible to a common type, or one is coercible to the other. In order to capture this situation, we define the $\uparrow$ operator on types, which returns the least upper bound of the types to which its arguments are coercible:

$$\tau_1 \uparrow \tau_2 = \min\{\tau \mid \vdash_{\leq} \tau_1 \leq \tau \text{ and } \vdash_{\leq} \tau_2 \leq \tau\}$$

This requires that the coercion relation $\leq$ is regarded as a partial order on types, thus requiring the reflexivity, transitivity and anti-symmetry properties to hold. As we have seen before, the coercion relation is reflexive; the transitivity and anti-symmetry requirements are also easy to add and well suited to our intuitive notion of coercion. With these properties in place, and for the particular set of coercions allowed in CAO, we have that $\tau_1 \uparrow \tau_2$ is always uniquely defined. In typing rules, we therefore abbreviate the following pattern

$$\frac{\ldots \quad \Gamma, \Delta \vdash e :: \tau_1 \quad \vdash_{\leq} \tau_1 \leq \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots} \text{ by } \frac{\ldots \quad \Gamma, \Delta \vdash e \leq \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots}.$$

Casts. The CAO language includes a cast mechanism that allows for explicitly converting values from one type to another. However, not all casts are possible: the set of admissible type cast operations has been carefully designed to account for those conversions that are conceptually meaningful in the mathematical sense and/or are important for the implementation of cryptographic software in a natural way. We define a type cast relation $\Rightarrow$,

Table 3: A few cases for the cast relation, $\vdash_c t_1 \Rightarrow t_2$.

$\mathbf{t_1}$	$\mathbf{t_2}$	**Condition**
Int	Bits $[i]$	
Int	Mod $[n]$	
Vector $[i]$ of τ_1	Mod $[\tau_2/pol]$	$\vdash_c \tau_1 \Rightarrow \tau_2$ and $i = degree(pol)$
Mod $[\tau_1/pol]$	Vector $[i]$ of τ_2	$\vdash_c \tau_1 \Rightarrow \tau_2$ and $i = degree(pol)$
Matrix $[1, j]$ of α	Vector $[j]$ of τ	$\vdash_c \alpha \Rightarrow \tau$ and $\alpha \in \mathcal{A}$
Vector $[i]$ of τ	Matrix $[i, 1]$ of α	$\vdash_c \tau \Rightarrow \alpha$ and $\alpha \in \mathcal{A}$
Vector $[i]$ of τ_1	Vector $[i]$ of τ_2	$\vdash_c \tau_1 \Rightarrow \tau_2$
Matrix $[i, j]$ of α_1	Matrix $[i, j]$ of α_2	$\vdash_c \alpha_1 \Rightarrow \alpha_2$ and $\alpha_1, \alpha_2 \in \mathcal{A}$

which is associated with a new kind of judgment $\vdash_c$. Table 3 shows the part of the definition of the cast relation. Using the cast relation, we only have to provide one typing rule for cast expressions.

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2}{\vdash_c \tau_1 \Rightarrow \tau_2} \qquad \frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e :: (\tau', \mathsf{c}) \quad \vdash_c \tau' \Rightarrow \tau}{\Gamma, \Delta \vdash (t)\ e :: (\tau, \mathsf{c})}$$

The additional rule on the left is needed so that coercions can be made explicit, which also implies that a certain type can be cast to itself.

Sizes of bit strings, vectors and matrices. Since type declarations are mandatory and container types have explicit sizes, we can verify if operations deal consistently with these sizes. Furthermore, the type system can feed this information to subsequent components in the CAO tool chain.

For instance, the operation that concatenates two vectors should return a new vector whose size is the sum of the sizes of the individual vectors, and whose type is the least upper bound of the types of the two vectors, with respect to the coercion ordering $\leq$:

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Vector}[j] \text{ of } \tau_2, \mathsf{c}_2) \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma, \Delta \vdash e_1 \mathbin{@} e_2 :: (\mathsf{Vector}[i + j] \text{ of } \tau, \max(c_1, c_2))}$$

The concatenation of bit strings is similar. Moreover, in the case of matrix algebraic operations, e.g. multiplication, the dimension of the matrices can be checked for correctness.

When range selection is used over bit strings, vectors or matrices, we require that the integer expressions must be evaluated at compile-time so that the size of the expression, and therefore its type can be determined. In this case, the limits of the range are compared against the bounds of the associated type. For instance, for a range access to a vector we have:

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Vector}[k] \text{ of } \tau, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma, \Delta \vdash e[e_1..e_2] :: (\mathsf{Vector}[j - i + 1] \text{ of } \tau, \mathsf{c})} k > j, j \geq i \geq 0$$

This is also a limited form of dependent typing since the type associated with the expression depends on the expression itself.

Rings, Finite Fields and Extensions. One of the most unusual features of the CAO language is the support for ring and finite field types and their possible extensions. Our type checking rules allow us to ensure that operations over values of these types are well-defined and that values from different (instances of these) types are not being erroneously mixed due to programming errors. For instance, the typing rule for division is:

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Mod}\ [m_1], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Mod}\ [m_2], \mathsf{c}_2) \quad \mathsf{Mod}\ [m_1] \uparrow \mathsf{Mod}\ [m_2] = \mathsf{Mod}\ [m]}{\Gamma, \Delta \vdash e_1 \,/\, e_2 :: (\mathsf{Mod}\ [m], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

The use of the least upper bound captures the fact that the types may be equal, or one may be an extension of the other.

Variables and function calls. The classification of expressions depends on the environment accessed when retrieving the value of a variable. If a local variable is accessed, the code is considered pure; if a global variable is read, the code is classified as read-only.

$$\frac{\Gamma_G(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{ReadOnly})}\ x \in dom(\Gamma_G)$$

$$\frac{\Gamma_L(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{Pure})}\ x \in dom(\Gamma_L)$$

Since in expression, we can only use functions that do not cause side-effects, the typing rule for function application has a side condition to ensure that the body of the function is not a procedure (i.e., it does not modify a global variable):

$$\frac{\Gamma_G(f) = ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{c}) \quad \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)}{\Gamma_G, \Gamma_L, \Delta \vdash f(e_1, \ldots, e_n) :: (\tau, \max(\mathsf{c}, \mathsf{c}_1, \ldots, \mathsf{c}_n))}$$

$\max(\mathsf{c}, \mathsf{c}_1, \ldots, \mathsf{c}_n) < \mathsf{Procedure}$ and $f \in dom(\Gamma_G)$

Functions, procedures and statements. We introduce symbol $\bullet$ as a possible (empty) return type to detect misuses of the return statement. We distinguish the cases when a block has explicitly executed a return statement from the cases where no return statement has been executed. In the former case we take the type of the parameter passed to the return statement or $\bullet$ if no such parameter exists. In the latter case we also use the $\bullet$ symbol. Thus, a return statement is typed with the same type as its argument, which must coincide with the expected return type for the block.

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{cc}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_{(\tau_1, \ldots, \tau_n)} \mathsf{return}\ e_1, \ldots, e_n :: ((\tau_1, \ldots, \tau_n), \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma)}$$

Since CAO has a call-by-value semantics, returning multiple values is allowed in order to make references or additional structures unnecessary.

The typing rule for a function definition therefore verifies if the type of its body is not $\bullet$ to ensure that a return statement was used to exit the function. Moreover, the returned type has to be equal (or coercible) to the declared type (recall the use of judgement $\Vdash_\tau$).

The seq statement permits iterating over an integer variable varying between two statically determined bounds. The index starts with the value of the lower (resp. upper) bound and at each step is incremented (resp. decremented) by the amount of the step value until it reaches the upper (resp. lower) bound. The interesting feature of this mechanism is that the iterator is regarded as a constant at each iteration step. In the typing rules, this allows us to add the index and its respective value to the environment Δ at each iteration:

$$\frac{\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \forall_{n \in \{i \ldots j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$
$$\rho \in \{\tau, \bullet\},\ x \notin dom\ \Gamma_L,\ i \leq j$$

Therefore, declarations and access expressions inside the body of the sequence statement may depend on the index but may still be statically typeable. As highlighted in Section 3, the combination of range selection and assignment operators for bit strings, vectors and matrices with this simplified loop construction is a good example of how the CAO language design allowed us to fine tune the type checker to provide extra assistance to the programmer. Note, however, that sequential statements can make the type checking process slow, as sequences must be explicitly unfolded and typed for each possible value of the iterator.

A Detailed Example. We now present a detailed example of the how our type system handles the hsalsa20 fragment introduced in Section 3. The syntactic form of the program is

```
seq i := 0 to 3 {
    x[i+1]  :=  from_littleendian( k[i*4..i*4+3]);
    x[i+6]  :=  from_littleendian(in[i*4..i*4+3]);
    x[i+11] :=  from_littleendian( k[i*4+16..i*4+19]); }
```

where we desire type annotations for each node in the parse tree. The inference process traverses the tree matching rules against syntax. This traversal highlights aspects of the inference at three levels in the tree. Before reaching this fragment the declarations have already been produced and thus the initial environment is

$$\begin{aligned}
\Gamma_L &= \{\mathsf{k} :: \mathsf{Vec}[32]\ \mathsf{of}\ \mathsf{UBits}[8],\ \mathsf{in} :: \mathsf{Vec}[16]\ \mathsf{of}\ \mathsf{UBits}[8],\ \mathsf{x} :: \mathsf{Vec}[8]\ \mathsf{of}\ \mathsf{UBits}[32]\} \\
\Gamma_G &= \{\mathsf{to_littleendian} :: \mathsf{UBits}[32] \to \mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8], \\
&\qquad \mathsf{from_littleendian} :: \mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8] \to \mathsf{UBits}[32]\} \\
\Delta &= \{\}
\end{aligned}$$

The first step matches the entire fragment against $\mathsf{seq\ i} := 0\ \mathsf{to}\ 3\ \{s_1; s_2; s_3\}$

$$\frac{\forall_{n \in \{0 \ldots 3\}} \Gamma_G, \Gamma_L[\mathsf{i} :: \mathsf{Int}], \Delta[\mathsf{i} := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq\ i} := 0\ \mathsf{to}\ 3\ \{s_1; s_2; s_3\} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$

This entails, for each of the $n \in \{0, 1, 2, 3\}$ cases, that for assignments $(l_i{:=}r_i) = s_i$ in each of the s_1, s_2, s_3 preconditions, each statement is matched by

$$\frac{\Gamma_n, \Delta_n \vdash l_i :: (\tau, \mathsf{cl}) \quad \Gamma_n, \Delta_n \vdash r_i \leq (\tau, \mathsf{c})}{\Gamma_n, \Delta_n \Vdash_\tau l_i := r_i :: (\bullet, \max(\mathsf{cl}, \mathsf{c}), \Gamma)}$$

Here $\Gamma_n = \Gamma_G, \Gamma_L[\mathsf{i} :: \mathsf{Int}]$ and $\Delta_n = \Delta[\mathsf{i} := n]$. Now, for each of the l_i we obtain something of the form $\mathsf{x[i{+}1]}$ where $\Gamma_L(\mathsf{x}) = \mathsf{Vec}[8]\ \mathsf{of}\ \mathsf{UBits}[32]$ and an index expression $\mathsf{i{+}1} :: (\mathsf{Int}, \mathsf{Pure})$, thus we can match

$$\frac{\Gamma_n, \Delta_n \vdash \mathsf{x} :: (\mathsf{Vec}[8]\ \mathsf{of}\ \mathsf{UBits}[32], \mathsf{Pure}) \quad \Gamma_n, \Delta_n \vdash \mathsf{i} + 1 \leq (\mathsf{Int}, \mathsf{Pure})}{\Gamma_n, \Delta_n \vdash \mathsf{x[i} + 1\mathsf{]} :: (\mathsf{UBits}[32], \max(\mathsf{Pure}, \mathsf{Pure}))}$$

Finally, for each of the r_i the function parameter e_i is either $\Gamma_G[\mathsf{k}]$ or $\Gamma_G[\mathsf{in}]$ :: $\mathsf{Vec}[16]$ of $\mathsf{UBits}[8]$, Furthermore, the index expression is defined only over i, whose value is known, and integer literals. Thus each expression of the form $\mathsf{k}[\mathsf{i} * 4..\mathsf{i} * 4 + 3]$ becomes a slice over determined indices after application of ϕ_Δ and $\mathsf{k}[\mathsf{i} * 4..\mathsf{i} * 4 + 3] :: (\mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8], \mathsf{Pure})$. Hence

$$\frac{\begin{array}{c}\Gamma_G(\mathsf{from\_littleendian}) = (\mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8] \to \mathsf{UBits}[32], \mathsf{Pure}) \\ \Gamma_G, \Gamma_L, \Delta_1 \vdash \mathsf{k}[\mathsf{i} * 4..\mathsf{i} * 4 + 3] \leq (\mathsf{Vec}[4]\ \mathsf{of}\ \mathsf{UBits}[8], \mathsf{Pure})\end{array}}{\Gamma_G, \Gamma_L[\mathsf{i} :: \mathsf{Int}], \Delta_1 \vdash \mathsf{from\_littleendian}(\mathsf{k}[\mathsf{i} * 4..\mathsf{i} * 4 + 3]) :: (\mathsf{UBits}[32], \max(\mathsf{Pure}, \mathsf{Pure}))}$$

5 Implementation

The CAO type-checker was fully implemented in the Haskell functional language, which provides a plethora of libraries and built-in language features. Among these, we found some to be particularly useful, such as *classes*, specific syntax for handling monadic data types and the *monad Error* data type. These Haskell assets, not only simplified the implementation process, but also helped improving substantially the readability of the code and its comparison with the formal specification of the type checking rules described in the previous section.

To generally illustrate Haskell's ability to deal with the formal type checking rules that we specified in the previous section, consider the following code snippet, which implements the rule for type checking CAO while statements.

```
tcStatement s@(WhileStatement info cond wstms) h rt =
  do (cond', condt, cb) <- tcExp cond h
   checkMatchType info condt Boolean
   (wstms', wst, cc, h') <- tcStatements wstms h rt
   return (mkWhileStatement (buildTcNodeInfo info Bullet)
                            cond' wstms', Bullet, max cb cc ,h)
```

The interpretation of the above code is quite immediate. Function tcStatement is our formal statement type checking function $\Vdash$, rt represents the expected return type, which in the formal definition subscripts $\Vdash$ and h corresponds to the type environments Γ and Δ. Note that, even though we have made clear the distinction between Γ and Δ in the formal rules, this was mainly justified by presentational reasons. Still on the arguments side, one finds (WhileStatement info cond wstms), trivially matching while b $\{c\}$, except for the info identifier, which is an add-on of the implementation for storing the exact place where the CAO code being analysed appears in the input file.

Regarding the function body, in accordance to the formal rule, which relies on premises referring to $\vdash$ and $\Vdash$, so does the implementation, referring to functions tcExp and tcStatements respectively. Here, however, one resorts to Haskell's monadic operator <- over the monad Error data type. In this way we combine calls to different type checking functions that may return type checking errors, ensuring that if an error occurs in one of the calls, the error is propagated down to the end of the type checker execution, without interfering with any other type checking rule in between.

Function checkMatchType corresponds to our order comparison operator $\leq$ over data types, while Bullet is our functional representation of symbol $\bullet$. Function max ensures that type classifiers, which allow the type system to recognise various types of functions,

are properly propagated. Instead of returning the type of the expression being evaluated, the implementation returns the expression received annotated with its type, to be used by subsequent compilation steps. Nevertheless, the above rule implementation illustrates how we have kept the implementation reasonably close to the formal definition, therefore favoring a direct validation by inspection of the implementation.

6 Soundness of the Type System

As usual, the CAO type system aims to ensure that *"well-typed programs do not go wrong"* [6]. This is formalised as a soundness theorem relating static (typing) and dynamic semantics. For the moment, our result only ensures that the evaluation of well-typed program does not fall into a certain class of errors: formally, we are proving a *weak soundness theorem*. Concretely, we have shown that only a well-defined set of run-time errors (trapped errors, denoted by ϵ in the semantic domain $\mathbf{V}$) can occur when evaluating a correctly typed program. These are explicitly captured in the semantics of the language, and they are limited to divisions by zero and out of bounds accesses to containers. In this Section, we first shortly present some aspects of our formalization of the CAO semantics necessary to provide support to the subsequent discussion of our soundness theorem and proof sketch. The complete description of both can be found in Appendices B and C, respectively.

CAO Semantics Evaluation of a CAO program is defined by an *evaluation relation* that relates an initial configuration (a CAO program together with a description of the initial state) with a final configuration (a semantic value and a final state). The domain of *semantic values* is defined as a solution of the domain equation $\mathbf{V} = \mathbb{Z} + \mathbf{V}^\star + \mathcal{E}$, where $\mathbb{Z}$ denotes the domain of integers, $\mathbf{V}^\star$ denotes sequences of values of type $\mathbf{V}$ of the form $[v_0, \ldots, v_{n-1}]$ and $\mathcal{E}$ is the type of the trapped error value ϵ. A *trapped error* is an execution error that results in an immediate fault (run-time error); an *untrapped error* is an execution error that does not immediately result in a fault, corresponding to an unexpected behavior. We denote such an error by $\perp$.

We define three mutually recursive evaluation relations, each of them responsible for characterising the evaluation of different syntactic classes: *expressions*, *statements* and *declarations*:

- $\langle\ e \mid \rho\ \rangle \rightarrow r$ evaluates expression e in state ρ to the value r. Expression evaluation is side-effect free, and hence the state is not changed.
- $\langle\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ means that the evaluation of statement c in state ρ transforms the state into ρ', and (possibly) produces result r.
- $\langle\ d \mid \rho\ \rangle \Rrightarrow \langle\ \rho'\ \rangle$ means that the evaluation of declaration d in state ρ transforms the state into ρ'.

