Mechanized Proofs of Adversarial Complexity and Application to Universal Composability

Manuel Barbosa
University of Porto (FCUP) and INESC TEC
mbb@fc.up.pt

Gilles Barthe
MPI-SP and IMDEA Software Institute
gjbarthe@gmail.com

Benjamin Grégoire
INRIA Sophia Antipolis
benjamin.gregoire@inria.fr

Adrien Koutsos
INRIA Paris
adrien.koutsos@inria.fr

Pierre-Yves Strub
Ecole Polytechnique
pierre-yves@strub.nu

ABSTRACT

EasyCrypt is a proof assistant used for verifying computational security proofs of cryptographic constructions. It has been applied to several prominent examples, including the SHA3 standard and a critical component of AWS Key Management Services.

In this paper we enhance the EasyCrypt proof assistant to reason about computational complexity of adversaries. The key technical tool is a Hoare logic for reasoning about computational complexity (execution time and oracle calls) of adversarial computations. Our Hoare logic is built on top of the module system used by EasyCrypt for modeling adversaries. We prove that our logic is sound w.r.t. the semantics of EasyCrypt programs — we also provide full semantics for the EasyCrypt module system, which was previously lacking.

We showcase (for the first time in EasyCrypt and in other computer-aided cryptographic tools) how our approach can express precise relationships between the probability of adversarial success and their execution time. In particular, we can quantify existentially over adversaries in a complexity class, and express general composition statements in simulation-based frameworks. Moreover, such statements can be composed to derive standard concrete security bounds for cryptographic constructions whose security is proved in a modular way. As a main benefit of our approach, we revisit security proofs of some well-known cryptographic constructions and we present a new formalization of Universal Composability (UC).

KEYWORDS

Verification of Cryptographic Primitives, Formal Methods, Interactive Proof System, Complexity Analysis

1 INTRODUCTION

Cryptographic designs are typically supported by mathematical proofs of security. Unfortunately, these proofs are error-prone and subtle flaws can go unnoticed for many years, in spite of careful and extensive scrutiny from experts. Therefore, it is desirable that cryptographic proofs are formally verified using computer-aided tools [20]. Over the last decades, many formalisms and tools have been developed for mechanizing cryptographic proofs [4]. In this paper we focus on the EasyCrypt proof assistant [6, 8], which has been used to prove security of a diverse set of cryptographic constructions in the computational model of cryptography [1, 2]. In this setting, cryptographic designs and their corresponding security notions are modeled as probabilistic programs. Moreover, security proofs provide an upper bound on the probability that an adversary breaks a cryptographic design, often assuming that the attacker has limited resources that are insufficient to solve a mathematical problem. While EasyCrypt excels at quantifying the probability of adversarial success, it lacks support for keeping track of the complexity of adversarial computations. This is a limitation that is common to other tools in computer-aided cryptography, and it means that manual inspection is required to check that the formalized claims refer to probabilistic programs that fall in the correct complexity classes. While this may be acceptable for very simple constructions, for more intricate proofs it may be difficult to interpret what a proved claim means in the cryptographic sense; in particular, existing computer-aided tools cannot fully express the subtleties that arise in compositional approaches such as Universal Composability [15]. This is an important limitation, as these compositional approaches are ideally suited for proving security of complex cryptographic designs that involve multiple-layers of simpler building blocks. This work overcomes this limitation and showcases the benefits of reasoning about computational complexity in EasyCrypt.

Contributions

Our work makes three broad contributions.

Formal verification of complexity statements. We define a formal system for specifying and proving complexity claims. Our formal system is based on an expressive module system, which enriches the existing EasyCrypt module system with declarations of memory footprints (specifying what is read and written) and cost (specifying which oracles can be called and how often). This richer module system is the basis for modeling the cost of a program as a tuple. The first component of the tuple represents the intrinsic cost of the program, i.e. its cost in a model where oracle and adversary calls are free. The remaining components of the tuple represent the number of calls to oracles and adversaries. This style of modeling is compatible with cryptographic practice and supports reasoning compositionally about (open) programs.

Our formal system is built on top of the module system and takes the form of a Hoare logic for proving complexity claims that upper bound the cost of expressions and commands. Furthermore, an embedding of the formal system into a higher-order logic provides support reductionist statements relating adversarial advantage and execution cost, for instance:

$$\forall \mathcal{A}.\exists \mathcal{B}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{B}) + \epsilon\ \land\ \mathsf{cost}(\mathcal{B}) \leq \mathsf{cost}(\mathcal{A}) + \delta$$

where typically ϵ and δ are polynomial expressions in the number of oracle calls. The above statement says that every adversary $\mathcal{A}$ can be turned into an adversary $\mathcal{B}$, with sensibly equivalent resources, such that the advantage of $\mathcal{A}$ against a cryptographic scheme $\mathcal{S}$ is upper bounded by the advantage of $\mathcal{B}$ against a hardness assumption $\mathcal{H}$. Note that the statement is only meaningful because the cost of $\mathcal{B}$ is conditioned on the cost of $\mathcal{A}$, as the advantage of an unbounded adversary is typically 1. The ability to prove and instantiate such $\forall\exists$-statements is essential for capturing compositional reasoning principles.

We show correctness of our formal system w.r.t. an interpretation of programs. Our interpretation provides the first complete semantics for the EasyCrypt module system, which was previously lacking. This semantics is of independent interest and could be used to prove soundness of the two program logics supported by EasyCrypt: a Relational Hoare Logic [9] and a Union Bound logic [7].

Implementation in the EasyCrypt proof assistant. We have implemented our formal system as an extension to the EasyCrypt proof assistant, which provides mechanisms for declaring the cost of operators and for helping users derive the cost of expressions and programs. Our implementation brings several contributions of independent interest, including an improvement of the memory restriction system of EasyCrypt, and a library and automation support to reason about extended integers that are used for reasoning about cost. For the latter we follow [24] and reduce formulae about extended integers to integer formulae that can be sent to SMT solvers. Another key step is to embed our Hoare logic for cost into the ambient higher-order logic—similar to what is done for the other program logics of EasyCrypt. This allows us to combine judgments from different program logics, and it enhances the expressiveness of the approach. The implementation and examples (including those of the paper as well as classic examples from the EasyCrypt distribution, including Bellare and Rogaway BR93 Encryption, Hashed ElGamal encryption, Cramer-Shoup encryption, and hybrid arguments) are given as **supplementary material** [3] and we intend to open-source our implementation upon publication.

Case study: Universal Composability. Universal Composability [14, 16] (UC) is a popular framework for reasoning about cryptographic systems. Its central notion, called UC-emulation, formalizes when a protocol π_1 can safely replace a protocol π_2. Informally, UC-emulation imposes that there exists a simulator $\mathcal{S}$ capable of *fooling* any environment $\mathcal{Z}$ by presenting to it a view that is fully consistent with an interaction with π_1, while it is in fact interacting with $\mathcal{S}(\pi_2)$. This intuition, however, must be formalized with tight control over the capabilities of the environment and the simulator. If this were not the case, the definition would make no sense: existential quantification over unrestricted simulators is too weak (it is crucial for the compositional security semantics that simulators use comparable resources to real-world attackers), whereas universal quantification over unrestricted environments results in a definition that is too strong to be satisfied [14, 15]. Moreover, when writing proofs in the UC setting, it is often necessary to consider the joint resources of a sub-part of a complex system that involves a mixture of concrete probabilistic algorithms and abstract adversarial entities, when they are grouped together to build an attacker for a reductionistic proof. In these cases, it is very hard to determine by inspection whether the constructed adversaries are within the complexity classes for which the underlying computational assumptions are assumed to hold. Therefore, tool support for complexity claims is of particular importance with UC — conversely, UC is a particularly challenging example for complexity claims.

Using our enriched implementation of EasyCrypt, we develop a new *fully mechanized* formalization of UC—in contrast to [18], which chooses to follow closely the classic execution model for UC, our mechanization adopts a more EasyCrypt-friendly approach that is closer to the simplified version of UC proposed by Canetti, Cohen and Lindell in [17]; this is further discussed in Section 5. Our mechanization covers the core notions of UC, the classic composition lemmas, transitivity and composability, which respectively state that UC-emulation (as a binary relation between cryptographic systems) is closed under transitivity and arbitrary adversarial contexts. As an illustrative application of our approach we revisit the example used in [18], where modular proofs for Diffie-Hellman key exchange and encryption over ideal authenticated channels are composed to construct a UC secure channel.

2 WARM UP EXAMPLE: PKE FROM A ONE-WAY TRAPDOOR PERMUTATION

To illustrate our approach we will use a public-key encryption (PKE) scheme proposed by [11] (BR93) that uses a one-way trapdoor permutation and a hash function modeled as a random oracle (RO). Intuitively, the RO is used to convert the message into a random input for the trapdoor permutation so as to allow a reduction to the one-wayness property. This proof strategy is used in BR93 and many other schemes, including OAEP [11]. Figure 1 shows the code of an inverter for the trapdoor permutation that is constructed from an attacker against the encryption scheme. This inverter simulates the single random oracle used by the encryption scheme for the attacker and recovers the preimage to y with essentially the same probability as the attacker breaks the encryption scheme.

We first define module types for random oracles RO, schemes Scheme, and adversaries Adv. The module type for random oracles declares a single procedure o with cost $\leq t_o$. The module type for schemes declares three procedures for key generation, encryption, and decryption, and is parametrized by a random oracle H. No cost declaration is necessary. The module type for (chosen-plaintext) adversaries declares two procedures: choose for choosing two plaintexts m_0 and m_1, and guess for guessing the (uniformly sampled) bit b given an encryption of m_b. The cost of these procedures is a pair: the second component is an upper bound on the number of times it can call the random oracle, and the first is an upper bound on its intrinsic cost, i.e. its cost assuming that oracle calls (modeled as functor parameters) have a cost of 0. This style of modeling is routinely used in cryptography and is better suited to reason about open code. This cost model is also more fine-grained than counting the total cost of the procedure including the cost of the oracles, as we have a guarantee on the number of time oracles are called.

Next, we define the inverter Inv for the one-way trapdoor permutation. It runs the adversary A, keeping track of all the calls that A

```
module type RO = {
 proc o (r:rand) : plaintext compl[intr : t_o]}.

module type Scheme (H: RO) = {
 proc kg() : pkey * skey
 proc enc(pk:pkey, m:plaintext) : ciphertext
 proc dec(sk:skey, c:ciphertext) : plaintext option}.

module type Adv (H: RO)  = {
 proc choose(p:pkey) : (plaintext * plaintext) compl[intr : t_c, H.o : k_c]
 proc guess(c:ciphertext) : bool compl[intr : t_g, H.o : k_g]}.

module (Inv : INV) (H : RO) (A:Adv) = {
 var qs : rand list
 module QH = {
  proc o(x:rand) = { qs ← x::qs; r  ← H.o(x); return r; }}

 proc invert(pk:pkey,y:rand): rand = {
  qs      ← [];
  (m_0,m_1) ← A(QH).choose(pk);
  h       ←$ dplaintext;
  b       ← A(QH).guess(y || h);
  while (qs ≠ []) {
   r ← head qs; if (f pk r = y) return r; qs ← tail qs; }
 }}.
```

Figure 1: Inverter for trapdoor permutation.

makes to H in a list qs (using the sub-module QH), and then searches in the list qs for a pre-image of y under f pk. Search is done through a while loop, which we write in a slightly beautified syntax. This inverter can be used to state the following reductionist security theorem relating the advantage and execution cost of an adversary against chosen-plaintext security of the PKE with the advantage of the inverter against one-wayness.

Theorem 2.1 (Security of BR93). *Let t_f represent the cost of applying the one-way function f and t_o denote the cost of H.o, i.e. the implementation of a query to a lazily sampled random oracle. Fix the type for adversaries $\tau_{\mathcal{A}}$ such that:*

$$\begin{aligned} &\text{cost}\,\mathcal{A}.choose \le \textcolor{red}{\text{compl}}[intr : t_c, H.o : k_c] \\ \textit{and}\quad &\text{cost}\,\mathcal{A}.guess \le \textcolor{red}{\text{compl}}[intr : t_g, H.o : k_g] \end{aligned}$$

and fix $\tau_{\mathcal{I}}$ such that:

$$\text{cost}\,\mathcal{I}.invert \le \textcolor{red}{\text{compl}}[intr : (5+t_f)\cdot(k_c+k_g)+4+t_o\cdot(k_c+k_g)+t_c+t_g]\,.$$

Then, $\forall \mathcal{A} \in \tau_{\mathcal{A}}, \exists \mathcal{I} \in \tau_{\mathcal{I}},\ \mathsf{adv}^{\mathrm{BR93}}_{\mathit{IND\text{-}CPA}}(\mathcal{A}) \le \mathsf{adv}^{f}_{\mathit{OW}}(\mathcal{I})$.

Here, IND-CPA refers to the standard notion of ciphertext indistinguishability under chosen-plaintext attacks for PKE, where the adversary is given the public key and asked to guess which of two messages of its choice has been encrypted in a challenge ciphertext; OW refers to the standard one-wayness definition for trapdoor permutations, where the attacker is given the public parameters and the image of a random pre-image, which it must invert. In the former, advantage is the absolute bias of the adversary's boolean output w.r.t. 1/2; in the latter, advantage is the probability of successful inversion.

We prove the statement by providing Inv($\mathcal{A}$) as a witness for the existential quantification, which creates two subgoals. The first sub-goal establishes the advantage bound, which we prove using relational Hoare logic. The second sub-goal establishes that our inverter satisfies the cost restrictions stated in the theorem, and so we use our Hoare logic for complexity to discharge it. We declare the type of Inv as:

$$\begin{aligned} \text{cost Inv.invert} \le \textcolor{red}{\text{compl}}[&\text{intr} : (5+t_f)\cdot(k_c+k_g)+4, \\ &\text{H.o} = k_c+k_g, \mathcal{A}.choose = 1, \mathcal{A}.guess = 1] \end{aligned}$$

and so we first must establish that Inv belongs to this functor type. It is easy to show that A.choose and A.guess are called exactly once, and that H.o is called at most $k_c + k_g$ times. So we turn to the intrinsic complexity of Inv. The key step for this proof is to show that the loop does at most $k_c + k_g$ iterations. We use the length of qs as a variant: the length of the list is initially 0, and incremented by 1 by each call to the random oracle, therefore its length at the start of the loop is at most $k_c + k_g$. Moreover, the length decreases by 1 at each iteration, so we are done. The remaining reasoning is standard,[1] using the cost of each operator—fixed by choice in this particular example to 1, except for the operator f. Our modeling of cost enforces useful invariants that simplify reasoning. For instance, proving upper bounds on the execution cost of Inv requires proving an upper bound on the number of iterations of the loop, and therefore on the length of qs upon entering the loop. We derive the complexity statement in the theorem, which shows only the intrinsic cost of Inv, by instantiating the complexity type of Inv with the cost of its module parameter $\mathcal{A}$. This illustrates how our finer-grained notion of cost is useful for compositional reasoning.

Comparison with EasyCrypt. Our formalization follows the same pattern as the BR93 formalization from the EasyCrypt library. However, the classic module system of EasyCrypt only tracks read-and-write effects and lacks first-class support for bounding the number of oracle calls and for reasoning about the complexity of programs. To compensate for this first point, classic EasyCrypt proofs use wrappers to explicitly count the number of calls and to return dummy answers when the number of adversarial calls to an oracle exceeds a threshold. The use of wrappers suffices for reasoning about adversarial advantage. However, no similar solution can be used for reasoning about the computational cost of adversaries.

Therefore, the BR93 formalization from the EasyCrypt library makes use of the explicit definition of $\mathcal{B}$, and users must analyze the complexity of $\mathcal{B}$ outside the tool. As a result, machine-checked security statements are partial (complexity analysis is missing), cluttered (existential quantification is replaced by explicit witnesses), and compositional reasoning is hard (existential quantification over module types cannot be used meaningfully).

3 ENRICHED EASYCRYPT MODULE SYSTEM

We present a formalisation of our extended module system for EasyCrypt. It is based on EasyCrypt current imperative probabilistic programming language and module system, which we enrich to track the read-and-write memory footprint and complexity cost of module components through *module restrictions*. These module restrictions are checked through a type system: memory footprint type-checking is fully automatic, while type-checking a complexity restriction generates a proof obligation that is discharged to the user — using the cost Hoare logic we present later, in Section 4.

Expressions *(distribution expressions are similar)*:

$$e ::= v \in \mathcal{V} \quad \text{(variable)}$$
$$\mid f(e_1, \ldots, e_n) \quad (\text{if } f \in \mathcal{F}_\mathsf{E})$$

Statements:

$$\begin{aligned} \mathsf{s} ::=\ & \textbf{abort} && \text{(abort)} \\ \mid\ & \textbf{skip} && \text{(skip)} \\ \mid\ & \mathsf{s}_1; \mathsf{s}_2 && \text{(sequence)} \\ \mid\ & x \leftarrow e && \text{(assignment)} \\ \mid\ & x \xleftarrow{\$} d && \text{(sampling)} \\ \mid\ & x \leftarrow \textbf{call}\ \mathsf{F}(\vec{e}\,) && \text{(proc. call)} \\ \mid\ & \textbf{if}\ e\ \textbf{then}\ \mathsf{s}_1\ \textbf{else}\ \mathsf{s}_2 && \text{(cond.)} \\ \mid\ & \textbf{while}\ e\ \textbf{do}\ \mathsf{s} && \text{(loop)} \end{aligned}$$

Procedure body:

$$\mathsf{body} ::= \{\ \mathsf{var}\ (\vec{v} : \vec{\tau});\ \mathsf{s};\ \mathsf{return}\ e\ \}$$

Function paths:

$$\mathsf{F} ::= \mathsf{p}.f \quad \text{(proc. lookup)}$$

Module paths:

$$\begin{aligned} \mathsf{p} ::=\ & \mathsf{x} && \text{(mod. ident.)} \\ \mid\ & \mathsf{p.x} && \text{(mod. comp.)} \\ \mid\ & \mathsf{p(p)} && \text{(func. app.)} \end{aligned}$$

Module expressions:

$$\begin{aligned} \mathsf{m} ::=\ & \mathsf{p} && \text{(mod. path)} \\ \mid\ & \mathsf{struct\ st\ end} && \text{(structure)} \\ \mid\ & \mathsf{func(x : M)\ m} && \text{(functor)} \end{aligned}$$

Module structures:

$$\mathsf{st} ::= \mathsf{d}_1; \ldots; \mathsf{d}_n \quad (n \in \mathbb{N})$$

Module declarations:

$$\begin{aligned} \mathsf{d} ::=\ & \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body} && \text{(proc.)} \\ \mid\ & \mathsf{module\ x = m} && \text{(module)} \end{aligned}$$

Figure 2: Program and module syntax

3.1 Syntax of Programs and Modules

The syntax of our language and module system is (quite) standard and summarized in Figure 2. We describe it in more detail below. We assume given a set of operators $\mathcal{F}_\mathsf{E}$ and a set of distribution operators $\mathcal{F}_\mathsf{D}$. For any $g \in \mathcal{F}_\mathsf{E} \cup \mathcal{F}_\mathsf{D}$, we assume given its type: $\mathsf{type}(g) = \tau_1 \times \cdots \times \tau_n \to \tau$ where $\tau_1, \ldots, \tau_n, \tau \in \mathbb{B}$ with $\mathbb{B}$ the set of base types. We require that bool is a base type, and otherwise leave $\mathbb{B}$ unspecified.

We consider well-typed arity-respecting expressions built from $\mathcal{F}_\mathsf{E}$ and variables in $\mathcal{V}$. Similarly, distribution expressions d are built upon $\mathcal{F}_\mathsf{D}$ and $\mathcal{V}$. For any expression e, we let $\mathsf{vars}(e)$ be the set of variables appearing in e (idem for distribution expression).

We assume a simple language for program statements. A statement s can be an abort, a skip, a statement sequence $\mathsf{s}_1; \mathsf{s}_2$, an assignment $x \leftarrow e$ of an expression to a variable, a random sampling $x \xleftarrow{\$} d$ from a distribution expression, a conditional, a while loop, or a procedure call $x \leftarrow \textbf{call}\ \mathsf{F}(\vec{e}\,)$.

The module system. In a procedure call, F is a function path of the form $\mathsf{p}.f$ where f is the procedure name and p is a module path. Basically, when calling $\mathsf{p}.f$, the module system will resolve p to a module structure, which must declare the procedure f (this will be guarantee by our type system). Formally, a module structure st is a list of module declarations, and a module declaration d is either a procedure (with typed arguments, and a body which comprises a list of local variables with their types $\vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l}$, a statement s and a return expression e) or a sub-module declaration.

The component c of a module x can be accessed through the module path expression $\mathsf{x}.c$. Since a module can contain sub-modules, we can have nested accesses, as in $\mathsf{x}. \ldots .z.c$. Hence, a module path is either a module identifier, a component access of another module path p, or a functor application. Finally, a module expression m is either a module path, a module structure or a functor.

[1]Notice that the condition of the loop is executed at most $k_c + k_g$ time.

Signature structures *(for any $n \in \mathbb{N}$)*:

$$\mathsf{S} ::= \mathsf{D}_1; \ldots; \mathsf{D}_n$$

Module signature declarations:

$$\mathsf{D} ::= \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r \mid \mathsf{module\ x : M}$$

Module signatures:

$$\mathsf{M} ::= \mathsf{sig\ S}\ \mathsf{restr}\ \theta\ \mathsf{end} \mid \mathsf{func(x : M)\ M'}$$

Module restrictions:

$$\theta \ ::= \epsilon \mid \theta, (f : \lambda) \qquad \lambda \ ::= \top \mid \lambda_\mathsf{m} \wedge \lambda_\mathsf{c}$$

Memory restrictions *(for any $l \in \mathbb{N}$)*:

$$\lambda_\mathsf{m} ::= +\mathsf{all\ mem} \backslash \{v_1, \ldots, v_l\} \mid \{v_1, \ldots, v_l\}$$

Complexity restrictions *(for any $l, k, k_1, \ldots, k_l \in \mathbb{N}$)*:

$$\lambda_\mathsf{c} ::= \top \mid \mathsf{compl}[\mathsf{intr} : k, \mathsf{x}_1.f_1 : k_1, \ldots, \mathsf{x}_l.f_l : k_l]$$

Figure 3: Module signatures and restrictions

```
module type HSM = {
  proc enc (x:msg) : cipher }.
module Hsm : HSM = {
  proc enc (x:msg) : cipher = { . . . } }.
module type Adv (H : HSM) {+all mem, -Hsm} = {
  proc guess () : skey      [intr : k0, H.enc : k] }.
```

Figure 4: Example of adversary with restrictions.

3.2 Module Signatures and Restrictions

The novel part of our system is the use of module restrictions in module signatures. Objects related to module restriction are highlighted in red throughout this paper. The syntax of module signatures and restrictions is given in Figure 3. A module structure signature S is a list of module signature declarations, which are procedure signatures or sub-module signatures. Then, a module signature M is either a functor signature, or a structure signature with a module restriction θ attached.

Module restrictions. A module restriction restricts the effects of a module's procedures. We are interested in two types of effects. First, we characterize the memory footprint (i.e. global variables which are read or written to) of a module's procedures through *memory restrictions*. Second, we upper bound the execution cost of a procedure, and the number of calls a functor's procedure can make to the functor's parameters, through *complexity restrictions*.

Restrictions are useful for compositional reasoning, as they allow stating and verifying properties of a module's procedures at declaration time. In the case of an abstract module, restrictions allow to constrain, through the type system, its possible instantiations. This is a key idea of our approach, which we exploit to prove complexity properties of cryptographic reductions.

For example, we give in Figure 4 EasyCrypt code corresponding to an adversary against a hardware security module. In this scenario the goal of the adversary is to recover the secret key stored in the module Hsm. The example uses two types of restrictions. The module-level restriction {+all mem, -Hsm} states that such an adversary can access all the memory, except for the memory used by the module Hsm. The procedure-level restriction [intr : k_0, H.enc : k] attached to guess, states that guess execution time is at most k_0 (excluding calls to H.enc), and that guess can make at most k queries to the procedure H.enc.

Formally, a module restriction is a list of pairs comprising a procedure identifier f and a procedure restriction λ, and a procedure restriction λ is either $\top$ (no restriction), or the conjunction of a memory restriction λ_{m} and a complexity restriction λ_{c}:

Memory. A memory restriction λ_{m}, attached to a procedure f, restricts the variables that f can access *directly*. We allow for positive memory restrictions $\{v_1, \ldots, v_l\}$, which states that f can only access the variables $v_1, \ldots, v_l$; and negative memory restrictions $+\mathsf{all\ mem}\backslash\{v_1, \ldots, v_l\}$, which states that f can access any global variables except the variables $v_1, \ldots, v_l$.

Note that λ_{m} only restricts f's *direct* memory accesses: this excludes the memory accessed by the procedure oracles (which are modeled as functor's parameters). This is crucial, as otherwise, an adversary that is not allowed to access some oracle's memory (a standard assumption in security proofs) would not be allowed to call this oracle. E.g., the adversary of Figure 4 can call the oracle H.enc (which can be instantiated by Hsm), even though it cannot access directly Hsm's memory.

Complexity. A complexity restriction λ_{c} attached to a procedure f restricts its execution time and the number of calls that f can make to its parameters: it is either $\top$, i.e. no restriction; or the restriction $\mathsf{compl}[\mathsf{intr} : k, \mathsf{x}_1.f_1 : k_1, \ldots, \mathsf{x}_l.f_l : k_l]$, which states that: i) its execution time (excluding calls to the parameters) must be at most k; ii) f can call, for every i, the parameter's procedure $\mathsf{x}_i.f_i$ at most k_i times. We require that all parameter's procedures appear in the restriction.

3.3 Typing Enriched Module Restrictions

We check that modules verify their signatures through a type system. The novelty of our approach lies in the enriched restrictions attached to module signatures, and the typing rules that check them. For space reasons, we only present the two main restriction checking rules here (the full type system is in Appendix B).

Environments. Typing is done in an environment $\mathcal{E}$.[2] Essentially, an environment is a list of declarations, which are either variable, module or abstract module declarations.

$$\begin{aligned}\mathcal{E} ::= \epsilon \mid \mathcal{E}, \mathsf{var}\ v : \tau \mid \mathcal{E}, \mathsf{module\ x} = \mathsf{m} : \mathsf{M} \\ \mid \mathcal{E}, \mathsf{module\ x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_l}\end{aligned}$$

An abstract module declaration $\mathsf{module\ x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_l}$ states that x is a module with signature $\mathsf{M_l}$ whose code is unknown, and allows to model open code.[3] For any $\mathcal{E}$, we let $\mathsf{abs}(\mathcal{E}) = \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ be the set of abstract module names declared in $\mathcal{E}$.

Restrictions. The RestrMem rule checks that a procedure body $\{\ \_;\ \mathsf{s};\ \mathsf{return}\ e\ \}$ (where s is the procedure's instructions, and e the returned expression) verifies a *memory* restriction through a fully automatic syntactic check.

$$\textsc{RestrMem}\quad \frac{(\mathsf{mem}_{\mathcal{E}}(\mathsf{s}) \sqcup \mathsf{vars}(e)) \sqsubseteq \lambda_{\mathsf{m}}}{\mathcal{E} \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ e\ \} \rhd \lambda_{\mathsf{m}}}$$

This syntactic check uses $\mathsf{mem}_{\mathcal{E}}(\mathsf{s})$ and $\mathsf{vars}(e)$, which are sound over-approximations of an instruction and expression memory footprint (the approximation is not complete, e.g. it will include memory accesses done by unreachable code).

The RestrCompl rule checks that an instruction verifies some *complexity* restriction. The rule generates proof obligations in a Hoare logic for cost. These proof obligations are discharged interactively using the proof system we present later, in Section 4.

$$\textsc{RestrCompl}\quad \frac{\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\} \qquad \vdash \{\psi\}\ r \leq t_r \qquad (t + t_r \cdot \mathbb{1}_{\mathsf{conc}}) \leq_{\mathsf{compl}} \lambda_{\mathsf{c}}}{\mathcal{E} \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ r\ \} \rhd \lambda_{\mathsf{c}}}$$

Here, the proof obligation $\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\}$ states that the execution of s in any memory has a complexity upper bounded by t, and that the post-condition ψ holds after s's execution. The proof obligation $\vdash \{\psi\}\ r \leq t_r$ upper-bounds the cost of evaluating the return expression r. Finally, the rule checks that the sum of t and t_r is compatible with the complexity restriction λ_{c} through the premise $(t + t_r \cdot \mathbb{1}_{\mathsf{conc}}) \leq_{\mathsf{compl}} \lambda_{\mathsf{c}}$. We leave the precise definition of $\leq_{\mathsf{compl}}$ to Section 4 (see Figure 8). Intuitively, t is a record of entries of the form $(\mathsf{x}.f \mapsto l_f)$, each stating that the abstract module x's procedure f has been called at most l_c times, plus a special entry $(\mathsf{conc} \mapsto l_c)$ stating that s execution time, excluding abstract calls, is at most l_c. Then, $t_0 \leq_{\mathsf{compl}} \lambda_{\mathsf{c}}$ checks that $t_0[\mathsf{x}.f] \leq \lambda_{\mathsf{c}}[\mathsf{x}.f]$ for every *functor parameter* $\mathsf{x}.f$, and that $\lambda_{\mathsf{c}}[\mathsf{intr}]$ upper-bounds everything else in t_0.

4 COMPLEXITY REASONING IN EASYCRYPT

We now present our Hoare logic for cost, which allows to formally prove complexity upper-bound of programs. This logic manipulates judgment of the form $\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}$, where s is a statement, ϕ, ψ are assertions, and t is a cost. We leave the assertion language unspecified, and only require that the models of an assertion formula ϕ are memories, and write $\nu \in \phi$ whenever ν satisfies ϕ.

4.1 Cost Judgment

A key point of our Hoare logic for cost is that it allows to split the cost of a program s between its concrete and abstract costs, i.e. between the time spent in concrete code, and the time spent in abstract procedures. To reflect this separation, a *cost t* is a record of entries mapping each abstract procedure $\mathsf{x}.f$ to the number of times this procedure was called, and mapping a special element conc to the concrete execution time (i.e. excluding abstract procedure calls).