CAO has a *call by value* semantics, where there are no references and each variable identifier denotes a value. Assignments mean that old values are replaced by the new ones in the state. Since expressions are effect-free, simultaneous value assignments are possible (however, here we will stick to the simpler single-assignment version of the evaluation rule). In CAO, a

run-time trapped error can occur only in three cases: 1) accessing a vector, matrix or bit string out of the bounds; 2) division (or remainder of division) by zero; and 3) assigning a value to a vector, matrix or bit string out of bounds. We present example rules for the latter two cases below, noting that the frame update operator is defined to return ϵ when l identifies an update to an invalid index in a container representation.

$$\textsc{Assign-Err}\ \frac{\langle\ e \mid \rho\ \rangle \to v}{\langle\ l\ \texttt{:=}\ e \mid \rho\ \rangle \Rightarrow \langle\ \epsilon\ ,\ \_\ \rangle} \quad \rho[v/l] = \epsilon$$

$$\textsc{Assign}\ \frac{\langle\ e \mid \rho\ \rangle \to v}{\langle\ l\ \texttt{:=}\ e \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho[r/l]\ \rangle} \quad \rho[v/l] \neq \epsilon$$

$$\textsc{Div}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to v_1 \quad \langle\ e_2 \mid \rho\ \rangle \to v_2}{\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \to [\![/]\!][v_1, v_2]}$$

$$\textsc{Div-Zero}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to v_1 \quad \langle\ e_2 \mid \rho\ \rangle \to 0}{\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \to \epsilon}$$

where function at returns the n-th element of a sequence. Range accesses actually cannot cause trapped errors, as the type system enforces that the limits must be statically defined in order to determine the size of the result, which means that such errors can be detected. Trapped errors are propagated throughout evaluation rules, i.e., whenever a premiss evaluates to ϵ the overall rule also evaluates to ϵ. All cases that fall outside of our semantic rules are implicitly evaluated to untrapped errors ($\perp$ value).

Soundness theorem and proof sketch Our result is stated in the following theorem, where $\vdash \rho :: \Gamma_G$ denotes consistency and $\circ$ denotes empty store/state.

Theorem 1. *Given a program p if $\circ, \circ, \circ \vdash p :: (\bullet, \Gamma_G)$ and $\langle\ p \mid \circ\ \rangle \Rrightarrow \langle\ \rho\ \rangle$ terminates, then $\vdash \rho :: \Gamma_G$ or ρ is an error state.*

Proof (Sketch). The full proof is presented in Appendix C. The proof is by induction on typing derivations. The base case for induction is that prior to execution, every type-checked program has an initial evaluation environment that is (trivially) consistent with the typing environment. Here, consistency means that all variables in the evaluation environment have associated values compatible with their corresponding type in the typing environment. The inductive cases are considered for each transition defined in the semantics of the language. In each case we show that one of two cases can occur: 1) either a consistent environment is produced at the end of each transition; or 2) a trapped error has been generated and is returned by the program. We present two cases, illustrating how the proof proceeds for division expressions and assignment statements that may raise trapped errors.

Division Expressions. We have to prove that if $\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}$. Two semantic rules can be applied for each operator, one in the case of division by 0; the other in the general case:

– If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to 0$ terminate, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $\epsilon \in \mathbf{V}$ by semantic Div-Zero.

– If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to v_2$ terminate, with $v_2 \neq 0$, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $[\![/]\!][v_1, v_2]$ by semantic rule Div. Here $[\![/]\!]$ gives the interpretation of the / operator with respect to the values v_1 and v_2. By induction hypothesis, v_1 and v_2 are in the semantic domain $\mathbf{V}$, corresponding to representations of integer values. Since division is well-defined for integer representations, then $[\![/]\!][v_1, v_2]$ evaluates to another value v which is again a representation of an integer and $v \in \mathbf{V} \backslash \mathcal{E}$.

Assignment Statements. We have to prove that if $\langle\ l := e \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then, either the statement raises a trapped error due to an invalid access on the left value, or the returned environment ρ' is consistent with the typing environment. Two semantic rules are applicable, Assign and Assign-Err, the latter only when the target is an invalid position in a container. If $\langle\ e \mid \rho\ \rangle \to v$ terminates, then $v \in \mathbf{V} \backslash \mathcal{E}$ and v represents a value of type τ. The semantic rule to apply depends on the result of the frame update operation $\rho[v/l]$. If this returns ϵ, then semantic rule Assign-Err is applied, and the statement evaluates to $\langle\ \epsilon\ ,\ \_\ \rangle$. Otherwise it will return an updated state ρ', in which case semantic rule Assign is applied, and the statement evaluates to $\langle\ \bullet\ ,\ \rho[v/l]\ \rangle$. It remains to prove that this resulting evaluation environment is consistent with the typing environment. Here we resort to the induction hypothesis $\vdash \rho :: \Gamma$, which guarantees the value currently stored for l represents a value of type τ. Since v also represents a value of type τ, the update of left value l for value v preserves consistency.

7 Related Work

Cryptol [4] is a domain-specific language and tool suite developed for the specification and implementation of cryptographic algorithms. It is a functional DSL without global state or side-effects, which was developed with the main purpose of producing formally verified hardware implementations of symmetric cryptographic primitives such as block ciphers and hash functions. CAO is an imperative language that targets a wider application domain, although also restricted to cryptography. Indeed, the CAO language features have been designed to permit expressing, not only symmetric but also asymmetric cryptographic primitives, in a natural way. Furthermore, CAO tools are released under an open-source policy.

Dependent types offer a powerful approach to ensure program properties. However, this power in not incorporated in any of the mainstream languages, while the prototypical languages that do it are mostly functional. The first prototype of an imperative language to use dependent types was Xanadu [8], allowing, e.g., to statically verify that accesses to arrays are within bounds. So far, CAO offers a modest form of dependent types, where all type parameters values must be statically known. Ongoing work aims extend CAO with a more powerful approach to dependent types inspired by [8]. This new version of the type system allows for symbolic parametrisation, dropping the requirement that all sizes are known at compilation, using an SMT solver to handle associated constraints.

The use of Generalized Algebraic Data Types (GADTs) in Haskell, together with type families and existential types, allows the implementation of embedded DSL's with some

dependent typing features. Moreover, since this approach relies on Haskell's type system, this permits avoiding the full implementation of a type checker. CAO does not follow this embedded approach because it would make it harder to preserve characteristics of the language that pre-dated formal work on the type system. For example, the CAO syntax tries to follow the cryptographic specification standards, and GADTs would impose their own syntax, which is more suitable for building combinator systems. One could of course try to use a GADT-based intermediate representation, but it is not clear that this would pay out in terms of the global implementation effort. In particular, we anticipate that dealing with coercions and casts would complicate the type checking apparatus [7]. Moreover, it would probably be difficult using an embedded approach to keep the implementation structure close to the formal specification.

The use of an embedded implementation in a dependently typed language, e.g. Coq or Agda, could also be an option for the implementation of our type system. However, this would suffer from the same drawbacks previously presented for GADTs, and would also require specific expertise that are not realistic to assume in the target audience for CAO. The need to reason about the correctness and termination of CAO programs at this level would also be an overkill for most applications. In the CAO tool-chain, this sort of analysis is enabled by an independent deductive formal verification tool called CAOVerif.

8 Conclusion

CAO is a language aimed at closing the gap between the usual way of specifying cryptographic algorithms and their actual implementation, reducing the possibility of errors and increasing the understanding of the source code. This language offers high-level features and a type system tailored to the implementation of cryptographic concepts, statically ruling out some important classes of errors. In this paper, we have presented a short overview of CAO and the specification, validation and implementation of a type-system designed to support the implementation of front-ends for CAO compilation and formal verification tools.

References

1. D. J. Bernstein. The Poly1305-AES message-authentication code. In H. Gilbert and H. Handschuh, editors, *FSE*, volume 3557 of *LNCS*. Springer, 2005.
2. D. J. Bernstein. Cryptography in NaCl, 2009. `http://nacl.cr.yp.to`.
3. J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specification Version 2.1, 2003.
4. J. Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *FMSE '07*, page 41. ACM, 2007.
5. A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.
6. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.
7. P. F. Silva and J. N. Oliveira. 'Galculator': functional prototype of a Galois-connection based proof assistant. In *PPDP '08*, pages 44–55. ACM, 2008.
8. H. Xi. Imperative programming with dependent types. In *LICS*, pages 375–387, 2000.

A Type system

Environments and type judgements. We follow a standard presentation style for the type inference rules. We use Γ for the type environment relation, which collects all the declarations so far (variables, functions, procedures, etc.), together with their associated type. The Γ environment is partitioned in two parts: Γ_G for global definitions and Γ_L for local definitions. Whenever this distinction is not important we will just write Γ to abbreviate Γ_G, Γ_L. Another environment Δ records the values associated to integer constants, which will be used in the evaluation of integer expressions. Notation $\Gamma[x :: \tau]$ is used to extend the environment Γ with a new variable x of type τ, providing that x is not in the original environment (i.e., $x \notin dom(\Gamma)$). Similarly, $\Delta[x := n]$ is used to extend the environment Δ with a new constant x with value n, also provided that x is not in the domain of environment Δ. Notation $\Gamma(x)$ and $\Delta(x)$ represent, respectively, the type and the integer value associated with identifier x, assuming that x belongs to the domain of the respective environment. We use the symbol $\vdash$ for the type judgement of expressions, retrieving an expression and its associated type. We use $x :: \tau$ to denote that x has type τ, rather than the more usual notation $x : \tau$, in order to avoid confusion with the CAO syntax notation.

Types. Table 4 shows the CAO types that we use in our type checking rules. These co-

Bool	Booleans
Int	Arbitrary precision integers
UBits $[i]$	Unsigned bit strings of length i
SBits $[i]$	Signed bit strings of length i
Mod $[n]$	Rings or fields defined by integer n
Mod $[t/pol]$	Extension fields defined by t/pol
Vector $[i]$ of τ	Vector of i elements of type τ
Matrix $[i,j]$ of α	Matrix of $i \times j$ elements of type $\alpha \in \mathcal{A}$

$\mathcal{A} = \{\mathsf{Int}, \mathsf{Mod}\ [n], \mathsf{Mod}\ [t/pol], \mathsf{Matrix}\ [i,j]\ \mathsf{of}\ \alpha\ \mid \alpha \in \mathcal{A}\}$

Table 4: CAO types for type checking rules

incide with the CAO types adopted in the syntax description in Figure 1. $\mathcal{A}$ denotes the set of algebraic types, which are the only ones that can be used to construct matrices, i.e., $\mathcal{A} = \{\mathsf{Int}, \mathsf{Mod}[n], \mathsf{Mod}\ [t/pol], \mathsf{Matrix}[i,j]\ \mathsf{of}\ \alpha\ \mid \alpha \in \mathcal{A}\}$. These are types for which addition, multiplication and symmetric operators are closed. In order to emphasize the situations when the type must be algebraic, we will use α instead of τ. Conversely, vectors are generic containers that can be constructed from all types. The type parameter m in modular types may represent either an integer, or the more complex type construction information required for field extensions (t/pol). To simplify the presentation, we only distinguish between signed and unsigned bits types when needed. The only difference between these types resides in the conversion operations to the integer type, as will be clarified

when we present type checking rules for cast operators below. Thus, we will usually use only $\mathsf{Bits}[i]$ to refer to both kind of bit strings.

TYPE EQUALITY. Equality between types is defined in the expected way.

$$\frac{}{\vdash_{eq} \mathsf{Int} = \mathsf{Int}}$$

$$\frac{}{\vdash_{eq} \mathsf{Bool} = \mathsf{Bool}}$$

$$\frac{}{\vdash_{eq} \mathsf{UBits}[i] = \mathsf{UBits}[i]}$$

$$\frac{}{\vdash_{eq} \mathsf{SBits}[i] = \mathsf{SBits}[i]}$$

$$\frac{}{\vdash_{eq} \mathsf{Mod}[n] = \mathsf{Mod}[n]}$$

$$\frac{\vdash_{eq} \tau_1 = \tau_2}{\vdash_{eq} \mathsf{Mod}[\tau_1/pol_1] = \mathsf{Mod}[\tau_2/pol_2]} \quad pol_1 = pol_2$$

$$\frac{\vdash_{eq} \tau_1 = \tau_2}{\vdash_{eq} \mathsf{Vector}[i] \text{ of } \tau_1 = \mathsf{Vector}[i] \text{ of } \tau_2}$$

$$\frac{\vdash_{eq} \alpha_1 = \alpha_2}{\vdash_{eq} \mathsf{Matrix}[i,j] \text{ of } \alpha_1 = \mathsf{Matrix}\ [i,j] \text{ of } \alpha_2} \quad \alpha_1, \alpha_2 \in \mathcal{A}$$

Type equality is used implicitly in rules whenever the same type variable appears in two type judgements. For instance, the following rule

$$\frac{\ldots \quad \Gamma, \Delta \vdash e_1 :: \tau \quad \Gamma, \Delta \vdash e_2 :: \tau \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

implicitly defines an equality relation between the type of expressions e_1 and e_2, i.e.,

$$\frac{\ldots \quad \Gamma, \Delta \vdash e_1 :: \tau_1 \quad \Gamma, \Delta \vdash e_2 :: \tau_2 \quad \vdash_{eq} \tau_1 = \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

TYPE COERCIONS. Coercions between types occur when values of some type can be used as values of another type without requiring an explicit cast, while ensuring that no information is lost in the process. We define a type coercion relation $\leq$ between coercible types. This

relation is reflexive, transitive and antisymmetric, thus being a partial order on types:

$$\frac{\vdash_{eq} \tau_1 = \tau_2}{\vdash_{\leq} \tau_1 \leq \tau_2}$$

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2 \quad \vdash_{\leq} \tau_2 \leq \tau_3}{\vdash_{\leq} \tau_1 \leq \tau_3}$$

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2 \quad \vdash_{\leq} \tau_2 \leq \tau_1}{\vdash_{eq} \tau_1 = \tau_2}$$

The following coercions are allowed in CAO:

$$\frac{}{\vdash_{\leq} \mathsf{UBits}[n] \leq \mathsf{Int}}$$

$$\frac{}{\vdash_{\leq} \mathsf{SBits}[n] \leq \mathsf{Int}}$$

$$\frac{\vdash_{\leq} \tau \leq \tau'}{\vdash_{\leq} \tau \leq \mathsf{Mod}[\tau'/pol]}$$

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2}{\vdash_{\leq} \mathsf{Vector}[n] \text{ of } \tau_1 \leq \mathsf{Vector}[n] \text{ of } \tau_2}$$

$$\frac{\vdash_{\leq} \alpha_1 \leq \alpha_2}{\vdash_{\leq} \mathsf{Matrix}\ [i,j] \text{ of } \alpha_1 \leq \mathsf{Matrix}\ [i,j] \text{ of } \alpha_2} \quad \alpha_1, \alpha_2 \in \mathcal{A}$$

Supremum coercion. In some cases, ambiguities can occur in rules if the type coercion relation is not used with care. For instance, let us consider the following typing rule for an operator $\oplus$:

$$\frac{\Gamma, \Delta \vdash e_1 :: \tau_1 \quad \Gamma, \Delta \vdash e_2 :: \tau_2 \quad \vdash_{\leq} \tau_1 \leq \tau \quad \vdash_{\leq} \tau_2 \leq \tau}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: \tau}$$

If τ_1 and τ_2 are types such that $\tau_1 \leq \tau_2$ holds, and there is a rule that states that $\tau_2 \leq \tau_3$, the above rule would allow $e_1 \oplus e_2$ to have two different types, i.e., τ_2 and τ_3, since by transitivity $\tau_1 \leq \tau_3$. Since we want expressions to have a single principal type, we introduce a functional operator that, given two types, returns their supremum with respect to the coercion order.

Introducing the supremum type operator $\uparrow$, the above rule becomes:

$$\frac{\Gamma, \Delta \vdash e_1 :: \tau_1 \quad \Gamma, \Delta \vdash e_2 :: \tau_2 \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: \tau}$$

For types τ_1 and τ_2, we define $\uparrow$ as their least upper bound with respect to the type coercion order $\leq$:

$$\tau_1 \uparrow \tau_2 = \min\{\tau \mid \vdash_{\leq} \tau_1 \leq \tau \text{ and } \vdash_{\leq} \tau_2 \leq \tau\}$$

Evaluation of integer expressions. The CAO language allows declaring the sizes of bit strings, vectors and matrices as the result of the evaluation of a integer expression. However, only a limited number of expressions is allowed, namely, basic arithmetic expressions with integer literals, and constants defined in the environment Δ. Thus, we define a partial function ϕ_Δ that given an expression e and the environment Δ computes the respective integer value:

$$\begin{aligned}
\phi_\Delta(x) &= \Delta(x) \quad x \in dom(\Delta) \\
\phi_\Delta(-e) &= -\phi_\Delta(e) \\
\phi_\Delta(e_1 + e_2) &= \phi_\Delta(e_1) + \phi_\Delta(e_2) \\
\phi_\Delta(e_1 - e_2) &= \phi_\Delta(e_1) - \phi_\Delta(e_2) \\
\phi_\Delta(e_1 * e_2) &= \phi_\Delta(e_1) \times \phi_\Delta(e_2) \\
\phi_\Delta(e_1 \,/\, e_2) &= \phi_\Delta(e_1) \,/\, \phi_\Delta(e_2) \\
\phi_\Delta(e_1 \,\%\, e_2) &= \phi_\Delta(e_1) \mod \phi_\Delta(e_2) \\
\phi_\Delta(e_1 ** e_2) &= (\phi_\Delta(e_1))^{(\phi_\Delta(e_2))} \\
\phi_\Delta(n) &= n
\end{aligned}$$

The function ϕ_Δ fails for all other operations.

Function Classification. The type checker is able to automatically classify CAO functions with respect to their interaction with global variables. The type checking rules classify functions as either

- Pure functions – Do not depend on global variables in any way. In particular, they can only call other pure functions. These functions are, not only side-effect free, but also guaranteed to return the same result in every invocation with the same input.
- Read-only functions – Can read values from global variables, but they cannot assign values to them. They can call pure functions and other read-only functions, but not procedures. These functions are side-effect free.
- Procedures – Can read and assign values from/to global variables. They can call pure functions, read-only functions and other procedures.

For the CAO type checker, the most important distinction is that between procedures and other functions. This is because procedures are only admitted in restricted contexts, such as simple assignment constructions. This distinction was a syntactic one in the first version of the CAO formal specification, but is now completely automated in the type-checking rules. Introducing this change into the type-checking rules implied creating a hierarchy (chain order) of classifiers

$$\mathsf{Pure} < \mathsf{ReadOnly} < \mathsf{Procedure}$$

that are associated to CAO constructions by the type-checking rules. Put simply, the type checking system enforces the following rules:

- When type-checking a construction depending on local variables, the classifier Pure is assigned.
- When reading the value of a global variable, the classifier is set to Read-only.
- When a global variable is used in a left-value, the classifier is set to Procedure.
- Expressions and statements procedures are classified with respect to their sub-elements using the *maximum operator* defined over the chain order specified above.

Note that this classification system is conservative in the sense that, for example, it will fail to correctly classify a function as pure when it reads a global variable but does not use its value in any computation.

NOTATION. In our rules, the following pattern is very frequent:

$$\frac{\ldots \quad \Gamma, \Delta \vdash e :: \tau_1 \quad \vdash_{\leq} \tau_1 \leq \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

Therefore, we will use syntactic sugar for this:

$$\frac{\ldots \quad \Gamma, \Delta \vdash e \leq \tau_2 \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

When evaluating integer expressions, we will just write:

$$\frac{\ldots \quad \phi_{\Delta}(e) = n \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

which implicitly implies that expression e is of integer type with a *pure classifier*, i.e.,

$$\frac{\ldots \quad \Gamma, \Delta \vdash e :: (\mathsf{Int}, \mathsf{Pure}) \quad \phi_{\Delta}(e) = n \quad \ldots}{\Gamma, \Delta \vdash \ldots}$$

TYPE DECLARATIONS. In CAO types we cannot say that int :: Int because int is not a value of the integer type. However, type declarations can be mixed with expressions whenever a cast is used, and we have to cater for this in our formalism. We could abuse notation and write (Int) e to denote a cast of an expression e to an integer value instead of (int) e, but this would introduce some overhead, specially when dealing with types which depend on integer expressions. Thus, we introduce a new type judgement $\vdash_t$ and a functional symbol $\rightsquigarrow$ that allows us to write $\Delta \vdash_t t \rightsquigarrow \tau$ where t is a (syntactic) type declaration in CAO and τ is the corresponding (theoretical) data type. We use the Δ environment because only constants are needed to evaluate integer expressions. Predicate *prime* tests if a given

number is prime and predicate *irreducible* tests whether a given polynomial is irreducible.

$$\frac{}{\Delta \vdash_t \mathsf{int} \rightsquigarrow \mathsf{Int}}$$

$$\frac{}{\Delta \vdash_t \mathsf{bool} \rightsquigarrow \mathsf{Bool}}$$

$$\frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{unsigned\ bits}\ [e] \rightsquigarrow \mathsf{UBits}[n]} \quad n \geq 1$$

$$\frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{signed\ bits}\ [e] \rightsquigarrow \mathsf{SBits}[n]} \quad n \geq 1$$

$$\frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{mod}\ [e] \rightsquigarrow \mathsf{Mod}[n]} \quad n \geq 2$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Mod}\ [n]}{\Delta \vdash_t \mathsf{mod}\ [t\ /\ pol] \rightsquigarrow \mathsf{Mod}\ [\mathsf{Mod}\ [n]\ /\ pol]} \quad \mathit{prime}(n) \text{ and } \mathit{irreducible}(pol)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Mod}\ [\tau\ /\ pol']}{\Delta \vdash_t \mathsf{mod}\ [t\ /\ pol] \rightsquigarrow \mathsf{Mod}\ [\mathsf{Mod}\ [\tau\ /\ pol']\ /\ pol]} \quad \mathit{irreducible}(pol)$$

$$\frac{\phi_\Delta(e) = n \quad \Delta \vdash_t t \rightsquigarrow \tau}{\Delta \vdash_t \mathsf{vector}\ [e]\ \mathsf{of}\ t \rightsquigarrow \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau} \quad n \geq 1$$

$$\frac{\phi_\Delta(e_1) = n \quad \phi_\Delta(e_2) = m \quad \Delta \vdash_t t \rightsquigarrow \alpha}{\Delta \vdash_t \mathsf{matrix}\ [e_1, e_2]\ \mathsf{of}\ t \rightsquigarrow \mathsf{Matrix}\ [n, m]\ \mathsf{of}\ \alpha} \quad \alpha \in \mathcal{A}, n \geq 1, m \geq 1$$

A.1 Expressions

Literals. For conciseness, we do not provide an exhaustive definition of all the type checking rules for literals. This is consistent with the approach we followed in presenting the language syntax, and it is justified because simple and well-known syntax and type checking rules can be used for these cases. However, we provide the following rules for boolean values and bit strings as a flavor of the set of type checking rules for literals.

$$\frac{}{\Gamma, \Delta \vdash \mathsf{true} :: (\mathsf{Bool}, \mathsf{Pure})}$$

$$\frac{}{\Gamma, \Delta \vdash \mathsf{false} :: (\mathsf{Bool}, \mathsf{Pure})}$$

$$\frac{}{\Gamma, \Delta \vdash \mathsf{0b}(\mathsf{0}|\mathsf{1})^i :: (\mathsf{Bits}[i], \mathsf{Pure})}$$

Variables, function calls and struct projections. In general expressions, one can only use functions that cannot cause any side-effects (for the particular case of procedures

see the statement typing rules below). When type checking struct projections, we resort to the previously stored type information about the particular field selector *fi* (see the program type checking rules below).

$$\frac{\Gamma_G(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{ReadOnly})} \quad x \in dom(\Gamma_G)$$

$$\frac{\Gamma_L(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{Pure})} \quad x \in dom(\Gamma_L)$$

$$\frac{\begin{array}{c}\Gamma_G(f) = ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{c}) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c_1}) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c_n})\end{array}}{\Gamma_G, \Gamma_L, \Delta \vdash f(e_1, \ldots, e_n) :: (\tau, \max(\mathsf{c}, \mathsf{c}_1, \ldots, \mathsf{c}_n))} \quad \mathsf{c} < \mathsf{Procedure}, f \in dom(\Gamma_G)$$

$$\frac{\Gamma_G(fi) = (\tau_1 \to \tau_2, \mathsf{Pure}) \quad \Gamma_G, \Gamma_L, \Delta \vdash e :: (\tau_1, \mathsf{c})}{\Gamma_G, \Gamma_L, \Delta \vdash e.fi :: (\tau_2, \mathsf{c})} \quad fi \in dom(\Gamma_G)$$

Arithmetic operations. Arithmetic operators are overloaded, and their semantics varies for the different types according to the mathematical notions that they capture. For modular types there is a subtlety in that division may not be defined if parameter n does not construct a field, e.g. if n is a composite integer. This, however, is not addressed by the type checking system.

$$\frac{\Gamma, \Delta \vdash e_1 :: (\alpha_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\alpha_2, \mathsf{c}_2) \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha, \alpha_1, \alpha_2 \in \mathcal{A} \quad \oplus \in \{+, -\}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\alpha_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\alpha_2, \mathsf{c}_2) \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma, \Delta \vdash e_1 * e_2 :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha, \alpha_1, \alpha_2 \in \mathcal{A}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Matrix}[i, j] \text{ of } \alpha_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Matrix}[j, k] \text{ of } \alpha_2, \mathsf{c}_2) \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma, \Delta \vdash e_1 * e_2 :: (\mathsf{Matrix}[i, k] \text{ of } \alpha, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

where $\alpha, \alpha_1, \alpha_2 \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\tau_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\tau_2, \mathsf{c}_2) \quad \vdash_{\leq} \tau_1 \leq \mathsf{Int} \quad \vdash_{\leq} \tau_2 \leq \mathsf{Int}}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{+, -\}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\tau_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\tau_2, \mathsf{c}_2) \quad \vdash_{\leq} \tau_1 \leq \mathsf{Int} \quad \vdash_{\leq} \tau_2 \leq \mathsf{Int}}{\Gamma, \Delta \vdash e_1 * e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\alpha, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 ** e_2 :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha \in \mathcal{A}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\tau, \mathsf{c}_1) \quad \vdash_{\leq} \tau \leq \mathsf{Int} \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 ** e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \tau \notin \mathcal{A}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Matrix}[i, i] \text{ of } \alpha, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 ** e_2 :: (\mathsf{Matrix}[i, i] \text{ of } \alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha \in \mathcal{A}$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \,/\, e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Mod}\ [m_1], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Mod}\ [m_2], \mathsf{c}_2) \quad \mathsf{Mod}\ [m_1] \uparrow \mathsf{Mod}\ [m_2] = \mathsf{Mod}\ [m]}{\Gamma, \Delta \vdash e_1 \,/\, e_2 :: (\mathsf{Mod}\ [m], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

where m_1, m_2 can be of the form n or t/pol

$$\frac{\Gamma, \Delta \vdash e :: (\alpha, \mathsf{c})}{\Gamma, \Delta \vdash -e :: (\alpha, \mathsf{c})} \quad \alpha \in \mathcal{A}$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \,\%\, e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

Boolean operations. Operations involving boolean values are standard.

$$\frac{\Gamma, \Delta \vdash e_1 :: (\tau_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\tau_2, \mathsf{c}_2) \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bool}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{==, !=\}$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bool}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{<, \leq, >, \geq\}$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\mathsf{Bool}, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Bool}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bool}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{||, \&\&, \hat{}\hat{}\}$$

$$\frac{\Gamma, \Delta \vdash e \leq (\mathsf{Bool}, \mathsf{c})}{\Gamma, \Delta \vdash !e :: (\mathsf{Bool}, \mathsf{c})}$$

Bit string operations. All bit string operators are closed over the same representation, i.e. one cannot mix signed and unsigned bit strings unless through an explicit cast[5]. The bit-wise operations (negation, and, or, exclusive or and shift) have identical semantics to those of the C language, and they are only defined for strings of the same size. The additional concatenation operator works in the obvious way by constructing a bit string using list concatenation. Selection and range selection over bit strings both return a bit string of the same type and appropriate size (size 1 for individual selection).

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Bits}[i], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Bits}[i], \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bits}[i], \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{|, \&, \hat{}\}$$

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Bits}[i], \mathsf{c})}{\Gamma, \Delta \vdash \sim e :: (\mathsf{Bits}[i], \mathsf{c})}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Bits}[i], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma \vdash e_1 \oplus e_2 :: (\mathsf{Bits}[i], \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{\ll, \gg, <|, |>\}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Bits}[i], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Bits}[j], \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \mathbin{@} e_2 :: (\mathsf{Bits}[i + j], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

[5] Such a cast does not alter the bit string itself but only the effect of conversion to an integer value, see the section on casts below.

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Bits}[i], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1[e_2] :: (\mathsf{Bits}[1], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Bits}[k], \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma, \Delta \vdash e[e_1..e_2] :: (\mathsf{Bits}[j - i + 1], \mathsf{c})} \quad k > j, j \geq i \geq 0$$

Vector operations. Vectors are the generic container type. They allow for a mixed set of operations. Similarly to bit strings, one can perform shifts and concatenations (note that for all types CAO defines a default zero value that can be used in shift operators). Range selection is also defined in a natural way: it returns a vector of the same type and appropriate size. Element selection operations over vectors may also return vector and matrix values, i.e. we can have vectors of vectors and vectors of matrices.

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Vector}[i] \text{ of } \tau, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{\ll, \gg, <|, |>\}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau_1, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 :: (\mathsf{Vector}[j] \text{ of } \tau_2, \mathsf{c}_2) \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma, \Delta \vdash e_1 \mathbin{@} e_2 :: (\mathsf{Vector}[i + j] \text{ of } \tau, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash e_1[e_2] :: (\tau, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Vector}[k] \text{ of } \tau, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma, \Delta \vdash e[e_1..e_2] :: (\mathsf{Vector}[j - i + 1] \text{ of } \tau, \mathsf{c})} \quad k > j, j \geq i \geq 0$$

Matrix operations. Matrix expressions include the algebraic operators for matrix addition, subtraction, multiplication and exponentiation, as seen above. This is why the contents of matrices can only include algebraic types. Element selection operations over matrices may also return matrix values, i.e. we can have matrices of matrices. Range selection over a matrix returns a matrix of appropriate dimensions and type.

$$\frac{\Gamma, \Delta \vdash e_1 :: (\mathsf{Matrix}[i, j] \text{ of } \alpha, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2) \quad \Gamma, \Delta \vdash e_3 \leq (\mathsf{Int}, \mathsf{c}_3)}{\Gamma, \Delta \vdash e_1[e_2, e_3] :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3))}$$

where $\alpha \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Matrix}[u, v] \text{ of } \alpha, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \quad \phi_\Delta(e_4) = n}{\Gamma, \Delta \vdash e[e_1..e_2, e_3..e_4] :: (\mathsf{Matrix}[j - i + 1, n - k + 1] \text{ of } \alpha, \mathsf{c})}$$

where $u > j, j \geq i \geq 0, v > n, n \geq k \geq 0, \alpha \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Matrix}[u, v] \text{ of } \alpha, \mathsf{c}) \quad \Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \phi_\Delta(e_2) = k \quad \phi_\Delta(e_3) = n}{\Gamma, \Delta \vdash e[e_1, e_2..e_3] :: (\mathsf{Matrix}[1, n - k + 1] \text{ of } \alpha, \max(\mathsf{c}, \mathsf{c}_1))}$$

where $v > n, n \geq k \geq 0, \alpha \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash e :: (\mathsf{Matrix}[u, v] \text{ of } \alpha, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \Gamma, \Delta \vdash e_3 \leq (\mathsf{Int}, \mathsf{c}_3)}{\Gamma, \Delta \vdash e[e_1..e_2, e_3] :: (\mathsf{Matrix}[j - i + 1, 1] \text{ of } \alpha, \max(\mathsf{c}, \mathsf{c}_3))}$$

where $u > j, j \geq i \geq 0, \alpha \in \mathcal{A}$

Casts. Type conversions in CAO are carried out using C-like casts. We have defined a type cast relation denoted by $\Rightarrow$ to specify which types can be converted. The type cast relation is reflexive (meaning that any type can be casted to itself) and preserves coercions, i.e., an explicit cast can be used whenever types are coercible. Since coercion subsumes reflexivity, only the following rule is needed:

$$\frac{\vdash_\leq \tau_1 \leq \tau_2}{\vdash_c \tau_1 \Rightarrow \tau_2}$$

Using the type cast relation, only one typing rule has to be provided for cast expressions.

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e \leq (\tau', \mathsf{c}) \quad \vdash_c \tau' \Rightarrow \tau}{\Gamma, \Delta \vdash (t)\ e :: (\tau, \mathsf{c})}$$

Bit strings are converted to integers differently, depending on whether they are signed or unsigned. Indeed this is the only distinction between the two flavours of bit strings. For unsigned bit strings, all bits are considered to construct the magnitude of a positive integer. For signed bit strings, two's complement representation is considered. Conversions in the reverse direction, i.e. from integers to bit strings, are always performed by first calculating the two's complement representation, and then taking the i least significant bits.

$$\frac{}{\vdash_c \mathsf{Int} \Rightarrow \mathsf{Bits}\ [i]}$$

Conversion from modular types to integer types in cases where the modulus is an integer is performed in the trivial way, whereas in the reverse direction one may incur in a modular reduction operation. Conversion between different modular types can occur from base field elements to elements of corresponding field extension, again in the trivial way. Conversion in the reverse direction is possible by casting into a vector of the size of the extension degree and contents of the type of the base field. A full element of an extended field can also be constructed from a vector of the type of the base field and size equal to the degree of the field extension. The same applies for line or column matrices.

$$\frac{}{\vdash_c \mathsf{Mod}\ [n] \Rightarrow \mathsf{Int}}$$

$$\frac{}{\vdash_c \mathsf{Mod}\ [n] \Rightarrow \mathsf{Mod}\ [\tau/pol]}$$

$$\frac{}{\vdash_c \mathsf{Int} \Rightarrow \mathsf{Mod}\ [n]}$$

$$\frac{}{\vdash_c \mathsf{Int} \Rightarrow \mathsf{Mod}[\tau/pol]}$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Vector}[i] \text{ of } \tau_1 \Rightarrow \mathsf{Mod}[\tau_2/pol]} \quad i = degree(pol)$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Mod}[\tau_1/pol] \Rightarrow \mathsf{Vector}[i] \text{ of } \tau_2} \quad i = degree(pol)$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Matrix}[1,j] \text{ of } \tau_1 \Rightarrow \mathsf{Mod}[\tau_2/pol]} \quad j = degree(pol)$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Matrix}[i,1] \text{ of } \tau_1 \Rightarrow \mathsf{Mod}[\tau_2/pol]} \quad i = degree(pol)$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Mod}[\tau_1/pol] \Rightarrow \mathsf{Matrix}[1,j] \text{ of } \tau_2} \quad j = degree(pol)$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Mod}[\tau_1/pol] \Rightarrow \mathsf{Matrix}[i,1] \text{ of } \tau_2} \quad i = degree(pol)$$

Finally, conversions between vectors and matrices are only possible for line or column matrices, in the natural way.

$$\frac{\vdash_c \alpha \Rightarrow \tau}{\vdash_c \mathsf{Matrix}[1,i] \text{ of } \alpha \Rightarrow \mathsf{Vector}[i] \text{ of } \tau} \quad \alpha \in \mathcal{A}$$

$$\frac{\vdash_c \alpha \Rightarrow \tau}{\vdash_c \mathsf{Matrix}[i,1] \text{ of } \alpha \Rightarrow \mathsf{Vector}[i] \text{ of } \tau} \quad \alpha \in \mathcal{A}$$

$$\frac{\vdash_c \tau \Rightarrow \alpha}{\vdash_c \mathsf{Vector}[i] \text{ of } \tau \Rightarrow \mathsf{Matrix}[i,1] \text{ of } \alpha} \quad \alpha \in \mathcal{A}$$

$$\frac{\vdash_c \tau \Rightarrow \alpha}{\vdash_c \mathsf{Vector}[i] \text{ of } \tau \Rightarrow \mathsf{Matrix}[1,i] \text{ of } \alpha} \quad \alpha \in \mathcal{A}$$

$$\frac{\vdash_c \tau_1 \Rightarrow \tau_2}{\vdash_c \mathsf{Vector}[i] \text{ of } \tau_1 \Rightarrow \mathsf{Vector}[i] \text{ of } \tau_2}$$

$$\frac{\vdash_c \alpha_1 \Rightarrow \alpha_2}{\vdash_c \mathsf{Matrix}[i,j] \text{ of } \alpha_1 \Rightarrow \mathsf{Matrix}[i,j] \text{ of } \alpha_2} \quad \alpha_1, \alpha_2 \in \mathcal{A}$$

A.2 Statements

Left values. Left values are used in assignments and the typing rules are similar to the ones already presented for struct fields, bit strings, vectors and matrices.

$$\frac{\Gamma_G(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{Procedure})} \quad \tau \in dom(\Gamma_G)$$

$$\frac{\Gamma_L(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{Pure})} \quad \tau \in dom(\Gamma_L)$$

$$\frac{\Gamma_G(fi) = (\tau_1 \to \tau_2, \mathsf{Pure}) \quad \Gamma_G, \Gamma_L, \Delta \vdash l :: (\tau_1, \mathsf{c})}{\Gamma_G, \Gamma_L, \Delta \vdash l.fi :: (\tau_2, \mathsf{c})} \quad fi \in dom(\Gamma_G)$$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Bits}[i], \mathsf{c}_1) \quad \Gamma, \Delta \vdash e \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash l[e] :: (\mathsf{Bits}[1], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Vector}[i] \text{ of } \tau, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma, \Delta \vdash l[e] :: (\tau, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Bits}[k], \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma, \Delta \vdash l[e_1..e_2] :: (\mathsf{Bits}[j - i + 1], \mathsf{c})} \quad k > j, j \geq i \geq 0$$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Vector}[k] \text{ of } \tau, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma, \Delta \vdash l[e_1..e_2] :: (\mathsf{Vector}[j - i + 1] \text{ of } \tau, \mathsf{c})} \quad k > j, j \geq i \geq 0$$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Matrix}[i, j] \text{ of } \alpha, \mathsf{c}_1) \quad \Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_2) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_3)}{\Gamma, \Delta \vdash l[e_1, e_2] :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3))}$$

where $\alpha \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Matrix}[u, v] \text{ of } \alpha, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \quad \phi_\Delta(e_4) = n}{\Gamma, \Delta \vdash l[e_1..e_2, e_3..e_4] :: (\mathsf{Matrix}[j - i + 1, n - k + 1] \text{ of } \alpha, \mathsf{c})}$$

where $u > j, j \geq i \geq 0, v > n, n \geq k \geq 0, \alpha \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Matrix}[u, v] \text{ of } \alpha, \mathsf{c}) \quad \Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \phi_\Delta(e_2) = k \quad \phi_\Delta(e_3) = n}{\Gamma, \Delta \vdash l[e_1, e_2..e_3] :: (\mathsf{Matrix}[1, n - k + 1] \text{ of } \alpha, \max(\mathsf{c}, \mathsf{c}_1))}$$

where $v > n, n \geq k \geq 0, \alpha \in \mathcal{A}$

$$\frac{\Gamma, \Delta \vdash l :: (\mathsf{Matrix}[u, v] \text{ of } \alpha, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \Gamma, \Delta \vdash e_3 \leq (\mathsf{Int}, \mathsf{c}_3)}{\Gamma, \Delta \vdash l[e_1..e_2, e_3] :: (\mathsf{Matrix}[j - i + 1, 1] \text{ of } \alpha, \max(\mathsf{c}, \mathsf{c}_3))}$$

where $u > j, j \geq i \geq 0, \alpha \in \mathcal{A}$

STATEMENTS. We introduce some additional notation in type checking rules for statements. The operator $\Vdash_\beta$ denotes type judgements of special statements that may modify the type environment relation: it retrieves not only a typed statement, but also a new type environment relation. The subscript β in operator $\Vdash_\beta$ represents the return type of the function in which the statement was defined. This information is particular useful, allowing the type checker to guarantee that the several return statements that may appear in a function are always in accordance with the return type of the corresponding function declaration. We also introduce symbol $\bullet$ as a possible return type. The objective of this

symbol is to distinguish the cases when a block has explicitly executed a return statement (in which case we use the type of the parameter passed to the return statement) from the cases where no return statement has been executed in the block (in which case we use the $\bullet$ to signal this situation). We include these rules in Figures 2 and 3.