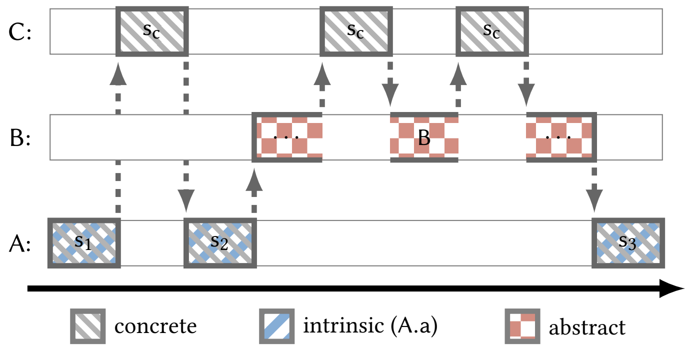
Definition 4.1. A $\mathcal{E}$-cost is an element of the form:

$$t ::= [\mathsf{conc} \mapsto k, \mathsf{x}_1.f_1 \mapsto k_1, \ldots \mathsf{x}_l.f_l \mapsto k_l]$$

where $\mathcal{E}$ is an environment, $k, k_1, \ldots, k_l$ are integers, and the $\mathsf{x}_i.f_i$ are all the abstract procedures declared in $\mathcal{E}$.

[2] Actually, the type system in Appendix B uses more complex environment, called *typing environment*, to account for sub-modules.

[3] Abstract module must have *low-order* signatures, i.e. module structures, or functors whose parameters are module structures (see Appendix B). This choice is motivated by the fact that further generality is not necessary for cryptographic proofs (adversaries and simulations usually return base values, not procedures); and, it allows the abstract call rule of our cost Hoare logic Abs (in Figure 7) to remain tractable.




```
module type T_C = { proc c () : unit }.

module C = { proc c () = { s_c } }.

module type T_B (C_0 : T_C) = {
 proc b () : unit [intr : t_b, C_0.c : 2] }.

module type T_A (B_0 : T_B) (C_0 : T_C) = {
 proc a () : unit
  [intr : t_a, B_0.b : 1, C_0.c : 1] }.

module A (B_0 : T_B) (C_0 : T_C) : T_A = {
 proc a () = {
  s_1;  C_0.c();  s_2;  B_0(C_0).b();  s_3;
 }}.
```

Judgment $\mathcal{E} \vdash \{\top\}$ A(B, C).a $\{\top \mid [\text{conc} \mapsto t_{\text{conc}}, \text{B.b} \mapsto 1]\}$ where $\mathcal{E} = (\text{module B} = \text{abs}_{\text{open}} : \text{T}_\text{B})$.

Figure 5: Graphical representation of the different cost measurements.

Example 4.1. Consider $\mathcal{E}$ with two abstract modules x and y:

$$\begin{aligned}\mathcal{E} = \ &(\text{module x} = \text{abs}_{\text{open}} : \text{sig (proc } f\ \_; \text{proc } g\ \_) \text{ restr } \_ \text{ end});\\ &(\text{module y} = \text{abs}_{\text{open}} : \text{sig (proc } h\ \_) \text{ restr } \_ \text{ end})\end{aligned}$$

Then $[\text{conc} \mapsto 10; \text{x}.f \mapsto 0; \text{x}.g \mapsto 2; \text{y}.h \mapsto 1]$ represents a concrete cost of 10, at most two calls to x.g, one to y.h, and none to x.f.

Definition 4.2. A *cost judgment* for a statement is an element of the form $\mathcal{E} \vdash \{\phi\}$ s $\{\psi \mid t\}$ where $\mathcal{E}$ must be well-typed, s must be well-typed in $\mathcal{E}$ and t must be an $\mathcal{E}$-cost. We define similarly a cost judgment for a procedure $\mathcal{E} \vdash \{\phi\}$ F $\{\psi \mid t\}$.

In Figure 5, we give a graphical representation of a cost judgment for the procedure A(B, C).a, where A and C are concrete modules, and B is an abstract functor with access to C as a parameter. Then, intuitively, the cost judgment:

$$\mathcal{E} \vdash \{\top\} \text{ A(B, C).a } \{\top \mid [\text{conc} \mapsto t_{\text{conc}}, \text{B.b} \mapsto 1]\}$$

is valid whenever t_{conc} upper-bound the concrete cost (in hatched gray) which is the sum of: i) the intrinsic cost of A.a, which is the cost of A.a without counting parameter's calls, represented in hatched blue in the figure, and must be at most t_a as stated in T_A's restriction; and ii) the sum of the cost of the three calls to C.c.

The cost of the execution of the abstract procedure B.b (in hatched red), which excludes the two calls B.b makes to C.c, are accounted for by the entry (B.b $\mapsto$ 1) in the cost judgment. Note that it is crucial that this excludes the cost of the two calls to C.c, which are already counted in the concrete cost t_{conc} .

Expression cost. We have a second kind of judgment $\vdash \{\phi\}\ e \le t_e$, which states that the cost of evaluating e in any memory satisfying ϕ is at most t_e, where t_e is an *integer*, not a $\mathcal{E}$-cost (indeed, an expression cost is always fully concrete, as expressions do not contain procedure calls). We do not provide a complete set of rules for such judgments, as this depends on low-level implementation details and choices, such as data-type representation and libraries implementations. In practice, we give rules for some built-ins, a way for the user to add new rules, and an automatic rewriting mechanism which automatically prove such judgments from the user rules in most cases.

4.2 Hoare Logic for Cost Judgment

We now present our Hoare logic for cost, which allows to prove cost judgments of programs. We first present basic rules, and then

$$\textsc{Skip}\ \frac{}{\mathcal{E} \vdash \{\phi\} \textbf{ skip } \{\phi \mid 0\}} \qquad \textsc{Weak}\ \frac{\mathcal{E} \vdash \{\phi'\} \text{ s } \{\psi' \mid t'\} \quad \phi \Rightarrow \phi' \quad \psi' \Rightarrow \psi \quad t' \le t}{\mathcal{E} \vdash \{\phi\} \text{ s } \{\psi \mid t\}}$$

$$\textsc{False}\ \frac{}{\mathcal{E} \vdash \{\bot\} \text{ s } \{\psi \mid t\}} \qquad \textsc{Assign}\ \frac{\vdash \{\phi\}\ e \le t_e}{\mathcal{E} \vdash \{\phi \wedge \psi[x \leftarrow e]\}\ x \leftarrow e\ \{\psi \mid t_e\}}$$

$$\textsc{Rand}\ \frac{\vdash \{\phi_0\}\ d \le t \qquad \phi = (\phi_0 \wedge \forall v \in \text{dom}(d).\psi[x \leftarrow v])}{\mathcal{E} \vdash \{\phi\}\ x \xleftarrow{\$} d\ \{\psi \mid t\}} \qquad \textsc{Seq}\ \frac{\mathcal{E} \vdash \{\phi\} \text{ s}_1 \{\phi' \mid t_1\} \qquad \mathcal{E} \vdash \{\phi'\} \text{ s}_2 \{\psi \mid t_2\}}{\mathcal{E} \vdash \{\phi\} \text{ s}_1; \text{s}_2 \{\psi \mid t_1 + t_2\}}$$

$$\textsc{If}\ \frac{\mathcal{E} \vdash \{\phi \wedge e\} \text{ s}_1 \{\psi \mid t\} \qquad \mathcal{E} \vdash \{\phi \wedge \neg e\} \text{ s}_2 \{\psi \mid t\} \qquad \vdash \{\phi\}\ e \le t_e}{\mathcal{E} \vdash \{\phi\} \textbf{ if } e \textbf{ then } \text{s}_1 \textbf{ else } \text{s}_2 \{\psi \mid t + t_e\}}$$

$$\textsc{While}\ \frac{\begin{array}{c} I \wedge e \Rightarrow c \le N \qquad \forall k, \mathcal{E} \vdash \{I \wedge e \wedge c = k\} \text{ s } \{I \wedge k < c \mid t(k)\} \\ \forall k \le N, \vdash \{I \wedge e \wedge c = k\}\ e \le t_e(k) \qquad \vdash \{I \wedge \neg e\}\ e \le t_e(N+1)\end{array}}{\mathcal{E} \vdash \{I \wedge 0 \le c\} \textbf{ while } e \textbf{ do } \text{s } \{I \wedge \neg e \mid \sum_{i=0}^{N} t(i) + \sum_{i=0}^{N+1} t_e(i)\}}$$

$$\textsc{Call}\ \frac{\text{args}_{\mathcal{E}}(\text{F}) = \vec{v} \qquad \vdash \{\phi[\vec{v} \leftarrow \vec{e}]\}\ \vec{e} \le t_e \qquad \mathcal{E} \vdash \{\phi\} \text{ F } \{\psi[x \leftarrow \text{ret}] \mid t\}}{\mathcal{E} \vdash \{\phi[\vec{v} \leftarrow \vec{e}]\}\ x \leftarrow \textbf{call } \text{F}(\vec{e})\ \{\psi \mid t_e + t\}}$$

$$\textsc{Conc}\ \frac{\text{f-res}_{\mathcal{E}}(\text{F}) = (\text{proc } f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_; \text{s}; \text{ return } r\ \}) \qquad \mathcal{E} \vdash \{\phi\} \text{ s } \{\psi[\text{ret} \leftarrow r] \mid t\} \qquad \vdash \{\psi\}\ r \le t_{\text{ret}}}{\mathcal{E} \vdash \{\phi\} \text{ F } \{\psi \mid t + t_{\text{ret}}\}}$$

Convention: ret cannot appear in programs (i.e. ret $\notin \mathcal{V}$).

Figure 6: Basic rules for cost judgment.

turn to the two core rules of our logic, namely the abstract call and instantiation rules.

The basic rules are given in Figure 6. Basically, our cost judgment are standard Hoare logic judgment with the additional cost information, and both aspects must be handled by the rules of our logic. In some cases, these can be handled separately. E.g., the assignment rule Assign lets the user provide a dedicated pre-condition ϕ used to upper-bound the cost of evaluating e[4]; and the weakening rule Weak is the standard Hoare logic weakening rule, with an

[4]If the rule forced to take $\phi = \psi[x \leftarrow e]$, then it would not be complete, as prior information on the value on x (e.g. coming from a previous assignment to x) is erased, which may prevent us from proving a precise upper-bound on $\vdash \{\phi\}\ e \le t_e$.

Abs

$$\frac{\begin{array}{c}\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \\ \mathcal{E}(\mathsf{x}) = \mathsf{abs}_{\mathsf{open}}\mathsf{x} : (\mathsf{func}(\vec{\mathsf{y}} : \_)\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}) \\ \theta[f] = \lambda_{\mathsf{m}} \wedge \lambda_{\mathsf{c}} \qquad \lambda_{\mathsf{c}} = \mathsf{compl}[\mathsf{intr} : K, \mathsf{z}_{j_1}.f_1 : K_1, \dots, \mathsf{z}_{j_l}.f_l : K_l] \\ \mathsf{FV}(I) \cap \lambda_{\mathsf{m}} = \emptyset \qquad \vec{k} \text{ fresh in } I \\ \forall i, \forall \vec{k} \leq (K_1, \dots, K_l),\ \vec{k}[i] < K_i \rightarrow \mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{\mathsf{p}}[j_i].f_i\ \{I\ (\vec{k} + \mathbb{1}_i) \mid t_i\ k\}\end{array}}{\mathcal{E} \vdash \{I\ \vec{0}\}\ \mathsf{F}\ \{\exists \vec{k},\ I\ \vec{k} \wedge \vec{0} \leq \vec{k} \leq (K_1, \dots, K_l) \mid T_{\mathsf{abs}}\}}$$

where:

$$T_{\mathsf{abs}} = \left\{\mathsf{x}.f \mapsto 1;\ \left(\mathsf{G} \mapsto \textstyle\sum_{i=1}^{l} \sum_{k=0}^{K_i - 1} (t_i\ k)[\mathsf{G}]\right)_{\mathsf{G} \neq \mathsf{x}.f}\right\}$$

Conventions: $\vec{\mathsf{y}}$ can be empty (this corresponds to the non-functor case).

Figure 7: Abstract call rule for cost judgment.

additional premise $t' \leq t$. Other rules require the user to show simultaneously invariants of the memory state of the program and cost upper-bounds, as expected.

Abstract call rule without cost. This is the case of our rule for upper-bounding the cost of a call to an abstract procedure F. To ease the presentation, we first present a version of the rule for usual Hoare judgment without costs, and explain how to add costs after.

ABS-PARTIAL

$$\frac{\begin{array}{c}\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \\ \mathcal{E}(\mathsf{x}) = \mathsf{abs}_{\mathsf{open}}\ \mathsf{x} : (\mathsf{func}(\vec{\mathsf{y}} : \_)\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}) \qquad \theta[f] = \lambda_{\mathsf{m}} \wedge \_ \\ \mathsf{FV}(I) \cap \lambda_{\mathsf{m}} = \emptyset \qquad \forall \mathsf{p}_0 \in \vec{\mathsf{p}}, \forall g \in \mathsf{procs}_{\mathcal{E}}(\mathsf{p}_0),\ \mathcal{E} \vdash \{I\}\ \mathsf{p}_0.g\ \{I\}\end{array}}{\mathcal{E} \vdash \{I\}\ \mathsf{F}\ \{I\}}$$

First, the function path F is resolved to $(\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f$, i.e. a call to the procedure f of an abstract functor x applied to the modules $\vec{\mathsf{p}}$ (the case where x is not a functor is handled by taking $\vec{\mathsf{p}} = \epsilon$). Then, x's module type is lookup in $\mathcal{E}$, and we retrieve the module restriction θ attached to it. The rule allows to prove that some formula I is an invariant of the abstract call, by showing two things.

First, we show that I is an invariant of $\mathsf{x}.f$, excluding calls to the functor parameters. This is done by checking that $\mathsf{x}.f$ cannot access the variables used in I, using its memory restriction λ_{m} (looked-up by the premise $\theta[f] = \lambda_{\mathsf{m}} \wedge \_$) and requiring that $\mathsf{FV}(I) \cap \lambda_{\mathsf{m}} = \emptyset$.

Then, we prove that I is an invariant of $\mathsf{x}.f$'s calls to functor parameters. This is guaranteed by requiring that for every functor parameter $\mathsf{p}_0 \in \vec{\mathsf{p}}$, for any of p_0's procedure $g \in \mathsf{procs}_{\mathcal{E}}(\mathsf{p}_0)$, the judgment $\mathcal{E} \vdash \{I\}\ \mathsf{p}_0.g\ \{I\}$ is valid.

Abstract call. We now present our Abs rule for *cost judgments*, which is given in Figure 7. Essentially, the cost of the call to $\mathsf{x}(\vec{\mathsf{p}}).f$ is decomposed between:

- the intrinsic cost of $\mathsf{x}.f$ excluding the cost of the calls to x's functor parameters. This is accounted for by the entry $(\mathsf{x}.f \mapsto 1)$ in the final cost T_{abs}.
- the cost of the calls to $\mathsf{x}.f$ functor parameters, which are enumerated in the restriction:

  $$\lambda_{\mathsf{c}} = \mathsf{compl}[\mathsf{intr} : K, \mathsf{z}_{j_1}.f_1 : K_1, \dots, \mathsf{z}_{j_l}.f_l : K_l]$$

  We require, for every i, a bound on the cost of the k-th call to the functor argument z_{j_i} procedure's f_i, where k can range anywhere between 0 and the maximum number of calls $\mathsf{x}.f$ can make to z_{j_i}, which is K_i. The cost of the k-th call to $\mathsf{z}_{j_i}.f_i$ is bounded by $(t_i\ k)$ where $k = \vec{k}[i]$ and:

  $$\mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{\mathsf{p}}[j_i].f_i\ \{I\ (\vec{k} + \mathbb{1}_i) \mid t_i\ k\}$$

  To improve precision, we let the invariant I depends on the numbers of calls to the functor parameters through the integer vector $\vec{k}$. After calling $\vec{\mathsf{p}}[j_i].f_i$, we update $\vec{k}$ by adding one to its i-th entry ($\mathbb{1}_i$ is the vector where the i-th entry is one and all other entries are zero).

The final cost T_{abs} (except for $\mathsf{x}.f$) is obtained by taking the sum, over all functor parameters and number of calls to this functor parameter, of the cost of each call.

Instantiation

$$\frac{\begin{array}{c}\mathsf{M_I} = \mathsf{func}(\vec{\mathsf{y}} : \vec{\mathsf{M}})\ \mathsf{sig}\ \mathsf{S_I}\ \mathsf{restr}\ \theta\ \mathsf{end} \\ \mathcal{E} \vdash_{\mathsf{x}} \mathsf{m} : \mathsf{erase}_{\mathsf{compl}}(\mathsf{M_I}) \qquad \vec{\mathsf{z}} \text{ fresh in } \mathcal{E} \\ \forall f \in \mathsf{procs}(\mathsf{S_I}),\ (\mathcal{E}, \mathsf{module}\ \vec{\mathsf{z}} : \mathsf{abs}_{\mathsf{open}}\ \vec{\mathsf{M}} \vdash \{\top\}\ \mathsf{m}(\vec{\mathsf{z}}).f\ \{\top \mid t_f\}) \\ \forall f \in \mathsf{procs}(\mathsf{S_I}),\ t_f \leq_{\mathsf{compl}} \theta[f] \\ \mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t_s\}\end{array}}{\mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid T_{\mathsf{ins}}\}}$$

where:

$$T_{\mathsf{ins}} = \left\{\mathsf{G} \mapsto t_s[\mathsf{G}] + \textstyle\sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot t_f[\mathsf{G}]\right\}$$

$$t_f \leq_{\mathsf{compl}} \theta[f] = \begin{array}{c}\forall \mathsf{z}_0 \in \vec{\mathsf{z}},\ t_f[\mathsf{z}_0] \leq \theta[f][\mathsf{z}_0]\ \wedge \\ t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ h \in \mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_f[\mathsf{A}.h] \cdot \mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h) \leq \theta[f][\mathsf{intr}]\end{array}$$

Conventions: $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is the intr field in the complexity restriction of the abstract module procedure $\mathsf{A}.h$ in $\mathcal{E}$.

Figure 8: Instantiation rule for cost judgment.

Instantiation rule. The Instantiation rule allows to instantiate an abstract module x by a concrete module m, and is given in Figure 8. Assume that we can upper-bound the cost of a statement s by t_s, when x is abstract:

$$\mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t_s\}$$

Then we can instantiate x by a concrete module m as long as m complies with the module signature $\mathsf{M_I}$, which is checked through two conditions.

First, we check that m has the correct module type, except for complexity restrictions, through the premise $\mathcal{E} \vdash_{\mathsf{x}} \mathsf{m}{:}\mathsf{erase}_{\mathsf{compl}}(\mathsf{M_I})$

Then, we check that m satisfies the complexity restriction θ in $\mathsf{M_I}$, by requiring that for any procedure f of x:

$$\mathcal{E}, \mathsf{module}\ \vec{\mathsf{z}} : \mathsf{abs}_{\mathsf{open}}\ \vec{\mathsf{M}} \vdash \{\top\}\ \mathsf{m}(\vec{\mathsf{z}}).f\ \{\top \mid t_f\}$$

where t_f must respects $\theta[f]$, which is guaranteed by $t_f \leq_{\mathsf{compl}} \theta[f]$, which does two checks:

- first, it ensures that the number of calls to any functor parameter z_0 of x done by $\mathsf{m}.f$ is upper-bounded by $\theta[f][\mathsf{z}_0]$.
- then, it verifies that the bound of x's intrinsic cost $\theta[f][\mathsf{intr}]$ upper-bounds the cost of the execution of $\mathsf{m}.f$, excluding functor parameter calls, through the condition:

  $$t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ h \in \mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_f[\mathsf{A}.h] \cdot \mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h) \leq \theta[f][\mathsf{intr}]$$

where $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is the upper-bound on $\mathsf{A}.h$ intrinsic cost declared in $\mathcal{E}$.[5] In other words, the concrete execution time $t_f[\mathsf{conc}]$ of

[5]If no intrinsic bound is declared for $\mathsf{A}.h$ in $\mathcal{E}$, then $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is undefined (hence $\mathsf{A}.h$ execution time can be arbitrarily large), and the Instantiation rule cannot be applied.

$\mathsf{x}.f$, plus the abstract execution time of $\mathsf{x}.f$ (excluding functor parameters, already accounted for), must be bounded by $\theta[f][\mathsf{intr}]$.

The final cost T_{ins} (in Figure 8) is the sum of the cost t_s of s (which excludes the cost of x's procedures), plus the sum, for any procedure f of x, of the number of times s called $\mathsf{x}.f$ (which is $t_s[\mathsf{x}.f]$), times the cost of $\mathsf{x}.f$ (which is t_f).

4.3 Soundness

We show the soundness of our rules w.r.t. a denotational semantics of our language. We quickly introduce the semantics below, and state our soundness theorem. For space reason, the full semantics and soundness proofs are in Appendix D and E.

The semantics $[\![\mathsf{s}]\!]_\nu^{\mathcal{E},\rho}$ of our language depends on the initial memory ν, the environment $\mathcal{E}$, and on the interpretation ρ of $\mathcal{E}$'s abstract modules. Essentially, $[\![\mathsf{s}]\!]_\nu^{\mathcal{E},\rho}$ is a discrete distribution over $\mathcal{M} \times \mathbb{N}$, where the integer component is the cost of evaluating s in $(\mathcal{E}, \rho)$, starting from the memory ν. Then, the $\mathcal{E}$-cost of an instruction s under memory ν and interpretation of $\mathcal{E}$'s abstract modules ρ, denoted by $\mathsf{cost}_\nu^{\mathcal{E},\rho}(\mathsf{s}) \in \mathbb{N} \cup \{+\infty\}$, is the maximum execution cost in any final memory, defined as:

$$\mathsf{cost}_\nu^{\mathcal{E},\rho}(\mathsf{s}) = \inf \left\{ c' \mid \Pr\left((\_, c) \leftarrow [\![\mathsf{s}]\!]_\nu^{\mathcal{E},\rho}; c \leq c'\right) = 1 \right\}$$

Judgments. Basically, the judgment $\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}$ states that: i) the memory ν obtained after executing s in an initial memory $\nu \in \phi$ must satisfy ψ; ii) the complexity of the instruction s is upper-bounded by the complexity of the concrete code in s, plus the sum over all abstract oracles $\mathsf{A}.f$ of the number of calls to $\mathsf{A}.f$ times the intrinsic complexity of $\mathsf{A}.f$. Formally:

$$\mathsf{cost}_\nu^{\mathcal{E},\rho}(\mathsf{s}) \leq t[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ f \in \mathsf{procs}(\mathcal{E}(\mathsf{A}))}} t[\mathsf{A}.f] \cdot \mathsf{compl}_{\mathsf{A}.f}^{\mathcal{E},\rho}$$

where $\mathsf{compl}_{\mathsf{A}.f}^{\mathcal{E},\rho}$ is the intrinsic complexity of the procedure $\mathsf{A}.f$, i.e. its complexity excluding calls to A's functor parameters.

We now state the soundness of our Hoare logic for cost (a rule is sound if its conclusion is valid whenever its premises are valid).

Theorem 4.1. *The proof rules in Figure 6, 7 and 8 are sound.*

5 EXAMPLE: UNIVERSAL COMPOSABILITY

UC security guarantees that a protocol π_1 can safely replace a protocol π_2 while preserving both the functionality and the security of the overall system. The most common application of this framework is to set π_2 to be an idealized protocol that assumes a trusted-third-party (TTP) to which parties delegate the computation; the specification of the TTP is called an *ideal functionality* $\mathcal{F}$. An ideal functionality $\mathcal{F}$ defines what protocol π_1 should achieve both in terms of correctness and security to securely replace the TTP. Moreover, $\mathcal{F}$ can be used as an ideal sub-component when designing higher-level protocols, which then can be instantiated with protocol π_1 to obtain a fully concrete real-world protocol.

The UC framework defines an execution model where protocol participants, attackers and contexts are modeled as Interactive Turing Machines (ITM). The model was carefully tailored to give a good balance between expressive power—e.g., one can capture complex interactions in distributed protocols involving multiple parties in a variety of communication models, various forms of corruption, etc.—and a tailored (and relatively simple) resource analysis mechanism that permits keeping track of the computing resources available to both honest and malicious parties.

The model is described in detail in [14, 15]. However, most UC proofs found in the literature refer only to a common understanding of the semantics of the execution model and a set of high-level restrictions that are inherent to the model. These include the allowed interactions between different machines, the order in which machines are activated, predefined sequences of events, etc. More fine-grained descriptions of the execution model are sometimes introduced locally in proofs, when they are needed to deal with more subtle points or technicalities that can only be clarified at the cost of extra details. This stands in contrast with typical game-based proofs for simpler cryptographic primitives [9], where security proofs are given in great detail. This is one of the reasons why, while there has been impressive progress in machine-checking game-based proofs [4], we are only now giving the first steps in formalizing proofs in the UC setting [18, 19, 21]. Another reason is that the ITM model for communication is difficult to express in procedure-based semantics offered by tools that target game-based proofs.[6]

To overcome these difficulties, we propose a new approach to machine-checking UC proofs that shares many features of the simplified version of UC proposed by Canetti, Cohen and Lindell in [17]. As in [17], we statically fix the machines/modules in the execution model and we allow an adversarial entity to control which module gets to be executed next, rather than allowing machines to pass control between them more freely as in the original UC execution model. The crucial difference to the ITM execution model is that the above interactions are procedure-based, which means that whenever the environment passes control to the protocol, the internal protocol structure will follow a procedure call tree that guarantees (excluding the possibility of non-terminating code) that control returns to the environment. As in [17], we lose some expressiveness, but we do not go as far as hard-wiring a specific communications model for protocols based on authenticated channels; instead, we leave it to the protocol designer to specify the communications model by using an appropriate module structure. We recover the authenticated communications model model of [17] by explicitly defining a hybrid real-world, in which concrete modules for ideal authenticated channels are available to the communicating parties.

5.1 Mechanized Formalization in EasyCrypt

We propose a natural simplification of the UC execution model that is based on EasyCrypt modules and show that this opens the way for a lightweight formalization of UC proofs. This formalisation has been conducted in our extension of EasyCrypt (the proofs of the lemmas and theorems of this section are fully mechanized).

Protocols and Functionalities as EasyCrypt modules. The basic component in our UC execution model is a module of type PROTOCOL given in Figure 9. Inhabitants of this type represent a full real-world configuration—a distributed protocol executed by a fixed number of parties—or an ideal-world configuration—an ideal functionality executing a protocol as a trusted-third party. The type of

[6] Intuitively, the UC model expresses a single line of execution using a token-passing mechanism that allows one machine to *transfer* computational resources to another, and even to create new machines.

```
module type IO = {
  proc inputs (i:inputs) : unit
  proc outputs(o:ask_outputs) : outputs option }.
module type BACKDOORS = {
  proc step (m:step) : unit
  proc backdoor (m:ask_backdoor) : backdoor option }.
module type E_INTERFACE = {
  include IO
  include BACKDOORS }.
module type PROTOCOL = {
  proc init() : unit
  include E_INTERFACE }.
```

Figure 9: PROTOCOL type in EasyCrypt.

```
module UC_emul (E:ENV) (P:PROTOCOL) = {
  proc main() = {
    var b;
    P.init(); b ← E(P).distinguish(); return b; }}.
```

```
module CompS(F:IDEAL.PROTOCOL, S:SIMULATOR) : PROTOCOL = {
  proc init() = { F.init(); S(F).init(); }
  include F [ inputs, outputs]
  include S(F) [step, backdoor]}.
```

Figure 10: Execution model for real/ideal worlds (top) and composition of functionality with a simulator (bottom).

a protocol has a fixed interface, but it is parametric on the types of values exchanged via this interface. The fixed interface is divided into three parts: i) init allows modeling some global protocol setup; ii) IO captures the interaction of a higher level protocol using this protocol as a sub-component; and iii) BACKDOORS captures the interaction of an adversary with the protocol during its execution.

When we define real-world protocols, a module of type PROTOCOL will be constructed from sub-modules that emulate the various parties and the communications channels between them. In this case, BACKDOORS models adversarial power in this communication model. For ideal-world protocols, a PROTOCOL is typically a flat description of the ideal computation in a single module; here BACKDOORS models unavoidable leakage (e.g., the length of secret inputs or the states of parties in an interactive protocol) and external influence over the operation of the trusted-third party (e.g., blocking the computation to model a possible denial of service attack).[7]

Execution Model. The real- and ideal-world configurations are composed by a statically determined set of modules, which communicate with each-other using a set of hardwired interfaces. The execution model is defined by an experiment in which an external environment interacts with the protocol via its IO and BACKDOORS interfaces until, eventually, it outputs a boolean value (Figure 10). The IO interface allows the environment to pass an input to the protocol using inputs or to retrieve an output produced by the protocol using outputs. For example in the real-world, the environment can use these procedures to give input to or obtain an output from one of the sub-modules that represent the computing parties involved in the protocol. The BACKDOORS interface allows the environment to read some message that may be produced by the protocol using backdoor or make one of the protocol sub-components (parties) advance in its execution using step to deliver a message.

We describe now the typical sequence of events in a real-world execution; the ideal-world will become clear when we describe the notion of UC emulation below. When the adversarial environment uses the IO interface to pass input to a computing party, this may trigger the computing party to perform some computations and, in turn, provide inputs to other sub-modules included in the protocol description; in most cases this will correspond to sending a message using an idealized communications channel represented by an ideal functionality.[8] Our convention is that inputs calls do not allow obtaining information back (the return type is unit). This means that any outputs produced by parties need to be *pulled* by the environment with separate calls to outputs. Similarly, when the environment asks a party for an output, the party may perform some computation and call the outputs interface of a hybrid ideal functionality (e.g., to see if a message has been delivered) before returning the output to the environment.

The BACKDOORS interface follows these conventions closely. The backdoor method allows the environment to retrieve leakage that may be available for it to collect (e.g., the public part of a party's state, or a buffered message in an authenticated channel). The step procedure allows the environment to pass control to any module inside the protocol; this is important to make sure that the environment always has full control of the liveness of the execution model and can schedule the execution of the various processes at will whenever there are several possible lines of execution.

UC emulation. The central notion to Universal Composability is called UC-emulation, which is a relation between two protocols π_1 and π_2: if π_1 UC-emulates π_2 with small advantage ϵ then π_1 can replace π_2 in any context (within a complexity class).