A.3 Programs/declarations

Programs. A program consists of procedure, function, variable, and struct declarations. In our approach to type checking a program, the order in which these constructs appear may have an impact on the outcome of the type checker. This is because we are augmenting our type environment Γ each time we encounter one of these declarations, and proceeding immediately to type check the declaration contents. In general this would be a problematic situation, as it would mean that the order by which the constructs are defined in a program would be relevant and no function would be able to rely on something that is subsequently declared. However, because CAO disallows recursive definitions of all of these construct declarations, this aspect is reduced to a mere presentation detail of CAO programs. Moreover, we rely in this fact to allow for just one pass in the source code to determine function classifiers. This means that when functions are called their respective classification is already known.

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n \quad \Delta \vdash_t t \rightsquigarrow \tau \qquad \Gamma_G, \circ[x_1 :: \tau_1, \ldots, x_n :: \tau_n], \Delta \Vdash_\tau c :: (\tau, \mathsf{cc}, \Gamma'_G)}{\Gamma_G, \circ, \Delta \Vdash \mathsf{def}\ fp(x_1 : t_1, \ldots, x_n : t_n) : t\ \{c\} :: (\bullet, \Gamma_G[fp :: ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{cc})])}$$

where $t \neq \mathsf{void}$ and $fp \notin dom(\Gamma_G)$

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n \qquad \Gamma_G, \circ[x_1 :: \tau_1, \ldots, x_n :: \tau_n], \Delta \Vdash_{()} c; \mathsf{return}() :: ((), \mathsf{Procedure}, \Gamma'_G)}{\Gamma_G, \circ, \Delta \Vdash \mathsf{def}\ fp(x_1 : t_1, \ldots, x_n : t_n) : \mathsf{void}\ \{c\} :: (\bullet, \Gamma_G[fp :: ((\tau_1, \ldots, \tau_n) \to (), \mathsf{Procedure})])}$$

where $fp \notin dom(\Gamma_G)$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \Vdash \mathsf{typedef}\ tid\ :=\ t :: (\bullet, \Gamma)}$$

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n}{\begin{array}{c}\Gamma_G, \Gamma_L, \Delta \Vdash \mathsf{typedef}\ sid := \mathsf{struct}[fi_1 : t_1; \ldots; fi_n : t_n] :: \\ (\bullet, \Gamma_G[fi_1 :: (sid \to \tau_1, \mathsf{Pure}), \ldots, fi_n :: (sid \to \tau_n, \mathsf{Pure})], \Gamma_L)\end{array}}$$

where $sid, fi_1, \ldots, fi_n \notin dom(\Gamma)$ where $fi_i \neq fi_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

$$\frac{\circ, \circ, \circ \Vdash d_1 :: (\bullet, \Gamma_{G_1}) \quad \ldots \quad \Gamma_{G_{n-1}}, \circ, \circ \Vdash d_n :: (\bullet, \Gamma_G)}{\circ, \circ, \circ \Vdash d_1; \ldots; d_n :: (\bullet, \Gamma_G)} \quad \mathsf{main} :: () \to () \in \Gamma$$

B Semantics

In this section, we assume that the CAO program has been previously type-checked using the rules presented in the previous section, and that types of expressions are available at all times. Whenever type information is needed, we denote it in superscript.

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x : t :: (\bullet, \mathsf{Pure}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x_1, \ldots, x_n : t :: (\bullet, \mathsf{Pure}, \Gamma[x_1 :: \tau, \ldots, x_n :: \tau])}$$

where $x_1, \ldots, x_n \notin dom(\Gamma), x_i \neq x_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e \leq (\tau, \mathsf{cc})}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x : t := e :: (\bullet, \mathsf{cc}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau \quad \Gamma, \Delta \vdash e_1 \leq (\tau, \mathsf{cc}_1) \ldots \Gamma, \Delta \vdash e_n \leq (\tau, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x : t := \{e_1, \ldots, e_n\} :: (\bullet, \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma[x :: \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau])}$$

where $x \notin dom(\Gamma)$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Matrix}\ [i, j]\ \mathsf{of}\ \alpha \quad \Gamma, \Delta \vdash e_1 \leq (\alpha, \mathsf{cc}_1) \ldots \Gamma, \Delta \vdash e_n \leq (\alpha, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_\alpha \mathsf{def}\ x : t := \{e_1, \ldots, e_n\} :: (\bullet, \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma[x :: \mathsf{Matrix}\ [i, j]\ \mathsf{of}\ \alpha])}$$

where $\alpha \in \mathcal{A}, x \notin dom(\Gamma), i \times j = n$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash l_1 :: (\tau_1, \mathsf{cl}_1) \quad \ldots \quad \Gamma, \Delta \vdash l_n :: (\tau_n, \mathsf{cl}_n) \\ \Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma, \Delta \Vdash_\tau l_1, \ldots, l_n := e_1, \ldots, e_n :: (\bullet, \max(\mathsf{cl}_1 \ldots, \mathsf{cl}_n, \mathsf{c}_1, \ldots, \mathsf{c}_n), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash l_1 :: (\tau_1, \mathsf{cl}_1) \quad \ldots \quad \Gamma, \Delta \vdash l_n :: (\tau_n, \mathsf{cl}_n) \quad \Gamma, \Delta \vdash e \leq ((\tau_1, \ldots, \tau_n), \mathsf{c})}{\Gamma, \Delta \Vdash_\tau l_1, \ldots, l_n := e :: (\bullet, \max(\mathsf{cl}_1, \ldots, \mathsf{cl}_n, \mathsf{c}), \Gamma)}$$

$$\frac{\begin{array}{c}\Gamma_G(fp) = ((\tau_1, \ldots, \tau_n) \rightarrow (\tau_{n+1}, \ldots, \tau_m), \mathsf{Procedure}) \\ \Gamma_G, \Gamma_L, \Delta \vdash l_{n+1} :: (\tau_{n+1}, \mathsf{cl}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash l_m :: (\tau_m, \mathsf{cl}_m) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau l_{n+1}, \ldots, l_m := fp(e_1, \ldots, e_n) :: (\bullet, \mathsf{Procedure}, \Gamma_G, \Gamma_L)} \quad fp \in dom(\Gamma_G)$$

$$\frac{\begin{array}{c}\Gamma_G(fp) = ((\tau_1, \ldots, \tau_n) \rightarrow (), \mathsf{Procedure}) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau fp(e_1, \ldots, e_n) :: (\bullet, \mathsf{Procedure}, \Gamma_G, \Gamma_L)} \quad fp \in dom(\Gamma_G)$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{cc}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_{(\tau_1, \ldots, \tau_n)} \mathsf{return}\ e_1, \ldots, e_n :: ((\tau_1, \ldots, \tau_n), \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma)}$$

$$\frac{\Gamma, \Delta \Vdash_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma', \Delta \Vdash_\tau c_2; \ldots; c_n :: (\rho, \mathsf{cc}_{2n}, \Gamma'')}{\Gamma, \Delta \Vdash_\tau c_1; \ldots; c_n :: (\rho, \max(\mathsf{cc}_1, \mathsf{cc}_{2n}), \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

$$\frac{\Gamma, \Delta \Vdash_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma', \Delta \Vdash_\tau c_2; \ldots; c_n :: (\rho, \mathsf{cc}_{2n}, \Gamma'')}{\Gamma, \Delta \Vdash_\tau c_1; \ldots; c_n :: (\tau, \max(\mathsf{cc}_1, \mathsf{cc}_{2n}), \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

Fig. 2: Type checking rules for CAO statements (Part I).

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\bullet, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\tau, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\bullet, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\tau, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\tau, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}), \Gamma)} \quad \rho \in \{\tau, \bullet\}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma')}{\Gamma, \Delta \Vdash_\tau \mathsf{while}\ b\ \{c\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}), \Gamma)} \quad \rho \in \{\tau, \bullet\}$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \\ \forall_{n \in \{i, i+k, \ldots, j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$
$$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i \leq j,\ k \geq 1$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \\ \forall_{n \in \{i, i-k, \ldots, j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$
$$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i > j,\ k \geq 1$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \\ \forall_{n \in \{i, i+1, \ldots, j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$
$$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i \leq j$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \\ \forall_{n \in \{i, i-1, \ldots, j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$
$$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i > j$$

Fig. 3: Type checking rules for CAO statements (Part II).

Source annotation. As discussed in the presentation of the type system, CAO allows for automatic coercions between certain types, not requiring the introduction of explicit casts. Since casts can imply conversions between values of different types, these have a well-defined evaluation semantics, as will be presented in this section. However, since coercions are implicit casts that are only considered during type-checking, this poses a problem when evaluating a CAO program because operations in the semantics expect values with the same representation, which does not happen when we have two operands with different but coercible types.

Similarly, in CAO there is a syntactic overloading of native operators that is resolved by the type system. In the presentation of the semantics we preserve this overloading, but only for compactness purposes. In fact, the CAO semantics requires full explicit knowledge of the declared types of operators when evaluating a program. The same discussion applies to the type name of a structure when a projection is being applied.

In our implementation, this is solved by annotating the abstract representation of the source program during type-checking with explicit casts (corresponding to coercions) and also with the full types of operators and structures; the evaluation of the semantics uses this annotated code. In the theoretical presentation of the semantics, we will consider that the code does not have any type coercion: either because the programmer has provided explicit casts, or because there was a program transformation that has introduced these casts implicitly. Furthermore, we will assume that the CAO program that is evaluated has been annotated so as to eliminate the ambiguity as to which native operator or structure is being used at all points in the program. This program transformation does not change the semantics of the program and can be assumed without loss of generality: it makes explicit the (unique) implicit meaning that the type checking rules are able to reveal, but that the semantic rules require.

Semantic domain. The domain of *values* is defined as a solution of the following domain equation

$$\mathbf{V} = \mathbb{Z} + \mathbf{V}^\star + \mathcal{E}$$

where $\mathbb{Z}$ denotes the domain of integers, $\mathbf{V}^\star$ denotes sequences of values of type $\mathbf{V}$ of the form $[v_0, \ldots, v_{n-1}]$ and $\mathcal{E}$ is the type of the run-time error value ϵ. We use v or $[v_0, \ldots, v_{n-1}]$ to range over values, and leave co-product injections implicit. We note that the simplicity of the semantic domain is obtained at the expense of a non-disjoint interpretation of different CAO types. This implies that to recover the meaning of a denotation of a CAO expression (an element of $\mathbf{V}$) we need to consider its type.

Interpretation of literals. CAO literals are encoded in $\mathbf{V}$ by an appropriate interpretation function:

$$[\![l^\sigma]\!] : \mathbf{Lit} \to \mathbf{V}$$

We do not present these encodings in detail, as they can be trivially implemented in various ways. As an example, we might have:

$$[\![\texttt{true}^{\texttt{bool}}]\!] = 1,\ [\![\texttt{1}^{\texttt{int}}]\!] = 1,\ [\![\texttt{11}^{\texttt{mod 7}}]\!] = 4,\ [\![\texttt{[1,0,1]}^{\texttt{bits[3]}}]\!] = [1,0,1],\ [\![sid]\!] = [102,[17,27]]$$

where *sid* is some struct type with two fields, the second of which is itself of a struct type.

Interpretation of operations. We shall also consider a set of primitive operations that assign meaning to CAO operators. Again, these will be characterised by an interpretation function:

$$[\![\mathrm{op}^{(\sigma_1,\ldots,\sigma_n)\to\sigma}]\!] : \mathbf{V}^\star \to \mathbf{V}$$

The semantics of the native operators in CAO have been informally described in the previous section, as the type-checking rules were presented. As for literal encodings, the implementation of these operations over the semantic domain can be done in a trivial way, which we omit for simplicity.

Stores and frames. The meaning of variables is kept in a *store*, which is a partial function mapping general identifiers to values. In order to handle a richer scope discipline, we also consider *frames* as stacks (lists) of stores: $\mathbf{St} = \mathbf{Id} \to \mathbf{V}$, $\mathbf{Fr} = \mathbf{St}^+$.

A frame keeps track of the full hierarchy of scopes (from the innermost to the outermost) in a given program point. The last store should always be present and represents the global variable store. We do not insist that the domains of these stores are disjoint, because the store access and update operations will take the first store they encounter that defines the variable in question. It is also convenient to consider some auxiliary functions acting on frames. Table 5 lists these functions together with the notation used and its informal meaning. We will also consider iterated versions of the update functions, for example we

Table 5: Auxiliary functions for frame manipulation

Function	Notation	Description
$\mathrm{frAcc}(\rho, v)$	$\rho(v)$	Returns the value of a variable v in frame ρ
$\mathrm{frUpd}(\rho, v, x)$	$\rho[x/v]$	Updates an existing variable v with value x in frame ρ
$\mathrm{frAdd}(\rho, v, x)$	$\rho[v := x]$	Inserts a new variable v in the first store of frame ρ
$\mathrm{frLVAcc}(\rho, l)$	$\rho(l)$	Returns the value of an lvalue l in frame ρ
$\mathrm{frLVUpd}(\rho, l, x)$	$\rho[x/l]$	Updates an lvalue l with value x in frame ρ
$\mathrm{push}(st, \rho)$		Pushes store st in frame ρ
$\mathrm{pop}(\rho)$		Deletes the top store in frame ρ
$\mathrm{global}(\rho)$		Returns a frame containing only the global store of ρ
$\mathrm{local}(\rho)$		Returns a frame containing only the local stores of ρ
$\mathrm{at}(n, [x_0, \ldots, x_{n-1}])$		Returns the n-th element of a list
$\mathrm{getPos}(x, [x_0, \ldots, x_{n-1}])$		Finds the position of x in a list

will use $\rho[x_1, \ldots, x_n/v_1, \ldots, v_n]$ to denote the iterated substitution $((\rho[x_1/v_1])\ldots)[x_n/v_n]$.

Evaluation relation. The evaluation of a CAO program will be defined by an appropriate *evaluation relation* that will relate an initial configuration (a CAO program together with a description of the initial state) with a final configuration (a semantic value and a final state). More concretely, we will define three mutually recursive evaluation relations, each of them responsible for characterising the evaluation of different syntactic classes: *expressions*, *statements* and *declarations*:

- $\langle\ e \mid \rho\ \rangle \to r$ means that the evaluation of expression e in state (frame) ρ evaluates to the value r. Expression evaluation is side-effect free, and hence the state is not changed.
- $\langle\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ means that the evaluation of statement c in state (frame) ρ transforms the state into ρ', and (possibly) produces result r.
- $\langle\ d \mid \rho\ \rangle \Rrightarrow \langle\ \rho'\ \rangle$ means that the evaluation of declaration d in state (frame) ρ transforms the state into ρ'.

Lookup functions. Finally, we assume the existence of functions that access type and function declarations, namely:

$$\begin{aligned} \text{lookupType} &: \mathbf{Id} \to \mathbf{Id}^{+} \\ \text{lookupFun} &: \mathbf{Id} \to \mathbf{Id}^{\star} \times \mathbf{Stm}^{+} \end{aligned}$$

where lookupType receives a variable identifier and retrieves a non-empty list containing variable identifiers. The reason for having this function returning a list rather than a single value is explained by the case where it receives a struct identifier, in which case it has to return all of the struct field identifiers. Function lookupFun shares a similar purpose to lookupType, but this time we use it to retrieve two different kinds of information associated to function declarations: the declared parameters of the function (possibly none), and its associated body statements.

Errors. A *trapped error* is an execution error that results in an immediate fault (run-time error), and it is denoted by ϵ; an *untrapped error* is an execution error that does not immediately result in a fault, corresponding to an unexpected behaviour. We denote such an error by $\perp$ (considering the lift version of the semantic domain $\mathbf{V}_{\perp}$). CAO errors are propagated through rules, i.e., whenever a premiss evaluates to ϵ the overall rule also evaluates to ϵ.
All cases for which an evaluation relation is not defined are implicitly evaluated to untrapped errors ($\perp$ value).

B.1 Expressions

Expression evaluation. Figure 4 shows the expression evaluation rules. Note that in rule Fun, the evaluation of the function body necessarily returns a result, since we only allow side-effect-free functions in expressions. The non-void return type enforces that every possible control path terminates with a *return* statement.

Expression evaluation can only lead to run-time errors when trying to access an index of a container type (vector, matrix or bit string) outside of bounds, when trying to divide or get the remainder of the division by zero, or when the divisor and moduli of a modular type are not coprime.

Semantics of type casts. In CAO type casts are only allowed between predefined data types, carrying a transformation in the values, i.e., a conversion between the formats in the two data types. Therefore, the exact meaning of a cast must be formalized in the language

$$\textsc{Lit}\ \frac{}{\langle\ l^\sigma \mid \rho\ \rangle \to [\![l^\sigma]\!]} \qquad \textsc{Var}\ \frac{}{\langle\ y \mid \rho\ \rangle \to \rho(y)}$$

$$\textsc{VSel}\ \frac{\langle\ ea \mid \rho\ \rangle \to [r_0, \ldots, r_{n-1}] \qquad \langle\ ei \mid \rho\ \rangle \to i}{\langle\ ea[ei] \mid \rho\ \rangle \to \mathrm{at}(i, [r_0, \ldots, r_{n-1}])} \quad 0 \le i < n$$

$$\textsc{VSel-Err}\ \frac{\langle\ ea \mid \rho\ \rangle \to [r_0, \ldots, r_{n-1}] \qquad \langle\ ei \mid \rho\ \rangle \to i}{\langle\ ea[ei] \mid \rho\ \rangle \to \epsilon} \quad i < 0 \vee i \ge n$$

$$\textsc{SSel}\ \frac{\langle\ ea \mid \rho\ \rangle \to [r_0, \ldots, r_{n-1}]}{\langle\ ea^\sigma.n \mid \rho\ \rangle \to \mathrm{at}(\mathrm{getPos}(n, [n_0, \ldots, n_{n-1}]), [r_0, \ldots, r_{n-1}])}$$

where $[n_0, \ldots, n_{n-1}] = \mathrm{lookupType}(\sigma)$

$$\textsc{Op}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_0 \quad \cdots \quad \langle\ e_n \mid \rho\ \rangle \to r_{n-1}}{\langle\ \mathrm{op}(e_1, \ldots, e_n) \mid \rho\ \rangle \to [\![\mathrm{op}]\!]([r_0, \ldots, r_{n-1}])}$$

where op is a primitive operation, except division and remainder by zero

$$\textsc{Div-Zero}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_0 \quad \langle\ e_2 \mid \rho\ \rangle \to 0}{\langle\ e_1 \,/\, e_2 \mid \rho\ \rangle \to \epsilon}$$

$$\textsc{Div-Mod}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_0 \quad \langle\ e_2 \mid \rho\ \rangle \to r_1}{\langle\ e_1 \,/\, e_2 \mid \rho\ \rangle \to \epsilon}$$

where r_0 and r_1 are representations of values of type $\mathsf{Mod}[n]$ and $\gcd(r_1, n) \neq 1$

$$\textsc{Rem-Zero}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_0 \quad \langle\ e_2 \mid \rho\ \rangle \to 0}{\langle\ e_1 \,\%\, e_2 \mid \rho\ \rangle \to \epsilon}$$

$$\textsc{Fun}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_0 \quad \cdots \quad \langle\ e_n \mid \rho\ \rangle \to r_{n-1} \qquad \langle\ \mathit{body} \mid \mathrm{push}(\circ[p_1, \ldots, p_n := r_0, \ldots, r_{n-1}], \mathrm{global}(\rho))\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}{\langle\ f(e_1, \ldots, e_n) \mid \rho\ \rangle \to r}$$

where $([p_1, \ldots, p_n], \{\mathit{body}\}) = \mathrm{lookupFun}(f)$

Fig. 4: Expression evaluation rules

semantics, by providing the corresponding conversion function on the semantic domain between values of types τ and τ':

$$conversion_{\tau \ to \ \tau'} : \mathbf{V} \rightarrow \mathbf{V}$$

This must be a total function, i.e., it is defined for every values of the domain (τ data type). We must note that although coercions are implicit conversions, these also correspond to the application of a conversion function.

The conversion functions are dependent on the particular encoding chosen for values belonging to a certain data type. Since, we do not enforce a particular encoding for values in this specification, leaving it open to several possibilities, we also do not present the definition of the possible conversions functions. However, these are quite straightforward since it is easy and intuitive to define a conversion function for any coercion and cast allowed in CAO.

B.2 Statements

Left values. The evaluation of left-values requires checking some side-conditions regarding the index accesses. Therefore, functions frLVAcc and frLVUpd return an error (signaled by value ϵ) whenever trying to access or assign to a position out of bounds of the representation. In the case of assignments, value ϵ denotes both an error value and an error state.

Statement evaluation. Figures 5 and 6 show the statement evaluation rules. We reuse symbol $\bullet$ with a similar purpose from the one employed in the previous section, i.e. we use this symbol to explicitly capture the non-execution of a return statement in statement blocks. The semantics restrict the use of functions that return multiple values to parallel assignment statements. In the evaluation of while statements we use two different evaluation rules to capture the correct execution of return statements inside while loop bodies. Note that rule Proc creates a new frame where only global variables are preserved, thus disallowing nested functions from accessing local variables of its caller. Because statement blocks may themselves declare new local (with respect to the block and its nested blocks) variables, a new store is created each time a block is evaluated. By the end of the block evaluation this store is discharged so that it cannot influence statements outside the block. As expected, sequences of statements are immediately interrupted as soon as a value is explicitly returned.

Statement evaluation can only lead to run-time errors when trying to assign a value to an index outside of bounds of a container type. Unlimited memory space is assumed, meaning that no errors can occur due to space limitation.

B.3 Programs/declarations

Program evaluation. Figure 7 presents the program evaluation rules. We consider functions, procedures and global type declarations as being transparently available to the

$$\textsc{LocalVar-Un}\ \frac{}{\langle\ \texttt{def}\ \ y{:}\sigma := \ \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho[y := d^{\sigma}]\ \rangle}$$

where $d^{\sigma} \in \mathbf{V}$ is the default value associated to type σ.

$$\textsc{LocalVar}\ \frac{\langle\ e \mid \rho\ \rangle \to r}{\langle\ \texttt{def}\ \ y{:}\sigma := e \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho[y := r]\ \rangle}$$

$$\textsc{Assign}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_1 \qquad \cdots \qquad \langle\ e_n \mid \rho\ \rangle \to r_n}{\langle\ l_1, \ldots, l_n\ \texttt{:=}\ e_1, \ldots, e_n \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho[r_1, \ldots, r_n / l_1, \ldots, l_n]\ \rangle} \quad \rho[r_1, \ldots, r_n / l_1, \ldots, l_n] \neq \epsilon$$

$$\textsc{Assign-Err}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_1 \qquad \cdots \qquad \langle\ e_n \mid \rho\ \rangle \to r_n}{\langle\ l_1, \ldots, l_n\ \texttt{:=}\ e_1, \ldots, e_n \mid \rho\ \rangle \Rightarrow \langle\ \epsilon\ ,\ \_\ \rangle} \quad \rho[r_1, \ldots, r_n / l_1, \ldots, l_n] = \epsilon$$

$$\textsc{AssignTuple}\ \frac{\langle\ e \mid \rho\ \rangle \to [r_1, \ldots, r_n]}{\langle\ l_1, \ldots, l_n\ \texttt{:=}\ e \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho[r_1, \ldots, r_n / l_1, \ldots, l_n]\ \rangle} \quad \rho[r_1, \ldots, r_n / l_1, \ldots, l_n] \neq \epsilon$$

$$\textsc{AssignTuple-Err}\ \frac{\langle\ e \mid \rho\ \rangle \to [r_1, \ldots, r_n]}{\langle\ l_1, \ldots, l_n\ \texttt{:=}\ e \mid \rho\ \rangle \Rightarrow \langle\ \epsilon\ ,\ \_\ \rangle} \quad \rho[r_1, \ldots, r_n / l_1, \ldots, l_n] = \epsilon$$

$$\textsc{AssignProc}\ \frac{\begin{array}{c}\langle\ e_1 \mid \rho\ \rangle \to r_1 \quad \cdots \quad \langle\ e_n \mid \rho\ \rangle \to r_n \\ \langle\ \mathit{body} \mid \mathrm{push}(\circ[p_1, \ldots, p_n := r_1, \ldots, r_n], \mathrm{global}(\rho))\ \rangle \Rightarrow \langle\ [s_1, \ldots, s_m]\ ,\ \rho'\ \rangle\end{array}}{\langle\ l_1, \ldots, l_m\ \texttt{:=}\ \mathit{fp}(e_1, \ldots, e_n) \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \mathrm{push}(\mathrm{local}(\rho), \mathrm{global}(\rho'))[s_1, \ldots, s_m / l_1, \ldots, l_m]\ \rangle}$$

where $([p_1, \ldots, p_n], \{\mathit{body}\}) = \mathrm{lookupFun}(\mathit{fp})$ and $\mathrm{push}(\mathrm{local}(\rho), \mathrm{global}(\rho'))[s_1, \ldots, s_m / l_1, \ldots, l_m] \neq \epsilon$

$$\textsc{AssignProc-Err}\ \frac{\begin{array}{c}\langle\ e_1 \mid \rho\ \rangle \to r_1 \quad \cdots \quad \langle\ e_n \mid \rho\ \rangle \to r_n \\ \langle\ \mathit{body} \mid \mathrm{push}(\circ[p_1, \ldots, p_n := r_1, \ldots, r_n], \mathrm{global}(\rho))\ \rangle \Rightarrow \langle\ [s_1, \ldots, s_m]\ ,\ \rho'\ \rangle\end{array}}{\langle\ l_1, \ldots, l_m\ \texttt{:=}\ \mathit{fp}(e_1, \ldots, e_n) \mid \rho\ \rangle \Rightarrow \langle\ \epsilon\ ,\ \_\ \rangle}$$

where $([p_1, \ldots, p_n,], \{\mathit{body}\}) = \mathrm{lookupFun}(\mathit{fp})$ and $\mathrm{push}(\mathrm{local}(\rho), \mathrm{global}(\rho'))[s_1, \ldots, s_m / l_1, \ldots, l_m] = \epsilon$

$$\textsc{Proc}\ \frac{\begin{array}{c}\langle\ e_1 \mid \rho\ \rangle \to r_1 \quad \cdots \quad \langle\ e_n \mid \rho\ \rangle \to r_n \\ \langle\ \mathit{body} \mid \mathrm{push}(\circ[p_1, \ldots, p_n := r_1, \ldots, r_n], \mathrm{global}(\rho))\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle\end{array}}{\langle\ \mathit{fp}(e_1, \ldots, e_n) \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \mathrm{push}(\mathrm{local}(\rho), \mathrm{global}(\rho'))\ \rangle}$$

where $([p_1, \ldots, p_n], \{\mathit{body}\}) = \mathrm{lookupFun}(\mathit{fp})$

$$\textsc{Return}\ \frac{\langle\ e_1 \mid \rho\ \rangle \to r_1 \quad \cdots \quad \langle\ e_n \mid \rho\ \rangle \to r_n}{\langle\ \texttt{return}\ e_1, \ldots, e_n \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \rho\ \rangle}$$

$$\textsc{StmtBlock}\ \frac{\langle\ c_1; \ldots; c_n \mid \mathrm{push}(\circ, \rho)\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}{\langle\ \{c_1; \ldots; c_n\} \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \mathrm{pop}(\rho')\ \rangle}$$

$$\textsc{StmtSeqRet}\ \frac{\langle\ c_1 \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_k]\ ,\ \rho'\ \rangle}{\langle\ c_1; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_k]\ ,\ \rho'\ \rangle}$$

$$\textsc{StmtSeq}\ \frac{\langle\ c_1 \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho'\ \rangle \qquad \langle\ c_2; \ldots; c_n \mid \rho'\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle}{\langle\ c_1; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle}$$

Fig. 5: Statement evaluation rules (Part I)

$$\textsc{IfTrue}\ \frac{\langle\ e \mid \rho\ \rangle \to 1 \qquad \langle\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}{\langle\ \texttt{if}\ \ (e)\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}$$

$$\textsc{IfFalse}\ \frac{\langle\ e \mid \rho\ \rangle \to 0}{\langle\ \texttt{if}\ \ (e)\ c \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho\ \rangle}$$

$$\textsc{IfElseTrue}\ \frac{\langle\ e \mid \rho\ \rangle \to 1 \qquad \langle\ c_1 \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}{\langle\ \texttt{if}\ \ (e)\ c_1\ \texttt{else}\ c_2 \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}$$

$$\textsc{IfElseFalse}\ \frac{\langle\ e \mid \rho\ \rangle \to 0 \qquad \langle\ c_2 \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}{\langle\ \texttt{if}\ \ (e)\ c_1\ \texttt{else}\ c_2 \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle}$$

$$\textsc{WhileTrue}\ \frac{\langle\ e \mid \rho\ \rangle \to 1 \qquad \langle\ c \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho'\ \rangle \qquad \langle\ \texttt{while}\ \ (e)\ c \mid \rho'\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle}{\langle\ \texttt{while}\ \ (e)\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle}$$

$$\textsc{WhileTrueRes}\ \frac{\langle\ e \mid \rho\ \rangle \to 1 \qquad \langle\ c \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \rho'\ \rangle}{\langle\ \texttt{while}\ \ (e)\ c \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \rho'\ \rangle}$$

$$\textsc{WhileFalse}\ \frac{\langle\ e \mid \rho\ \rangle \to 0}{\langle\ \texttt{while}\ \ (e)\ c \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho'\ \rangle}$$

$$\textsc{SeqRet}\ \frac{\begin{array}{c}\langle\ e_1 \mid \rho\ \rangle \to s_1 \qquad \langle\ e_2 \mid \rho\ \rangle \to s_2 \qquad \langle\ e_3 \mid \rho\ \rangle \to s_3 \\ \exists_{n \in \{s_1, s_1+s_3, \ldots, s_2\}} : \forall_{n_0 \in \{s_1, s_1+s_3, \ldots, n-s_3\}} \langle\ c \mid \mathrm{push}(\circ[x := n_0], \mathrm{pop}(\rho_{n_0}))\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho_{n_0+s_3}\ \rangle \\ \wedge \quad \langle\ c \mid \mathrm{push}(\circ[x := n], \mathrm{pop}(\rho_n))\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \rho'\ \rangle\end{array}}{\langle\ \texttt{seq}\ x := e_1\ \texttt{to}\ e_2\ \texttt{by}\ e_3\ \{c\})\ \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \mathrm{pop}(\rho')\ \rangle}$$

where $\rho_{s_1} = \mathrm{push}(\circ, \rho)$

$$\textsc{Seq}\ \frac{\begin{array}{c}\langle\ e_1 \mid \rho\ \rangle \to s_1 \qquad \langle\ e_2 \mid \rho\ \rangle \to s_2 \qquad \langle\ e_3 \mid \rho\ \rangle \to s_3 \\ \forall_{n \in \{s_1, s_1+s_3, \ldots, s_2\}} \langle\ c \mid \mathrm{push}(\circ[x := n], \mathrm{pop}(\rho_n))\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho_{n+s_3}\ \rangle\end{array}}{\langle\ \texttt{seq}\ x := e_1\ \texttt{to}\ e_2\ \texttt{by}\ e_3\ \{c\})\ \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \mathrm{pop}(\rho_{s_2})\ \rangle}$$

where $\rho_{s_1} = \mathrm{push}(\circ, \rho)$

Fig. 6: Statement evaluation rules (Part II)

$$\textsc{GlobalVar}\ \frac{\langle\ e \mid \rho\ \rangle \to r}{\langle\ \texttt{def}\ \ x{:}\sigma := e \mid \rho\ \rangle \Rrightarrow \langle\ \rho[x := r]\ \rangle}$$

$$\textsc{Program}\ \frac{\langle\ d_1 \mid \rho_1\ \rangle \Rrightarrow \langle\ \rho_2\ \rangle \cdots \langle\ d_n \mid \rho_n\ \rangle \Rrightarrow \langle\ \rho'\ \rangle \quad \langle\ \mathit{body} \mid \mathrm{push}(\circ, \rho')\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle}{\langle\ d_1; \ldots; d_n \mid \rho_1\ \rangle \Rrightarrow \langle\ \mathrm{global}(\rho'')\ \rangle}$$

where $([], \{\mathit{body}\}) = \mathrm{lookupFun}(\texttt{main})$

Fig. 7: Program evaluation rules

semantic rules. The interpretation of these declarations can be easily included in the operational semantics by adding rules that update a global environment as the syntactic declarations of these constructs are evaluated. The evaluation of global variable declarations constructs a store with global variables. In order to allow for the execution of additional transformations on this store (which can be caused by the execution of procedures) we further assume the existence of a special procedure main that acts as the entry-point of the program. This procedure can be seen as emulating calls to an API implemented in CAO.

C Type System Soundness

After defining the static (typing) and dynamic (operational) semantics of the CAO language, we aim to prove an important property relating them: type soundness. In our case, we are only interested in proving that the evaluation of well-typed programs does not originate certain kind of errors: we are aiming for a *weak soundness* guarantee.

Concretely, we aim to prove that type-checking excludes the occurrence of untrapped errors (denoted by $\perp$) defined as being the result of trying to evaluate any expression or statement that falls outside the cases captured by the language operational semantics. In particular, the only errors that can occur in well-typed CAO program are trapped errors (denoted by ϵ in the semantic domain $\mathbf{V}$) which are explicitly detected and originated during semantic evaluation, corresponding to the cases that the type checker cannot admittedly detect. These are the cases of access and update of out-of-bound vector or matrix values and invalid divisions as defined in the operational semantics.

Encoding validity. The CAO semantics are defined over a very simple semantic domain, leading to interpretation functions which are not injective. Therefore, a possible interpretation may map both the integer 1 and the boolean value `true` in CAO to the semantic value $1 \in \mathbf{V}$. Thus, from a semantic value $v \in \mathbf{V}$ it is not possible to recover the original type of the CAO value that v is representing. However, this is not a problem as we are assuming that the declared types of all interpreted operators are known at the time of evaluation.

More in detail, the properties of the interpretation function ensure that it returns a valid and consistent representation of its argument. This means that, for a literal l^σ its interpretation $[\![l^\sigma]\!] \in \mathbf{V} \backslash \mathcal{E}$ corresponds to a value that correctly encodes the original type σ. Furthermore, for any operator $\mathrm{op}^{(\sigma_1,\ldots,\sigma_n)\to\sigma}$, it is guaranteed that its interpretation

$$[\![\mathrm{op}^{(\sigma_1,\ldots,\sigma_n)\to\sigma}]\!] : \mathbf{V}^\star \to \mathbf{V}$$

will output a valid representation of the result with respect to its output type σ, as long as it is fed with valid encodings of its input parameters with respect to their expected types $(\sigma_1, \ldots, \sigma_n)$. This discussion justifies the notion of consistency introduced below.