Definition 5.1 (UC emulation). Protocol π_1 UC emulates π_2 under complexity restrictions c_{sim} and c_{env} and advantage bound ϵ if

$$\exists \mathcal{S} \in \tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}}}, \forall \mathcal{Z} \in \tau_{\mathsf{env}}^{\pi_1,\pi_2,\mathcal{S},c_{\mathsf{env}}},$$
$$|\,\Pr[\,\mathcal{Z}(\pi_1)\ :\top\,] - \Pr[\,\mathcal{Z}(\langle \pi_2 \parallel \mathcal{S}(\pi_2)\rangle)\ :\top\,]\,| \le \epsilon$$

We write this as $\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\pi_1,\pi_2) \le \epsilon$.

The first probability term corresponds to the event that the environment returns true in the real-world execution model described above, i.e., in game UC_emul parametrized with ENV $= \mathcal{Z}$ and P $= \pi_1$. The second probability term corresponds to the equivalent event in the ideal-world (or reference) execution model where, as shown in Figure 11 (right), π_2 is typically an ideal functionality; this corresponds to game UC_emul parametrized with ENV $= \mathcal{Z}$ and a protocol P that results from attaching $\mathcal{S}$ to the BACKDOORS interface of π_2. We denote this ideal-world P by $\langle \pi_2 \parallel \mathcal{S}(\pi_2)\rangle$, corresponding to the EasyCrypt functor CompS also shown in Figure 10.

UC-emulation imposes that a simulator $\mathcal{S}$ is capable to *fool* any environment by presenting a view that is fully consistent with the real-world, while learning only what the BACKDOORS interface of π_2 allows. If such a simulator exists, then clearly π_2 cannot be worse than π_1 in the information it reveals to the environment via

[7] Ideal-world backdoors are used to weaken the security requirements and are usually tailored to bring the security definition down to a level that can be met by real-world protocols. Note that the definition of meaningful ideal functionalities is a crucial aspect of UC security theory; here we just provide a mechanism that permits formalizing such definitions in EasyCrypt.

[8] Such real-world settings where ideal functionalities are used as sub-components are called *hybrid*.

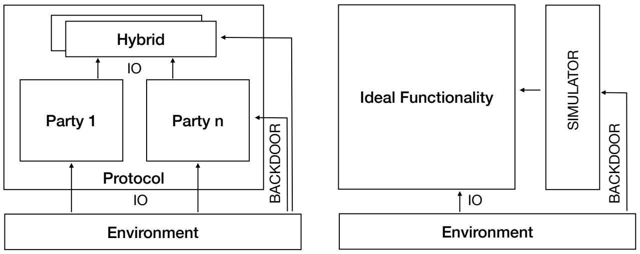


Figure 11: Module restrictions. Arrows indicate ability to make procedure calls via the interface specified as a label; all other cross-boundary memory access is disallowed.

its BACKDOORS interface.[9] Our UC-emulation definition quantifies over simulators and environments using types that give a full characterization of their use of resources, including the ability to access memory, number and types of procedure calls and intrinsic computational costs. The memory access restrictions are depicted in Figure 11, and they can be easily matched to the standard restrictions in the UC framework. Not shown are the cost restrictions, which give explicit bounds for the resources used by various parts of the execution model; these are crucial for obtaining, not only a meaningful definition, but also for obtaining meaningful reductions to computational assumptions, as will be seen below.

Let us examine the types of $\mathcal{Z}$ and $\mathcal{S}$ in more detail. We first note that the definition of emulation is parametric in the resource restrictions c_{sim} and c_{env}. Clearly the IO interface of π_2 must match the type of the IO interface of π_1, which is consistent with the goal that π_1 can replace π_2 in any context, and this is enforced by our type system. This need not be the case for the BACKDOORS interface and, in fact, if π_2 is an ideal functionality, the BACKDOORS interface in the ideal world is of a different nature altogether than the one in the real world: it specifies leakage and adversarial control that are unavoidable even when the functionality is executed by a trusted third-party on behalf of the parties. The type of the simulator $\mathcal{S}$ is given by $\tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}}}$, which defines the type of modules that has access to the BACKDOORS interface of π_2, exposes the BACKDOORS interface of π_1 and is restricted memory-wise to exclude the memory of π_2 and resource-wise by c_{sim}. Note that, if $\mathcal{S}$ could *look inside* the ideal functionality, then it would know all the information that is also given to the real-world protocol: a trivial simulator would always exist and the definition would be meaningless because all protocols would be secure. The type of the environment is given by $\tau_{\mathsf{env}}^{\pi_1,\pi_2,\mathcal{S},c_{\mathsf{env}}}$, the type of modules that have oracle access to the IO and BACKDOORS interfaces of π_1, and are restricted memory-wise to exclude the memories of π_1, π_2 and $\mathcal{S}$, and resource-wise by c_{env}. In this case, if the environment could *look inside* π_1, π_2 or $\mathcal{S}$ it could directly detect with which world it is interacting, and no protocol would be secure. For concreteness, the cost restriction on the type of the environment imposed by c_{env} is of the form:

$$c_{\mathsf{env}} := \mathsf{compl}[\mathsf{intr} : c_1, \pi.\mathsf{inputs} : c_2, \pi.\mathsf{outputs} : c_3, \pi.\mathsf{backdoor} : c_4, \pi.\mathsf{step} : c_5]$$

where type refinements can set c_i to depend on the types of other modules in the context.

Warm-up: Transitivity of UC emulation. It is easy to show that UC-emulation is a transitive relation: if π_1 UC-emulates π_2 and this, in turn, UC-emulates π_3, then π_1 UC-emulates π_3. When stating this lemma in EasyCrypt as follows we move the existential quantifications over the simulators in the hypotheses to global universal quantifications; this logically equivalent formulation allows us to refer to the memory of these simulators when quantifying over all adversarial environments in the consequence: we quantify only over those that cannot *look inside* the simulators that are assumed to exist by hypothesis, which is a natural (and necessary) restriction. In other examples we use the same approach. The lemma is stated in EasyCrypt as follows (we adapt the $\mathsf{Adv}^{\mathsf{uc},\mathcal{S}}_{\cdot,\cdot}(\cdot,\cdot)$ notation by indicating the universally quantified simulator $\mathcal{S}$ in superscript).

Lemma 5.1 (Transitivity). *For all $\epsilon_{1,2}, \epsilon_{2,3} \in \mathbb{R}^+$, all protocols π_1, π_2 and π_3 s.t. the IO interfaces of all three protocols are of the same type, all cost restrictions $c_{\mathsf{sim}(1,2)}$, $c_{\mathsf{sim}(2,3)}$ and all simulators $\mathcal{S}_{1,2} \in \tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}(1,2)}}$, $\mathcal{S}_{2,3} \in \tau_{\mathsf{sim}}^{\pi_2,\pi_3,c_{\mathsf{sim}(2,3)}}$, we have that:*

$$\mathsf{Adv}^{\mathsf{uc},\mathcal{S}_{1,2}}_{c_{\mathsf{sim}(1,2)},\hat{c}_{\mathsf{env}(1,2)}}(\pi_1,\pi_2) \le \epsilon_{1,2} \Rightarrow \mathsf{Adv}^{\mathsf{uc},\mathcal{S}_{2,3}}_{c_{\mathsf{sim}(2,3)},\hat{c}_{\mathsf{env}(2,3)}}(\pi_2,\pi_3) \le \epsilon_{2,3} \Rightarrow \mathsf{Adv}^{\mathsf{uc}}_{\hat{c}_{\mathsf{sim}(1,3)},c_{\mathsf{env}(1,3)}}(\pi_1,\pi_3) \le \epsilon_{1,2} + \epsilon_{2,3}$$

where $\hat{c}_{\mathsf{sim}(1,3)}$ corresponds to the cost of sequentially composing $\mathcal{S}_{1,2}$ and $\mathcal{S}_{2,3}$, $\hat{c}_{\mathsf{env}(2,3)}$ must allow for an adversarial environment that results from converting a distinguisher between π_1 and π_3 in $c_{\mathsf{env}(1,3)}$ and composing it with $\mathcal{S}_{1,2}$, and $\hat{c}_{\mathsf{env}(1,2)} = c_{\mathsf{env}(1,3)}$.

In the statement of the lemma we use notation $\hat{c}$ to denote the fact that these cost restrictions are fixed as a function of the costs of other algorithms: intuitively, the cost of the environment in the consequence is free and it constrains the costs of environments in the hypotheses; then, if for some cost restrictions $c_{\mathsf{sim}(1,2)}$ and $c_{\mathsf{sim}(2,3)}$ the hypotheses hold, these in turn fix the cost of the simulator we give as a witness. This pattern is observable in the remaining examples we give in this section.

The proof gives a witness simulator $\mathcal{S}_{1,3} = \mathsf{SeqS}(\mathcal{S}_{2,3}, \mathcal{S}_{1,2})$ that results from plugging together the two simulators implied by the assumptions: intuitively, $\mathcal{S}_{2,3}$ is able to interact with π_3 and emulate the BACKDOORS of π_2, and this is sufficient to enable $\mathcal{S}_{1,2}$ to emulate the BACKDOORS interface of π_1, as required. Technically, the proof shows first that one can break down $\mathcal{S}_{1,3}$ and put π_2 in the place of $\mathsf{CompS}(\pi_3, \mathcal{S}_{2,3})$. To show this, we aggregate $\mathcal{S}_{1,2}$ into the environment to construct a new environment that would break π_2 if such a modification was noticeable, contradicting the second hypothesis. The proof then follows by applying the first hypothesis. Note that this proof strategy is visible in the resources used by $\mathcal{S}_{1,3}$, since they are those required to run the composed module

[9] The emulation notions in [14, 15] quantify over a restricted class of *balanced* environments. Intuitively, such environments must be *fair* to the simulator in that polynomial-time execution in the size of its inputs is comparable to the execution time of the real-world adversary. Without this restriction, the definition would require the existence of a simulator that uses much less resources than the real-world attacker, which makes the definition too strong. Balanced environments guarantee that the resources given to the simulator match those given to the real-world adversary; moreover, the dummy adversary is formally explicit in the real-world to enable this resource accounting. In our setting we deal with this issue differently: the EasyCrypt resource model is concrete, which means that one can explicitly state in the security definition which resources are used by the simulator and assess what this means in terms of protocol security. We refer the interested reader to [14, Section 4.4] for a discussion of quantitative UC definitions such as the one we adopt. For this reason we also do not need to keep the dummy adversary explicitly in the real world.

$\mathsf{SeqS}(\mathcal{S}_{2,3}, \mathcal{S}_{1,2})$. Moreover, the quantification over the resources of the environments in the second hypothesis must accommodate an environment that *absorbs* simulator $\mathcal{S}_{1,2}$ and runs it internally.

In Appendix A we give a more elaborate example of the properties of UC emulation definition, by showing that our formalization inherits an important property from the general UC framework: that including an explicit adversary in the real world that colludes with an arbitrary environment to break the protocol leads to an equivalent definition to the one we have, which assumes an (implicit) dummy adversary that just follows the instructions of the adversarial environment. Moreover, in our setting with concrete costs, this is equivalent to our execution model where the dummy adversary is implicit.

Universal Composability. The fundamental theorem of Universal Composability is stated in our EasyCrypt formalization as follows.

THEOREM 5.2 (UNIVERSAL COMPOSABILITY). *For all $\epsilon_\rho, \epsilon_\pi \in \mathbb{R}^+$, all ideal functionalities $f, \mathcal{F}$, all protocols $\rho(f)$ and π, such that the IO interfaces of π and f (resp. ρ and $\mathcal{F}$) are of the same type, all cost restrictions $c_{\mathsf{sim}(\rho)}$, $c_{\mathsf{sim}(\pi)}$, and all simulators $\mathcal{S}_\rho \in \tau_{\mathsf{sim}}^{\rho(f),\mathcal{F},c_{\mathsf{sim}(\rho)}}$ and $\mathcal{S}_\pi \in \tau_{\mathsf{sim}}^{\pi,f,c_{\mathsf{sim}(\pi)}}$, we have:*

$$\mathsf{Adv}^{\mathsf{uc},\mathcal{S}_\pi}_{c_{\mathsf{sim}(\pi)},\hat{c}_{\mathsf{env}(\pi)}}(\pi, f) \le \epsilon_\pi \Rightarrow \mathsf{Adv}^{\mathsf{uc},\mathcal{S}_\rho}_{c_{\mathsf{sim}(\rho)},\hat{c}_{\mathsf{env}(\rho)}}(\rho(f), \mathcal{F}) \le \epsilon_\rho$$
$$\Rightarrow \mathsf{Adv}^{\mathsf{uc}}_{\hat{c}_{\mathsf{sim}},c_{\mathsf{env}}}(\rho(\pi), \mathcal{F}) \le \epsilon_\rho + \epsilon_\pi$$

where $\hat{c}_{\mathsf{env}(\pi)}$ accommodates an environment c_{env} that internally uses c_{env} resources and additionally runs ρ, $\hat{c}_{\mathsf{sim}}$ corresponds to the cost of composing $\mathcal{S}_\pi$ and $\mathcal{S}_\rho$, $\hat{c}_{\mathsf{env}(\rho)}$ allows for an adversarial environment built by composing $\mathcal{S}_\pi$ with an environment in c_{env}.

This theorem establishes that any protocol $\rho(f)$ that UC-emulates a functionality $\mathcal{F}$ when relying on an ideal sub-component f offers the same level of security when it is instantiated with a protocol π that UC-emulates f. The proof first shows that the simulator $\mathcal{S}_\pi$ that exists by hypothesis can be converted into a simulator that justifies that $\rho(\pi)$ UC-emulates $\rho(f)$: intuitively this new simulator uses $\mathcal{S}_\pi$ when interacting with the backdoors of f and just passes along the environment's interactions with the backdoors of ρ. This part of the proof combines any successful environment $\mathcal{Z}$ against the composed protocol into a successful environment that absorbs ρ and breaks π. This justifies the cost restriction on c_{env}. Then, we know by hypothesis that $\rho(f)$ UC emulates $\mathcal{F}$, and the result follows by applying the transitivity lemma, which also explains the remaining cost restrictions.

Example: Composing key exchange with encryption. We conclude this section with an example of the use of our framework and general lemmas stated above for concrete protocols. Consider the code snippets in Figure 12. On the left we show the inner structure of a two-party protocol formalization (Diffie-Hellman) when one assumes an ideal sub-component (in this case a bi-directional ideal authenticated channel F2Auth exposing IO interface Pi.REAL.IO). The full real-world configuration is obtained by applying a functor CompRF that composes this protocol with F2Auth and exposes the backdoors of both DHKE and F2Auth in a combined BACKDOORS interface. The IO interface to this real-world protocol is simply the input/output interface for both parties; parties take as input a role (initiator/responder) and the identities of parties involved in the protocol (type unit pkg); they output a session key when the protocol completes. On the right-hand side of Figure 12 we give an example ideal functionality for a simple one-shot unidirectional authenticated channel; one party provides input with the party identities and the message to transmit (type msg pkg), and the other party can obtain the message if it calls outputs with matching identities (type unit pkg.) The attacker can use the BACKDOORS interface to observe the state of the channel, including the transmitted message and the party identities and control when the message is delivered.

The example starts with a proof that the Diffie-Hellman protocol from Figure 12 UC-emulates the ideal functionality for key exchange shown in Figure 13 in an hybrid-real world where the parties have access to authenticated channels. The FKE functionality runs internally a state machine that waits for both parties to provide input, and allows an adversary/simulator interacting with its BACKDOORS interface to control when the different parties obtain a fresh shared secret key. This result is stated as follows; note the accounting of resources spent by the combined Diffie-Hellman attacker, making it explicit that the DDH assumption must be valid for such an attacker.

LEMMA 5.3 (SECURITY OF DHKE). *Fix $c_{\mathsf{ddh}} \in \mathbb{R}^+$ and let ϵ_{DDH} be the maximum advantage of any DDH attacker against the group over which we implement DHKE. Then, we have that*

$$\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim(DHKE)}},c_{\mathsf{env(DHKE)}}}(DHKE(F2Auth), FKE) \le \epsilon_{\mathsf{DDH}}$$

where $c_{\mathsf{sim(DHKE)}}$ is the cost of a concrete simulator $\mathcal{S}_{\mathsf{DHKE}}$ that just samples random group elements as the protocol messages and mimics the states of the real-world parties and F2Auth; $c_{\mathsf{env(DHKE)}}$ must be such that c_{ddh} accommodates the cost of an adversary that runs internally the entire UC emulation experiment (including the environment) and interpolates between the real and ideal worlds, depending on the external DDH challenge.

The second result shows that the ideal functionality for key exchange can be combined with one-time-pad encryption to transform a one-shot authenticated channel into a one-shot secure channel that also guarantees confidentiality. Formally:

LEMMA 5.4 (SECURITY OF OTP). *Fix any $c_{\mathsf{env(OTP)}}$. Then we have*

$$\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim(OTP)}},c_{\mathsf{env(OTP)}}}(OTP(FKE, FAuth), FSC) = 0$$

where $c_{\mathsf{sim(OTP)}}$ is the cost of a concrete simulator $\mathcal{S}_{\mathsf{OTP}}$ that just samples a random string in place of the ciphertext and mimics the states of the real-world parties, FKE and FAuth.

Here, FSC represents the secure channel ideal functionality, which operates exactly as Fauth, but does not leak the transmitted message; leakage includes only information on the state of the channel. The protocol runs in an hybrid world where it has access to both FKE and Fauth, uses the former to obtain a shared key between the two parties, and then transmits the one-time-padded message using Fauth. We apply our Universal Composability theorem to derive that FKE can be replaced by the DHKE protocol, resulting in a protocol that still UC-emulates the secure channel functionality. The final theorem is stated as follows.

THEOREM 5.5 (SECURITY OF OTP COMPOSED WITH DHKE). *Fix $c_{\mathsf{ddh}} \in \mathbb{R}^+$ and let ϵ_{DDH} be the maximum advantage of any DDH*

```
module (DHKE : RHO) (F2Auth: Pi.REAL.IO) = {
  module Initiator = { · · · }
  module Responder = { · · · }
  proc init() : unit = { Initiator.init(); Responder.init(); }
  proc inputs(r : role, p : unit pkg) : unit = {
    if (r = I) { Initiator.inputs(p); }
    else { Responder.inputs(p); } }
  proc outputs(r : role) : group option = { · · · }
  proc step(r : role) : unit = { · · · }
  proc backdoor(r : role) : unit option = {
    var rr;
    if (r = I) { rr ← Initiator.backdoor(); }
    else { rr ← Responder.backdoor(); }
    return rr; }}.
```

```
module FAuth : PROTOCOL = {
  var st : state
  proc init() : unit = { st ← init_st; }
  proc inputs(r : role, p : msg pkg) : unit = {
    st ← set_msg st r p;
  }
  proc outputs(r : role, p : unit pkg) : msg option = {
    return get_msg st r p;
  }
  proc step() : unit = {
    st ← unblock st; }
  proc backdoor() : leakage option = {
    return leak st;
  }}.
```

Figure 12: Examples of a real-world protocol (left) and an ideal-world protocol (right). Left: structure of a Diffie-Hellman protocol defined in a hybrid world where it has access to an ideal functionality Auth for authenticated communication between the parties (one shot each way). Right: ideal functionality for one-shot unidirectional authenticated communication FAuth.

```
module FKE : PROTOCOL = {
  var st : state
  proc init() : unit = { k <$ gen; st ← init k; }
  proc inputs(r : role, p : unit pkg) : unit = {
      st ← party_start st r p; }
  proc outputs(r : role) : key option = {
      return party_output st r; }
  proc step() : unit = { st ← unblock st; }
  proc backdoor() : leakage option = { return leak st; }}.
```

Figure 13: Key Exchange functionality.

attacker against the group over which we implement DHKE. Then

$$\mathrm{Adv}^{\mathrm{uc}}_{c_{\mathrm{sim}}, c_{\mathrm{env}}}(OTP(DHKE, FAuth), FSC) \leq \epsilon_{\mathrm{DDH}}$$

where c_{env} is constrained so that $c_{\mathrm{env(DHKE)}}$ accommodates an environment that internally uses c_{env} resources and additionally runs OTP, and c_{sim} corresponds to the cost of composing $\mathcal{S}_{\mathrm{OTP}}$ and $\mathcal{S}_{\mathrm{DHKE}}$.

The crucial application of the complexity restrictions is visible in the attacker against the DDH assumption, which now has a more complex structure that results from the application of the composition theorem: for this application of composition to be meaningful, it is crucial that the global environment is computationally bounded (even though the OTP protocol is information-theoretically secure) as a function of c_{ddh}, as otherwise the reduction to DDH would be meaningless. Indeed, the class of DDH attackers must allow for the extra resources required to run a simulation of OTP protocol in the reduction. Note also that the execution time of the global simulator is given by $\mathcal{S}_{\mathrm{OTP}}$ and $\mathcal{S}_{\mathrm{DHKE}}$, which are very efficient; this means that the UC emulation result has a small simulation overhead [15, 16].

For the proof we used an auxiliary lemma, which is a specialization of the Universal Composability theorem for the case where the hybrid functionality is the parallel composition of two ideal functionalities and we apply the Universal Composability theorem to instantiate only one of them.

Our formalization vs EasyUC. Our Diffie-Hellman example is an alternative formalization of the example given by [18] for the EasyUC framework. We borrow it because, as was the case in [18], it is a very good toy example with which to validate and demonstrate our formalization. Our goal is *not* to compare ourselves with [18], whose goals are clearly different from ours: a design goal of EasyUC is to follow the UC execution model as closely as possible, whereas in our work we simplify the model in order to take advantage of the EasyCrypt machinery. In particular, our approach has an impact in the way one writes down protocol descriptions and ideal functionalities. In fact, the two approaches should be seen as complementary: one can think of the EasyUC approach as a front-end for cryptographers, and our approach as a convenient back-end for conducting the machine-checked proofs. We leave it as an interesting future work to develop a sound translation between the two UC models for a representative class of protocols such as those considered in [17].

6 RELATED WORK

We focus on related work in computer-aided cryptography. We do not consider work on cost analyses and modules and on type-and-effect systems, as our work is only superficially related to them.

CryptoVerif [12] is an automated tool for computational security proofs. CryptoVerif uses approximate equivalences to find (or check) cryptographic reductions, and keeps track of the complexity of adversaries. Most other tools for computational security proofs, including CertiCrypt [9], Foundational Cryptography Framework [22], and CryptHOL [10], share their foundations and overall approach with EasyCrypt. However, these tools offer limited support for complexity reasoning and they do not support the use of modules for defining cryptographic schemes and notions. This is not a fundamental limitation, since these tools are embedded in a general-purpose proof assistant. However, extending these tools to achieve similar effects than our type-and-effect module system and program logic for complexity would represent a significant endeavour.

Our module system is inspired from EasyCrypt [6, 8]. However, the EasyCrypt module system lacks complexity restrictions, which hampers the use of compositional approaches. Beyond EasyCrypt, several other tools and approaches use structures similar to modules for formalizing cryptographic schemes and their security. Computational Indistinguishability Logic (CIL) [5] rely on oracle systems, which are very closely related to our modules. Interestingly, the main judgment of CIL establishes the approximate equivalence of two oracle systems, and is explicitly quantified by the resources of an adversary. State-separating proofs [13] pursue similar goals, using a notion of package. Packages have the expressivity of modules, but additionally support private functions. Our modules can emulate private functions using restrictions. At present, there is no tool support for state-separating proofs. [23] introduces the notion of interface, which is similar to module, for formalizing cryptography.

7 CONCLUSION

We have developed an extension of the EasyCrypt proof assistant to support reasoning complexity claims. The extension captures reductionist statements that faithfully match the cryptographic literature and supports compositional reasoning. As a main example, we have shown how to formalize key results from Universal Composability, a long-standing goal of computer-aided cryptography.

REFERENCES

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Grégoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. 2019. A Machine-Checked Proof of Security for AWS Key Management Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 63–78. https://doi.org/10.1145/3319535.3354228

[2] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1607–1622. https://doi.org/10.1145/3319535.3363211

[3] anonymous authors. [n. d.]. Source code of our Easycrypt extention, and proof artifacts. ([n. d.]). https://www.dropbox.com/s/mdmgpab4cfv4t56/easycrypt-2021-01-20.tar.bz2?dl=0.

[4] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *42th IEEE Symposium on Security and Privacy (S&P 2021)*. IEEE Computer Society. To appear.

[5] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, and Yassine Lakhnech. 2010. Computational indistinguishability logic. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 375–386. https://doi.org/10.1145/1866307.1866350

[6] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.), Vol. 8604. Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[7] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy (LIPIcs)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.), Vol. 55. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 107:1–107:15. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

[8] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings (Lecture Notes in Computer Science)*, Phillip Rogaway (Ed.), Vol. 6841. Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. New York, 90–101.

[10] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *J. Cryptology* 33, 2 (2020), 494–566. https://doi.org/10.1007/s00145-019-09341-z

[11] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby (Eds.). ACM, 62–73. https://doi.org/10.1145/168588.168596

[12] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 140–154. https://doi.org/10.1109/SP.2006.1

[13] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III (Lecture Notes in Computer Science)*, Thomas Peyrin and Steven D. Galbraith (Eds.), Vol. 11274. Springer, 222–249. https://doi.org/10.1007/978-3-030-03332-3_9

[14] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. (2000). https://eprint.iacr.org/2000/067.

[15] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. IEEE Computer Society, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[16] Ran Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145.

[17] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2015. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Advances in Cryptology – CRYPTO 2015*, Rosario Gennaro and Matthew Robshaw (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–22.

[18] R. Canetti, A. Stoughton, and M. Varia. 2019. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 167–16716.

[19] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. 2018. Computer-Aided Proofs for Multiparty Computation with Active Security. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 119–131. https://doi.org/10.1109/CSF.2018.00016

[20] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.* 2005 (2005), 181. http://eprint.iacr.org/2005/181

[21] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: a calculus for composable, computational cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 640–654. https://doi.org/10.1145/3314221.3314607

[22] Adam Petcher and Greg Morrisett. 2015. A Mechanized Proof of Security for Searchable Symmetric Encryption. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, Cédric Fournet, Michael W. Hicks, and Luca Viganò (Eds.). IEEE Computer Society, 481–494.

[23] Mike Rosulek. 2020. *The Joy of Cryptography.*

[24] Asankhaya Sharma, Shengyi Wang, Andreea Costea, Aquinas Hobor, and Wei-Ngan Chin. 2015. Certified Reasoning with Infinity. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings (Lecture Notes in Computer Science)*, Nikolaj Bjørner and Frank S. de Boer (Eds.), Vol. 9109. Springer, 496–513. https://doi.org/10.1007/978-3-319-19249-9_31

A THE DUMMY ADVERSARY IN UC

The standard notion of UC emulation [14, 15] enriches the real-world with an explicit adversary $\mathcal{A}$ representing an attacker that has access to the real-world BACKDOORS interface and colludes with the environment to break the protocol. In this case, the real- and ideal- world execution models become structurally identical, in that the environment interacts with the BACKDOORS interface via adversarial entities in both worlds.[10] The order of the quantifiers in the emulation definition is crucial for its compositional properties: it requires that, for all adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that, for all environments $\mathcal{Z}$, the real- and ideal- worlds are indistinguishable. We now show that the same result holds in our setting.

Consider the functor in Figure 15, which extends any real-world protocol with abstract adversary $\mathcal{A}$ (A in EasyCrypt notation) at its BACKDOORS interface. The type of $\mathcal{A}$ is parametric in the BACKDOORS offered by the protocol in our basic execution model, and it fixes the type of the BACKDOORS interface in the extended execution model NONDUMMY.PROTOCOL. This means that when we

[10] For this reason the simulator is often called an *ideal world adversary*; we do not adopt this terminology here to avoid confusion.