Environment consistency. In order to prove the type soundness theorem, we need to establish a relation between the typing and the evaluation environment.

Definition 1. *Given a typing environment Γ and an evaluation environment (frame) ρ, we say that Γ and ρ are* consistent *(or simply that ρ is a consistent evaluation environment),*

written $\vdash \rho :: \Gamma$, *if* $dom(\rho) \subseteq dom(\Gamma)$ *and for all* $x \in dom(\rho)$ *then* $\rho(x) \neq \epsilon$ *and* $\rho(x)$ *is a valid encoding of a value of type* $\Gamma(x) = \tau$.

Lemma 1. *If a typing environment* Γ *and an evaluation environment* ρ *are consistent, i.e.,* $\vdash \rho :: \Gamma$, *then for every variable* $x \notin dom(\Gamma)$ *of an arbitrary (valid) type* τ, $\vdash \rho :: \Gamma[x :: \tau]$ *holds.*

Proof. By definition of consistency, $dom(\rho) \subseteq dom(\Gamma)$. Since, by hypothesis, for every variable x, $x \notin dom(\Gamma)$ then $x \notin dom(\rho)$ either. Thus, the definition of consistency trivially holds.

Lemma 2. *Given a typing environment* Γ *and an evaluation environment* ρ *which are consistent, i.e.,* $\vdash \rho :: \Gamma$, *for every variable* $x \notin dom(\Gamma)$ *of an arbitrary (valid) type* τ *and a valid encoding* r *of a value of type* τ, *then* $\vdash \rho[x := r] :: \Gamma[x :: \tau]$.

Proof. By Lemma (1), we already know that in these conditions $\vdash \rho :: \Gamma[x :: \tau]$. We just have to prove that also extending ρ keeps the consistency.

In this case, we are extending ρ such that $\rho[x := r](x) = r$ and r is, by hypothesis, a valid encoding of type $\Gamma[x :: \tau](x) = \tau$ (this also means that $r \neq \epsilon$). Since $x \in dom(\Gamma[x :: \tau])$ and $x \in dom(\rho[x := r])$ trivially hold, the consistency of the extended environment follows.

Lemma 3. *Given a typing environment* Γ *and a evaluation environment* ρ *which are consistent, i.e.,* $\vdash \rho :: \Gamma$, *if a variable* $l \in dom(\rho)$ *is updated with a value* r *which is a valid encoding of a value of type* $\Gamma(l)$, *then* $\vdash \rho[r/l] :: \Gamma$.

Proof. The update does not change the domain of ρ; only the previous value is replaced by another encoding which is valid under the original type (also meaning that $r \neq \epsilon$).

Lemma 4. *Given a frame* ρ *then*

$$\rho = \text{push}(\text{local}(\rho), \text{global}(\rho))$$

Proof. Trivial by the properties of frames.

Lemma 5. *Given a frame* ρ *which is consistent with typing environment* Γ, *i.e,* $\vdash \rho :: \Gamma$, *then* $\vdash \text{local}(\rho) :: \Gamma$ *and* $\vdash \text{global}(\rho) :: \Gamma$.

Proof. By Lemma 4 we know that $\rho = \text{push}(\text{local}(\rho), \text{global}(\rho))$. By properties of push function, both $dom(\text{local}(\rho))$ and $dom(\text{global}(\rho))$ are subsets of $dom(\rho)$, meaning that they are also subsets of $dom(\Gamma)$ by transitivity of the subset relation. or the remaining conditions, since they hold for ρ, they also hold for its subsets.

Lemma 6. *Given a typing environment* Γ *and two frames* ρ *and* ρ' *such that* $\vdash \rho :: \Gamma$ *and* $\vdash \rho' :: \Gamma$ *then* $\vdash \text{push}(\rho, \rho') :: \Gamma$.

Proof. By properties of the push function, $dom(\text{push}(\rho, \rho')) = dom(\rho) \cup dom(\rho')$. Since both ρ and ρ' are consistent with environment Γ, then $dom(\rho) \subseteq dom(\Gamma)$ and $dom(\rho') \subseteq dom(\Gamma)$, meaning that $dom(\text{push}(\rho, \rho')) \subseteq dom(\Gamma)$. Moreover, if all other consistency conditions hold individually for frames ρ and ρ', then they also must hold for $\text{push}(\rho, \rho')$.

C.1 Expressions

The type soundness theorem for expressions states that the evaluation of a well-typed expression in CAO, either terminates normally, explicitly raises a trapped error or runs forever. Furthermore, only out-of-bounds accesses and invalid division (division by zero or when the divisor and moduli of a modular type are not coprime) can raise errors; all other cases return a value or run forever.

We should notice that function recursion is not allowed in the language, meaning that while loops are the only possible source of divergence in computation. Since these are part of the statement subset of the language, non-termination in expression evaluation can only occur due to function calls.

Theorem 2. *Let Γ be a typing environment and ρ a consistent evaluation environment such that $\vdash \rho :: \Gamma$. For any expression e, semantic value v and type τ, if $\Gamma \vdash e :: \tau$ and $\langle\ e \mid \rho\ \rangle \to v$* terminates *then $v \in \boldsymbol{V}$. Furthermore, v can only be a trapped error (ϵ) if it is originated by an out-of-bounds accesses or an invalid division; all other evaluations result in a value $v \in \boldsymbol{V}\backslash\mathcal{E}$.*

Proof. The proof follows by induction on typing derivations. The base cases for induction are the evaluation of literal and variables, since their interpretation is the only valid way of introducing new semantic values. The inductive cases correspond to the evaluation of expressions which depend on the evaluation of its sub-expressions which is assumed by hypothesis that is not an error. This means that a semantic function is applied in the semantic values coming from the evaluation of sub-expressions and produces a new semantic value. This will lead to the conclusion that, except in the case of accesses, updates and divisions, whenever the sub-expressions evaluated to non-error values, the returned result is also not an error. Finally, an inductive case is added that corresponds to the implicit case when a trapped error is propagated through semantic rules resulting in a final error value.

Literals. Intuitively, the evaluation of a literal cannot lead to an (trapped or untrapped) error, since any literal without a well-established format would have been rejected by the type checker. Moreover, the semantics of a literal just states that it is mapped to the value it represents.

Formally, for the integer literal case

$$\frac{}{\Gamma \vdash n :: \mathsf{Int}}$$

we have to prove that if $\langle\ n^{\mathsf{Int}} \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}\backslash\mathcal{E}$.

By Lit rule, $\langle\ n^{\mathsf{Int}} \mid \rho\ \rangle$ evaluates to $[\![n^{\mathsf{Int}}]\!]$. Since, by hypothesis, n is of type Int, $[\![n^{\mathsf{Int}}]\!]$ corresponds to the representation of an integer value in the semantic domain $\mathbf{V}$. Since the representation of integers does not belong to $\mathcal{E}$, then $v \in \mathbf{V}\backslash\mathcal{E}$.

The proof is identical for the other literals.

Variables. Intuitively, type checking ensures that all program variables have a well-defined type and that any used variable is within the scope of its definition. Since, by

hypothesis, the evaluation environment is consistent with the typing environment, then all variables are also within the scope during run-time. Moreover, since, by the same hypothesis, the evaluation environment cannot contain error values, the evaluation of a variable cannot lead to an error.

Formally, for the rule

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}$$

we have to prove that if $\langle\ x \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V} \backslash \mathcal{E}$.

By VAR rule, $\langle\ x \mid \rho\ \rangle$ evaluates to $\rho(x)$. By hypothesis, the evaluation environment respects the typing environment, meaning that all variables in the domain of ρ are also defined in domain of Γ and that ρ cannot contain error values. Since, by hypothesis, variable x is defined in Γ, thus, $\rho(x)$ evaluates to a value of the semantic domain $\mathbf{V}$ that, by definition of consistency, is a valid representation of the type $\Gamma(x) = \tau$ and thus not equal to ϵ.

FUNCTION CALLS. For the rule

$$\frac{\begin{array}{c}\Gamma(f) = (\tau_1, \ldots, \tau_n) \to \tau \\ \Gamma \vdash e_1 \leq \tau_1 \quad \ldots \quad \Gamma \vdash e_n \leq \tau_n\end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) :: \tau} \quad f \in dom(\Gamma)$$

we have to prove that if $\langle\ f(e_1, \ldots, e_n) \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V} \backslash \mathcal{E}$.

Since the program type-checks by hypothesis, then function f must be defined and therefore looking up the operational information about this function will not fail, i.e.,

$$([p_1, , \ldots, p_n], \{body\}) = \text{lookupFun}(f)$$

will return the correct result. By FUN rule, $\langle\ f(e_1, \ldots, e_n) \mid \rho\ \rangle$ evaluates to a semantic value v if $\langle\ e_1 \mid st\ \rangle \to r_0, \ldots, \langle\ e_n \mid st\ \rangle \to r_{n-1}$ and $\langle\ body \mid \text{push}(\circ[p_1, \ldots, p_n := r_0, \ldots, r_{n-1}], \text{global}(st))\ \rangle \Rightarrow \langle\ r\ ,\ st'\ \rangle$ terminate. Condition $\langle\ e_1 \mid \rho\ \rangle \to r_0, \ldots, \langle\ e_n \mid \rho\ \rangle \to r_{n-1}$ holds by induction hypothesis, with $r_0, \ldots, r_{n-1}$ being value representations of types $\tau_1, \ldots, \tau_n$, respectively. By induction hypothesis the evaluation of the body of function f with the respective parameters instantiated, returns a value in the semantic domain $\mathbf{V} \backslash \mathcal{E}$.

STRUCT PROJECTIONS. For rule,

$$\frac{\Gamma(fi) = \tau \to \tau' \quad \Gamma \vdash e :: \tau}{\Gamma \vdash e.fi :: \tau'} \quad fi \in dom(\Gamma)$$

we have to prove that if $\langle\ e.fi \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V} \backslash \mathcal{E}$.

Recall that in the semantic evaluation of struct projections, we assume that there is an annotation in the source about the type name of the struct to which the expression e belongs, i.e.,. e^{τ}. This is used to lookup all the possible projections and respective declared types

$$[n_0, \ldots, n_{n-1}] = \text{lookupType}(\tau)$$

necessary by evaluation rule SSel.

By SSel rule, $\langle\ e^{\tau}.\mathit{fi} \mid \rho\ \rangle$ evaluates to $\mathrm{at}(\mathrm{getPos}(\mathit{fi}, [n_0, \ldots, n_{n-1}]), [r_0, \ldots, r_{n-1}])$ if $\langle\ e \mid \rho\ \rangle \to [r_0, \ldots, r_{n-1}]$. This condition holds by induction hypothesis, because $\langle\ e \mid \rho\ \rangle$ evaluates to v_1 which is a valid representation of a struct of type τ. The evaluation of function $\mathrm{getPos}(\mathit{fi}, [n_0, \ldots, n_{n-1}])$ returns the index associated with the projection fi in the projection list associated with struct type τ. Since the program type-checks by hypothesis, projection fi must exist in the definition of the structure contained in the program, and hence this evaluation must return a value belonging to the interval $0, \ldots, n-1$. This is used as argument of function at in order to lookup the correct value in $[r_0, \ldots, r_{n-1}]$; since the interval of indexes are the same, function at will always return a value. By induction hypothesis, all value $r_0, \ldots, r_{n-1}$ belong to $\mathbf{V}\backslash\mathcal{E}$, and therefore this is also true for the evaluation of function at in $[r_0, \ldots, r_{n-1}]$.

Arithmetic operations. For rule

$$\frac{\Gamma \vdash e_1 :: \alpha_1 \quad \Gamma \vdash e_2 :: \alpha_2 \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma \vdash e_1 \oplus e_2 :: \alpha} \quad \alpha, \alpha_1, \alpha_2 \in \mathcal{A} \quad \oplus \in \{+, -, *\}$$

we have to prove that if $\langle\ e_1 \oplus e_2 \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}\backslash\mathcal{E}$.

If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to v_2$ terminate, then $\langle\ e_1 \oplus e_2 \mid \rho\ \rangle$ evaluates to $[\![\oplus]\!][v_1, v_2]$ by semantic rule Op. Here $[\![\oplus]\!]$ gives the interpretation of the operation $\oplus \in \{+, -, *\}$ with respect to the values v_1 and v_2. By induction hypothesis, v_1 and v_2 are in the semantic domain $\mathbf{V}\backslash\mathcal{E}$. Since semantic evaluation occurs in a program where coercions were replaced by explicit casts, both v_1 and v_2 must correspond to representations of values of algebraic type α. Because all of the above operations are well-defined and total for algebraic types, then $[\![\oplus]\!][v_1, v_2]$ evaluates to another value v which is again a representation of an algebraic type α and hence $v \in \mathbf{V}\backslash\mathcal{E}$.

For rule,

$$\frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2 \quad \vdash_{\leq} \tau_1 \leq \mathsf{Int} \quad \vdash_{\leq} \tau_2 \leq \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 :: \mathsf{Int}} \quad \oplus \in \{+, -, *\}$$

we have to prove that if $\langle\ e_1 \oplus e_2 \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}\backslash\mathcal{E}$.

If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to v_2$ terminate, then $\langle\ e_1 \oplus e_2 \mid \rho\ \rangle$ evaluates to $[\![\oplus]\!][v_1, v_2]$ by semantic rule Op. Here $[\![\oplus]\!]$ gives the interpretation of the operation $\oplus \in \{+, -, *\}$ with respect to the values v_1 and v_2. By induction hypothesis, v_1 and v_2 are in the semantic domain $\mathbf{V}\backslash\mathcal{E}$, corresponding to representations of integer values, since semantic evaluation occur in a program without coercions. Since all the above operations are well-defined and total for integer values, then $[\![\oplus]\!][v_1, v_2]$ evaluates to another value v which is again a representation of an integer and hence $v \in \mathbf{V}\backslash\mathcal{E}$.

The proof for these operations on other types follows the same line of reasoning, as well as the proofs for the exponential and symmetric operators.

Division. The only partial arithmetic operators are division and the remainder of the division, leading situations with distinct semantic rules, where trapped errors can occur.

For rule,

$$\frac{\Gamma \vdash e_1 \leq \mathsf{Int} \quad \Gamma \vdash e_2 \leq \mathsf{Int}}{\Gamma \vdash e_1 \oslash e_2 :: \mathsf{Int}} \quad \oslash \in \{/, \%\}$$

we have to prove that if $\langle\ e_1 \oslash e_2 \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}$.

Two semantic rules can be applied for each operator, one in the particular case when the second parameter evaluates to 0; the other in the general case:

- If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to 0$ terminate, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $\epsilon \in \mathbf{V}$ by semantic Div-Zero.
- If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to v_2$ terminate, with $v_2 \neq 0$, then $\langle\ ||e_1/e_2|| \mid \rho\ \rangle$ evaluates to $[\![/]\!][v_1, v_2]$ by semantic rule Op. Here $[\![/]\!]$ gives the interpretation of the / operator with respect to the values v_1 and v_2. By induction hypothesis, v_1 and v_2 are in the semantic domain $\mathbf{V}\backslash\mathcal{E}$, corresponding to representations of integer values. Since division is well-defined for integer representations, then $[\![/]\!][v_1, v_2]$ evaluates to another value v which is again a representation of an integer and hence $v \in \mathbf{V}\backslash\mathcal{E}$.

For divisions of values belonging to modular types, we only have to consider the division when the moduli is an integer, because the extension case only allows field extensions meaning that division is well-defined for all non-zero values; the zero case is captured by the above proof. For rule,

$$\frac{\begin{array}{c}\Gamma \vdash e_1 :: \mathsf{Mod}\ [m_1] \quad \Gamma \vdash e_2 :: \mathsf{Mod}\ [m_2] \\ \mathsf{Mod}\ [m_1] \uparrow \mathsf{Mod}\ [m_2] = \mathsf{Mod}\ [m]\end{array}}{\Gamma \vdash e_1\ /\ e_2 :: \mathsf{Mod}\ [m]}$$

where m_1, m_2 and m are of the form n

we have to prove that if $\langle\ e_1\ /\ e_2 \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}$.

Assuming that the semantic evaluation has access to the moduli n of the modular type of which value v_2 is representation, two semantic rules can be applied for each operator, one in the particular case when the division is not coprime with the moduli; the other in the general case:

- If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to v_2$ terminate, with $\gcd(v_2, n) \neq 1$, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $\epsilon \in \mathbf{V}$ by semantic Div-Mod.
- If $\langle\ e_1 \mid \rho\ \rangle \to v_1$ and $\langle\ e_2 \mid \rho\ \rangle \to v_2$ terminate, with $\gcd(v_2, n) = 1$, then $\langle\ e_1/e_2 \mid \rho\ \rangle$ evaluates to $[\![/]\!][v_1, v_2]$ by semantic rule Op. Here $[\![/]\!]$ gives the interpretation of the / operator with respect to the values v_1 and v_2. By induction hypothesis, v_1 and v_2 are in the semantic domain $\mathbf{V}\backslash\mathcal{E}$, corresponding to representations of the same modular type. Since division is well-defined for modular values with the same moduli, then $[\![/]\!][v_1, v_2]$ evaluates to another value v which is again a representation of a modular value with the same moduli, and hence $v \in \mathbf{V}\backslash\mathcal{E}$.

Boolean operations. The semantic rule applicable to all Boolean operation is the Op rule, as it is the case of arithmetic operations (except in the particular case of division by

zero). Since all boolean operations are total and well-defined for the representation of values of types specified in the typing rules, the soundness of the system for these operations is ensured. The proof is similar to the arithmetic operators cases.

Bit string operations. The same reasoning about arithmetic and Boolean operators applies to bit string operators, since they all evaluate through the Op rule. Thus, we only have to provide the proof that bit string accesses are safe. For the rule

$$\frac{\Gamma \vdash e_1 :: \mathsf{Bits}[n] \quad \Gamma \vdash e_2 \leq \mathsf{Int}}{\Gamma \vdash e_1[e_2] :: \mathsf{Bits}[1]}$$

we have to prove that if $\langle\ e_1[e_2] \mid \rho\ \rangle \to v$ terminates then $v \in \mathbf{V}$.