Expressions *(distribution expressions are similar)*:

$$e ::= v \in \mathcal{V} \quad \text{(variable)}$$
$$\mid f(e_1, \ldots, e_n) \quad (\text{if } f_{|n} \in \mathcal{F}_\mathsf{E})$$

Statements:

$$\mathsf{s} ::= \textbf{abort} \quad \text{(abort)}$$
$$\mid \textbf{skip} \quad \text{(skip)}$$
$$\mid \mathsf{s}_1; \mathsf{s}_2 \quad \text{(sequence)}$$
$$\mid x \leftarrow e \quad \text{(assignment)}$$
$$\mid x \overset{\$}{\leftarrow} d \quad \text{(sampling)}$$
$$\mid x \leftarrow \textbf{call } \mathsf{F}(\vec{e}\,) \quad \text{(procedure call)}$$
$$\mid \textbf{if } e \textbf{ then } \mathsf{s}_1 \textbf{ else } \mathsf{s}_2 \quad \text{(conditional)}$$
$$\mid \textbf{while } e \textbf{ do } \mathsf{s} \quad \text{(loop)}$$

Procedure body:

$$\mathsf{body} ::= \{\ \mathsf{var}\ (\vec{v} : \vec{\tau});\ \mathsf{s};\ \mathsf{return}\ e\ \}$$

Module paths:

$$\mathsf{p} ::= \mathsf{x} \quad \text{(mod. ident.)}$$
$$\mid \mathsf{p.x} \quad \text{(mod. comp.)}$$
$$\mid \mathsf{p(p)} \quad \text{(func. app.)}$$

Function paths:

$$\mathsf{F} ::= \mathsf{p}.f \quad \text{(proc. lookup)}$$

Module expressions:

$$\mathsf{m} ::= \mathsf{p} \quad \text{(mod. path)}$$
$$\mid \mathsf{struct\ st\ end} \quad \text{(structure)}$$
$$\mid \mathsf{func(x : M)\ m} \quad \text{(functor)}$$

Module structures:

$$\mathsf{st} ::= \mathsf{d}_1; \ldots; \mathsf{d}_n \quad (\text{for any } n \in \mathbb{N})$$

Module declarations:

$$\mathsf{d} ::= \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \rightarrow \tau_r = \mathsf{body} \quad \text{(proc.)}$$
$$\mid \mathsf{module\ x = m} \quad \text{(module)}$$

Signature structures:

$$\mathsf{S} ::= \mathsf{D}_1; \ldots; \mathsf{D}_n \quad (\text{for any } n \in \mathbb{N})$$

Module signature declarations:

$$\mathsf{D} ::= \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \rightarrow \tau_r \quad \text{(procedure decl.)}$$
$$\mid \mathsf{module\ x : M} \quad \text{(mod. decl.)}$$

Module signatures:

$$\mathsf{M} ::= \mathsf{sig\ S\ restr}\ \theta\ \mathsf{end} \quad \text{(restr. sig. struct.)}$$
$$\mid \mathsf{func(x : M)\ M'} \quad \text{(functor)}$$

Typing environment declarations:

$$\delta ::= \mathsf{var}\ v : \tau \quad \text{(variable decl.)}$$
$$\mid \mathsf{module\ p = m : M} \quad \text{(module decl.)}$$
$$\mid \mathsf{module\ x = abs_K : M} \quad \text{(abs. mod.)}$$
$$\mid \mathsf{proc\ p}.f(\vec{v} : \vec{\tau}) \rightarrow \tau_r = \mathsf{body} \quad \text{(proc. decl.)}$$

Abstract module kind:

$$\mathsf{K} ::= \mathsf{open} \quad \text{(open module)}$$
$$\mid \mathsf{param} \quad \text{(module parameter)}$$

Typing Environment:

$$\Gamma ::= \epsilon \mid \Gamma, \delta$$

Module restrictions:

$$\theta \ ::= \epsilon \mid \theta, (f : \lambda)$$
$$\lambda \ ::= \top \mid \lambda_\mathsf{m} \wedge \lambda_\mathsf{c}$$

Memory restrictions:

$$\lambda_\mathsf{m} ::= +\mathsf{all\ mem} \backslash \{v_1, \ldots, v_l\} \quad (\text{for any } l \in \mathbb{N})$$
$$\mid \{v_1, \ldots, v_l\} \quad (\text{for any } l \in \mathbb{N})$$

Complexity restrictions:

$$\lambda_\mathsf{c} ::= \top \mid \mathsf{compl}[\mathsf{intr} : k, \mathsf{x}_1.f_1 : k_1, \ldots, \mathsf{x}_l.f_l : k_l]$$
$$(\text{for any } l, k, k_1, \ldots, k_l \in \mathbb{N})$$

Low-order module signatures:

$$\mathsf{M_l} ::= \mathsf{sig\ S_l\ restr}\ \theta\ \mathsf{end} \quad \text{(restr. sig. struct.)}$$
$$\mid \mathsf{func(x : sig\ S_l\ restr}\ \theta\ \mathsf{end)\ M_l} \quad \text{(low-order func.)}$$

Low-order signature structures:

$$\mathsf{S_l} ::= \mathsf{D_{l}}_1; \ldots; \mathsf{D_{l}}_n \quad (\text{for any } n \in \mathbb{N})$$

Low-order module signature declarations:

$$\mathsf{D_l} ::= \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \rightarrow \tau_r \quad \text{(procedure decl.)}$$

Extended module expressions:

$$\bar{\mathsf{m}} ::= \mathsf{m} \mid \mathsf{abs_K\ x}$$

Figure 14: Syntax of statements and modules.

quantify over such adversaries, we quantify also over the potential forms of environment-to-adversary information exchange. We prove the following theorem showing that we do not lose generality by working with an (implicit) dummy adversary in our execution model.

Theorem A.1 (Dummy Adversary). *UC emulation is equivalent to UC emulation with an explicit real-world adversary. More precisely, the following implications hold.*

- *Emulation with an implicit dummy adversary implies emulation with an explicit arbitrary adversary: For all* $\epsilon \in \mathbb{R}^+$*,*

```
module type ADV(B : BACKDOORS) = {
  include NONDUMMY.BACKDOORS
}.

module A_PROTOCOL(A : ADV, P : PROTOCOL)
                  : NONDUMMY.PROTOCOL = {
  proc init() : unit = {
    P.init();
  }
  include P [inputs, outputs]
  include A(P) [step,backdoor]
}.
```

Figure 15: Real-world protocol + adversary.

all protocols π_1 and π_2 with IO interfaces of the same type, all complexity restrictions c_{sim}, c_{env} and all simulators $\mathcal{S} \in \tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}}}$, we have

$$\mathsf{Adv}^{\mathsf{uc},\mathcal{S}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\pi_1,\pi_2) \leq \epsilon \Rightarrow \\ \forall \mathcal{A} \in \tau_{adv}, \mathsf{Adv}^{\mathsf{uc}}_{\hat{c}_{\mathsf{sim}},c_{\mathsf{env}}}(\langle \pi_1 \parallel \mathcal{A}(\pi_1)\rangle, \pi_2) \leq \epsilon$$

where $\hat{c}_{\mathsf{sim}}$ allows for a simulator $\mathcal{S}'$ that combines adversary $\mathcal{A}$ and simulator $\mathcal{S}$.

- *Emulation with an implicit dummy adversary is implied by emulation with an explicit arbitrary adversary: For all $\epsilon \in \mathbb{R}^+$, all protocols π_1 and π_2 with IO interfaces of the same type, all complexity restrictions c_{sim}, c_{env} and all simulator memory spaces $\mathcal{M}$, we have*

$$\forall \mathcal{A} \in \tau_{adv}, \mathsf{Adv}^{\mathsf{uc},\mathcal{M}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\langle \pi_1 \parallel \mathcal{A}(\pi_1)\rangle, \pi_2) \leq \epsilon \Rightarrow \\ \mathsf{Adv}^{\mathsf{uc},\mathcal{M}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\pi_1,\pi_2) \leq \epsilon$$

where τ_{adv} accommodates the dummy adversary.

Our proof gives a simulator $\mathcal{S}'$ for the first part of the theorem that joins together simulator $\mathcal{S}$ and adversary $\mathcal{A}$: intuitively the new simulator uses the existing one to fool the (non-dummy) real-world adversary into thinking it is interacting with the real-world protocol and, in this way, it can offer the expected BACKDOORS view generated by $\mathcal{A}$ to the environment. The resources used by this new simulator are therefore those required to run the composition of $\mathcal{S}$ and $\mathcal{A}$. The proof of the second part of the theorem is more interesting: we construct an explicit dummy adversary and use this to instantiate the hypothesis and obtain a simulator for this adversary, which we then show must also work when the dummy adversary is only implicit: this second step is an equivalence proof showing that, if the simulator matches the explicit dummy adversary which just passes information along, then it is also good when the environment is calling the protocols' BACKDOORS interface directly. The resulting simulator is therefore guaranteed to belong to the same cost-annotated type over which we quantify existentially in the hypothesis.

We note a technicality in the statement of the second part of the theorem: since the hypothesis quantifies over adversaries before quantifying existentially over simulators, we cannot use the approach adopted in the transitivity proof and in the first part of the theorem, where we use global universal quantifications over hypothesized simulators. Instead, we quantify globally over a memory space $\mathcal{M}$, restrict simulators in the hypothesis to only use this memory space, and prevent other algorithms to interfere with this memory space where appropriate (we abuse notation by indicating $\mathcal{M}$ in $\mathsf{Adv}^{\mathsf{uc}}$ to denote this).

B TYPING RULES

To help the reader, we give a summary of all the syntactic categories of our programming language and module system in Figure 14.

B.1 Program and Module Typing

We now present the core rules of our module type system, which are summarized in Figure 16 and Figure 17. The rest of the rules is postponed to Appendix B.2. For clarity of presentation, our module type system requires module paths to always be long modules paths, from the root of the program to the sub-module called (we give a simple example in Figure 18). This allows to have a simpler module resolution mechanism, by removing any scoping issues. This is done without loss of generality: in practice, one can always replace short module paths with long module paths when parsing a program.

A typing environment Γ is a list of typing declarations. A typing declaration, denoted δ, is either a variable, module, abstract module or procedure declaration, with a type.

$$\delta ::= \mathsf{var}\ v : \tau \mid \mathsf{module\ p = m : M} \mid \mathsf{module\ x} = \mathsf{abs_K} : \mathsf{M} \\ \mid \mathsf{proc\ p}.f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}$$

$$\mathsf{K} ::= \mathsf{open} \mid \mathsf{param} \qquad\qquad \Gamma ::= \epsilon \mid \Gamma, \delta$$

Note that module and procedure declarations can be rooted at an arbitrary path p.

An abstract module declaration module $\mathsf{x} = \mathsf{abs_K} : \mathsf{M}$ states that x is a module with signature M whose code is unknown. This is used either for open code, or to represent a functor parameter at typing time. Open modules and parameters are treated differently by the type system: memory restriction ignore the memory footprint of a functor parameter; and a complexity restriction restricts the number of calls that can be made to parameter's procedure. Therefore, we annotate an abstract module with its kind, which can be open or param. Finally, module and procedure declarations come with the absolute path from the root of the program to the parent module where the declaration is made (variable and abstract modules are always declared at top-level).

For example, the entry (module $\mathsf{p.x = m : M}$) means that there is a sub-module m named x and with type M declared at path p. As usual we require that typing environments do not contain two declarations with the same path. This allows to see a typing environment Γ as a partial function from variable names v, module paths p or procedure paths $\mathsf{p}.f$ to (base, module, abstract modules or procedure) values and their types.

$$\begin{array}{lcll}
\Gamma(v) & = & \tau & (\text{ if } \Gamma = (\Gamma_1;\ \mathsf{var}\ v : \tau;\ \Gamma_2)) \\
\Gamma(\mathsf{p}) & = & \mathsf{m : M} & (\text{ if } \Gamma = (\Gamma_1;\ \mathsf{module\ p = m : M};\ \Gamma_2)) \\
\Gamma(\mathsf{x}) & = & \mathsf{abs_K}\ \mathsf{x : M} & (\text{ if } \Gamma = (\Gamma_1;\ \mathsf{module\ x} = \mathsf{abs_K} : \mathsf{M};\ \Gamma_2)) \\
\Gamma(\mathsf{p}.f) & = & \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body} & \\
 & & & (\text{ if } \Gamma = (\Gamma_1;\ \mathsf{proc\ p}.f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body};\ \Gamma_2))
\end{array}$$

Module path typing $\Gamma \vdash \mathsf{p} : \mathsf{M}$.

$$\textsc{Name}\quad \frac{\Gamma(\mathsf{p}) = \_ : \mathsf{M}}{\Gamma \vdash \mathsf{p} : \mathsf{M}} \qquad \textsc{Compnt}\quad \frac{\Gamma \vdash \mathsf{p} : \mathsf{sig}\ \mathsf{S}_1;\ \mathsf{module}\ \mathsf{x} : \mathsf{M};\ \mathsf{S}_2\ \mathsf{restr}\ \theta\ \mathsf{end}}{\Gamma \vdash \mathsf{p.x} : \mathsf{M}}$$

$$\textsc{FuncApp}\quad \frac{\Gamma \vdash \mathsf{p} : \mathsf{func}(\mathsf{x} : \mathsf{M}')\ \mathsf{M} \qquad \Gamma \vdash \mathsf{p}' : \mathsf{M}'}{\Gamma \vdash \mathsf{p}(\mathsf{p}') : \mathsf{M}[\mathsf{x} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')]}$$

Module expression typing $\Gamma \vdash_{\mathsf{p}} \mathsf{m} : \mathsf{M}$.
We omit the rules $\Gamma \vdash \mathsf{M}$ *to check that a module signature* M *is well-formed.*

$$\textsc{Alias}\quad \frac{\Gamma \vdash \mathsf{p}_a : \mathsf{M}}{\Gamma \vdash_{\mathsf{p}} \mathsf{p}_a : \mathsf{M}} \qquad \textsc{Struct}\quad \frac{\Gamma \vdash_{\mathsf{p},\theta} \mathsf{st} : \mathsf{S}}{\Gamma \vdash_{\mathsf{p}} \mathsf{struct}\ \mathsf{st}\ \mathsf{end} : \mathsf{sig}\ \mathsf{S}\ \mathsf{restr}\ \theta\ \mathsf{end}}$$

$$\textsc{Func}\quad \frac{\Gamma \vdash \mathsf{M}_0 \qquad \Gamma(\mathsf{x}) \not\downarrow_{\mathsf{undef}} \qquad \Gamma, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{param}} : \mathsf{M}_0 \vdash_{\mathsf{p(x)}} \mathsf{m} : \mathsf{M}}{\Gamma \vdash_{\mathsf{p}} \mathsf{func}(\mathsf{x} : \mathsf{M}_0)\ \mathsf{m} : \mathsf{func}(\mathsf{x} : \mathsf{M}_0)\ \mathsf{M}} \qquad \textsc{Sub}\quad \frac{\Gamma \vdash_{\mathsf{p}} \mathsf{m} : \mathsf{M}_0 \qquad \vdash \mathsf{M}_0 <: \mathsf{M}}{\Gamma \vdash_{\mathsf{p}} \mathsf{m} : \mathsf{M}}$$

Module structure typing $\Gamma \vdash_{\mathsf{p},\theta} \mathsf{st} : \mathsf{S}$.

$$\textsc{ProcDecl}\quad \frac{\begin{array}{c} \mathsf{body} = \{\ \mathsf{var}\ (\vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l});\ \mathsf{s};\ \mathsf{return}\ r\ \} \\ \vec{v}, \vec{v}_\mathsf{l}\ \text{fresh in}\ \Gamma \qquad \Gamma_f = \Gamma, \mathsf{var}\ \vec{v} : \vec{\tau}, \mathsf{var}\ \vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l} \\ \Gamma_f \vdash \mathsf{s} \qquad \Gamma_f \vdash r : \tau_r \qquad \Gamma \vdash \mathsf{body} \rhd \theta[f] \\ \Gamma(\mathsf{p}.f) \not\downarrow_{\mathsf{undef}} \qquad \Gamma, \mathsf{proc}\ \mathsf{p}.f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body} \vdash_{\mathsf{p},\theta} \mathsf{st} : \mathsf{S} \end{array}}{\Gamma \vdash_{\mathsf{p},\theta} (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body};\ \mathsf{st}) : (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r;\ \mathsf{S})}$$

$$\textsc{ModDecl}\quad \frac{\Gamma(\mathsf{p.x}) \not\downarrow_{\mathsf{undef}} \qquad \begin{array}{c} \Gamma \vdash_{\mathsf{p.x}} \mathsf{m} : \mathsf{M} \\ \Gamma, \mathsf{module}\ \mathsf{p.x} = \mathsf{m} : \mathsf{M} \vdash_{\mathsf{p},\theta} \mathsf{st} : \mathsf{S} \end{array}}{\Gamma \vdash_{\mathsf{p},\theta} (\mathsf{module}\ \mathsf{x} = \mathsf{m};\ \mathsf{st}) : (\mathsf{module}\ \mathsf{x} : \mathsf{M};\ \mathsf{S})}$$

$$\textsc{StructEmp}\quad \frac{}{\Gamma \vdash_{\mathsf{p},\theta} \epsilon : \epsilon}$$

Environments typing $\vdash \mathcal{E}$

$$\textsc{EnvEmp}\quad \frac{}{\vdash \epsilon} \qquad \textsc{EnvSeq}\quad \frac{\vdash \mathcal{E} \qquad \mathcal{E} \vdash \delta}{\vdash \mathcal{E}, \delta} \qquad \textsc{EnvVar}\quad \frac{\mathcal{E}(v) \not\downarrow_{\mathsf{undef}}}{\mathcal{E} \vdash \mathsf{var}\ v : \tau}$$

$$\textsc{EnvMod}\quad \frac{\mathcal{E} \vdash_{\mathsf{x}} \mathsf{m} : \mathsf{M} \qquad \mathcal{E}(\mathsf{x}) \not\downarrow_{\mathsf{undef}}}{\mathcal{E} \vdash (\mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M})} \qquad \textsc{EnvAbs}\quad \frac{\mathcal{E} \vdash \mathsf{M}_\mathsf{l} \qquad \mathcal{E}(\mathsf{x}) \not\downarrow_{\mathsf{undef}}}{\mathcal{E} \vdash (\mathsf{module}\ \mathsf{x} = \mathsf{abs}_K : \mathsf{M})}$$

Figure 16: Core typing rules.

and $\Gamma(z) = \mathsf{undef}$ otherwise. Also, we write $\Gamma(z) \not\downarrow_{\mathsf{undef}}$ when $\Gamma(z') = \mathsf{undef}$. for any prefix z' of z.[11]

[11] Meaning that the (variable, module or proc.) path z is not declared by Γ, even through a sub-module or functor application.

Restriction checking $\Gamma \vdash \{\ \mathsf{var}\ (\vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l});\ \mathsf{s};\ \mathsf{return}\ e\ \} \rhd \theta$.

$$\textsc{RestrCheck}\quad \frac{\Gamma \vdash \mathsf{body} \rhd \lambda_\mathsf{m} \qquad \Gamma \vdash \mathsf{body} \rhd \lambda_\mathsf{c}}{\Gamma \vdash \mathsf{body} \rhd \lambda_\mathsf{m} \wedge \lambda_\mathsf{c}} \qquad \textsc{RestrMemExt}\quad \frac{\Gamma \vdash \mathsf{s} \rhd \lambda_\mathsf{m} \qquad \Gamma \vdash e \rhd \lambda_\mathsf{m}}{\Gamma \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ e\ \} \rhd \lambda_\mathsf{m}}$$

$$\textsc{RestrMemS}\quad \frac{\mathsf{mem}_\Gamma(\mathsf{s}) \sqsubseteq \lambda_\mathsf{m}}{\Gamma \vdash \mathsf{s} \rhd \lambda_\mathsf{m}} \qquad \textsc{RestrMemE}\quad \frac{\mathsf{vars}(e) \sqsubseteq \lambda_\mathsf{m}}{\Gamma \vdash e \rhd \lambda_\mathsf{m}} \qquad \textsc{RestrComplTop}\quad \frac{}{\Gamma \vdash \mathsf{body} \rhd \top}$$

$$\textsc{RestrCompl}\quad \frac{\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\} \qquad \vdash \{\psi\}\ r \le t_r \qquad (t + t_r \cdot \mathbb{1}_{\mathsf{conc}}) \le_{\mathsf{compl}} \lambda_\mathsf{c}}{\mathcal{E} \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ r\ \} \rhd \lambda_\mathsf{c}}$$

Notes: *the relation* $\sqsubseteq$ *checks the inclusion of a memory restriction into another, and is defined in Figure 20.*
Also, *$\mathsf{mem}_\Gamma(\mathsf{s})$ computes an over-approximation of a instruction's memory footprint, and is defined in Figure 21.*

Figure 17: Restriction checking rules.

```
module A = {
 module B = { · · · }

 module C = {
  module E = A.B      (* Valid full path *)
  module F = B        (* Invalid path *)
 }}
```

Figure 18: Example of valid and invalid paths.

Abstract modules for open code. Abstract modules representing open code (i.e. with kind open) are restricted to low-order signatures:

$$\mathsf{M}_\mathsf{l} ::= \mathsf{sig}\ \mathsf{S}_\mathsf{l}\ \mathsf{restr}\ \theta\ \mathsf{end} \mid \mathsf{func}(\mathsf{x} : \mathsf{sig}\ \mathsf{S}_\mathsf{l}\ \mathsf{restr}\ \theta\ \mathsf{end})\ \mathsf{M}_\mathsf{l}$$
$$\mathsf{S}_\mathsf{l} ::= \mathsf{D}_{\mathsf{l}1}; \dots; \mathsf{D}_{\mathsf{l}n} \qquad \mathsf{D}_\mathsf{l} ::= \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r$$

Basically, we only allow module structures, or functors whose parameters are module structures. This restriction is motivated by the fact that further generality is not necessary for cryptographic proofs (adversaries and simulations usually return base values, not procedures); and, more importantly, this restriction allows the abstract call rule of our instrumented Hoare logic Abs presented in Figure 7 to remain tractable.

For any M_l, we let $\mathsf{procs}(\mathsf{M}_\mathsf{l}) = \{f_1, \dots, f_n\}$ be the set of procedure names declared in M_l.

Typing module paths. The typing judgment $\Gamma \vdash \mathsf{p} : \mathsf{M}$ states that the module path p refers to a module with type M. Its typing rules, which are given in Figure 16, are standard, except for the functor application typing rule FuncApp:

$$\textsc{FuncApp}\quad \frac{\Gamma \vdash \mathsf{p} : \mathsf{func}(\mathsf{x} : \mathsf{M}')\ \mathsf{M} \qquad \Gamma \vdash \mathsf{p}' : \mathsf{M}'}{\Gamma \vdash \mathsf{p}(\mathsf{p}') : \mathsf{M}[\mathsf{x} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')]}$$

A key point here is that we need to substitute x in the module signature. The substitution function is standard (see Figure 22), except for module restrictions, which are modified as follows:

- a memory restriction restricts the variables that a procedure can access *directly* — however, memory accesses done through functor parameters are purposely not restricted. Hence, when we instantiate a functor parameter x by a module path p′, we must add its memory footprint, which is $\mathsf{mem}_\Gamma(\mathsf{p}')$. This is handled when substituting x in a memory restriction:

$$\lambda_{\mathsf{m}}[\mathsf{x} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')] = \lambda_{\mathsf{m}} \sqcup \mathsf{mem}_\Gamma(\mathsf{p}')$$

- a complexity restriction gives upper bounds on a procedure execution time, and on the number of calls it can make to its functors' parameters. When we instantiate a functor, we remove a functor parameter, and therefore remove the corresponding entries in the complexity restrictions.

$$\begin{aligned}&\mathsf{compl}[\mathsf{intr} : k, \mathsf{y}_1.f_1 : k_1, \dots, \mathsf{y}_l.f_l : k_l][\mathsf{x} \mapsto \_] = \\ &\quad \mathsf{compl}[\mathsf{intr} : k, (\mathsf{y}_1.f_1 : k_1)[\mathsf{x} \mapsto \_], \dots, (\mathsf{y}_l.f_l : k_l)[\mathsf{x} \mapsto \_]]\end{aligned}$$

$$\text{where} \quad (\mathsf{y}.f : k)[\mathsf{x} \mapsto \_] \; = \; \begin{cases}\epsilon & \text{if } \mathsf{y} = \mathsf{x} \\ \mathsf{y}.f : k & \text{otherwise}\end{cases}$$

Also, note that when substituting x into p in p.y, we do not substitute the module component identifier y (essentially, only top-level module names are substituted). Similarly, when we substitute x into p in a module declaration (module y = m), we ignore y.

Other typing rules. The typing judgment for module expressions $\Gamma \vdash_{\mathsf{p}} \mathsf{m} : \mathsf{M}$ states that the module expression m, declared at path p, has type M. Functor are typed by the Func rule. Note that the functor body is typed in an extended typing environment, where the module parameter x has been declared as an abstract module with kind param.

The typing judgment for module structures $\Gamma \vdash_{\mathsf{p},\theta} \mathsf{st} : \mathsf{S}$ is annotated by both the module path of the structure being typed, and the module restriction θ that the structure must verify. Remark that when we type a procedure using ProcDecl, we check that the procedure f's body satisfies the module restriction $\theta[f]$ by requiring that the restriction checking judgment $\Gamma \vdash \mathsf{body} \triangleright \theta[f]$ holds.

The rule RestrMemExt in Figure 17 is more general than the RestrMem rule presented in the body, as it allows typing a memory restriction in any typing environment Γ, not only in an environment $\mathcal{E}$. Crucially, the complexity checking rule RestrCompl is *not* extended to typing environment, because the the cost Hoare judgment $\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\}$ are not defined for typing environment.

Remark B.1. While we could probably extend RestrCompl to allow typing in a *typing environment* Γ, this would complicate a lot the soundness proof of our logic. Indeed, as it stands, we do not need to show closure of Hoare logic derivations under substitution of a module parameter x of type $\mathsf{abs}_{\mathsf{param}} : \mathsf{M}$ by a concrete module m of the same type M (because an environment $\mathcal{E}$ cannot contain a declaration of an abstract module of kind param, only of open modules of kind open, which are never substituted, only instantiated). Instead, we only need to show closure under such substitution for *typing judgment*, which makes the proof simpler.

B.2 Additional Typing Rules

The memory restriction union ⊔, intersection ⊓ and the memory restriction subset ⊑ operations are defined in Figure 20. In Figure 21, we present our sub-typing rules, our typing rules for statements

```
module A = {
 module B = { proc f () : unit = · · · }

 module C = {
  proc g () : unit = · · ·

  proc h () : unit = {
   A.B.f();
   A.C.g();
  }}}
```

Figure 19: Example of procedures typing.

and expressions, and the definition of the function $\mathsf{mem}_\Gamma(\mathsf{p})$ which computes the memory footprint of p in Γ. Note that we need two different rules to type function paths: T-Proc1 does a lookup of the procedure as a component of an already typed module; and T-Proc2 does a lookup of the procedure in the typing environment, in case the procedure is declared in one of the parent modules of the current sub-module being typed (consequently, these modules are not yet fully typed).

Example B.1. Consider the modules and procedures given in Figure 19. When typing h, the typing environment contains one module declaration and one procedure declaration:

$$\begin{aligned}\Gamma = \; &(\mathsf{module\ A.B} = \_ : \mathsf{struct\ proc\ f}() \to \mathsf{unit\ end}); \\ &(\mathsf{proc\ A.C.g}() \to \mathsf{unit} = \dots)\end{aligned}$$

Here, the call to f in h is typed using the T-Proc1 rule, while the call to g is typed using T-Proc2.

B.3 Module Resolution

Our module resolution mechanism, given in Figure 23, allows to evaluate any module expression m in a typing environment Γ (mostly, it takes care of functor applications).

Because a module expression m is evaluated in a typing environment Γ that can contain abstract modules (representing open code or functor parameters), the resolved module $\mathsf{res}_\Gamma(\mathsf{m})$ may not be a module expression according to the syntactic category defined in Figure 14. We let extended module expressions be the elements of the form:

$$\bar{\mathsf{m}} ::= \mathsf{m} \mid \mathsf{abs}_{\mathsf{K}}\ \mathsf{x}$$

Note that it would not make much sense to extend the syntax of module expressions to allow them to contain abstract modules, as abstract modules of kind param are reserved to the type system; and open modules must be introduced at the logical level (in the ambient higher-order logic).

Module resolution. The resolution functions $\mathsf{res}_\Gamma(\_)$ evaluates a module path, in Γ, into a (resolved) extended module expression, which can be a module structure, a functor, or an (potentially applied) abstract module. Mostly, $\mathsf{res}_\Gamma(\_)$ take care of functor application through the rules:

$$\begin{aligned}\mathsf{res}_\Gamma(\mathsf{p}(\mathsf{p}')) \;&=\; \mathsf{res}_\Gamma(\mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}']) && (\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = \mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m}_0) \\ \mathsf{res}_\Gamma(\mathsf{p}(\mathsf{p}')) \;&=\; (\mathsf{abs}_{\mathsf{K}}\ \mathsf{x})(\vec{\mathsf{p}}_0, \mathsf{p}') && (\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = (\mathsf{abs}_{\mathsf{K}}\ \mathsf{x})(\vec{\mathsf{p}}_0))\end{aligned}$$

Memory restriction union $\sqcup$

$$
\begin{aligned}
(+\mathsf{all\ mem}\backslash\{v_1,\dots,v_n\}) \sqcup (+\mathsf{all\ mem}\backslash\{v'_1,\dots,v'_m\}) &= +\mathsf{all\ mem}\backslash(\{v_1,\dots,v_n\} \cap \{v'_1,\dots,v'_m\})\\
\{v_1,\dots,v_n\} \sqcup \{v'_1,\dots,v'_m\} &= \{v_1,\dots,v_n\} \cup \{v'_1,\dots,v'_m\}\\
(+\mathsf{all\ mem}\backslash\{v_1,\dots,v_n\}) \sqcup \{v'_1,\dots,v'_m\} &= +\mathsf{all\ mem}\backslash(\{v_1,\dots,v_n\}\backslash\{v'_1,\dots,v'_m\})
\end{aligned}
$$

Memory restriction intersection $\sqcap$

$$
\begin{aligned}
(+\mathsf{all\ mem}\backslash\{v_1,\dots,v_n\}) \sqcap (+\mathsf{all\ mem}\backslash\{v'_1,\dots,v'_m\}) &= +\mathsf{all\ mem}\backslash(\{v_1,\dots,v_n\} \cup \{v'_1,\dots,v'_m\})\\
\{v_1,\dots,v_n\} \sqcap \{v'_1,\dots,v'_m\} &= \{v_1,\dots,v_n\} \cap \{v'_1,\dots,v'_m\}\\
(+\mathsf{all\ mem}\backslash\{v_1,\dots,v_n\}) \sqcap \{v'_1,\dots,v'_m\} &= \{v'_1,\dots,v'_m\}\backslash\{v_1,\dots,v_n\}
\end{aligned}
$$

Memory restriction subset $\sqsubseteq$

$$
\begin{aligned}
(+\mathsf{all\ mem}\backslash\{v_1,\dots,v_n\}) \sqsubseteq (+\mathsf{all\ mem}\backslash\{v'_1,\dots,v'_m\}) &= \{v'_1,\dots,v'_m\}) \subseteq \{v_1,\dots,v_n\})\\
\{v_1,\dots,v_n\} \sqsubseteq \{v'_1,\dots,v'_m\} &= \{v_1,\dots,v_n\} \subseteq \{v'_1,\dots,v'_m\}\\
(+\mathsf{all\ mem}\backslash\{v_1,\dots,v_n\}) \sqsubseteq \{v'_1,\dots,v'_m\} &= \bot\\
\{v_1,\dots,v_n\} \sqsubseteq (+\mathsf{all\ mem}\backslash\{v'_1,\dots,v'_m\}) &= \{v_1,\dots,v_n\} \cap \{v'_1,\dots,v'_m\}) = \emptyset
\end{aligned}
$$

Figure 20: Memory restriction operations and type erasure functions.

(the full definition is in Figure 23). In the concrete functor case, we must substitute the module identifier x into a path p′ in a module expression m_0.

Example B.2. Consider a typing environment Γ, and the path x.y(z)(v)(w), which must be read as (((x.y)(z))(v))(w). Then, assuming that $\Gamma(\mathsf{z}) = \mathsf{abs}_{\mathsf{open}}\ \mathsf{z}$, $\Gamma(\mathsf{v}) = \mathsf{m_v}$, $\Gamma(\mathsf{w}) = \mathsf{abs}_{\mathsf{param}}\ \mathsf{w}$ and:

$$\Gamma(\mathsf{x}) = \mathsf{struct\ module\ y = func(u : \_)\ u\ end}$$

where $\mathsf{m_v}$ is some module expression, then $\mathsf{res}_\Gamma(\mathsf{x.y(z)(v)(w)}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{z})(\mathsf{v},\mathsf{w})$.

We define the module procedure resolution function $\mathsf{f\text{-}res}_\Gamma(\mathsf{m}.f)$. A resolved module procedure $\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{m}.f)$ is: i) either a concrete procedure declaration ($\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}$); ii) or the procedure component f of a resolved (potentially applied) abstract module $(\mathsf{abs}_{\mathsf{K}}\ \mathsf{x})(\vec{\mathsf{p}}).f$.

Soundness. Then, we need show that our module resolution mechanism has the subject reduction property. Unfortunately, this does not hold, because of sub-modules declarations, as shown in the following example.

Example B.3. Consider a well-typed typing environment $\vdash \Gamma$, and a module path p where:

$$\Gamma \vdash \mathsf{p} : \mathsf{sig}\ \mathsf{S}_1; \mathsf{module\ x : M}; \_\ \mathsf{restr}\ \_\ \mathsf{end}$$

We are going to assume that some kind of subject reduction property holds for p. More precisely, we assume that:

$$\mathsf{f\text{-}res}_\Gamma(\mathsf{p}) = (\mathsf{struct\ st}_1; \mathsf{module\ x = m}; \_\ \mathsf{end})$$

and that we have a derivation:

$$\Gamma \vdash_{\mathsf{p}} \begin{array}{l}\mathsf{struct\ st}_1; \mathsf{module\ x = m}; \_\ \mathsf{end}\ \ : \\ \mathsf{sig}\ \mathsf{S}_1; \mathsf{module\ x : M}; \_\ \mathsf{restr}\ \_\ \mathsf{end}\end{array}$$

Then, we know that p.x resolves to m, i.e. $\mathsf{f\text{-}res}_\Gamma(\mathsf{p}) = \mathsf{m}$. But we do not have:

$$\Gamma \vdash_{\mathsf{p.x}} \mathsf{m : M}$$

The problem is that the sub-module m may use sub-modules declared in st_1. Consequently, it is not well-typed in Γ, but in an extended typing environment, where the sub-module declarations in st_1 (which have types S_1) have been added to Γ. For example, we can have:

$$\mathsf{st}_1 = (\mathsf{module\ z = m_0}) \qquad\qquad \mathsf{m = z}$$

Therefore, we cannot state a subject reduction property for the module resolution function w.r.t. the typing judgment $\Gamma \vdash_{\mathsf{p}} \mathsf{m : M}$.

Instead, we introduce another typing judgment, noted $\Gamma \Vdash \mathsf{m : M}$, which is similar to the typing judgment $\Gamma \vdash_{\mathsf{p}} \mathsf{m : M}$ of Figure 16, but is used to type a module expression in an environment which has already been typed, while $\Gamma \vdash_{\mathsf{p}} \mathsf{m : M}$ is used to type a module declaration in an environment where some modules have not yet been fully typed. We postpone its definition to Appendix C.1. Using this alternative typing judgment notion, we can state the subject reduction property we want (the proof is postponed to Appendix C).

Lemma B.1 (Resolution Soundness). *If $\Gamma \Vdash \mathcal{E}$ and $\Gamma \Vdash m : \mathsf{M}$ then $\Gamma \Vdash res_{\mathcal{E}}(m) : \mathsf{M}$ whenever $res_{\mathcal{E}}(m)$ is well-defined.*

C SUBJECT REDUCTION OF MODULE RESOLUTION

The goal of this section is to prove that the module resolution mechanism of Figure 23 has the subject reduction property (Lemma B.1), which we recall below:

Lemma (Resolution Soundness). *If $\Gamma \Vdash \mathcal{E}$ and $\Gamma \Vdash m : \mathsf{M}$ then $\Gamma \Vdash res_{\mathcal{E}}(m) : \mathsf{M}$ whenever $res_{\mathcal{E}}(m)$ is well-defined.*

The proof is essentially the proof that simply typed λ-calculus has the subject reduction properties, with some consequent additional work to handle module restrictions.

The rest of this section is organized as follows: we define the typing rules for $\Gamma \Vdash \mathsf{m{:}M}$ in Section C.1, and prove the link between $\Gamma \Vdash \mathsf{m : M}$ and $\Gamma \vdash_{\mathsf{p}} \mathsf{m : M}$; we prove that the module resolution procedure has the subject reduction property in Section C.3.

Module signature and structure sub-typing $\vdash \mathsf{M}_1 <: \mathsf{M}_2$ **and** $\vdash \mathsf{S}_1 <: \mathsf{S}_2$.

We omit the reflexivity and transitivity rules.

$$\textsc{SubSig}\quad \frac{\vdash \mathsf{S}_1 <: \mathsf{S}_2 \qquad \vdash \theta_1 <: \theta_2}{\vdash \mathsf{sig}\ \mathsf{S}_1\ \mathsf{restr}\ \theta_1\ \mathsf{end} <: \mathsf{sig}\ \mathsf{S}_2\ \mathsf{restr}\ \theta_2\ \mathsf{end}}$$

$$\textsc{SubFunc}\quad \frac{\vdash \mathsf{M}_0' <: \mathsf{M}_0 \qquad \vdash \mathsf{M} <: \mathsf{M}'}{\vdash \mathsf{func}(\mathsf{x} : \mathsf{M}_0)\ \mathsf{M} <: \mathsf{func}(\mathsf{x} : \mathsf{M}_0')\ \mathsf{M}'}$$

$$\textsc{SubStruct}\quad \frac{\forall i \in \{1; \ldots; n\},\ \vdash \mathsf{D}_i <: \mathsf{D}_i'}{\vdash \mathsf{D}_1;\ \ldots;\ \mathsf{D}_n <: \mathsf{D}_1';\ \ldots;\ \mathsf{D}_n'}$$

$$\textsc{SubModDecl}\quad \frac{\vdash \mathsf{M}_1 <: \mathsf{M}_2}{\vdash \mathsf{module}\ \mathsf{x} : \mathsf{M}_1 <: \mathsf{module}\ \mathsf{x} : \mathsf{M}_2}$$

Statements and function paths typing $\Gamma \vdash \mathsf{s}$ **and** $\Gamma \vdash \mathsf{F} : \_$.

$$\textsc{T-Abort}\quad \frac{}{\Gamma \vdash \mathbf{abort}} \qquad \textsc{T-Skip}\quad \frac{}{\Gamma \vdash \mathbf{skip}} \qquad \textsc{T-Seq}\quad \frac{\Gamma \vdash \mathsf{s}_1 \qquad \Gamma \vdash \mathsf{s}_2}{\Gamma \vdash \mathsf{s}_1; \mathsf{s}_2}$$

$$\textsc{T-Assign}\quad \frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e} \qquad \textsc{T-Rand}\quad \frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash d : \tau}{\Gamma \vdash x \xleftarrow{\$} d}$$

$$\textsc{T-Call}\quad \frac{\Gamma \vdash \mathsf{F} : \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r \qquad \Gamma \vdash x : \tau_r \qquad \Gamma \vdash \vec{e} : \vec{\tau}}{\Gamma \vdash x \leftarrow \mathbf{call}\ \mathsf{F}(\vec{e}\,)}$$

$$\textsc{T-Proc1}\quad \frac{\Gamma \vdash \mathsf{p} : \mathsf{sig}\ (\mathsf{S}_1;\ \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r;\ \mathsf{S}_2)\ \mathsf{restr}\ \theta\ \mathsf{end}}{\Gamma \vdash \mathsf{p}.f : (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r)}$$

$$\textsc{T-Proc2}\quad \frac{\Gamma(\mathsf{p}.f) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \_)}{\Gamma \vdash \mathsf{p}.f : (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r)}$$

$$\textsc{T-If}\quad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash \mathsf{s}_1 \qquad \Gamma \vdash \mathsf{s}_2}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2} \qquad \textsc{T-While}\quad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash \mathsf{s}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ \mathsf{s}}$$

Expressions typing $\Gamma \vdash e : \tau$.

$$\textsc{ExprApp}\quad \frac{\mathsf{type}(f) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \forall i \in \{1; \ldots; n\}, \Gamma \vdash e_i : \tau_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau} \qquad \textsc{ExprVar}\quad \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}$$

Restriction entailment $\vdash \theta <: \theta'$.

We omit the transitivity and reflexivity rules for $\vdash \theta <: \theta'$.

$$\sqsubseteq\textsc{-Proc}\quad \frac{\forall f \in \mathrm{dom}(\theta, \theta'), \vdash \theta[f] <: \theta'[f]}{\vdash \theta <: \theta'} \qquad \sqsubseteq\textsc{-Split}\quad \frac{\vdash \lambda_{\mathrm{m}} <: \lambda_{\mathrm{m}}' \qquad \vdash \lambda_{\mathrm{c}} <: \lambda_{\mathrm{c}}'}{\vdash \lambda_{\mathrm{m}} \wedge \lambda_{\mathrm{c}} <: \lambda_{\mathrm{m}}' \wedge \lambda_{\mathrm{c}}'}$$

$$\sqsubseteq\textsc{-Top}\quad \frac{}{\vdash \lambda <: \top} \qquad \sqsubseteq\textsc{-Mem}\quad \frac{\lambda_{\mathrm{m}} \sqsubseteq \lambda_{\mathrm{m}}'}{\vdash \lambda_{\mathrm{m}} <: \lambda_{\mathrm{m}}'} \qquad \sqsubseteq\textsc{-MemTop}\quad \frac{}{\vdash \lambda_{\mathrm{c}} <: \top}$$

$$\sqsubseteq\textsc{-Compl}\quad \frac{k \leq k' \qquad \forall i, k_i \leq k_i'}{\vdash \mathsf{compl}[\mathsf{intr} : k, \mathsf{x}_1.f_1 : k_1, \ldots, \mathsf{x}_n.f_n : k_n] <: \mathsf{compl}[\mathsf{intr} : k', \mathsf{x}_1.f_1 : k_1', \ldots, \mathsf{x}_n.f_n : k_n']}$$

Memory restriction.

$$\begin{array}{lcl|lcl}
\mathsf{mem}_\Gamma(\mathbf{abort}) & = & \emptyset & \mathsf{mem}_\Gamma(\mathbf{skip}) & = & \emptyset \\
\mathsf{mem}_\Gamma(x \leftarrow e) & = & \{x\} \sqcup \mathsf{vars}(e) & \mathsf{mem}_\Gamma(x \xleftarrow{\$} d) & = & \{x\} \\
\mathsf{mem}_\Gamma(\mathsf{s}_1; \mathsf{s}_2) & = & \mathsf{mem}_\Gamma(\mathsf{s}_1) \sqcup \mathsf{mem}_\Gamma(\mathsf{s}_1) & \mathsf{mem}_\Gamma(\mathbf{while}\ e\ \mathbf{do}\ \mathsf{s}) & = & \mathsf{vars}(e) \sqcup \mathsf{mem}_\Gamma(\mathsf{s})
\end{array}$$

$$\begin{array}{lcl}
\mathsf{mem}_\Gamma(\mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2) & = & \mathsf{vars}(e) \sqcup \mathsf{mem}_\Gamma(\mathsf{s}_1) \sqcup \mathsf{mem}_\Gamma(\mathsf{s}_1) \\
\mathsf{mem}_\Gamma(x \leftarrow \mathbf{call}\ \mathsf{p}.f(\vec{e}\,)) & = & \{x\} \sqcup \mathsf{mem}_\Gamma(\mathsf{p}.f) \sqcup \mathsf{vars}(\vec{e}\,) \\
\mathsf{mem}_\Gamma(\mathsf{p}) & = & \sqcup_{f \in \mathsf{procs}_\Gamma(\mathsf{p})}\ \mathsf{mem}_\Gamma(\mathsf{p}.f)
\end{array}$$

When $\mathsf{f\text{-}res}_\Gamma(\mathsf{p}.f) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\ \mathsf{var}\ (\vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l});\ \mathsf{s};\ \mathsf{return}\ r\ \})$:

$$\mathsf{mem}_\Gamma(\mathsf{p}.f) = (\mathsf{mem}_\Gamma(\mathsf{s}) \cup \mathsf{vars}(r)) \backslash \{\vec{v}; \vec{v}_\mathsf{l}\}$$

When $\mathsf{f\text{-}res}_\Gamma(\mathsf{p}.f) = (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0).f$, $\mathsf{K} = \mathsf{open}$ and $\Gamma(\mathsf{x}) = \mathsf{abs}_\mathsf{K} : \mathsf{func}(\_)\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}$:

$$\mathsf{mem}_\Gamma(\mathsf{p}.f) = \theta[f] \sqcup \mathsf{mem}_\Gamma(\vec{\mathsf{p}}_0)$$

When $\mathsf{f\text{-}res}_\Gamma(\mathsf{p}.f) = (\mathsf{abs}_{\mathsf{param}}\ \mathsf{x})(\vec{\mathsf{p}}_0).f$:

$$\mathsf{mem}_\Gamma(\mathsf{p}.f) = \mathsf{mem}_\Gamma(\vec{\mathsf{p}}_0)$$

Figure 21: Additional typing rules and operations.

Substitution in module signatures and declarations

$$
\begin{aligned}
(\mathsf{func}(\mathsf{y}:\mathsf{M}_0))\ \mathsf{M}_1[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \begin{cases}\mathsf{func}(\mathsf{y}:\mathsf{M}_0[\mathsf{x}\mapsto\lambda_{\mathsf{m}}])\ (\mathsf{M}_1[\mathsf{x}\mapsto\lambda_{\mathsf{m}}]) & \text{when } \mathsf{y}\neq\mathsf{x}\\ \mathsf{func}(\mathsf{y}:\mathsf{M}_0)\ \mathsf{M}_1 & \text{otherwise}\end{cases}\\
(\mathsf{sig}\ \mathsf{S}\ \mathsf{restr}\ \theta\ \mathsf{end})[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \mathsf{sig}\ (\mathsf{S}[\mathsf{x}\mapsto\lambda_{\mathsf{m}}])\ \mathsf{restr}\ (\theta[\mathsf{x}\mapsto\lambda_{\mathsf{m}}])\ \mathsf{end}\\
(\mathsf{D}_1;\dots;\mathsf{D}_n)[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \mathsf{D}_1[\mathsf{x}\mapsto\lambda_{\mathsf{m}}];\dots;\mathsf{D}_n[\mathsf{x}\mapsto\lambda_{\mathsf{m}}]\\
\mathsf{proc}\ f(\vec{v}:\vec{\tau})\to\tau_r[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \mathsf{proc}\ f(\vec{v}:\vec{\tau})\to\tau_r\\
\mathsf{module}\ \mathsf{y}:\mathsf{M}[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \mathsf{module}\ \mathsf{y}:(\mathsf{M}[\mathsf{x}\mapsto\lambda_{\mathsf{m}}])
\end{aligned}
$$

Substitution in module restriction

$$
\begin{aligned}
\big((f_1:\lambda_1);\dots;(f_n:\lambda_n)\big)[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= (f_1:\lambda_1[\mathsf{x}\mapsto\lambda_{\mathsf{m}}]);\dots;(f_n:\lambda_n[\mathsf{x}\mapsto\lambda_{\mathsf{m}}])\\
(\lambda^0_{\mathsf{m}}\wedge\lambda_{\mathsf{c}})[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \lambda^0_{\mathsf{m}}[\mathsf{x}\mapsto\lambda_{\mathsf{m}}]\wedge\lambda_{\mathsf{c}}[\mathsf{x}\mapsto\lambda_{\mathsf{m}}]\\
\top[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] &= \top
\end{aligned}
$$

Substitution in memory restriction

$$\lambda^0_{\mathsf{m}}[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] = \lambda^0_{\mathsf{m}}\sqcup\lambda_{\mathsf{m}}$$

Substitution in complexity restriction

$$\mathsf{compl}[\mathsf{intr}:k,\mathsf{y}_1.f_1:k_1,\dots,\mathsf{y}_l.f_l:k_l][\mathsf{x}\mapsto\lambda_{\mathsf{m}}] = \mathsf{compl}[\mathsf{intr}:k,(\mathsf{y}_1.f_1:k_1)[\mathsf{x}\mapsto\lambda_{\mathsf{m}}],\dots,(\mathsf{y}_l.f_l:k_l)[\mathsf{x}\mapsto\lambda_{\mathsf{m}}]]$$

$$\text{where}\quad (\mathsf{y}.f:k)[\mathsf{x}\mapsto\lambda_{\mathsf{m}}] = \begin{cases}\epsilon & \text{if } \mathsf{y}=\mathsf{x}\\ \mathsf{y}.f:k & \text{otherwise}\end{cases}$$

Substitution in module paths

$$\mathsf{y}[\mathsf{x}\mapsto\mathsf{p}] = \begin{cases}\mathsf{p} & \text{if } \mathsf{y}=\mathsf{x}\\ \mathsf{y} & \text{otherwise}\end{cases}\qquad
\begin{aligned}(\mathsf{p}'.\mathsf{y})[\mathsf{x}\mapsto\mathsf{p}] &= (\mathsf{p}'[\mathsf{x}\mapsto\mathsf{p}]).\mathsf{y}\\ (\mathsf{p}'(\mathsf{p}''))[\mathsf{x}\mapsto\mathsf{p}] &= (\mathsf{p}'[\mathsf{x}\mapsto\mathsf{p}])(\mathsf{p}''[\mathsf{x}\mapsto\mathsf{p}])\end{aligned}$$

Substitution in module expressions

$$(\mathsf{func}(\mathsf{y}:\mathsf{M})\ \mathsf{m})[\mathsf{x}\mapsto\mathsf{p}] = \mathsf{func}(\mathsf{y}:\mathsf{M})\ (\mathsf{m}[\mathsf{x}\mapsto\mathsf{p}])\qquad (\mathsf{struct}\ \mathsf{st}\ \mathsf{end})[\mathsf{x}\mapsto\mathsf{p}] = \mathsf{struct}\ \mathsf{st}[\mathsf{x}\mapsto\mathsf{p}]\ \mathsf{end}$$

Substitution in module structures, declarations and procedure body

$$
\begin{aligned}
(\mathsf{d}_1;\dots;\mathsf{d}_n)[\mathsf{x}\mapsto\mathsf{p}] &= \mathsf{d}_1[\mathsf{x}\mapsto\mathsf{p}];\dots;\mathsf{d}_n[\mathsf{x}\mapsto\mathsf{p}] & (\mathsf{proc}\ f(\vec{v}:\vec{\tau})\to\tau_r=\mathsf{body})[\mathsf{x}\mapsto\mathsf{p}] &= \mathsf{proc}\ f(\vec{v}:\vec{\tau})\to\tau_r=(\mathsf{body}[\mathsf{x}\mapsto\mathsf{p}])\\
(\mathsf{module}\ \mathsf{y}=\mathsf{m})[\mathsf{x}\mapsto\mathsf{p}] &= \mathsf{module}\ \mathsf{y}=(\mathsf{m}[\mathsf{x}\mapsto\mathsf{p}]) & \{\ \mathsf{var}\ (\vec{v}_{\mathsf{l}}:\vec{\tau}_{\mathsf{l}});\ \mathsf{s};\ \mathsf{return}\ e\ \}[\mathsf{x}\mapsto\mathsf{p}] &= \{\ \mathsf{var}\ (\vec{v}_{\mathsf{l}}:\vec{\tau}_{\mathsf{l}});\ \mathsf{s}[\mathsf{x}\mapsto\mathsf{p}];\ \mathsf{return}\ e\ \}
\end{aligned}
$$

Substitution in statements

$$
\begin{aligned}
\mathbf{abort}[\mathsf{x}\mapsto\mathsf{p}] &= \mathbf{abort} & \mathbf{skip}[\mathsf{x}\mapsto\mathsf{p}] &= \mathbf{skip}\\
x\leftarrow e[\mathsf{x}\mapsto\mathsf{p}] &= x\leftarrow e & x\xleftarrow{\$} d[\mathsf{x}\mapsto\mathsf{p}] &= x\xleftarrow{\$} d\\
(\mathsf{s}_1;\mathsf{s}_2)[\mathsf{x}\mapsto\mathsf{p}] &= \mathsf{s}_1[\mathsf{x}\mapsto\mathsf{p}];\mathsf{s}_2[\mathsf{x}\mapsto\mathsf{p}] & x\leftarrow\mathbf{call}\ \mathsf{p}'.f(\vec{e})[\mathsf{x}\mapsto\mathsf{p}] &= x\leftarrow\mathbf{call}\ (\mathsf{p}'[\mathsf{x}\mapsto\mathsf{p}]).f(\vec{e})\\
(\mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2)[\mathsf{x}\mapsto\mathsf{p}] &= \mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1[\mathsf{x}\mapsto\mathsf{p}]\ \mathbf{else}\ \mathsf{s}_2[\mathsf{x}\mapsto\mathsf{p}] & (\mathbf{while}\ e\ \mathbf{do}\ \mathsf{s})[\mathsf{x}\mapsto\mathsf{p}] &= \mathbf{while}\ e\ \mathbf{do}\ \mathsf{s}[\mathsf{x}\mapsto\mathsf{p}]
\end{aligned}
$$

Figure 22: Substitution functions.

C.1 Typing in Typed Environments

During the resolution of a module expression m, we may have to resolve applied module paths of the form p(p′). This is resolved by first evaluating p into a functor or an abstract module, and then perform the application. Consequently, we can have intermediate expressions of the form m(p). Similarly, when resolving a module component access p.x, we first resolve p into a module structure and then access its component x. This yields intermediate expressions of the form m.x.

Example C.1. We recall Example B.2 which we presented in Section 3. We recall that Γ is a typing environment Γ such that:

$$\Gamma(\mathsf{x}) = \mathsf{struct}\ \mathsf{module}\ \mathsf{y} = \mathsf{func}(\mathsf{u}:\_)\ \mathsf{u}\ \mathsf{end}\qquad \Gamma(\mathsf{z}) = \mathsf{abs}_{\mathsf{open}}\ \mathsf{z}$$

$$\Gamma(\mathsf{v}) = \mathsf{m}_{\mathsf{v}}\qquad \Gamma(\mathsf{w}) = \mathsf{abs}_{\mathsf{param}}\ \mathsf{w}$$

where m_{v} is some module expression. The resolution of the module expression x.y(z)(v)(w) in Γ is as follows:

$$
\begin{aligned}
&\mathsf{x.y(z)(v)(w)}\\
&\quad\Rightarrow_{\text{res. step}}\ \big(\mathsf{struct}\ \mathsf{module}\ \mathsf{y} = \mathsf{func}(\mathsf{u}:\_)\ \mathsf{u}\ \mathsf{end}\big).\mathsf{y(z)(v)(w)}\\
&\quad\Rightarrow_{\text{res. step}}\ \big(\mathsf{func}(\mathsf{u}:\_)\ \mathsf{u}\big)\mathsf{(z)(v)(w)}\\
&\quad\Rightarrow_{\text{res. step}}\ \mathsf{z(v)(w)}
\end{aligned}
$$

Module path resolution $\mathsf{res}_\Gamma(\mathsf{p})$ **to module expression**

$$\begin{array}{lcll}
\mathsf{res}_\Gamma(\mathsf{p}) &=& \mathsf{res}_\Gamma(\bar{\mathsf{m}}) & (\text{if } \Gamma(\mathsf{p}) = \bar{\mathsf{m}} : _) \\
\mathsf{res}_\Gamma(\mathsf{p.x}) &=& \mathsf{res}_\Gamma(\mathsf{m}) & \\
&& \multicolumn{2}{r}{(\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = \mathsf{struct}\ \mathsf{st}_1; \mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M}; \mathsf{st}_2\ \mathsf{end})} \\
\mathsf{res}_\Gamma(\mathsf{p}(\mathsf{p}')) &=& \mathsf{res}_\Gamma(\mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}']) & (\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = \mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m}_0) \\
\mathsf{res}_\Gamma(\mathsf{p}(\mathsf{p}')) &=& (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0, \mathsf{p}') & (\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0))
\end{array}$$

Module expression resolution $\mathsf{res}_\Gamma(\bar{\mathsf{m}})$

$$\begin{array}{rcl}
\mathsf{res}_\Gamma(\mathsf{struct}\ \mathsf{st}\ \mathsf{end}) &=& \mathsf{struct}\ \mathsf{st}\ \mathsf{end} \\
\mathsf{res}_\Gamma(\mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m}) &=& \mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m} \\
\mathsf{res}_\Gamma((\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}})) &=& (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}})
\end{array}$$

Module procedure resolution $\mathsf{f\text{-}res}_\Gamma(\mathsf{m}.f)$

(*note that this includes resolution for function paths f-res*$_\Gamma(\mathsf{p}.f)$)

$$\begin{array}{lcl}
\mathsf{f\text{-}res}_\Gamma(\mathsf{p}.f) &=& (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}) \\
&& \qquad (\text{if } \Gamma(\mathsf{p}.f) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body})) \\
\mathsf{f\text{-}res}_\Gamma(\mathsf{m}.f) &=& (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}) \\
&& (\text{if } \mathsf{res}_\Gamma(\mathsf{m}) = \mathsf{struct}\ \mathsf{st}_1; \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}; \mathsf{st}_2\ \mathsf{end}) \\
\mathsf{f\text{-}res}_\Gamma(\mathsf{m}.f) &=& (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}).f \qquad (\text{if } \mathsf{res}_\Gamma(\mathsf{m}) = (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}))
\end{array}$$

Figure 23: Resolution functions for paths, module expressions and module procedure.

$$\begin{array}{ll}
\Rightarrow_{\text{res. step}} & (\mathsf{abs}_{\mathsf{open}}\ \mathsf{z})(\mathsf{v})(\mathsf{w}) \\
\Rightarrow_{\text{res. step}} & ((\mathsf{abs}_{\mathsf{open}}\ \mathsf{z})(\mathsf{v}))(\mathsf{w}) \\
\Rightarrow_{\text{res. step}} & (\mathsf{abs}_{\mathsf{open}}\ \mathsf{z})(\mathsf{v}, \mathsf{w})
\end{array}$$

Therefore, to prove subject reduction for the module resolution procedure, we consider an extended syntax for modules (and add the corresponding typing rules).

Definition C.1. A *partially resolved* module expression $\tilde{\mathsf{m}}$ is an element of the form:

$$\tilde{\mathsf{m}} ::= \bar{\mathsf{m}} \mid \tilde{\mathsf{m}}(\mathsf{p}) \mid \tilde{\mathsf{m}}.\mathsf{x}$$

It is enough to allow only top-level applications of module expressions to module paths and module component accesses.

We give the rules for typing in already typed environment in Figure 24. Roughly, these rules are the module expression, module structure and module environment typing rules given in Figure 16, with the following changes:

- when typing a module declaration of a module structure, the typing environment Γ is not extended with the inferred type, as it must already be present in Γ. Consequently, we do not need to keep track of the current path of the sub-module expression being typed, which simplifies the rules.
- we added a rule for abstract modules, for the application of a module *expression* to a module path, and for component access. This allows to type the intermediate terms that appear during the module resolution procedure.
- when typing an environment, the typing environment is not extended with the inferred types.

Before going further, we give the definition of well-formed typing environments. Essentially, a typing environment Γ is well-formed if it contains no duplicate declarations, and it contains only well-formed module signature.

Module expression typing $\Gamma \Vdash \tilde{\mathsf{m}} : \mathsf{M}$.

$$\textsc{D-Alias}\ \frac{\Gamma \vdash \mathsf{p}_a : \mathsf{M}}{\Gamma \Vdash \mathsf{p}_a : \mathsf{M}} \qquad \textsc{D-Struct}\ \frac{\Gamma \Vdash_\theta \mathsf{st} : \mathsf{S}}{\Gamma \Vdash \mathsf{struct}\ \mathsf{st}\ \mathsf{end} : \mathsf{sig}\ \mathsf{S}\ \mathsf{restr}\ \theta\ \mathsf{end}}$$

$$\textsc{D-Func}\ \frac{\Gamma \vdash \mathsf{M}_0 \qquad \Gamma(\mathsf{x}) \not\downarrow_{\mathsf{undef}} \qquad \Gamma, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{param}} : \mathsf{M}_0 \Vdash \mathsf{m} : \mathsf{M}}{\Gamma \Vdash \mathsf{func}(\mathsf{x} : \mathsf{M}_0)\ \mathsf{m} : \mathsf{func}(\mathsf{x} : \mathsf{M}_0)\ \mathsf{M}}$$

$$\textsc{D-Sub}\ \frac{\Gamma \Vdash \tilde{\mathsf{m}} : \mathsf{M}_0 \qquad \vdash \mathsf{M}_0 <: \mathsf{M}}{\Gamma \Vdash \tilde{\mathsf{m}} : \mathsf{M}} \qquad \textsc{D-AbsEmp}\ \frac{\Gamma \Vdash \mathsf{x} : \mathsf{M}_\mathsf{I}}{\Gamma \Vdash \mathsf{abs}_\mathsf{K}\ \mathsf{x} : \mathsf{M}_\mathsf{I}}$$

$$\textsc{D-Abs}\ \frac{\Gamma \Vdash (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}) : \mathsf{func}(\mathsf{y} : \mathsf{M}')\ \mathsf{M} \qquad \Gamma \Vdash \mathsf{p}' : \mathsf{M}'}{\Gamma \Vdash (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}, \mathsf{p}') : \mathsf{M}[\mathsf{y} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')]}$$

$$\textsc{D-Compnt}\ \frac{\Gamma \Vdash \tilde{\mathsf{m}} : \mathsf{sig}\ \mathsf{S}_1;\ \mathsf{module}\ \mathsf{x} : \mathsf{M};\ \mathsf{S}_2\ \mathsf{restr}\ \theta\ \mathsf{end}}{\Gamma \Vdash \tilde{\mathsf{m}}.\mathsf{x} : \mathsf{M}}$$

$$\textsc{D-FuncApp}\ \frac{\Gamma \Vdash \tilde{\mathsf{m}} : \mathsf{func}(\mathsf{x} : \mathsf{M}')\ \mathsf{M} \qquad \Gamma \vdash \mathsf{p}' : \mathsf{M}'}{\Gamma \Vdash \tilde{\mathsf{m}}(\mathsf{p}') : \mathsf{M}[\mathsf{x} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')]}$$

Module structure typing $\Gamma \Vdash_\theta \mathsf{st} : \mathsf{S}$.

$$\textsc{D-ProcDecl}\ \frac{\begin{array}{c} \mathsf{body} = \{\ \mathsf{var}\ (\vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l});\ \mathsf{s};\ \mathsf{return}\ r\ \} \\ \vec{v}, \vec{v}_\mathsf{l}\ \text{fresh in}\ \Gamma \qquad \Gamma_f = \Gamma, \mathsf{var}\ \vec{v} : \vec{\tau}, \mathsf{var}\ \vec{v}_\mathsf{l} : \vec{\tau}_\mathsf{l} \\ \Gamma_f \vdash \mathsf{s} \qquad \Gamma_f \vdash r : \tau_r \qquad \Gamma \vdash \mathsf{body} \rhd \theta[f] \qquad \Gamma \Vdash_\theta \mathsf{st} : \mathsf{S} \end{array}}{\Gamma \Vdash_\theta (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body};\ \mathsf{st}) : (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r;\ \mathsf{S})}$$

$$\textsc{D-ModDecl}\ \frac{\Gamma \Vdash \mathsf{m} : \mathsf{M} \qquad \Gamma \Vdash_\theta \mathsf{st} : \mathsf{S}}{\Gamma \Vdash_\theta (\mathsf{module}\ \mathsf{x} = \mathsf{m};\ \mathsf{st}) : (\mathsf{module}\ \mathsf{x} : \mathsf{M};\ \mathsf{S})} \qquad \textsc{D-StructEmp}\ \frac{}{\Gamma \Vdash_\theta \epsilon : \epsilon}$$

Environments typing $\Gamma \Vdash \mathcal{E}$

$$\textsc{D-EnvSeq}\ \frac{\Gamma \Vdash \Gamma_1 \qquad \Gamma \Vdash \Gamma_2}{\Gamma \Vdash (\Gamma_1; \Gamma_2)} \qquad \textsc{D-EnvEmp}\ \frac{}{\Gamma \Vdash \epsilon} \qquad \textsc{D-EnvVar}\ \frac{\Gamma(v) = (\mathsf{var}\ v : \tau)}{\Gamma \Vdash (\mathsf{var}\ v : \tau)}$$

$$\textsc{D-EnvMod}\ \frac{\Gamma \Vdash \mathsf{m} : \mathsf{M}}{\Gamma \Vdash (\mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M})} \qquad \textsc{D-EnvAbs}\ \frac{\Gamma \vdash \mathsf{M}_\mathsf{I} \qquad \Gamma(\mathsf{x}) = \mathsf{abs}_\mathsf{K}\ \mathsf{x} : \mathsf{M}_\mathsf{I}}{\Gamma \Vdash (\mathsf{module}\ \mathsf{x} = \mathsf{abs}_\mathsf{K} : \mathsf{M}_\mathsf{I})}$$

Figure 24: Typing rules in typed environments.