In bit string accesses, two semantic rules are applicable, VSel and VSel-Error, depending on whether the access is within or outside the bounds:

- If $\langle\ e_1 \mid \rho\ \rangle \to [r_0, \ldots, r_{n-1}]$ and $\langle\ e_2 \mid \rho\ \rangle \to i$ terminate, for $0 \leq i < n$, then $\langle\ e_1[e_2] \mid \rho\ \rangle$ evaluates to $\mathsf{at}(i, [r_0, \ldots, r_{n-1}])$ by semantic rule VSel, where $[r_0, \ldots, r_{n-1}]$ represents a bit string with $r_0 \in \mathbf{V}\backslash\mathcal{E}, \ldots r_{n-1} \in \mathbf{V}\backslash\mathcal{E}$, and $i \in \mathbf{V}\backslash\mathcal{E}$ represents an integer, both by induction hypothesis. This case follows from the observation that the at function only fails if the integer index is outside the range of positions of the argument sequence and by hypothesis this cannot happen. Therefore, the returned value always belongs to $\mathbf{V}\backslash\mathcal{E}$.
- If $\langle\ e_1 \mid \rho\ \rangle \to [r_0, \ldots, r_{n-1}]$ and $\langle\ e_2 \mid \rho\ \rangle \to i$ terminate, whenever $i < 0 \vee i \leq n$, by VSel-Error, $\langle\ e_1[e_2] \mid \rho\ \rangle$ evaluates to ϵ which belongs to the semantic domain and hence this case also holds.

The case when a range of bit string values is selected is reduced to the previous case, with the difference that we want to prove that no *trapped* **or** *untrapped* error can occur. We can look at the range as a sequence of individual element accesses together with a concatenation of results. The concatenation operation has been proved to maintain the soundness of the system. Since accesses are sequential, the potential problems can occur in the extrema, meaning that if these accesses are within bounds, all the other accesses are within bounds. Considering the side condition of the typing rule, the only possible applicable semantic rule is VSel, meaning that all accesses return semantic values different from ϵ. Thus, bit string range access maintain the soundness of the system.

Vector operations. Operations over vectors are similar to operations over bit strings, the differences being: vector concatenation includes a possible coercion; individual accesses in vectors return values of the container type, not a vector with a single element. Despite this differences, the soundness proofs are quite similar, thus they will be omitted.

Matrix operations. General arithmetic operations over matrices were covered in the arithmetic section. The remainder cases of operations over matrices are accesses, which allow for all possible combinations of individual and range accesses in the two dimensions of matrices. Since these situation can be directly reduced to two separate cases in just one dimension, and this can be mapped to the vector case, the proof for matrices can be safely reduced to the proof for vectors.

Casts. For the rule,

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e \leq \tau' \quad \vdash_c \tau' \Rightarrow \tau}{\Gamma, \Delta \vdash (t)\ e :: \tau}$$

we have to prove that if $\langle\ (t)\ e \mid \rho\ \rangle \rightarrow v$ terminates then $v \in \mathbf{V} \backslash \mathcal{E}$.

By the semantic definition of the cast judgment, $\vdash_c \tau' \Rightarrow \tau$ denotes a conversion function from type τ' to type τ, *conversion* $_{\tau' to \tau}$. By induction hypothesis, $\langle\ e \mid \rho\ \rangle$ corresponds to a value representation of the type τ', belonging to $\mathbf{V} \backslash \mathcal{E}$. This means that $\langle\ (t)\ e \mid \rho\ \rangle$ is the application of the conversion function to the evaluated value, i.e., *conversion* $_{\tau' to \tau}(v) = v'$. Because this function is total, it maps all values that are representations of τ' to representations of values of τ, meaning that v' is also in $\mathbf{V} \backslash \mathcal{E}$ and that v' is a valid encoding of the type τ.

Error propagation. All the previous cases assumed, by induction hypothesis, that evaluation of sub-expressions returns values belonging to $\mathbf{V} \backslash \mathcal{E}$. In the case when the semantic evaluation of any sub-expression returns ϵ, we implicitly consider that the overall semantic rule also returns ϵ. Since $\epsilon \in \mathbf{V}$ these cases maintain the soundness of the system.

C.2 Statements

The type soundness theorem for statements establishes that the evaluation of a well-typed statements in CAO, either terminates normally in a new valid state, raises a trapped error or runs forever. Furthermore, only out-of-bounds assignments can raise trapped errors; all other cases either return a value in a new valid state, return a no-result identifier in a new valid state, or run forever.

Proving soundness for statements, requires not only showing that returned values are not untrapped errors, but also that the consistency of the state (frame) is not violated through the application of semantic rules. In order to establish the main theorem, we first prove some auxiliary lemmas regarding the state manipulation by statement evaluation. The most important of these lemmas establishes that the evaluation of blocks of statements, due to the scoping mechanism in frames, can only lead to a frame that differs from the initial one in the update of possibly some variables in its global part. This lemma will be useful to establish consistency whenever the evaluation of blocks of statements is needed.

Lemma 7. *The evaluation of a statement respects the invariants for frames:*

- *Access and updates operations take the first store they encounter that defines the variable in question;*
- *New variables are created in the outermost store;*
- *Frames are stacks whose invariants must be met.*

Proof. By direct inspection of the semantic rules for statement evaluation and because operators of Table 5, by definition, respect these invariants.

Lemma 8. *The evaluation of a statement does not change the number of levels of the state (frame). Formally, given a frame ρ and a statement c, if $\langle\ c \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ terminates and $r \neq \epsilon$ then $length(\rho) = length(\rho')$.*

Proof. By induction on derivation of the statement evaluation. The base cases correspond to the rules that do not add or remove stores from the input frame. The induction steps are for rules AssignProc, Proc, StmtBlock, SeqRet and Seq.

Lemma 9. *If the evaluation of a statement requires adding a new local store to the initial frame, and the evaluation terminates and does not fail, then*

1. *The local store is discarded in the end;*
2. *The returned frame only differs from the initial one in updated variable values.*

Proof. 1. By Lemma 8 we know that the evaluation of a statement does not change the number of levels of frame. By adding a new local store, we increase the number of levels of the store, requiring that, in the end, some store has to be discarded. However, by Lemma 7, we know that operations in frames respect the invariants of a stack, meaning that the only possible operation to discard a store from the frame is by removing the outermost store, i.e., the local store.

2. By Lemma 7, new variables are created in the outermost store which will be discarded. Again by Lemma 7, updates of a variable occur in the first store that defines the variable. Thus, only updates of variables will change the non-local store. This can be guaranteed because all rules respect the invariants of frames (7), meaning that it is not possible to reorder stores in a frame and that the creation and deletions follow the stack invariant.

Lemma 10. *Given a typing environment Γ and an evaluation environment ρ which are consistent, i.e., $\vdash \rho :: \Gamma$, and a block statement $\{c_1; \ldots; c_n\}$, if $\Gamma \vdash c_1; \ldots; c_n :: (\tau, \Gamma')$ and $\langle\ \{c_1; \ldots; c_n\} \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ terminates with $r \neq \epsilon$ then $\vdash \rho' :: \Gamma$.*

Proof. According with the semantics, the evaluation of a block of statements is performed using the semantic rule StmtBlock. This means that $\langle\ \{c_1; \ldots; c_n\} \mid \rho\ \rangle$ evaluates to $\langle\ r\ ,\ \mathrm{pop}(\rho'')\ \rangle$ if $\langle\ c_1; \ldots; c_n \mid \mathrm{push}(\circ, \rho)\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle$. In the case that this evaluation does not terminate or $r = \epsilon$ the hypothesis fails and the lemma trivially holds. Otherwise, this means that frame $\rho' = \mathrm{pop}(\rho'')$ results from the evaluation of the statement block using an empty local store added to the frame ρ using function push. At the end of the evaluation, function pop is used to remove the local store (with local variables) from the returned store ρ''. This means that, by Lemma 9, ρ' can only differ from ρ in the update of variables already defined in ρ. Since by Lemma 3 updates do not alter consistency, then $\vdash \rho' :: \Gamma$.

We should notice that this lemma does not rely in an inductive reasoning on typing derivations, since we are not proving anything about the resulting typing environment Γ' which is ignored for proof purposes. The proof only establishes the consistency of the final state with the initial typing environment, and this results from adding a local store to the current frame in beginning of the evaluation and removing it from the current store in the end of the evaluation.

Theorem 3. *Let Γ be a typing environment and ρ a consistent evaluation environment such that $\vdash \rho :: \Gamma$. For any typing environment Γ, evaluation environments ρ and ρ', statement c, semantic value v and type τ, if $\Gamma \Vdash c :: (\tau, \Gamma')$ and $\langle\ c \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then $v = \bullet$ (no value) or $v \in \boldsymbol{V}$, and $\vdash \rho' :: \Gamma'$. Furthermore, v can only be an untrapped error (ϵ) if it is originated by out-of-bounds assignments; all other evaluations result in value $v \in \boldsymbol{V} \backslash \mathcal{E}$.*

Proof. Uninitialized local variable. For the rule,

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x : t :: (\bullet, \mathsf{Pure}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma)$$

we have to prove that if $\langle\ \mathsf{def}\ x : t \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V} \backslash \mathcal{E}$;
2. $\vdash \rho' :: \Gamma[x :: \tau]$.

By semantic rule LocalVar-Un, $\langle\ \mathsf{def}\ x : t \mid \rho\ \rangle$ evaluates to $\langle\ \bullet\ ,\ \rho[x := d^\tau]\ \rangle$, The first case, is trivial. For the second case, d^τ is a representation of a value of type τ, and by induction hypothesis, $\vdash \rho :: \Gamma$. Thus, by Lemma (2), $\vdash \rho[x := d^\tau] :: \Gamma[x :: \tau]$.

The proof for the rule

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x_1, \ldots, x_n : t :: (\bullet, \mathsf{Pure}, \Gamma[x_1 :: \tau, \ldots, x_n :: \tau])}$$

where $x_1, \ldots, x_n \notin dom(\Gamma), x_i \neq x_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

follows by iterating the reasoning used in the previous case. Since all added variables are different and do not exist in the environment, consistency of environments holds.

Initialized local variable. For the rule,

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e \leq (\tau, \mathsf{cc})}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x : t := e :: (\bullet, \mathsf{cc}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma)$$

we have to prove that if $\langle\ \mathsf{def}\ x : t := e \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V} \backslash \mathcal{E}$;
2. $\vdash \rho' :: \Gamma[x :: \tau]$.

If $\langle\ e \mid \rho\ \rangle \rightarrow v$ terminates then $\langle\ \mathsf{def}\ x : t := e \mid \rho\ \rangle$ evaluates to $\langle\ \bullet\ ,\ \rho[x := v]\ \rangle$, by semantic rule LocalVar. The first case is trivial. For the second case, by induction hypothesis $\vdash \rho :: \Gamma$ and $v \in \mathbf{V} \backslash \mathcal{E}$ is a value of type τ. Thus, by Lemma (2), $\vdash \rho[x := v] :: \Gamma[x :: \tau]$.

Proofs for rules

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau \quad \Gamma, \Delta \vdash e_1 \leq (\tau, \mathsf{cc}_1) \ldots \Gamma, \Delta \vdash e_n \leq (\tau, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_\rho \mathsf{def}\ x : t := \{e_1, \ldots, e_n\} :: (\bullet, \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma[x :: \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau])}$$

where $x \notin dom(\Gamma)$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Matrix}\ [i, j]\ \mathsf{of}\ \alpha \quad \Gamma, \Delta \vdash e_1 \leq (\alpha, \mathsf{cc}_1) \ldots \Gamma, \Delta \vdash e_n \leq (\alpha, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_\alpha \mathsf{def}\ x : t := \{e_1, \ldots, e_n\} :: (\bullet, \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma[x :: \mathsf{Matrix}\ [i, j]\ \mathsf{of}\ \alpha])}$$

where $\alpha \in \mathcal{A}, x \notin dom(\Gamma), i \times j = n$

follow the same principle as the previous one. The differences are the multiple expression evaluation and the particular representation for vectors and matrices.

Assign. For the rule,

$$\frac{\begin{array}{ccc} \Gamma, \Delta \vdash l_1 :: (\tau_1, \mathsf{cl}_1) & \ldots & \Gamma, \Delta \vdash l_n :: (\tau_n, \mathsf{cl}_n) \\ \Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) & \ldots & \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n) \end{array}}{\Gamma, \Delta \Vdash_\tau l_1, \ldots, l_n := e_1, \ldots, e_n :: (\bullet, \max(\mathsf{cl}_1 \ldots, \mathsf{cl}_n, \mathsf{c}_1, \ldots, \mathsf{c}_n), \Gamma)}$$

we have to prove that if $\langle\ l_1, \ldots, l_n := e_1, \ldots, e_n \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}$;
2. $\vdash \rho' :: \Gamma$.

In variable assignment, two semantic rules are applicable, Assign and Assign-Err, depending on whether the access of vectors, matrices or bit strings is within or outside the bounds. This validity of the access is given as result by update function frLVUpd.

If $\langle\ e_1 \mid \rho\ \rangle \rightarrow r_1, \ldots, \langle\ e_n \mid \rho\ \rangle \rightarrow r_n$ terminate then $r_1, \ldots, r_n \in \mathbf{V} \backslash \mathcal{E}$, where $r_1, \ldots, r_n$ correspond to values of types $\tau_1, \ldots, \tau_n$. The semantic rule to apply (and consequently the final state of the evaluation ρ') depends on the result of the left value update function frLUpd in its iterated and more friendly syntax. Case $\rho[r_1, \ldots, r_n / l_1, \ldots, l_n]$

- Returns ϵ, then semantic rule Assign-Err is applied, and $\langle\ l_1, \ldots, l_n := e_1, \ldots, e_n \mid \rho\ \rangle$ is evaluated to $\langle\ \epsilon\ ,\ \_\ \rangle$. This case holds since $\epsilon \in \mathbf{V}$ and the evaluation environment is trivially consistent (after an error the state is ignored; we can consider the empty evaluation environment which is consistent with any typing environment).
- Returns an updated state ρ', then semantic rule Assign is applied, and $\langle\ l_1, \ldots, l_n := e_1, \ldots, e_n \mid \rho\ \rangle$ is evaluated to $\langle\ \bullet\ ,\ \rho[r_1, \ldots, r_n / l_1, \ldots, l_n]\ \rangle$.
  The first proof obligation is trivial since $v = \bullet$. The second proof obligation requires proving that $\vdash \rho[r_1, \ldots, r_n / l_1, \ldots, l_n] :: \Gamma$. By induction hypothesis, $\vdash \rho :: \Gamma$ and $r_1, \ldots, r_n \in \mathbf{V} \backslash \mathcal{E}$, where $r_1, \ldots, r_n$ correspond to values of types $\tau_1, \ldots, \tau_n$. Thus, by Lemma 3, the update of a left value l_i for a value r_i, for $0 < i \leq n$ does not affect the consistency of the environment, meaning that $\vdash \rho[r_1, \ldots, r_n / l_1, \ldots, l_n] :: \Gamma$ holds.

Assign tuple. For the rule

$$\frac{\Gamma, \Delta \vdash l_1 :: (\tau_1, \mathsf{cl}_1) \quad \ldots \quad \Gamma, \Delta \vdash l_n :: (\tau_n, \mathsf{cl}_n) \quad \Gamma, \Delta \vdash e \leq ((\tau_1, \ldots, \tau_n), \mathsf{c})}{\Gamma, \Delta \Vdash_\tau l_1, \ldots, l_n := e :: (\bullet, \max(\mathsf{cl}_1, \ldots, \mathsf{cl}_n, \mathsf{c}), \Gamma)}$$

the proof is identical to the simple assignment case. The difference is that instead of having n evaluations of expressions, we have a single evaluation which returns a sequence with n values, and we apply semantic rules AssignTupple and AssignTupple-Err.

Procedure. For the rule

$$\frac{\begin{array}{c}\Gamma_G(fp) = ((\tau_1, \ldots, \tau_n) \to (), \mathsf{Procedure}) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau fp(e_1, \ldots, e_n) :: (\bullet, \mathsf{Procedure}, \Gamma_G, \Gamma_L)} \quad fp \in dom(\Gamma_G)$$

we have to prove that if $\langle\ fp(e_1, \ldots, e_n) \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V} \backslash \mathcal{E}$;
2. $\vdash \rho' :: \Gamma$.

Since the procedure *fp* is defined in the typing environment, function lookupFun(fp) will not fail to find its definition. If

$$\langle\ e_1 \mid \rho\ \rangle \to r_1, \ldots, \langle\ e_n \mid \rho\ \rangle \to r_n$$

and

$$\langle\ body \mid \mathrm{push}(\circ[p_1, \ldots, p_n := r_1, \ldots, r_n], \mathrm{global}(\rho))\ \rangle \Rightarrow \langle\ r\ ,\ \rho''\ \rangle$$

terminate, then, by Proc rule, $\langle\ fp(e_1, \ldots, e_n) \mid \rho\ \rangle$ evaluates to $\langle\ \bullet\ ,\ \mathrm{push}(\mathrm{local}(\rho), \mathrm{global}(\rho''))\ \rangle$. Since $v = \bullet$ the first claim holds.

In the second part, we have to prove that for $\rho' = \mathrm{push}(\mathrm{local}(\rho), \mathrm{global}(\rho''))$, then $\vdash \rho' :: \Gamma$. By induction hypothesis, we have that $\vdash \rho :: \Gamma$, which means, by Lemma 5, that $\vdash \mathrm{local}(\rho) :: \Gamma$ and $\vdash \mathrm{global}(\rho) :: \Gamma$. By Lemma 8, we now that frames $\mathrm{push}(\circ[p_1, \ldots, p_n := r_1, \ldots, r_n], \mathrm{global}(\rho))$ and ρ'' must have the same length (the same number of stores). Moreover, Lemmas 7 and 9 ensure that the evaluation of the body, even when other functions or procedures are called, will maintain the structure of the frame and discard possible local stores. Since $\circ[p_1, \ldots, p_n := r_1, \ldots, r_n]$ is the local store created to evaluate the body of the procedure, then $\mathrm{global}(\rho'')$ only differs from $\mathrm{global}(\rho)$ in the update of variables already defined in $\mathrm{global}(\rho)$. By Lemma 3, the consistency is not changed by updates, meaning that $\vdash \mathrm{global}(\rho'') :: \Gamma$. Hence, by Lemma 6, we conclude that $\vdash \rho' :: \Gamma$.

Assign procedure. For the rule

$$\frac{\begin{array}{c}\Gamma_G(fp) = ((\tau_1, \ldots, \tau_n) \to (\tau_{n+1}, \ldots, \tau_m), \mathsf{Procedure}) \\ \Gamma_G, \Gamma_L, \Delta \vdash l_{n+1} :: (\tau_{n+1}, \mathsf{cl}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash l_m :: (\tau_m, \mathsf{cl}_m) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau l_{n+1}, \ldots, l_m := fp(e_1, \ldots, e_n) :: (\bullet, \mathsf{Procedure}, \Gamma_G, \Gamma_L)} \quad fp \in dom(\Gamma_G)$$

we have to prove that if $\langle\ l_{n+1}, \ldots, l_m := fp(e_1, \ldots, e_n) \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V} \backslash \mathcal{E}$;
2. $\vdash \rho' :: \Gamma$.