Definition C.2. A typing environment Γ is well-formed iff:

- whenever $\Gamma = (\Gamma_0;\ \mathsf{Decl}_\mathsf{z};\ \Gamma_1)$ then $(\Gamma_0; \Gamma_1)(\mathsf{z}) \not\downarrow_{\mathsf{undef}}$, where:

$\mathsf{Decl}_\mathsf{z} \in \left\{\mathsf{module}\ \mathsf{z} = \mathsf{m} : \mathsf{M};\ \mathsf{var}\ \mathsf{z} : \tau;\ \mathsf{proc}\ \mathsf{z}(\vec{v} : \vec{\tau}) \to \tau_r\right\}$

- whenever $\Gamma = \Gamma_0; \mathsf{module}\ \mathsf{x} = \mathsf{m}{:}\mathsf{M}; \Gamma_1$ then M is well-formed in Γ_0, i.e. $\Gamma_0 \vdash \mathsf{M}$.

It is straightforward to check that if Γ is well-typed, then Γ is well-formed.

Proposition C.1. *If* $\vdash \Gamma$ *then* Γ *is well-formed.*

Proof. This is by induction over $\vdash \Gamma$. □

We recall that given a module path p, $\Gamma(\mathrm{p})$ returns the typing declaration corresponding to p in Γ, if it exists. Note that this does not descend in sub-module declarations to retrieve p type. For example, assume that:

$$\Gamma = \big(\text{module p} = \text{struct module y} = \text{m end}: \\ (\text{sig module y} : \text{M} \text{ restr \_ end})\big)$$

then $\Gamma(\mathrm{p}) = \_$:(sig module y:M restr _ end), but $\Gamma(\mathrm{p.y})$ is not defined. We introduce a new notion of lookup in a typing environment, $\Gamma[\mathrm{p}]$, which descends in sub-module declarations. It also descend below functor definitions, in the case where the functor argument is a module identifier. Continuing the example above, we have $\Gamma[\mathrm{p.y}] = \mathrm{m} : \mathrm{M}$. Formally:

Definition C.3. For every well-formed typing environment Γ and module path p, we let:

$$\begin{array}{lcll} \Gamma[\mathrm{p}] & = & \mathrm{m} : \mathrm{M} & (\text{if } \Gamma(\mathrm{p}) = \mathrm{m} : \mathrm{M}) \\ \Gamma[\mathrm{p}(\mathrm{x})] & = & \mathrm{m} : \mathrm{M} & (\text{if } \Gamma[\mathrm{p}] = \text{func}(\mathrm{x} : \mathrm{M}_0)\ \mathrm{m} : \ \ \text{func}(\mathrm{x} : \mathrm{M}_0)\ \mathrm{M}) \end{array}$$

where the functor rule is module alpha-renaming. Moreover, if:

$$\Gamma[\mathrm{p}] = (\text{struct \_; module x} = \mathrm{m};\ \text{\_ end}) \\ :\ (\text{sig (\_; module x} : \mathrm{M};\ \text{\_) restr \_ end})$$

then:

$$\Gamma[\mathrm{p.x}] = \mathrm{m} : \mathrm{M}$$

The following proposition allows to replace a typing environment Γ_0 by Γ_1, as long as they coincide w.r.t. _[p] on every p such that $\Gamma_0[\mathrm{p}]$ is well-defined.

Proposition C.2. *For every well-formed environments* Γ_0 *and* Γ_1, *if* $\Gamma_0[\mathrm{p}] = \Gamma_1[\mathrm{p}]$ *for every* p *such that* $\Gamma_0[\mathrm{p}]$ *is well-defined, then for any* m *and* M, *if* $\Gamma_0 \Vdash m : \mathrm{M}$ *then* $\Gamma_1 \Vdash m : \mathrm{M}$

Proof. This is immediate by induction over the typing derivations $\Gamma_0 \Vdash \mathrm{m} : \mathrm{M}$. □

Example C.2. The following two typing environments Γ_0 and Γ_1 verify the proposition hypothesis:

$$\Gamma_0 = (\text{module p.x} = \mathrm{m} : \mathrm{M};\ \text{module p.y} = \mathrm{m}' : \mathrm{M}')$$

and:

$$\Gamma_1 = \Big(\text{module p} = (\text{struct module x} = \mathrm{m}; \text{module y} = \mathrm{m}' \text{ end}) \\ : (\text{sig module x} : \mathrm{M}; \text{module y} : \mathrm{M}' \text{ restr \_ end})\Big)$$

We can strengthen the context.

Proposition C.3. *For any well-formed environment* Γ, *if* (Γ, Γ_1) *is well-formed then* $\Gamma; \Gamma_1 \Vdash m : \mathrm{M}$ *whenever* $\Gamma \Vdash m : \mathrm{M}$.

Proof. This is straightforward by induction over the typing derivation $\Gamma \Vdash \mathrm{m} : \mathrm{M}$. □

We can replaying a typing derivation if we have a finer type for a module declaration in the context.

Proposition C.4. *For any well-formed environment* Γ, *if*

$$\Gamma, \text{module p} = m : \mathrm{M}, \Gamma_1 \Vdash m_0 : \mathrm{M}_0$$

then for any $\vdash \mathrm{M}' <: \mathrm{M}$, *we have*

$$\Gamma, \text{module p} = m : \mathrm{M}', \Gamma_1 \Vdash m_0 : \mathrm{M}_0$$

Proof. We show this by induction over the typing derivation $\Gamma, \text{module p} : \mathrm{M}, \Gamma_1 \Vdash \mathrm{m} : \mathrm{M}_0$, by replacing any application of the D-Alias typing rule on a path starting by p, i.e. of the form $\mathrm{p}(\mathrm{p}_0, \dots, \mathrm{p}_n).\mathrm{p}'$, by an application of D-Alias on p, followed by D-Sub and by the applications of D-FuncApp and D-Compnt to replay the path typing derivation of $\Gamma \vdash \mathrm{p}(\mathrm{p}_0, \dots, \mathrm{p}_n).\mathrm{p}' : \mathrm{M}$ as a module expression typing derivation of $\Gamma \Vdash \mathrm{p}(\mathrm{p}_0, \dots, \mathrm{p}_n).\mathrm{p}'$:M. □

Lemma C.5. *For every well-formed typing environment* Γ *such that* $\Gamma(\mathrm{p}) \not\downarrow_{undef}$*:*

- *if* $\Gamma \vdash_{\mathrm{p}} m : \mathrm{M}$ *then there exists* M' *well-formed in* Γ *such that* $\vdash \mathrm{M}' <: \mathrm{M}$ *and:*

$$\Gamma; \text{module p} = m : \mathrm{M}' \Vdash m : \mathrm{M}' \tag{1}$$

- *if* $\Gamma \vdash_{\mathrm{p},\theta} (\mathrm{d}_1, \dots, \mathrm{d}_n)$:$(\mathrm{D}_1, \dots, \mathrm{D}_n)$ *then there exists* $\mathrm{D}'_1, \dots, \mathrm{D}'_n$ *well-formed such that* $\vdash \mathrm{D}'_i <: \mathrm{D}_i$ *for every i, and:*

$$\Gamma; \delta_1, \dots, \delta_n \Vdash_\theta (\mathrm{d}_1, \dots, \mathrm{d}_n) : (\mathrm{D}_1, \dots, \mathrm{D}_n)$$

where for every i:

$$\delta_i = \text{module p.x} = m : \mathrm{M}$$

if $\mathrm{D}'_i = (\text{module x} : \mathrm{M})$ *and* $\mathrm{d}_i = (\text{module x} = m)$*, and*

$$\delta_i = \text{proc p.}f(\vec{v} : \vec{\tau}) \longrightarrow \tau_r = \text{body}$$

if $\mathrm{d}_i = (\text{proc } f(\vec{v} : \vec{\tau}) \longrightarrow \tau_r = \text{body})$.

Proof. The proof is by induction over the typing derivations in hypothesis.

- the Alias and StructEmp cases are immediate. Note that we use the hypothesis $\Gamma(\mathrm{p}) \not\downarrow_{\mathrm{undef}}$ in the Alias case, to ensure that the path lookup in Γ is still well-defined.
- we show the Struct case by applying the induction hypothesis, and using Proposition C.2.
- for ModDecl we have $\mathrm{st} = (\text{module x} = \mathrm{m}_0;\ \mathrm{st}_0)$, and $\mathrm{S} = (\text{module x} : \mathrm{M}_0;\ \mathrm{S}_0)$, and a derivation of the form:

ModDecl

$$\frac{\Gamma(\mathrm{p.x}) \not\downarrow_{\mathrm{undef}} \qquad \begin{array}{c} \Gamma \vdash_{\mathrm{p.x}} \mathrm{m}_0 : \mathrm{M}_0 \\ \Gamma, \text{module p.x} = \mathrm{m}_0 : \mathrm{M}_0 \vdash_{\mathrm{p},\theta} \mathrm{st}_0 : \mathrm{S}_0 \end{array}}{\Gamma \vdash_{\mathrm{p},\theta} (\text{module x} = \mathrm{M}_0;\ \mathrm{st}_0) : (\text{module x} : \mathrm{M}_0;\ \mathrm{S}_0)}$$

We know that $\mathrm{st}_0 = (\text{struct } \mathrm{d}_1, \dots, \mathrm{d}_n \text{ end})$ and:

$$\mathrm{S}_0 = (\text{sig } \mathrm{D}_1, \dots, \mathrm{D}_n \text{ restr \_ end})$$

By applying the induction hypothesis twice, we know that there are derivations of:

$$\Gamma, \text{module p.x} = \mathrm{m}_0 : \mathrm{M}'_0 \Vdash \mathrm{m}_0 : \mathrm{M}'_0 \tag{2}$$

$$\Gamma, \text{module p.x} = \mathrm{m}_0 : \mathrm{M}_0, \Gamma'' \Vdash_\theta (\mathrm{d}_1, \dots, \mathrm{d}_n) : (\mathrm{D}'_1, \dots, \mathrm{D}'_n) \tag{3}$$

where $\vdash \mathsf{M}_0' <: \mathsf{M}_0$, and $\Gamma'' = (\mathsf{D}_1'', \dots, \mathsf{D}_n'')$ where for every i, $\vdash \mathsf{D}_i' <: \mathsf{D}_i$ and:

$$\mathsf{D}_i'' = \begin{cases} \mathsf{module\ p.y = m : M} \\ \quad (\text{if } \mathsf{D}_i' = (\mathsf{module\ y : M}) \text{ and } \mathsf{d}_i = (\mathsf{module\ y = m})) \\ \mathsf{proc\ p.}f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body} \\ \quad (\text{if } \mathsf{d}_i = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body})) \end{cases}$$

By Proposition C.3, we deduce from Eq. (2) that there is a derivation of:

$$\Gamma, \mathsf{module\ p.x = m_0 : M_0'}, \Gamma'' \Vdash \mathsf{m_0 : M_0'}$$

By Proposition C.4, we deduce from Eq. (3) that there is a derivation of:

$$\Gamma, \mathsf{module\ p.x = m_0 : M_0'}, \Gamma'' \Vdash_\theta (\mathsf{d}_1, \dots, \mathsf{d}_n) : (\mathsf{D}_1', \dots, \mathsf{D}_n')$$

We conclude by applying the D-ModDecl typing rule.

- the ProcDecl case is similar to the ModDecl case. We omit the details.
- for Sub, we have:

$$\textsc{Sub}\ \frac{\Gamma \vdash_\mathsf{p} \mathsf{m : M_0} \qquad \vdash \mathsf{M_0 <: M}}{\Gamma \vdash_\mathsf{p} \mathsf{m : M}}$$

By induction hypothesis, we have a derivation of Γ, module p: $\mathsf{M}_0' \Vdash \mathsf{m} : \mathsf{M}_0'$ where $\vdash \mathsf{M}_0' <: \mathsf{M}_0$. Using the sub-typing transitivity rule, we have that $\vdash \mathsf{M}_0' <: \mathsf{M}$. This concludes this case.

- for Func we have $\mathsf{m = func(x : M_0)\ m'}$ and $\mathsf{M = func(x : M_0)\ M'}$, and the derivation:

$$\textsc{Func}\ \frac{\Gamma(\mathsf{x}) \not\downarrow_{\mathsf{undef}} \qquad \begin{array}{c}\Gamma \vdash \mathsf{M_0} \\ \Gamma, \mathsf{module\ x = abs_{param} : M_0} \vdash_{\mathsf{p(x)}} \mathsf{m' : M'}\end{array}}{\Gamma \vdash_\mathsf{p} \mathsf{func(x : M_0)\ m' : func(x : M_0)\ M'}}$$

By induction hypothesis, we have $\mathsf{M''}$ such that $\vdash \mathsf{M'' <: M'}$ and:

$$\Gamma, \mathsf{module\ x = abs_{param} : M_0}, \mathsf{module\ p(x) = m' : M''} \Vdash \mathsf{m' : M''}$$

is derivable. We re-order the declarations in the context, to obtain a derivation of:

$$\Gamma, \mathsf{module\ p(x) = m' : M''}, \mathsf{module\ x = abs_{param} : M_0} \Vdash \mathsf{m' : M''}$$

(note that the new typing environment is still well-formed). By applying the D-Func rule, there is a derivation of:

$$\Gamma, \mathsf{module\ p(x) = m' : M''} \Vdash \mathsf{func(x : M_0)\ m' : func(x : M_0)\ M''}$$

By Proposition C.2, there is a derivation of:

$$\Gamma, \mathsf{module\ p = (func(x : M_0)\ m') : (func(x : M_0)\ M'')} \Vdash \mathsf{func(x : M_0)\ m' : func(x : M_0)\ M''} \quad \square$$

We extend sub-typing judgement to typing environment, by requiring that $\vdash \Gamma_0 <: \Gamma_1$ whenever Γ_0 and Γ_1 are of the same length, and all declarations in Γ_0 are a subtype of the corresponding declaration in Γ_1.

Lemma C.6. *If* $\vdash \Gamma$ *then there exists* $\vdash \Gamma' <: \Gamma$ *such that* $\Gamma' \Vdash \Gamma$.

Proof. We use Lemma C.5 to replace the EnvMod rules by D-EnvMod rules. We omit the details. □

p	$\to_\Gamma$	m	if $\Gamma(\mathsf{p}) = \mathsf{m} : \_$
$\mathsf{m.x}$	$\to_\Gamma$	$\mathsf{m'}$	if $\mathsf{m = struct\ st_1; module\ x = m'; st_2\ end}$
$\mathsf{m.x}$	$\to_\Gamma$	$\mathsf{m'.x}$	if $\mathsf{m} \to_\Gamma \mathsf{m'}$
$\mathsf{m(p)}$	$\to_\Gamma$	$\mathsf{m_0[x \mapsto p]}$	if $\mathsf{m = func(x : M)\ m_0}$ and x is fresh in $\mathcal{E}$
$\tilde{\mathsf{m}}(\mathsf{p})$	$\to_\Gamma$	$(\mathsf{abs_K\ x})(\vec{\mathsf{p}}_0, \mathsf{p})$	if $\tilde{\mathsf{m}} = (\mathsf{abs_K\ x})(\vec{\mathsf{p}}_0)$
$\mathsf{m(p)}$	$\to_\Gamma$	$\mathsf{m'(p)}$	if $\mathsf{m} \to_\Gamma \mathsf{m'}$

Figure 25: The relation $\to_\Gamma$.

C.2 Module Resolution as a Rewrite Relation

We define in Figure 25 the relation $\to_\Gamma$ on partially resolved module expressions. This relation is exactly the evaluation strategy used by the module resolution procedure defined in Figure 23.

We let $\tilde{\mathsf{m}} \downarrow_\Gamma$ be the normal form of $\tilde{\mathsf{m}}$ w.r.t. the reflexive and transitive closure of $\to_\Gamma$, if it exists.

We now focus on properties of $\to_\Gamma$ in the case where Γ is an *environment*, i.e. top-level module declarations are not module aliases.

Proposition C.7. *The relation* $\to_\Gamma$ *is deterministic on well-formed module expression: for every well-formed* $\tilde{m}$, *if* $\tilde{m} \to_\Gamma \tilde{m}_0$ *and* $\tilde{m} \to_\Gamma \tilde{m}_1$ *then* $\tilde{m}_0 = \tilde{m}_1$.

Moreover, for any well-formed $\mathcal{E}$ *and* $\tilde{m}$, *if* $res_\mathcal{E}(\tilde{m})$ *is well-defined then* $res_\mathcal{E}(\tilde{m}) = \tilde{m} \downarrow_\mathcal{E}$.

Proof. The fact that $\to_\Gamma$ is deterministic is straightforward from the definition of $\to_\Gamma$ in Figure 25: indeed, we only need to observe that well-formed module expressions cannot have two module declarations with the same name.

Moreover, for any $\tilde{\mathsf{m}}_1$ and $\tilde{\mathsf{m}}_2$, if $\tilde{\mathsf{m}}_1 \to_\Gamma \tilde{\mathsf{m}}_2$ and $\tilde{\mathsf{m}}_1$ is well-formed then so is $\tilde{\mathsf{m}}_2$. Hence if $\to_\Gamma$ terminates on a well-formed $\tilde{\mathsf{m}}$ then $\tilde{\mathsf{m}} \downarrow_\Gamma$ exists.

Let $\mathcal{E}$ be a well-typed environment. We show the second point by induction over the length of the computation of $\mathrm{res}_\mathcal{E}(\tilde{\mathsf{m}})$. In the environment lookup case, we use the fact that $\mathcal{E}(\mathsf{p})$ is either a module structure, functor or an abstract module, and is therefore in $\to_\mathcal{E}$-normal form. We omit the rest of the proof. □

C.3 Subject Reduction

Before proving that our system has the subject reduction property, we state the following substitution lemmas.

Lemma C.8 (Substitution 1). *We have the following substitution properties:*

- *if* $\vdash \theta_1 <: \theta_2$ *then* $\vdash \theta_1[\mathsf{x} \mapsto \lambda_\mathsf{m}] <: \theta_2[\mathsf{x} \mapsto \lambda_\mathsf{m}]$.
- *if* $\vdash \lambda_1 <: \lambda_2$ *then* $\vdash \lambda_1[\mathsf{x} \mapsto \lambda_\mathsf{m}] <: \lambda_2[\mathsf{x} \mapsto \lambda_\mathsf{m}]$.
- *if* $\vdash \mathsf{M}_0 <: \mathsf{M}_0'$ *then* $\vdash \mathsf{M}_0[\mathsf{x} \mapsto \lambda_\mathsf{m}] <: \mathsf{M}_0'[\mathsf{x} \mapsto \lambda_\mathsf{m}]$.

Proof. The properties above are shown by induction over their respective typing derivation. □

Lemma C.9 (Substitution 2). *For every:*

$$\Gamma_1;\ \mathsf{module\ x} = \mathsf{abs}_{param} : \mathsf{M};\ \Gamma_2$$

which is a well-formed typing environment, if:

$$\Gamma_1;\ \mathsf{module\ x} = \mathsf{abs}_{param} : \mathsf{M};\ \Gamma_2 \vdash e : \tau$$

then:

$$\Gamma_1;\ \Gamma_2[\mathsf{x} \mapsto \lambda_{\mathsf{m}}] \vdash e : \tau$$

Proof. The proof is immediate by induction over the typing derivation, since module types in the typing environment are not used when typing expressions. □

Proposition C.10. *For any well-formed typing environment* Γ, *if* $\Gamma \vdash \mathsf{p} : \mathsf{M}$ *and* $mem_\Gamma(\mathsf{p}) = \lambda_{\mathsf{m}}$ *then:*

$$mem_\Gamma(\mathsf{F}[\mathsf{x} \mapsto \mathsf{p}]) \subseteq \left(mem_{\Gamma, \mathsf{module\ x} = \mathsf{abs}_{param} : \mathsf{M}}(\mathsf{F})\right)[\mathsf{x} \mapsto \lambda_{\mathsf{m}}]$$

Lemma C.11 (Substitution 3). *For every well-formed environment* Γ *of the form:*

$$\Gamma_1;\ \mathsf{module\ x} = \mathsf{abs}_{param} : \mathsf{M};\ \Gamma_2$$

for every module path p *such that* $\Gamma_1 \vdash \mathsf{p} : \mathsf{M}$, *if* $mem_{\Gamma_1}(\mathsf{p}) = \lambda_{\mathsf{m}}$ *then:*

- *if* $\Gamma \Vdash m : \mathsf{M}_0$ *then:*

$$\Gamma_1;\ \Gamma_2[\mathsf{x} \mapsto \lambda_{\mathsf{m}}] \Vdash m[\mathsf{x} \mapsto \mathsf{p}] : \mathsf{M}_0[\mathsf{x} \mapsto \lambda_{\mathsf{m}}]$$

- *if* $\Gamma \vdash \{\ var\,(\vec{v}_{\mathsf{l}} : \vec{\tau}_{\mathsf{l}});\ \mathsf{s};\ \mathsf{return}\ e\ \} \triangleright \lambda$ *then:*

$$\Gamma_1;\ \Gamma_2[\mathsf{x} \mapsto \lambda_{\mathsf{m}}] \vdash \{\ var\,(\vec{v}_{\mathsf{l}} : \vec{\tau}_{\mathsf{l}});\ \mathsf{s}[\mathsf{x} \mapsto \mathsf{p}];\ \mathsf{return}\ e\ \} \triangleright \lambda[\mathsf{x} \mapsto \lambda_{\mathsf{m}}]$$

Proof. We prove the two properties above simultaneously, by induction over the corresponding typing derivations. The proof is straightforward (we omit the details).

Note that we do not need to show closure under substitution of module parameters in *Hoare* derivations, only in typing derivations (see Remark B.1). □

Lemma C.12. *If* $\Gamma \Vdash \mathcal{E}$ *and* $\Gamma \Vdash m : \mathsf{M}$ *then if* $m \to_\Gamma m'$ *then* $\Gamma \Vdash m' : \mathsf{M}$.

Proof. W.l.o.g., we assume that the typing derivations $\Gamma \Vdash \mathcal{E}$ and $\Gamma \Vdash \mathsf{m} : \mathsf{M}$ never apply the D-Sub typing rules twice in a row (using the transitivity rule for sub-typing judgments). To simplify derivations, we also assume that D-Sub is applied once between each typing rule application (using the sub-typing reflexivity rule if necessary).

We do a case analysis on the reduction $\mathsf{m} \to_\Gamma \mathsf{m}'$.

- if $\mathsf{x} \to_\Gamma \mathcal{E}(\mathsf{x})$. We check that we must have a derivation of the following form:

$$\textsc{D-Sub}\ \dfrac{\textsc{D-Alias}\ \dfrac{\Gamma(\mathsf{x}) = \_ : \mathsf{M}_0}{\Gamma \Vdash \mathsf{x} : \mathsf{M}_0} \qquad \vdash \mathsf{M}_0 <: \mathsf{M}}{\Gamma \Vdash \mathsf{x} : \mathsf{M}}$$

  Using the fact that $\Gamma \Vdash \mathcal{E}$, we know that $\mathcal{E} = (\mathcal{E}_0; \mathsf{module\ x} = \mathsf{m} : \mathsf{M}_0; \mathcal{E}_1)$ and $\Gamma \Vdash \mathsf{m} : \mathsf{M}_0$. We conclude by applying D-Sub.
- if $\mathsf{m.x} \to_\Gamma \mathsf{m}'$ where $\mathsf{m} = (\mathsf{struct\ st}_1; \mathsf{module\ x} = \mathsf{m}'; \mathsf{st}_2\ \mathsf{end})$. We check that we must have a derivation of the following form:

$$\textsc{D-Sub}\ \dfrac{\textsc{D-Compnt}\ \dfrac{\Gamma \Vdash \mathsf{m} : \mathsf{sig}\ \mathsf{S}_1;\ \mathsf{module\ x} : \mathsf{M}_0;\ \mathsf{S}_2\ \mathsf{restr}\ \theta\ \mathsf{end}}{\Gamma \Vdash \mathsf{m.x} : \mathsf{M}_0} \qquad \vdash \mathsf{M}_0 <: \mathsf{M}}{\Gamma \Vdash \mathsf{m.x} : \mathsf{M}}$$

and:

$$\textsc{D-Sub}\ \dfrac{\textsc{D-Struct+}\ \dots\ \dfrac{\dots \qquad \Gamma \Vdash \mathsf{m}' : \mathsf{M}' \qquad \vdash \mathsf{M}' <: \mathsf{M}'_0}{\Gamma \Vdash \mathsf{m} : \mathsf{sig}\ \mathsf{S}'_1;\ \mathsf{module\ x} : \mathsf{M}'_0;\ \mathsf{S}'_2\ \mathsf{restr}\ \theta'\ \mathsf{end}} \qquad \vdash \mathsf{M}'_0 <: \mathsf{M}_0 \qquad \dots}{\Gamma \Vdash \mathsf{m} : \mathsf{sig}\ \mathsf{S}_1;\ \mathsf{module\ x} : \mathsf{M}_0;\ \mathsf{S}_2\ \mathsf{restr}\ \theta\ \mathsf{end}}$$

  Hence we have a derivation of $\Gamma \Vdash \mathsf{m}' : \mathsf{M}'$. We conclude by subtyping, using the fact that we can derive $\vdash \mathsf{M}' <: \mathsf{M}$.
- if $\mathsf{m.x} \to_\Gamma \mathsf{m}'.\mathsf{x}$ where $\mathsf{m} \to_\Gamma \mathsf{m}'$, we know that our derivation is of the form:

$$\textsc{D-Sub}\ \dfrac{\textsc{D-Compnt}\ \dfrac{\Gamma \Vdash \mathsf{m} : \mathsf{sig}\ \mathsf{S}_1;\ \mathsf{module\ x} : \mathsf{M}_0;\ \mathsf{S}_2\ \mathsf{restr}\ \theta\ \mathsf{end}}{\Gamma \Vdash \mathsf{m.x} : \mathsf{M}_0} \qquad \vdash \mathsf{M}_0 <: \mathsf{M}}{\Gamma \Vdash \mathsf{m.x} : \mathsf{M}}$$

  By induction hypothesis, we have a derivation of:

$$\Gamma \Vdash \mathsf{m}' : \mathsf{sig}\ \mathsf{S}_1;\ \mathsf{module\ x} : \mathsf{M}_0;\ \mathsf{S}_2\ \mathsf{restr}\ \theta\ \mathsf{end}$$

  We conclude immediately using D-Compnt and D-Sub.
- the case where $\mathsf{m}(\mathsf{p}) \to_\Gamma \mathsf{m}'(\mathsf{p})$ with $\mathsf{m} \to_\Gamma \mathsf{m}'$ is the same.
- if $\mathsf{m}(\mathsf{p}) \to_\Gamma \mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}]$ where $\mathsf{m} = \mathsf{func}(\mathsf{x} : \mathsf{M}_\mathsf{p})\ \mathsf{m}_0$, we have a derivation of the form:

$$\textsc{D-Sub}\ \dfrac{\textsc{D-FuncApp}\ \dfrac{\Gamma \Vdash \mathsf{m} : \mathsf{func}(\mathsf{x} : \mathsf{M}'_\mathsf{p})\ \mathsf{M}' \qquad \Gamma \vdash \mathsf{p} : \mathsf{M}'_\mathsf{p}}{\Gamma \Vdash \mathsf{m}(\mathsf{p}) : \mathsf{M}'[\mathsf{x} \mapsto \lambda_\mathsf{m}]} \qquad \vdash \mathsf{M}'[\mathsf{x} \mapsto \lambda_\mathsf{m}] <: \mathsf{M}}{\Gamma \Vdash \mathsf{m}(\mathsf{p}) : \mathsf{M}}$$

  where $\lambda_\mathsf{m} = mem_\Gamma(\mathsf{p}')$. Since $\mathsf{m} = \mathsf{func}(\mathsf{x} : \mathsf{M}_\mathsf{p})\ \mathsf{m}_0$, we must have a derivation of the form:

$$\textsc{D-Sub}\ \dfrac{\textsc{D-Func}\ \dfrac{\Gamma \vdash \mathsf{M}_\mathsf{p} \qquad \Gamma(\mathsf{x}) \not\downarrow_{\mathsf{undef}} \qquad \Gamma, \mathsf{module\ x} = \mathsf{abs}_{\mathsf{param}} : \mathsf{M}_\mathsf{p} \Vdash \mathsf{m}_0 : \mathsf{M}''}{\Gamma \Vdash \mathsf{func}(\mathsf{x} : \mathsf{M}_\mathsf{p})\ \mathsf{m}_0 : \mathsf{func}(\mathsf{x} : \mathsf{M}_\mathsf{p})\ \mathsf{M}''} \qquad \vdash \mathsf{M}'_\mathsf{p} <: \mathsf{M}_\mathsf{p} \qquad \vdash \mathsf{M}'' <: \mathsf{M}'}{\Gamma \Vdash \mathsf{func}(\mathsf{x} : \mathsf{M}_\mathsf{p})\ \mathsf{m}_0 : \mathsf{func}(\mathsf{x} : \mathsf{M}'_\mathsf{p})\ \mathsf{M}'}$$

  Since $\Gamma \vdash \mathsf{p} : \mathsf{M}'_\mathsf{p}$ and $\vdash \mathsf{M}'_\mathsf{p} <: \mathsf{M}_\mathsf{p}$, we know that we have a derivation of $\Gamma \vdash \mathsf{p} : \mathsf{M}_\mathsf{p}$. Since:

$$\Gamma, \mathsf{module\ x} = \mathsf{abs}_{\mathsf{param}} : \mathsf{M}_\mathsf{p} \Vdash \mathsf{m}_0 : \mathsf{M}''$$

  is derivable, we apply Lemma C.11 to get a derivation of:

$$\Gamma \Vdash \mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}] : \mathsf{M}''[\mathsf{x} \mapsto \lambda_\mathsf{m}]$$

  Using Lemma C.8, we know that since $\vdash \mathsf{M}'' <: \mathsf{M}'$, we have a derivation of:

$$\Gamma \Vdash \mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}] : \mathsf{M}'[\mathsf{x} \mapsto \lambda_\mathsf{m}]$$

- the case where $\bar{\mathsf{m}}(\mathsf{p}) \to_\Gamma (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0, \mathsf{p})$ and $\bar{\mathsf{m}} = (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0)$ is immediate. □

We recall and prove Lemma B.1:

Lemma (Resolution Soundness). *If* $\Gamma \Vdash \mathcal{E}$ *and* $\Gamma \Vdash m : \mathsf{M}$ *then* $\Gamma \Vdash res_\mathcal{E}(m) : \mathsf{M}$ *whenever* $res_\mathcal{E}(m)$ *is well-defined.*

Proof. From Proposition C.7, we know that $\mathsf{res}_\mathcal{E}(\mathsf{m}) = \mathsf{m} \downarrow_\mathcal{E}$. From Lemma C.12, we know that the type of the module expression is preserved. This concludes this proof. □

D INSTRUMENTED SEMANTICS

We now present the semantics of our programs and judgments. This requires several complexity measures: the total cost of a program execution, the number of calls a program execution makes to some abstract procedure, and the intrinsic cost of a program execution (i.e. the cost of the program without the cost of parameters calls). Instead of defining one instrumented semantics keeping track of all three measurements, we present the three instrumentations separately, to improve readability.

D.1 Semantics

For any set A, we denote by $\mathbb{D}(A)$ the set of discrete sub-distributions over A — i.e. the set of function $\mu : A \to [0,1]$ with discrete support s.t. μ is summable and $|\mu| = \sum_x \mu(x) \leq 1$. For $x \in A$, the *Dirac distribution at* x is written $\mathbb{1}^A_x$ or $\mathbb{1}_x$ when A is clear from the context. If $\mu \in \mathbb{D}(A)$ and $\mu' \in A \to \mathbb{D}(B)$, the expected distribution of $\mu' \in \mathbb{D}(B)$ when ranging over μ, written $\mathbb{E}_{x\sim\mu}[\mu'(x)]$ or $\mathbb{E}_\mu[\mu']$, is defined as $\mathbb{E}_\mu[\mu'] = b \in B \mapsto \sum_{a\in A} \mu(a)\, \mu'(a)(b)$. For $\mu' \in \mathbb{D}(A)$ and $f : A \to B$, the marginal of μ' w.r.t. f, written $f^\#(\mu') \in \mathbb{D}(B)$, is defined as $f^\#(\mu') = b \mapsto \sum_{a\in A | f(a)=b} \mu'(a)$. We write $\pi^\#_1$ (resp. $\pi^\#_2$) for resp. the first and second marginal — i.e. when f is resp. the first and second projection. For any base type $\tau \in \mathbb{B}$, we assume an interpretation domain $\mathbb{V}_\tau$. We let $\mathbb{V}$ be the set of all possible values $\cup_{\tau\in\mathbb{B}} \mathbb{V}_\tau$. A memory $\nu \in \mathcal{M}$ is a function from $\mathcal{V}$ to $\mathbb{V}$. We write $\nu[x]$ for $\nu(x)$. For $\nu \in \mathcal{M}$ and $v \in \mathbb{V}$, we write $\nu[x \leftarrow v]$ for the memory that maps x to v and y to $\nu[y]$ for $y \neq x$.

Expressions semantics. For any operator $f \in \mathcal{F}_\mathsf{E}$ with type $\tau_1 \times \cdots \times \tau_n \to \tau$, we assume given its semantics $(\!|f|\!) : \mathbb{V}_{\tau_1} \times\cdots\times \mathbb{V}_{\tau_n} \mapsto \mathbb{V}_\tau$, and the cost of its evaluation $c_\mathsf{E}(f,\cdot) : \mathbb{V}_{\tau_1} \times\cdots\times \mathbb{V}_{\tau_n} \mapsto \mathbb{N}$. The semantics $(\!|e|\!)_\nu : \mathcal{M} \to \mathbb{V}$ of a well-typed expression e in a memory ν is defined inductively by:

$$(\!|e|\!)_\nu \;=\; \begin{cases} \nu(x) & \text{if } e = x \in \mathcal{V} \\ (\!|f|\!)((\!|e_1|\!)_\nu, \ldots, (\!|e_n|\!)_\nu) & \text{if } e = f(e_1, \ldots, e_1) \end{cases}$$

And the cost of the evaluation of a well-typed expression $c_\mathsf{E}(e,\cdot) : \mathcal{M} \mapsto \mathbb{N}$ is defined by:

$$c_\mathsf{E}(e,\nu) \;=\; \begin{cases} 1 & \text{if } e = x \in \mathcal{V} \\ c_f + \sum_{1\leq i\leq n} c_\mathsf{E}(e_i,\nu) & \text{if } e = f(e_1,\ldots,e_n) \\ & \text{and } c_f = c_\mathsf{E}(f, (\!|e_1|\!)_\nu, \ldots, (\!|e_n|\!)_\nu) \end{cases}$$

For technical reasons, we assume that there exists one operator with a non-zero cost.[12]

For any distribution operator $d \in \mathcal{F}_\mathsf{D}$ with type $\tau_1 \times\cdots\times \tau_n \to \tau$, we assume given its semantics $(\!|d|\!) : \mathbb{V}_{\tau_1} \times\cdots\times \mathbb{V}_{\tau_n} \mapsto \mathbb{D}(\mathbb{V}_\tau)$, and the cost of its evaluation $c_\mathsf{D}(d,\cdot) : \mathbb{V}_{\tau_1} \times\cdots\times \mathbb{V}_{\tau_n} \mapsto \mathbb{N}$. We define similarly $(\!|d|\!)_\nu : \mathcal{M} \to \mathbb{D}(\mathbb{V})$ and $c_\mathsf{D}(d,\cdot) : \mathcal{M} \mapsto \mathbb{N}$.

Environment and $\mathcal{E}$-pre-interpretation. To give the semantics of a program in an environment $\mathcal{E}$, we need an interpretation of $\mathcal{E}$'s abstract modules. A $\mathcal{E}$-pre-interpretation is a function ρ from $\mathcal{E}$'s abstract modules to module expressions, with the correct types, *except for complexity restrictions*. We will specify what it means for a module expression to verify a complexity restriction later, after having defined the semantics of our language.

[12]Some of our lemmas do not hold if all programs have a cost of zero.

$$\begin{aligned}
[\![\mathbf{skip}]\!]^{\mathcal{E},\rho}_\nu &= \mathbb{1}_{(\nu,0)} \\
[\![\mathbf{abort}]\!]^{\mathcal{E},\rho}_\nu &= \mathbb{0} \\
[\![\mathsf{s}_1;\mathsf{s}_2]\!]^{\mathcal{E},\rho}_\nu &= \mathbb{E}_{(\nu',c')\sim[\![\mathsf{s}_1]\!]^{\mathcal{E},\rho}_\nu}[[\![\mathsf{s}_2]\!]^{\mathcal{E},\rho}_{\nu'} \oplus c'] \\
[\![x \leftarrow e]\!]^{\mathcal{E},\rho}_\nu &= \mathbb{1}_{(\nu[x\leftarrow(\!|e|\!)_\nu],\, c_\mathsf{E}(e,\nu))} \\
[\![x \xleftarrow{\$} d]\!]^{\mathcal{E},\rho}_\nu &= \mathbb{E}_{v\sim(\!|d|\!)_\nu}[\mathbb{1}_{(\nu[x\leftarrow v],\, c_\mathsf{D}(d,\nu))}] \\
[\![\mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2]\!]^{\mathcal{E},\rho}_\nu &= \begin{cases} [\![\mathsf{s}_1]\!]^{\mathcal{E},\rho}_\nu \oplus c_\mathsf{E}(e,\nu) & \text{if } (\!|e|\!)_\nu \neq 0 \\ [\![\mathsf{s}_2]\!]^{\mathcal{E},\rho}_\nu \oplus c_\mathsf{E}(e,\nu) & \text{otherwise} \end{cases} \\
[\![\mathbf{while}\ e\ \mathbf{do}\ \mathsf{s}]\!]^{\mathcal{E},\rho}_\nu &= \lim_{n\mapsto\infty} [\![\mathbf{loop}^{e,\mathsf{s}}_n]\!]^{\mathcal{E},\rho}_\nu
\end{aligned}$$

$$\begin{aligned}
\textit{where} \quad \mathbf{loop}^{e,\mathsf{s}}_{n+1} &= \mathbf{if}\ e\ \mathbf{then}\ (\mathsf{s};\mathbf{loop}^{e,\mathsf{s}}_n)\ \mathbf{else}\ \mathbf{skip} \\
\textit{and} \quad \mathbf{loop}^{e,\mathsf{s}}_0 &= \mathbf{if}\ e\ \mathbf{then}\ \mathbf{abort}\ \mathbf{else}\ \mathbf{skip}
\end{aligned}$$

Moreover, if $\text{f-res}_\mathcal{E}(\mathsf{m}.f) = \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\,\_;\ \mathsf{s};\ \mathsf{return}\ r\,\}$:

$$\begin{aligned}
&[\![x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,)]\!]^{\mathcal{E},\rho}_\nu = \\
&\quad \text{let } \nu_0 = \nu[\vec{v} \leftarrow (\!|\vec{e}\,|\!)_\nu] \text{ in} \\
&\quad \mathbb{E}_{(\nu',c')\sim[\![\mathsf{s}]\!]^{\mathcal{E},\rho}_{\nu_0}}[\mathbb{1}_{\nu'[x\leftarrow(\!|r|\!)_{\nu'}],\, c'+c_\mathsf{E}(\vec{e},\nu)+c_\mathsf{E}(r,\nu')}]
\end{aligned}$$

And if $\text{f-res}_\mathcal{E}(\mathsf{m}.f) = (\mathsf{abs_{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f$:

$$[\![x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,)]\!]^{\mathcal{E},\rho}_\nu = [\![x \leftarrow \mathbf{call}\ \rho(\mathsf{x})(\vec{\mathsf{p}}).f(\vec{e}\,)]\!]^{\mathcal{E},\rho}_\nu$$

Figure 26: $(\mathcal{E},\rho)$-denotational semantics $[\![\_]\!]^{\mathcal{E},\rho}_\nu$.

Definition D.1. Let $\mathsf{erase_{compl}}(\mathsf{M})$ be the module signature M where every complexity restriction λ_c has been erased, by replacing it by $\top$. Then ρ is a $\mathcal{E}$-pre-interpretation if and only if for every x such that $\mathcal{E} = \mathcal{E}_1; \mathsf{module}\ \mathsf{x} = \mathsf{abs_{open}} : \mathsf{M_I}; \mathcal{E}_2$, we have $\mathcal{E}_1 \vdash_\epsilon \rho(\mathsf{x}) : \mathsf{erase_{compl}}(\mathsf{M_I})$.

Note that we type $\rho(\mathsf{x})$ in $\mathcal{E}_1$, which lets the interpretation of x use any module or abstract module declared before x in $\mathcal{E}$.

Programs semantics. If $\mu \in \mathbb{D}(\mathcal{M}\times\mathbb{N})$ and $n \in \mathbb{N}$, we write $\mu \oplus n$ for the distribution $f^\#(\mu)$ where $f : (m,c) \mapsto (m, c+n)$. Let $\mathcal{E}$ be a well-typed environment, and s be a well-typed instruction in $\mathcal{E}$, i.e. such that $\mathcal{E} \vdash \mathsf{s}$. The $\mathcal{E}$-denotational semantics of an instruction s under the memory ν and $\mathcal{E}$-pre-interpretation ρ, written $[\![\mathsf{s}]\!]^{\mathcal{E},\rho}_\nu \in \mathbb{D}(\mathcal{M}\times\mathbb{N})$, is defined in Figure 26.

We give the semantics for an extended syntax, which allows procedure calls to be of the form $x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,)$ where m is a module expression. Note that this subsumes the syntax of statements, since a module expression m can be a module path p. This allows to concisely define the semantics of a call to an abstract procedure $(\mathsf{abs_{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f$ as the semantics of a call to $\rho(\mathsf{x})(\vec{\mathsf{p}}).f$.

The $\mathcal{E}$-cost of an instruction s under memory ν and $\mathcal{E}$-pre-interpretation ρ, denoted by $\mathsf{cost}^{\mathcal{E},\rho}_\nu(\mathsf{s}) \in \mathbb{N}\cup\{+\infty\}$, is defined as:

$$\mathsf{cost}^{\mathcal{E},\rho}_\nu(\mathsf{s}) = \sup(\mathsf{supp}(\pi^\#_2([\![\mathsf{s}]\!]^{\mathcal{E},\rho}_\nu)))$$

$$
\begin{aligned}
{}^{y.g}\llbracket \mathbf{skip} \rrbracket_{\nu}^{\mathcal{E},\rho} &= \mathbb{1}_{(\nu,0)} \\
{}^{y.g}\llbracket \mathbf{abort} \rrbracket_{\nu}^{\mathcal{E},\rho} &= \mathbb{0} \\
{}^{y.g}\llbracket \mathsf{s}_1; \mathsf{s}_2 \rrbracket_{\nu}^{\mathcal{E},\rho} &= \mathbb{E}_{(\nu',c') \sim {}^{y.g}\llbracket \mathsf{s}_1 \rrbracket_{\nu}^{\mathcal{E},\rho}} [{}^{y.g}\llbracket \mathsf{s}_2 \rrbracket_{\nu'}^{\mathcal{E},\rho} \oplus c'] \\
{}^{y.g}\llbracket x \leftarrow e \rrbracket_{\nu}^{\mathcal{E},\rho} &= \mathbb{1}_{(\nu[x \leftarrow (\!| e |\!)_{\nu}], 0)} \\
{}^{y.g}\llbracket x \xleftarrow{\$} d \rrbracket_{\nu}^{\mathcal{E},\rho} &= \mathbb{E}_{v \sim (\!| d |\!)_{\nu}} [\mathbb{1}_{(\nu[x \leftarrow v], 0)}] \\
{}^{y.g}\llbracket x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho} &= \text{let } \nu_0 = \nu[\vec{v} \leftarrow (\!| \vec{e}\, |\!)_{\nu}] \text{ in} \\
&\quad\ \mathbb{E}_{(\nu',c') \sim {}^{y.g}\llbracket \mathsf{s} \rrbracket_{\nu_0}^{\mathcal{E},\rho}} [\mathbb{1}_{(\nu'[x \leftarrow (\!| r |\!)_{\nu'}], c')}] \\
&\quad (\text{if f-res}_{\mathcal{E}}(\mathsf{m}.f) = \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_;\ \mathsf{s};\ \mathsf{return}\ r\ \}) \\
{}^{y.g}\llbracket x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho} &= {}^{y.g}\llbracket x \leftarrow \mathbf{call}\ \rho(\mathsf{x})(\vec{\mathsf{p}}).f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho} \oplus 1 \\
&\quad (\text{if f-res}_{\mathcal{E}}(\mathsf{m}.f) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \text{ and } \mathsf{x}.f = \mathsf{y}.g) \\
{}^{y.g}\llbracket x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho} &= {}^{y.g}\llbracket x \leftarrow \mathbf{call}\ \rho(\mathsf{x})(\vec{\mathsf{p}}).f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho} \\
&\quad (\text{if f-res}_{\mathcal{E}}(\mathsf{m}.f) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \text{ and } \mathsf{x}.f \neq \mathsf{y}.g) \\
{}^{y.g}\llbracket \mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2 \rrbracket_{\nu}^{\mathcal{E},\rho} &= \begin{cases} {}^{y.g}\llbracket \mathsf{s}_1 \rrbracket_{\nu}^{\mathcal{E},\rho} \oplus 0 & \text{if } (\!| e |\!)_{\nu} \neq 0 \\ {}^{y.g}\llbracket \mathsf{s}_2 \rrbracket_{\nu}^{\mathcal{E},\rho} \oplus 0 & \text{otherwise} \end{cases} \\
{}^{y.g}\llbracket \mathbf{while}\ e\ \mathbf{do}\ \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho} &= \lim_{n \mapsto \infty} {}^{y.g}\llbracket \mathbf{loop}_n^{e,\mathsf{s}} \rrbracket_{\nu}^{\mathcal{E},\rho}
\end{aligned}
$$

Figure 27: Function call counting semantics ${}^{y.g}\llbracket \_ \rrbracket_{\nu}^{\mathcal{E},\rho}$.

where supp is the support of a distribution (this definition is equivalent to the one given in Section 4.3).

D.2 Instrumented Semantics

We present two other instrumented semantics: ${}^{y.g}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho}$ counts the number of times s calls an abstract procedure $\mathsf{y}.g$; and ${}^{\mathsf{i}}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho}$ measures the intrinsic cost of an instruction (i.e. without counting the cost of function calls in a functor parameters).

Function call counting. The function call counting semantics ${}^{y.g}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho}$, given in in Figure 27, evaluates the instruction s under the memory ν and $\mathcal{E}$-pre-interpretation ρ, counting the number of calls to the abstract procedure $\mathsf{y}.g$.

The maximum number of calls of an instruction s or module procedure $\mathsf{m}.f$ to $\mathsf{y}.g$ in $(\mathcal{E}, \rho)$ is:

$$
\begin{aligned}
\#\mathsf{calls}_{\mathsf{y}.g,\nu}^{\mathcal{E},\rho}(\mathsf{s}) &= \sup(\mathrm{supp}(\pi_2^{\#}({}^{y.g}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho}))) && (\text{in memory } \nu) \\
\#\mathsf{calls}_{\mathsf{y}.g}^{\mathcal{E},\rho}(\mathsf{s}) &= \max_{\nu \in \mathcal{M}} (\#\mathsf{calls}_{\mathsf{y}.g,\nu}^{\mathcal{E},\rho}(\mathsf{s})) && (\text{in any memory})
\end{aligned}
$$

$$
\#\mathsf{calls}_{\mathsf{y}.g}^{\mathcal{E},\rho}(\mathsf{m}.f) =
\begin{cases}
\#\mathsf{calls}_{\mathsf{y}.g}^{\mathcal{E},\rho}(\mathsf{s}) & \text{when f-res}_{\mathcal{E}}(\mathsf{m}.f) = \\
& (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_;\ \mathsf{s};\ \_\}) \\
\#\mathsf{calls}_{\mathsf{y}.g}^{\mathcal{E},\rho}(\rho(\mathsf{x})(\vec{\mathsf{p}}).f)\nu & \text{when f-res}_{\mathcal{E}}(\mathsf{m}.f) = \\
& (\mathsf{abs}_{\mathsf{x}}\ \mathsf{open})(\vec{\mathsf{p}}).f
\end{cases}
$$

Note that when $\text{f-res}_{\mathcal{E}}(\mathsf{m}.f) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_;\ \mathsf{s};\ \_\})$, we ignore the return expression, since expression cannot contain procedure calls (only operator applications).

Intrinsic cost. The $(\mathcal{E}, \rho)$-denotational semantics of an instruction s with intrinsic cost under memory ν and parameters $\vec{\mathsf{x}}$, written ${}^{\mathsf{i}}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}$ is the cost of the execution of s under ν in ρ, without counting the costs of function calls to the parameters $\vec{\mathsf{x}}$. Formally, ${}^{\mathsf{i}}\llbracket \_ \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}$ is defined exactly like $\llbracket \_ \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}$ in Figure 26, except for the concrete procedure call case, which is replaced by:

$$
{}^{\mathsf{i}}\llbracket x \leftarrow \mathbf{call}\ \mathsf{m}.f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}} =
$$
$$
\begin{cases}
\mathbb{E}_{(\nu',c') \sim {}^{\mathsf{i}}\llbracket x \leftarrow \mathbf{call}\ \rho(\mathsf{x})(\vec{\mathsf{p}}).f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}} [(\nu', c_{\mathsf{E}}(\vec{e}, \nu))] & \text{if } \mathsf{z} \in \vec{\mathsf{x}} \\
{}^{\mathsf{i}}\llbracket x \leftarrow \mathbf{call}\ \rho(\mathsf{x})(\vec{\mathsf{p}}).f(\vec{e}\,) \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}} & \text{if } \mathsf{z} \notin \vec{\mathsf{x}}
\end{cases}
$$

where $\text{f-res}_{\mathcal{E}}(\mathsf{m}.f) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{z})(\vec{\mathsf{p}}).f$.

Remark that both semantics coincide on their first component. Indeed, for any $\mathcal{E}$-pre-interpretation ρ:

$$
\forall \nu\ \mathsf{s}.\ \pi_1^{\#}(\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho}) = \pi_1^{\#}({}^{\mathsf{i}}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}})
$$

The $(\mathcal{E}, \rho)$-intrinsic cost $\mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}(\_) \in \mathbb{N} \cup \{+\infty\}$ of an instruction s is:

$$
\mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}(\mathsf{s}) = \sup(\mathrm{supp}(\pi_2^{\#}({}^{\mathsf{i}}\llbracket \mathsf{s} \rrbracket_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}})))
$$

The intrinsic cost of a procedure $\mathsf{m}.f$, with parameters $\vec{\mathsf{x}}$, is:

- If $\text{f-res}_{\mathcal{E}}(\mathsf{m}.f) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_;\ \mathsf{s};\ \mathsf{return}\ r\ \})$ then:
$$
\mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}(\mathsf{m}.f) = \mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}(\mathsf{s}) + c_{\mathsf{E}}(r, \nu)
$$
- If $\text{f-res}_{\mathcal{E}}(\mathsf{m}.f) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f$ then:
$$
\mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}(\mathsf{m}.f) = \mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{x}}}(\rho(\mathsf{x})(\vec{\mathsf{p}}).f)
$$

And the intrinsic cost *in any memory* of an instruction s or a module procedure $\mathsf{m}.f$ is:

$$
\begin{aligned}
\mathsf{i\text{-}cost}^{\mathcal{E},\rho,\vec{\mathsf{z}}}(\mathsf{s}) &= \max_{\nu \in \mathcal{M}} \mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{z}}}(\mathsf{s}) \\
\mathsf{i\text{-}cost}^{\mathcal{E},\rho,\vec{\mathsf{z}}}(\mathsf{m}.f) &= \max_{\nu \in \mathcal{M}} \mathsf{i\text{-}cost}_{\nu}^{\mathcal{E},\rho,\vec{\mathsf{z}}}(\mathsf{m}.f)
\end{aligned}
$$

Interpretations. We now define when a pre-interpretation is an interpretation.

Definition D.2. Let $\mathcal{E}$ be an well-typed environment. A $\mathcal{E}$-pre-interpretation ρ is an $\mathcal{E}$-interpretation if for every module identifier x such that $\mathcal{E} = \mathcal{E}_1; \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I}; \mathcal{E}_2$ where:

$$
\mathsf{M_I} = \mathsf{func}(\vec{\mathsf{z}} : \vec{\mathsf{M}})\ \mathsf{sig}\ \mathsf{S_I}\ \mathsf{restr}\ \theta\ \mathsf{end}
$$

and for every procedure $f \in \mathrm{procs}(\mathsf{S})$, for every valuation $\vec{\mathsf{m}}$ of the functor's parameters such that, for every $1 \leq i \leq |\vec{\mathsf{z}}|$, if we let $\mathsf{z}_i = \vec{\mathsf{z}}[i]$, $\mathsf{m}_i = \vec{\mathsf{m}}[i]$ and $\mathsf{M}_i = \vec{\mathsf{M}}[i] = \mathsf{sig}\ \_\ \mathsf{restr}\ \lambda_{\mathsf{c}}^{i} \wedge \_\ \mathsf{end}$,[13] if:

$$
\mathcal{E} \vdash (\mathsf{module}\ \mathsf{z}_i = \mathsf{m}_i : \mathsf{erase}_{\mathsf{compl}}(\mathsf{M}_i))
$$

and

$$
\forall g \in \mathrm{procs}(\mathsf{M}_i), \mathsf{cost}_{\nu}^{\mathcal{E}}(\mathsf{m}_i.g) \leq \lambda_{\mathsf{c}}^{i}
$$

(with the convention that $j \leq \top$ for any integer j) then the execution of f in any memory verifies the complexity restriction in $\theta[f]$. Formally, let $\mathcal{E}' = \mathcal{E}; \mathsf{module}\ \vec{\mathsf{z}} = \mathsf{abs}_{\mathsf{open}} : \vec{\mathsf{M}}$ and $\rho' = \rho, (\vec{\mathsf{z}} : \vec{\mathsf{m}})$, and:

$$
\theta[f] = \_ \wedge \lambda_{\mathsf{c}} = \mathsf{compl}[\mathsf{intr} : k, \mathsf{y}_1.f_1 : k_1, \ldots, \mathsf{y}_l.f_l : k_l]
$$

[13] Indeed, since $\mathsf{M_I}$ is a low-order signature, M_i must be a module structure signature.

Then for every $1 \leq j \leq l$,

$$\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\mathsf{y}_j.f_j}(\mathsf{x}(\vec{\mathsf{z}}).f) \leq k_j \quad \text{and} \quad \mathsf{i\text{-}cost}^{\mathcal{E}',\rho',\vec{\mathsf{z}}}(\mathsf{x}(\vec{\mathsf{z}}).f) \leq k$$

Intrinsic cost of a functor. Finally, the $(\mathcal{E}, \rho)$-*intrinsic complexity* of a functor procedure $\mathsf{x}.f$, denoted by $\mathsf{compl}^{\mathcal{E},\rho}_{\mathsf{x}.f} \in \mathbb{N} \cup \{+\infty\}$, is the maximal intrinsic cost of $\mathsf{x}.f$'s body over all possible memories and instantiation of x's functor parameters. Let $\mathcal{E}(\mathsf{x}) = \mathsf{abs}_{\mathsf{open}}\ (\mathsf{func}(\vec{\mathsf{z}} : \vec{\mathsf{M}})\ \mathsf{sig}\ \_\ \mathsf{end})$ and $\mathcal{E}' = (\mathcal{E}; \mathsf{module}\ \vec{\mathsf{z}} = \mathsf{abs}_{\mathsf{open}} : \vec{\mathsf{M}})$. Also, let $\mathcal{I}$ be the set $\mathcal{E}'$-interpretation ρ' extending ρ. Then:

$$\mathsf{compl}^{\mathcal{E},\rho}_{\mathsf{x}.f} = \sup_{\rho' \in \mathcal{I}} \mathsf{i\text{-}cost}^{\mathcal{E}',\rho',\vec{\mathsf{z}}}(\mathsf{x}(\vec{\mathsf{z}}).f)$$

D.3 Soundness of our Proof System

Quite naturally, the judgment $\vdash \{\phi\}\ e \leq t_e$ stands for:

$$\forall v \in \phi, c_{\mathsf{E}}(e, v) \leq t_e$$

The judgment $\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}$ means that for any $\mathcal{E}$-interpretation ρ and $v \in \phi$:

$$\mathsf{supp}(\pi^{\#}_1(\llbracket \mathsf{s} \rrbracket^{\mathcal{E},\rho}_v)) \subseteq \psi \ \wedge$$
$$\mathsf{cost}^{\mathcal{E},\rho}_v(\mathsf{s}) \leq t[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ f \in \mathsf{procs}(\mathcal{E}(\mathsf{A}))}} t[\mathsf{A}.f] \cdot \mathsf{compl}^{\mathcal{E},\rho}_{\mathsf{A}.f}$$

(this definition is equivalent to the one given in Section 4.3). Basically, the complexity of the instruction s is upper-bounded by the complexity of the concrete code in s, plus the sum over all abstract oracles $\mathsf{A}.f$ of the number of calls to $\mathsf{A}.f$ times the intrinsic complexity of $\mathsf{A}.f$.

E PROOF RULES VALIDITY

We know prove the validity our Hoare logic rules. We recall Theorem 4.1.

Theorem. *The proof rules in Figure 6, 7 and 8 are valid.*

We focus on the two most complicated rules of our system, Abs (in Section E.1) and Instantiation (in Section E.2). The validity of the remaining rules is straightforward to show (we omit the proof).