The proof of this case follows by the combination of procedure call and assignment proofs.

RETURN. For rule,

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{cc}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{cc}_n)}{\Gamma, \Delta \Vdash_{(\tau_1,\ldots,\tau_n)} \mathsf{return}\ e_1, \ldots, e_n :: ((\tau_1, \ldots, \tau_n), \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma)}$$

we have to prove that if $\langle\ \mathsf{return}\ e_1, \ldots, e_n \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}\backslash\mathcal{E}$;
2. $\vdash \rho' :: \Gamma$.

By induction hypothesis, if evaluation terminates then $\langle\ e_1 \mid \rho\ \rangle \Rightarrow r_1, \ldots, \langle\ e_n \mid \rho\ \rangle \Rightarrow r_n$ and $r_1 \in \mathbf{V}\backslash\mathcal{E}, \ldots, r_n \in \mathbf{V}\backslash\mathcal{E}$. Moreover, $r_1, \ldots, r_n$ is, respectively, a value of type $\tau_1, \ldots, \tau_n$. Then, by semantic rule RETURN, $\langle\ \mathsf{return}\ e_1, \ldots, e_n \mid \rho\ \rangle$ evaluate to $\langle\ [r_1, \ldots, r_n]\ ,\ \rho\ \rangle$. Since sequences of value of $\mathbf{V}\backslash\mathcal{E}$ are also in semantic domain, then $[r_1, \ldots, r_n] \in \mathbf{V}\backslash\mathcal{E}$. The environment consistency trivially holds.

SEQUENCE OF STATEMENTS. For sequences of statements, two cases are possible: when a value is returned and when no value is returned.

For rule

$$\frac{\Gamma, \Delta \Vdash_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma', \Delta \Vdash_\tau c_2; \ldots; c_n :: (\rho, \mathsf{cc}_{2n}, \Gamma'')}{\Gamma, \Delta \Vdash_\tau c_1; \ldots; c_n :: (\rho, \max(\mathsf{cc}_1, \mathsf{cc}_{2n}), \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

we have to prove that if $\langle\ c_1; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}\backslash\mathcal{E}$;
2. $\vdash \rho' :: \Gamma''$

By induction hypothesis, $\Gamma \vdash c_1 :: (\bullet, \Gamma')$. Thus, the applicable semantic rule is STMT-SEQ. If $\langle\ c_1 \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho'\ \rangle$ and $\langle\ c_2; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho''\ \rangle$ terminates then $\langle\ c_1; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho''\ \rangle$. Both $v \in \mathbf{V}\backslash\mathcal{E}$ and $\vdash \rho'' :: \Gamma''$ holds by by induction hypothesis.

For rule

$$\frac{\Gamma, \Delta \Vdash_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma', \Delta \Vdash_\tau c_2; \ldots; c_n :: (\rho, \mathsf{cc}_{2n}, \Gamma'')}{\Gamma, \Delta \Vdash_\tau c_1; \ldots; c_n :: (\tau, \max(\mathsf{cc}_1, \mathsf{cc}_{2n}), \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

we have to prove that if $\langle\ c_1; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}\backslash\mathcal{E}$;
2. $\vdash \rho' :: \Gamma''$

By induction hypothesis, $\Gamma \vdash c_1 :: (\tau, \Gamma')$. Thus, the applicable semantic rule is STMTSEQRET. If $\langle\ c_1 \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_k]\ ,\ \rho'\ \rangle$ terminates then $\langle\ c_1; \ldots; c_n \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_k]\ ,\ \rho'\ \rangle$. By induction hypothesis $[r_1, \ldots, r_k] \in \mathbf{V}\backslash\mathcal{E}$ and $\vdash \rho' :: \Gamma'$. Since Γ'' extends Γ' ($dom(\Gamma') \subseteq dom(\Gamma'')$) without changing its definitions, then, as consequence of Lemma 1, ρ' is also consistent with Γ'', i.e., $\vdash \rho' :: \Gamma''$.

Conditional choice (if). For the rule

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\tau, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\tau, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

we have to prove that if $\langle\ \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}\backslash\mathcal{E}$;
2. $\vdash \rho' :: \Gamma$.

The evaluation rule which is applicable depends on the evaluation of the condition. Case $\langle\ b \mid \rho\ \rangle$ evaluates to

- Value 1 then rule IfElseTrue applies. If $\langle\ \{c_1\} \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ terminates, then $\langle\ \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} \mid \rho\ \rangle$ evaluates to $\langle\ r\ ,\ \rho'\ \rangle$. By induction hypothesis, the returned value r is in domain $\mathbf{V}\backslash\mathcal{E}$. Since by induction hypothesis $\vdash \rho :: \Gamma$ and $\{c_1\}$ is a block of statements, we can apply Lemma 10 to conclude that $\vdash \rho' :: \Gamma$.
- Value 0 then rule IfElseFalse applies If $\langle\ c_2 \mid \rho\ \rangle \Rightarrow \langle\ r\ ,\ \rho'\ \rangle$ terminates, then $\langle\ \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} \mid \rho\ \rangle$ evaluates to $\langle\ r\ ,\ \rho'\ \rangle$. By induction hypothesis, the returned value r is in domain $\mathbf{V}\backslash\mathcal{E}$. Since by induction hypothesis $\vdash \rho :: \Gamma$ and $\{c_2\}$ is a block of statements, we can apply Lemma 10 to conclude that $\vdash \rho' :: \Gamma$.

The other conditional choice rules

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\bullet, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\tau, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma, \Delta \Vdash_\tau c_2 :: (\bullet, \mathsf{cc}_2, \Gamma'')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}_1, \mathsf{cc}_2), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma')}{\Gamma, \Delta \Vdash_\tau \mathsf{if}\ b\ \{c\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}), \Gamma)} \quad \rho \in \{\tau, \bullet\}$$

have a similar proof, just differing in the way $\bullet$ and returned values are combined.

Iteration (while). For rule

$$\frac{\Gamma, \Delta \vdash b \leq (\mathsf{Bool}, \mathsf{cb}) \quad \Gamma, \Delta \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma')}{\Gamma, \Delta \Vdash_\tau \mathsf{while}\ b\ \{c\} :: (\bullet, \max(\mathsf{cb}, \mathsf{cc}), \Gamma)} \quad \rho \in \{\tau, \bullet\}$$

we have to prove that if $\langle\ \mathsf{while}\ b\ \{c\} \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}\backslash\mathcal{E}$;
2. $\vdash \rho' :: \Gamma$

The evaluation rule which is applicable depends on the evaluation of the condition. Case $\langle\ b \mid \rho\ \rangle$ evaluates to

– Value 1, then two cases are possible, depending if a value was returned. Case $\langle\ \{c\} \mid \rho\ \rangle$ terminates and evaluates to
  - Result $\langle\ \bullet\ ,\ \rho'\ \rangle$, then rule WhileTrue applies. If $\langle\ \mathsf{while}\ b\ \{c\} \mid \rho'\ \rangle \Rightarrow \langle\ v\ ,\ \rho''\ \rangle$ terminates, then $\langle\ \mathsf{while}\ b\ \{c\} \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho''\ \rangle$. By induction hypothesis $v \in \mathbf{V}\backslash\mathcal{E}$. Since by induction hypothesis $\vdash \rho :: \Gamma$ and $\{c\}$ is block of statements, it follows from Lemma 10 that $\vdash \rho' :: \Gamma$. Therefore, by induction hypothesis, $\vdash \rho'' :: \Gamma$ for the case of the recursive invocation of the rule starting with environment ρ'.
  - Result $\langle\ [r_1, \ldots, r_n]\ ,\ \rho'\ \rangle$, then rule WhileTrueRes applies, meaning that $\langle\ \mathsf{while}\ b\ \{c\} \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \rho'\ \rangle$. By induction hypothesis $[r_1, \ldots, r_n] \in \mathbf{V}\backslash\mathcal{E}$. Since $\vdash \rho :: \Gamma$, by induction hypothesis, and $\{c\}$ is a block of statements, then, by Lemma 10, $\vdash \rho' :: \Gamma$.
– Value 0 then rule WhileFalse applies: $\langle\ \mathsf{while}\ b\ \{c\} \mid \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \rho\ \rangle$. This case trivially holds.

Sequence. For rule

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \\ \forall_{n \in \{i, i+k, \ldots, j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)\end{array}}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$
$$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i \leq j,\ k \geq 1$$

we have to prove that if $\langle\ \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} \mid \rho\ \rangle \Rightarrow \langle\ v\ ,\ \rho'\ \rangle$ terminates then

1. $v = \bullet$ or $v \in \mathbf{V}\backslash\mathcal{E}$;
2. $\vdash \rho' :: \Gamma$.

If $\langle\ e_1 \mid \rho\ \rangle \to s_1$, $\langle\ e_2 \mid \rho\ \rangle \to s_2$ and $\langle\ e_3 \mid \rho\ \rangle \to s_3$ terminate, then we have two possible cases in the evaluation of $\langle\ c \mid \mathrm{push}(\circ[x := n], \mathrm{pop}(st_n))\ \rangle$:

– For some $n \in \{s_1, s_1 + s_3, \ldots, s_2\}$ it evaluates to $\langle\ [r_1, \ldots, r_n]\ ,\ \rho''\ \rangle$ then rule SeqRet applies. This means that $\langle\ \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} \mid \rho\ \rangle \Rightarrow \langle\ [r_1, \ldots, r_n]\ ,\ \mathrm{pop}(\rho'')\ \rangle$. By induction hypothesis $[r_1, \ldots, r_n] \in \mathbf{V}\backslash\mathcal{E}$.
  To prove that $\rho' = \mathrm{pop}(\rho'')$ is consistent with typing environment Γ, we have to make an inductive argument about each iteration. The sequence operation is, in fact, an indexed expansion of the block of statements evaluation. The base case occurs when $n = s_1$ meaning just one expansion, starting with frame $\rho_{s_1} = \mathrm{push}(\circ, \rho)$. Since by induction hypothesis (on type derivations), $\vdash \rho :: \Gamma$ and therefore $\vdash \rho_{s_1} :: \Gamma$, this means that the evaluation of $\langle\ c \mid \mathrm{push}(\circ[x := s_1], \mathrm{pop}(\rho_{s_1}))\ \rangle$ starts in a consistent environment because $\vdash \mathrm{push}(\circ[x := s_1], \mathrm{pop}(\rho_{s_1})) :: \Gamma[x :: \mathsf{Int}]$. Since statement c is a block of statements, by Lemma 10, we conclude that $\vdash \rho'' :: \Gamma[x :: \mathsf{Int}]$. Lemma 7 ensures that $\mathrm{pop}(\rho'') = \rho'$ just removes the store $\circ[x := s_1]$ from the top of ρ'', and thus $\vdash \rho' :: \Gamma$. This holds because the side condition of typing ensures that variable x was not in the domain of Γ, and assumed consistency means that it could not also be in the domain of ρ.

The induction step (on sequence indexes) takes as hypothesis that evaluation for $n-1 = s_n - s_3$ returns a frame $\rho_{(s_n-s_3)+s_3} = \rho_{s_n}$, which is consistent with typing environment $\Gamma[x :: Int]$. By an argument similar to the one used above, we can conclude that the evaluation of the block of statements starts with frame $\text{pop}(\rho_{s_n})$ extended with the index variable and that this is consistent with the typing environment. Again, removing the added index variable at the end, will maintain the consistency of the environment. Therefore, by induction on indexes of sequences, we can conclude that $\vdash \rho' :: \Gamma$.

– Otherwise, it evaluates to $\langle\ \bullet\ ,\ \rho_{n+s_3}\ \rangle$ and rule SEQ applies. This means that $\langle\ \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \}\ |\ \rho\ \rangle \Rightarrow \langle\ \bullet\ ,\ \text{pop}(\rho_{s_2})\ \rangle$. Since $v = \bullet$ the first part is trivial. The justification for $\vdash \text{pop}(\rho_{s_2}) :: \Gamma$ is similar to the above case, differing just in the fact that no value is returned.

The other sequence statement rules

$$\frac{\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \quad \forall_{n\in\{i,i-k,\ldots,j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$

$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i > j,\ k \geq 1$

$$\frac{\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \forall_{n\in\{i,i+1,\ldots,j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$

$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i \leq j$

$$\frac{\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \forall_{n\in\{i,i-1,\ldots,j\}} \Gamma_G, \Gamma_L[x :: \mathsf{Int}], \Delta[x := n] \Vdash_\tau c :: (\rho, \mathsf{cc}, \Gamma'_G, \Gamma'_L)}{\Gamma_G, \Gamma_L, \Delta \Vdash_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet, \mathsf{cc}, \Gamma_G, \Gamma_L)}$$

$\rho \in \{\tau, \bullet\},\ x \notin dom(\Gamma_L),\ i > j$

have a similar proof, just differing in the order and amount of evaluation.

ERROR PROPAGATION. Such as in the case of expression evaluation, it is assumed by induction hypothesis that the evaluation of sub-expressions and sub-statements always return values belonging to $\mathbf{V}\backslash\mathcal{E}$. Again, we implicitly consider that every time such an evaluation returns ϵ, the overall semantic rule also returns ϵ. Since $\epsilon \in \mathbf{V}$, error propagation maintains the soundness of the system.

C.3 Programs

DECLARATIONS. The following rules

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n \quad \Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma_G, \circ[x_1 :: \tau_1, \ldots, x_n :: \tau_n], \Delta \Vdash_\tau c :: (\tau, \mathsf{cc}, \Gamma'_G)}{\Gamma_G, \circ, \Delta \Vdash \mathsf{def}\ fp(x_1 : t_1, \ldots, x_n : t_n) : t\ \{c\} :: (\bullet, \Gamma_G[fp :: ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{cc})])}$$

where $t \neq \mathsf{void}$ and $fp \notin dom(\Gamma_G)$

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n \qquad \Gamma_G, \circ[x_1 :: \tau_1, \ldots, x_n :: \tau_n], \Delta \Vdash_{()} c; \mathsf{return}() :: ((), \mathsf{Procedure}, \Gamma'_G)}{\Gamma_G, \circ, \Delta \Vdash \mathsf{def}\ fp(x_1 : t_1, \ldots, x_n : t_n) : \mathsf{void}\ \{c\} :: (\bullet, \Gamma_G[fp :: ((\tau_1, \ldots, \tau_n) \to (), \mathsf{Procedure})])}$$

where $fp \notin dom(\Gamma_G)$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \Vdash \mathsf{typedef}\ tid\ :=\ t :: (\bullet, \Gamma)}$$

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n}{\Gamma_G, \Gamma_L, \Delta \Vdash \mathsf{typedef}\ sid := \mathsf{struct}[fi_1 : t_1; \ldots; fi_n : t_n] :: (\bullet, \Gamma_G[fi_1 :: (sid \to \tau_1, \mathsf{Pure}), \ldots, fi_n :: (sid \to \tau_n, \mathsf{Pure})], \Gamma_L)}$$

where $sid, fi_1, \ldots, fi_n \notin dom(\Gamma)$ where $fi_i \neq fi_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

do not have a directly associated semantic. However they are critical to ensure that the hypothesis we have appealed to in the above discussion regarding the validity of the results returned by the lookupType and lookupFun functions. Indeed, these rules in combination with expression and statement type checking rules guarantee that function and type look-up during evaluation will never fail. The same applies to struct projections.

Programs. The main theorem for CAO programs states that the evaluation of a program can return a consistent evaluation environment, return an error or fail to terminate. It should be noticed that an entry point (which we called main) must exist in order to perform the evaluation.

Theorem 4. *Given a CAO program p if $\circ, \circ, \circ \vdash p :: (\bullet, \Gamma_G)$ and $\langle\ p \mid \circ\ \rangle \Rrightarrow \langle\ \rho\ \rangle$ terminates, then $\vdash \rho :: \Gamma_G$ or ρ is an error state.*

Proof. The soundness proof for CAO programs takes global variable declarations as base cases, in which the initial state of the program is set. The inductive step is the program evaluation rule, which collects all these declarations building the initial state which is used in the evaluation of the main procedure.

Global variables. For rule,

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma_G, \Gamma_L, \Delta \vdash e \leq (\tau, \mathsf{cc})}{\Gamma_G, \Gamma_L, \Delta \Vdash_\rho \mathsf{def}\ x : t := e :: (\bullet, \mathsf{cc}, \Gamma_G[x :: \tau], \Gamma_L)} \quad x \notin dom(\Gamma_G)$$

we have to prove that if $\langle\ \mathsf{def}\ x : t := e \mid \rho\ \rangle \Rrightarrow \langle\ \rho'\ \rangle$ terminates then $\vdash \rho' :: \Gamma_G[x :: \tau], \Gamma_L$.

The proof is similar to the local variable case, except that the semantic rule Global-Var is used, no return value is considered, and one now uses the global typing environment instead of the local typing environment.

Programs.. For the rule,

$$\frac{\circ, \circ, \circ \Vdash d_1 :: (\bullet, \Gamma_{G_1}) \quad \ldots \quad \Gamma_{G_{n-1}}, \circ, \circ \Vdash d_n :: (\bullet, \Gamma_G)}{\circ, \circ, \circ \Vdash d_1; \ldots; d_n :: (\bullet, \Gamma_G)} \quad \mathsf{main} :: () \to () \in \Gamma_G$$

we have to prove that if $\langle\ d_1; \ldots; d_n \mid \rho\ \rangle \Rightarrow \langle\ \rho'\ \rangle$ terminates, then $\vdash \rho' :: \Gamma_G$ or ρ' is an error state.

According with the semantics, programs are evaluated using the Program semantic rule. Both evaluation and typing start with an empty environment, meaning that the evaluation environment is clearly consistent. By induction hypothesis, each declaration maintains consistency leading to the conclusion that $\vdash \rho :: \Gamma_G$ where ρ is the environment obtained after evaluating all global declarations. By the properties of type checking the lookup of function main always finds it definition. If $\langle\ \mathit{body} \mid \mathrm{push}(\circ, \rho) \Rightarrow \langle\ v\ ,\ \rho''\ \rangle\ \rangle$ terminates, by Theorem 3, then case v is equal to

- Error value ϵ then the state ρ'' is an error state and this case holds.
- No result $\bullet$ or a value in $\mathbf{V} \backslash \mathcal{E}$, then ρ'' is valid state. Since the body of the main procedure is evaluated by extending global frame ρ with an empty local store, then $\mathrm{global}(\rho'')$ only differs from ρ in the update of variables already defined in ρ (program global variables). By Lemma 3 the consistency is not changed by updates, which means that $\vdash \mathrm{global}(\rho'') :: \Gamma_G$.