First, some notation. We extend the program semantics $\llbracket \_ \rrbracket^{\mathcal{E},\rho}_v$ to function paths, by letting $\llbracket \mathsf{F} \rrbracket^{\mathcal{E},\rho}_v$ be the execution of F's body without the procedure's arguments evaluation phase:

$$\llbracket \mathsf{F} \rrbracket^{\mathcal{E},\rho}_v = \mathbb{E}_{(v',c') \sim \llbracket \mathsf{s} \rrbracket^{\mathcal{E},\rho}_v}[\mathbb{1}_{v'[x \leftarrow (\!|r|\!)_{v'}], c' + c_{\mathsf{E}}(r,v')}]$$
$$(\text{if } \mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_;\ \mathsf{s};\ \mathsf{return}\ r\ \}))$$
$$\llbracket \mathsf{F} \rrbracket^{\mathcal{E},\rho}_v = \llbracket \rho(\mathsf{x})(\vec{\mathsf{p}}).f \rrbracket^{\mathcal{E},\rho}_v \qquad (\text{if } \mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f)$$

The cost of F is defined as expected, by taking the maximum over all memories of the support of the second marginal of $\llbracket \mathsf{F} \rrbracket^{\mathcal{E},\rho}_v$:

$$\mathsf{cost}^{\mathcal{E},\rho}_v(\mathsf{F}) = \sup(\mathsf{supp}(\pi^{\#}_2(\llbracket \mathsf{F} \rrbracket^{\mathcal{E},\rho}_v)))$$

Finally, cost judgment $\mathcal{E} \vdash \{\phi\}\ \mathsf{F}\ \{\psi \mid t\}$ on function path have the same semantics than cost judgment on statements, i.e. for every well-typed environment $\mathcal{E}$, for every $\mathcal{E}$-interpretation ρ and $v \in \phi$:

$$\mathsf{supp}(\pi^{\#}_1(\llbracket \mathsf{F} \rrbracket^{\mathcal{E},\rho}_v)) \subseteq \psi \ \wedge$$
$$\mathsf{cost}^{\mathcal{E},\rho}_v(\mathsf{F}) \leq t[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ f \in \mathsf{procs}(\mathcal{E}(\mathsf{A}))}} t[\mathsf{A}.f] \cdot \mathsf{compl}^{\mathcal{E},\rho}_{\mathsf{A}.f} \qquad (4)$$

E.1 Abstract Call Rule Validity

We now prove that the abstract call rule Abs in Figure 7 is valid.

First, remark that since the rule Abs uses a different upper-bound $t_{\mathsf{H}.g}(k)$ on the cost of the k-th call to the oracle $\mathsf{H}.g$, and since the fact that we are in the k-th call is characterized by the invariant I, we must prove both properties (the invariant on the memories, and the upper-bound on the complexity of the call) simultaneously. The proofs is by induction on the size in $\mathcal{E}$ of the procedure $\rho(\mathsf{x}).f$ called, where procedure calls are inlined. Formally, for every environment $\mathcal{E}$ and $\mathcal{E}$-pre-interpretation ρ, we define:

$$\begin{aligned}
\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathbf{abort}) &= 1 \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathbf{skip}) &= 1 \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}_1; \mathsf{s}_2) &= 1 + \#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}_1) + \#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}_2) \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(x \leftarrow e) = 1 &= 1 \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(x \xleftarrow{\$} d) = 1 &= 1 \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(x \leftarrow \mathbf{call}\ \mathsf{F}(\vec{e})) &= 1 + \#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}) \\
&\quad (\text{ if } \mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = \mathsf{proc}\ f\ _ = \{_;\ \mathsf{s};\ _\}) \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(x \leftarrow \mathbf{call}\ \mathsf{F}(\vec{e})) &= 1 + \#\mathsf{size}^{\mathcal{E}}_{\rho}(x \leftarrow \mathbf{call}\ \rho(\mathsf{x})(\vec{\mathsf{p}}).f(\vec{e})) \\
&\quad (\text{ if } \mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f) \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2) &= 1 + \#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}_1) + \#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}_2) \\
\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathbf{while}\ e\ \mathbf{do}\ \mathsf{s}) &= 1 + \#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s})
\end{aligned}$$

Note that we do not care about the size of the expressions appearing in the statement.

We now define and prove our (generalized) induction property, which we need to prove the validity of Abs after.

Lemma E.1. *Let I a formula, $\mathcal{E}$ a well-typed environment, ρ an $\mathcal{E}$-interpretation and $\vec{\mathsf{x}}$ be functor parameters with module types $\mathsf{abs}_{open}\ \vec{\mathsf{M}}$ which are module structure signatures (i.e. not functors).*

For every statement s well-typed in $\mathcal{E}$ with additional functor parameters $\mathsf{abs}_{param}\ \vec{\mathsf{x}} : \vec{\mathsf{M}}$, and satisfying a memory restriction $\lambda^{\mathsf{s}}_{\mathsf{m}}$:

$$\Gamma \vdash \mathsf{s} \qquad \Gamma \vdash \mathsf{s} \rhd \lambda^{\mathsf{s}}_{\mathsf{m}} \qquad \textit{where} \qquad \Gamma = \mathcal{E}, \mathsf{module}\ \vec{\mathsf{x}} = \mathsf{abs}_{param} : \vec{\mathsf{M}}$$

Then for every valuation $\vec{\mathsf{p}}$ of the parameters $\vec{\mathsf{x}}$ well-typed in $\mathcal{E}$ (i.e. $\mathcal{E} \vdash \vec{\mathsf{p}} : \vec{\mathsf{M}}$), if we let $\mathcal{E}' = \mathcal{E}, \mathsf{module}\ \vec{\mathsf{x}} = \mathsf{abs}_{open} : \vec{\mathsf{M}}$, and $\vec{\mathsf{O}}$ be an enumeration of the parameters $\vec{\mathsf{x}}$'s procedures, i.e. of:

$$\{\mathsf{y}.h \mid \mathsf{y} \in \vec{\mathsf{x}} \wedge h \in \mathsf{procs}(\vec{\mathsf{M}}[\mathsf{y}])\}$$

Then if:

- *the memory of the statement s is independent of I, except for calls to the parameters $\vec{\mathsf{x}}$, i.e.:*

$$\lambda^{\mathsf{s}}_{\mathsf{m}} \cap \mathsf{FV}(I) = \emptyset$$

- *for every parameters' procedure $\mathsf{H}.g \in \mathsf{procs}_{\Gamma}(\vec{\mathsf{x}})$, the k-th call to $\mathsf{H}.g$ (when H is instantiated by $\vec{\mathsf{p}}[\mathsf{H}]$) preserves the invariant and has a cost upper-bounded by $t_{\mathsf{H}.g}[k]$ (for $k \leq \lambda_{\mathsf{c}}[\mathsf{H}.g]$):*

$$\forall \vec{k} \leq \lambda_{\mathsf{c}}[\vec{\mathsf{O}}],\ \vec{k}[\mathsf{H}.g] < \lambda_{\mathsf{c}}[\mathsf{H}.g] \to$$
$$\mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{\mathsf{p}}[\mathsf{H}].f\ \{I\ (\vec{k} + \mathbb{1}_{\mathsf{H}.g}) \mid t_{\mathsf{H}.g}(\vec{k}[\mathsf{H}.g])\} \qquad (5)$$

- *the number of calls from* s *to the parameters' procedure is upper-bounded by* $\lambda_c[\vec{\mathsf{O}}] - \vec{k}$ *(for memory satisfying the invariant* $I\,\vec{k}$*):*

$$\forall \vec{k} \leq \lambda_c[\vec{\mathsf{O}}],\ \forall v \in I\,\vec{k},\ \forall \mathsf{H}.g \in \mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}}),$$
$$\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\mathsf{H}.g,v}(\mathsf{s}) \leq \lambda_c[\mathsf{H}.g] - \vec{k}[\mathsf{H}.g] \quad (6)$$

For every $\mathsf{H}.g \in \mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}})$ *and* $j < \lambda_c[\mathsf{H}.g]$*, we define:*

$$\mathsf{call\text{-}cost}^{\mathcal{E},\rho}_{\mathsf{H}.g}(j) = t_{\mathsf{H}.g}(j)[conc] + \sum_{\substack{\mathsf{A}\in\mathsf{abs}(\mathcal{E})\\ f\in\mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_{\mathsf{H}.g}(j)[\mathsf{A}.f]\cdot \mathsf{compl}^{\mathcal{E},\rho}_{\mathsf{A}.f}$$

Let $\rho' = \rho, (\vec{\mathsf{x}} : \vec{\mathsf{p}})$*. We have that for every* $\vec{k} \leq \lambda_c[\vec{\mathsf{O}}]$ *and* $v \in I\,\vec{k}$*:*

$$\mathsf{supp}(\pi_1^\#([\![\mathsf{s}]\!]^{\mathcal{E}',\rho'}_v)) \subseteq I\,(\vec{k} + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s})) \quad (7)$$

and:

$$\mathsf{cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}) \leq \mathsf{i\text{-}cost}^{\mathcal{E}',\rho',\vec{\mathsf{x}}}_v(\mathsf{s}) + \sum_{\mathsf{H}.g\in\mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}})} \sum_{j=\vec{k}[\mathsf{H}.g]}^{\vec{k}[\mathsf{H}.g]+\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\mathsf{H}.g,v}(\mathsf{s})-1} \mathsf{call\text{-}cost}^{\mathcal{E},\rho}_{\mathsf{H}.g}(j) \quad (8)$$

That is, the cost of s executed in $\mathcal{E}', \rho'$ and memory v (satisfying the invariant $I\,\vec{k}$) is upper-bounded by the intrinsic cost of s with functors parameters $\vec{\mathsf{x}}$, of the sum over functor parameters' procedures $\mathsf{H}.g \in \mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}})$ of the sum over all calls to $\mathsf{H}.g$ (which ranges from $\vec{k}[\mathsf{H}.g]$ to $\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\mathsf{H}.g,v}(\mathsf{s})$ -1) of the concrete cost of (the j-th call to) $\mathsf{H}.g$ (upper-bounded by $t_{\mathsf{H}.g}(j)[\mathsf{conc}]$), plus the sum, over all abstract procedures $\mathsf{A}.f$ (in the original environment $\mathcal{E}$), of the number of times the j-th call to $\mathsf{H}.g$ called $\mathsf{A}.f$ (upper-bounded by $t_{\mathsf{H}.g}(j)[\mathsf{A}.f]$) times the maximal cost of $\mathsf{A}.f$ (which is $\mathsf{compl}^{\mathcal{E},\rho}_{\mathsf{A}.f}$).

Proof. We prove this induction over $\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s})$. We do a case analysis on s:

- if s is **abort** or **skip**, this is immediate.
- if $\mathsf{s} = \mathsf{s}_1; \mathsf{s}_2$. Let $\vec{k} \leq \lambda_c[\vec{\mathsf{O}}]$ and $v \in I\,\vec{k}$. We know that:

$$\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_1;\mathsf{s}_2]\!]^{\mathcal{E}',\rho'}_v)) = \bigcup_{v'\in\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_1]\!]^{\mathcal{E}',\rho'}_v))} \mathsf{supp}(\pi_1^\#([\![\mathsf{s}_2]\!]^{\mathcal{E}',\rho'}_{v'}))$$

  Let $\vec{k}_1 = \vec{k} + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}_1)$. By induction hypothesis applied on s_1,

$$\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_1]\!]^{\mathcal{E}',\rho'}_v)) \subseteq I\,\vec{k}_1 \quad (9)$$

  Let $v' \in \mathsf{supp}(\pi_1^\#([\![\mathsf{s}_1]\!]^{\mathcal{E}',\rho'}_v))$, we know that $\vec{k}_1 \leq \lambda_c[\vec{\mathsf{O}}]$. Hence, by induction hypothesis on s_2, we deduce that:

$$\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_2]\!]^{\mathcal{E}',\rho'}_{v'})) \subseteq I\,\vec{k}_2$$

  where $\vec{k}_2(v') = \vec{k} + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}_1) + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v'}(\mathsf{s}_2)$. Since:

$$\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}_1) + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v'}(\mathsf{s}_2) \leq \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}_1;\mathsf{s}_2) \quad (10)$$

  we deduce:

$$\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_2]\!]^{\mathcal{E}',\rho'}_{v'})) \subseteq I\,(\vec{k} + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}_1;\mathsf{s}_2))$$

  which concludes the proofs of the first point. It remains to prove that the complexity of $\mathsf{s}_1;\mathsf{s}_2$ is upper-bounded by the wanted quantity. First, we have:

$$\mathsf{cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1;\mathsf{s}_2) = \mathsf{cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1) + \max_{v'\in\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_1]\!]^{\mathcal{E}',\rho'}_v))} \mathsf{cost}^{\mathcal{E}',\rho'}_{v'}(\mathsf{s}_2)$$

  By applying the induction hypothesis on s_1 and s_2 with, respectively, $\vec{k}$ and $\vec{k}_1$, we get:

$$\mathsf{cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1) \leq \mathsf{i\text{-}cost}^{\mathcal{E}',\rho',\vec{\mathsf{x}}}_v(\mathsf{s}_1) + \sum_{\mathsf{H}.g\in\mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}})} \sum_{j=\vec{k}[\mathsf{H}.g]}^{\vec{k}_1[\mathsf{H}.g]-1} \mathsf{call\text{-}cost}^{\mathcal{E},\rho}_{\mathsf{H}.g}(j)$$

  and:

$$\mathsf{cost}^{\mathcal{E}',\rho'}_{v'}(\mathsf{s}_2) \leq \mathsf{i\text{-}cost}^{\mathcal{E}',\rho',\vec{\mathsf{x}}}_{v'}(\mathsf{s}_2) + \sum_{\mathsf{H}.g\in\mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}})} \sum_{j=\vec{k}_1[\mathsf{H}.g]}^{\vec{k}_2(v')[\mathsf{H}.g]-1} \mathsf{call\text{-}cost}^{\mathcal{E},\rho}_{\mathsf{H}.g}(j)$$

  Since $\mathsf{i\text{-}cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1;\mathsf{s}_2)$ is equal to:

$$\mathsf{i\text{-}cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1) + \max_{v'\in\mathsf{supp}(\pi_1^\#([\![\mathsf{s}_1]\!]^{\mathcal{E}',\rho'}_v))} \mathsf{i\text{-}cost}^{\mathcal{E}',\rho'}_{v'}(\mathsf{s}_2)$$

  and using Equ. (10), we deduce that:

$$\mathsf{cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1) + \mathsf{cost}^{\mathcal{E}',\rho'}_{v'}(\mathsf{s}_2) \leq \mathsf{i\text{-}cost}^{\mathcal{E}',\rho'}_v(\mathsf{s}_1;\mathsf{s}_2) + \sum_{\mathsf{H}.g\in\mathsf{procs}_{\mathcal{E}}(\vec{\mathsf{x}})} \sum_{j=\vec{k}[\mathsf{H}.g]}^{\vec{k}[\mathsf{H}.g]+\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}_1;\mathsf{s}_2)-1} \mathsf{call\text{-}cost}^{\mathcal{E},\rho}_{\mathsf{H}.g}(j)$$

  We conclude by taking the max over v' and using Equ. 9.
- if $\mathsf{s} = x \leftarrow e$. Let $\vec{k} \leq \lambda_c[\vec{\mathsf{O}}]$ and $v \in I\,\vec{k}$. Since $\mathsf{mem}_{\Gamma}(\mathsf{s}) \cap \mathsf{FV}(I) = \emptyset$, we know that $x \notin \mathsf{FV}(I)$. Hence the invariant is preserved, i.e.:

$$\forall v' \in \mathsf{supp}(\pi_1^\#([\![\mathsf{s}]\!]^{\mathcal{E}',\rho'}_v))),\ v' \in I\,\vec{k}$$

  Since $\#\mathsf{calls}^{\mathcal{E}',\rho'}_{\vec{\mathsf{O}},v}(\mathsf{s}) = 0$, this proves the first point. Moreover:

$$\mathsf{cost}^{\mathcal{E}',\rho'}_v(x \leftarrow e) = \mathsf{i\text{-}cost}^{\mathcal{E}',\rho',\vec{\mathsf{x}}}_v(x \leftarrow e)$$

  which concludes the proof.
- the random assignment $x \xleftarrow{\$} d$, conditional **if** e **then** s_1 **else** s_2 and while loop **while** e **do** s cases are similar. We omit the details.
- if $\mathsf{s} = x \leftarrow \textbf{call}\ \mathsf{F}(\vec{e})$ and $\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\ \mathsf{var}\ (\vec{v}_l : \vec{\tau}_l);\ \ \mathsf{s}';\ \ \mathsf{return}\ e\ \})$, then we proceed as in the previous case, applying the induction hypothesis on s'. First, we remark that s' is smaller than s, since:

$$\#\mathsf{size}^{\mathcal{E}}_{\rho}(\mathsf{s}') < \#\mathsf{size}^{\mathcal{E}}_{\rho}(x \leftarrow \textbf{call}\ \mathsf{F}(\vec{e}))$$

  It only remains to check that the induction hypothesis's hypotheses hold. The last two hypotheses are straightforward to show. It only remains to prove that $\Gamma \vdash s' \triangleright \lambda^s_m$ is derivable. Since $\Gamma \vdash s \triangleright \lambda^s_m$, we know that $\mathsf{mem}_\Gamma(s) \subseteq \lambda^s_m$, where:

$$\mathsf{mem}_\Gamma(s) = \{x\} \sqcup \mathsf{mem}_\Gamma(s') \sqcup \mathsf{vars}(\vec{e})$$

Which concludes this case.

- idem if $s = x \leftarrow \textbf{call } F(\vec{e})$ and $\text{f-res}_{\mathcal{E}}(F) = (\mathsf{abs}_{\mathsf{open}}\ H)(\_).f$ with $H \notin \vec{x}$.
- if $s = x \leftarrow \textbf{call } F(\vec{e})$ and $\text{f-res}_{\mathcal{E}}(F) = (\mathsf{abs}_{\mathsf{open}}\ H)(\vec{p}').g$ with $H \in \vec{x}$, then we use the hypothesis that procedures of the interpretation of the modules parameters $\vec{x}$ preserves the invariant. First, since $\vec{x}$ have types $\vec{M}$ and $\vec{M}$ are module structure signatures, we know that $\vec{p}'$ is empty (from the fact that s is well-typed in Γ). Hence $\text{f-res}_{\mathcal{E}}(F) = \mathsf{abs}_{\mathsf{open}}\ H.g$.
Let $\vec{k} \leq \lambda_c[\vec{O}]$ and $v \in I\ \vec{k}$. First, using the hypothesis in Eq. (6), we know that:

$$\vec{k}[H.g] + \#\mathsf{calls}^{\mathcal{E}',\rho'}_{H.g,v}(x \leftarrow \textbf{call } F(\vec{e})) \leq \lambda_c[H.g]$$

Hence $\vec{k}[H.g] < \lambda_c[H.g]$. By applying Equ. (5) on $H.g$, we know that:

$$\mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{p}[H].g\ \{I\ (\vec{k} + \mathbb{1}_{H.g}) \mid t_{H.g}(k)\} \tag{11}$$

Since ρ is an $\mathcal{E}$-interpretation, and since $\vec{x}$ have type $\vec{M}$ where $\vec{M}$ are module structure signatures, we can check that ρ' is an $\mathcal{E}'$-interpretation. Hence, from Equ. (11) and the semantics of the cost judgment given in Equ. (4) we get that:

$$\mathsf{supp}(\pi^\#_1(\llbracket \vec{p}[H].g \rrbracket^{\mathcal{E},\rho}_v)) \subseteq I\ (\vec{k} + \mathbb{1}_{H.g}) \tag{12}$$

and:

$$\begin{aligned} &\mathsf{cost}^{\mathcal{E},\rho}_v(\vec{p}[H].g) \\ \leq\ & t_{H.g}(\vec{k}[H.g])[\mathsf{conc}] \\ &+ \sum_{\substack{A \in \mathsf{abs}(\mathcal{E}) \\ f \in \mathsf{procs}_{\mathcal{E}}(A)}} t_{H.g}(\vec{k}[H.g])[A.f] \cdot \mathsf{compl}^{\mathcal{E},\rho}_{A.f} \\ \leq\ & \mathsf{call\text{-}cost}^{\mathcal{E},\rho}_{H.g}(\vec{k}[H.g]) \end{aligned} \tag{13}$$

Observe that since $\rho' = \rho, (\vec{x} : \vec{p})$, we have:

$$\llbracket x \leftarrow \textbf{call } F(\vec{e}) \rrbracket^{\mathcal{E}',\rho'}_v = \llbracket x \leftarrow \textbf{call } \vec{p}[H].g(\vec{e}) \rrbracket^{\mathcal{E}',\rho'}_v$$

Hence Equ. (7) follows directly from Equ. (12). It remains to show Equ. (8). First, note that:

$$\begin{aligned} \mathsf{cost}^{\mathcal{E}',\rho'}_v(x \leftarrow \textbf{call } F(\vec{e})) \ = \\ \text{i-cost}^{\mathcal{E}',\rho',\vec{x}}_v(x \leftarrow \textbf{call } F(\vec{e})) + \mathsf{cost}^{\mathcal{E}',\rho'}_v(\vec{p}[H].g) \end{aligned}$$

Since $\vec{p}$ is well-typed in $\mathcal{E}$, we know that $\mathsf{cost}^{\mathcal{E}',\rho'}_v(\vec{p}[H].g) = \mathsf{cost}^{\mathcal{E},\rho}_v(\vec{p}[H].g)$. Hence:

$$\begin{aligned} \mathsf{cost}^{\mathcal{E}',\rho'}_v(x \leftarrow \textbf{call } F(\vec{e})) \ = \\ \text{i-cost}^{\mathcal{E}',\rho',\vec{x}}_v(x \leftarrow \textbf{call } F(\vec{e})) + \mathsf{cost}^{\mathcal{E},\rho}_v(\vec{p}[H].g) \end{aligned}$$

We conclude the proof of Equ. (8) using the inequality above and Equ. (13). ◻

Lemma E.2. *The rule* Abs *given in Figure 7 is sound.*

Proof. We just apply Lemma E.1 on $\rho(x).f$. The first two hypotheses of the lemma hold thanks to the premises of the Abs rule, and using the fact our module system has the subject reduction property (Lemma B.1). The third hypothesis follows from the fact that ρ is an $\mathcal{E}$-interpretation. ◻

E.2 Instantiation Rule Validity

We prove the following technical lemma, which allows to extend an $\mathcal{E}$-interpretation ρ into an $(\mathcal{E}, \mathsf{module}\ x = \mathsf{abs}_{\mathsf{open}}{:}M_I)$-interpretation $\rho' = \rho, (x \mapsto m)$. This is possible whenever:

i) m has type $\mathsf{erase}_{\mathsf{compl}}(M_I)$;
ii) and we can show that m verifies M_I's complexity restriction by proving that:

$$\forall f \in \mathsf{procs}(M_I), \quad \mathcal{E}, \mathsf{module}\ \vec{x} = \mathsf{abs}_{\mathsf{open}}{:}\vec{M} \vdash \{\top\}\ m(\vec{x}).f\ \{\top \mid t_f\}$$

where $\vec{x}$ are M_I's functor parameters, and $\vec{M}$ their types.

Lemma E.3. *Let $\mathcal{E}$ be a well-typed environment, M_I be low-order module signature s.t.:*

$$M_I = \mathsf{func}(\vec{x} : \vec{M})\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}$$

and m be a module expression s.t. $\mathcal{E} \vdash_x m : erase_{compl}(M_I)$. Let $\mathcal{E}_a = \mathcal{E}, \mathsf{module}\ x = \mathsf{abs}_{open} : M_I$ and ρ be an $\mathcal{E}$-interpretation. If, for every $f \in \mathsf{procs}(M_I)$, we have:

$$\mathcal{E}, \mathsf{module}\ \vec{x} = \mathsf{abs}_{open} : \vec{M} \vdash \{\top\}\ m(\vec{x}).f\ \{\top \mid t_f\}\ \wedge \quad t_f \leq_{compl} \theta[f] \tag{14}$$

then $\rho_a = \rho, (x \mapsto m)$ is an $\mathcal{E}_a$-interpretation. Moreover, for any $f \in \mathsf{procs}(M_I)$:

$$\mathsf{compl}^{\mathcal{E}_a,\rho_a}_{x.f} \leq t_f[conc] + \sum_{\substack{A \in \mathsf{abs}(\mathcal{E}) \\ h \in \mathsf{procs}_{\mathcal{E}}(A)}} t_f[A.h] \cdot \mathsf{compl}^{\mathcal{E},\rho}_{A.h} \tag{15}$$

Proof. Let $\vec{m}$ be an evaluation of x's parameters s.t. for every $1 \leq i \leq |\vec{x}|$, if we let $x_i = \vec{x}[i]$, $m_i = \vec{m}[i]$ and $M_i = \vec{M}[i] = \mathsf{sig}\ \_\ \mathsf{restr}\ \lambda^i_c \wedge \_\ \mathsf{end}$:

$$\mathcal{E} \vdash (\mathsf{module}\ x_i = m_i : \mathsf{erase}_{\mathsf{compl}}(M_i))$$

and:

$$\forall g \in \mathsf{procs}(M_i), \mathsf{cost}^{\mathcal{E}}_v(m_i.g) \leq \lambda^i_c$$

Let $\mathcal{E}'_a = \mathcal{E}_a; \mathsf{module}\ \vec{x} = \mathsf{abs}_{\mathsf{open}} : \vec{M}$ and $\rho'_a = \rho_a, (\vec{x} : \vec{m})$. Also, let $y \in \vec{x}$ and $g \in \mathsf{procs}(\vec{M}[y])$. To prove that ρ_a is an $\mathcal{E}_a$-interpretation, we only need to prove that:

$$\#\mathsf{calls}^{\mathcal{E}'_a,\rho'_a}_{y.g}(x(\vec{x}).f) \leq \theta[f][y.g] \tag{16}$$

and that:

$$\text{i-cost}^{\mathcal{E}'_a,\rho'_a,\vec{x}}(x(\vec{x}).f) \leq \theta[f][\mathsf{intr}] \tag{17}$$

Restriction to module structures. First, note that w.l.o.g. we can assume that $\vec{m}$ are all module structure (i.e. of the form struct _ end). Basically, we show that we can build another interpretation ρ''_a extending ρ_a which satisfies the same hypotheses than ρ'_a, and such that for every $z \in \vec{x}$, $\rho''_a(z)$ is a module structure such that, for every $g \in \mathsf{procs}(\vec{M}[z])$:

$$\#\mathsf{calls}^{\mathcal{E}'_a,\rho'_a}_{z.g}(x(\vec{x}).f) = \#\mathsf{calls}^{\mathcal{E}'_a,\rho''_a}_{z.g}(x(\vec{x}).f)$$

and:

$$\text{i-cost}^{\mathcal{E}'_a,\rho'_a,\vec{x}}(x(\vec{x}).f) = \text{i-cost}^{\mathcal{E}'_a,\rho''_a,\vec{x}}(x(\vec{x}).f)$$

Indeed, assume that there exists some $y \in \vec{x}$ such that $\vec{m}[y]$ is not a module structure in ρ'_a. If $\vec{m}[y]$ is a module path p in ρ'_a, then we resolve it in $\mathcal{E}, \rho$ (which is always possible, since $\vec{m}[y]$ is well-typed in $\mathcal{E}$ and ρ is an $\mathcal{E}$-interpretation) until we get a module structure struct st end, and replace y by struct st end in ρ''_a. Finally, $\vec{m}[y]$ cannot be a functor (by typing hypothesis) in ρ'_a. We repeat the steps above until $(\rho'_a)[\vec{x}]$ are all module structures.

Proof of Equ. (16). Since $\rho'_a(x) = m$ and m is well-typed in $\mathcal{E}$, and since the module expressions $\vec{m}$ are well-typed in $\mathcal{E}$, we can removed x from the environment and the interpretation while keeping the semantics unchanged. That is, we have:

$$\#\text{calls}_{y.g}^{\mathcal{E}'_a,\rho'_a}(x(\vec{x}).f) = \#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f) \tag{18}$$

where:

$$\mathcal{E}' = \mathcal{E}; \text{module } \vec{x} = \text{abs}_{\text{open}} : \vec{M} \qquad \text{and} \qquad \rho' = \rho, (\vec{x} : \vec{m})$$

Since $\mathcal{E}$ is well-typed, and $\vec{m}$ have types $\vec{M}$ in $\mathcal{E}$, and since $\vec{M}$ is not a functor type, we can check that $\mathcal{E}'$ is well-typed, and ρ' is an $\mathcal{E}'$-interpretation. Using Equ. (14), we get:

$$\forall \nu,\ \text{cost}_\nu^{\mathcal{E}',\rho'}(m(\vec{x}).f) \le t_f[\text{conc}] + \sum_{\substack{A \in \text{abs}(\mathcal{E}') \\ h \in \text{procs}_{\mathcal{E}'}(A)}} t_f[A.h] \cdot \text{compl}_{A.h}^{\mathcal{E}',\rho'} \tag{19}$$

Let $N > 0$ be an non-zero positive integer. We are going to change the interpretation of y in ρ' by adding some code doing nothing and taking time N. Let st_y be such that $\rho'(y) = \text{struct st}_y \text{ end}$. By typing hypothesis, we know that st_y is of the shape:

$$\text{st}_y = \text{st}_1;\ \text{proc } g(\vec{v} : \vec{\tau}) \to \tau_r = \{\ \text{var } (\vec{v}_l : \vec{\tau}_l);\ s_g;\ \text{return } e_g\ \};\ \text{st}_2$$

Then, we let st_y^N be the module structure:

$$\text{st}_1;\ \text{proc } g(\vec{v} : \vec{\tau}) \to \tau_r = \{\ \text{var } (\vec{v}_l : \vec{\tau}_l);\ (\text{tic}^N; s_g);\ \text{return } e_g\ \};\ \text{st}_2$$

and let ρ'_N be the interpretation with the same domain as ρ' s.t.:

$$\forall w \in \text{dom}(\rho'), \rho'_N(w) = \begin{cases} \rho'(w) & \text{if } w \ne y \\ \text{struct st}_y^N \text{ end} & \text{if } w = y \end{cases}$$

Let ν be some arbitrary memory. Since ρ' is an $\mathcal{E}'$-interpretation, then so is ρ'_N. Using Equ. (14), we get:

$$\text{cost}_\nu^{\mathcal{E}',\rho'_N}(m(\vec{x}).f) \le t_f[\text{conc}] + \sum_{\substack{A \in \text{abs}(\mathcal{E}') \\ h \in \text{procs}_{\mathcal{E}'}(A)}} t_f[A.h] \cdot \text{compl}_{A.h}^{\mathcal{E}',\rho'_N} \tag{20}$$

Moreover, we can easily check that:

$$\#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f) = \#\text{calls}_{y.g}^{\mathcal{E}',\rho'_N}(m(\vec{x}).f)$$

and:

$$\text{cost}_\nu^{\mathcal{E}',\rho'_N}(m(\vec{x}).f) = \text{cost}_\nu^{\mathcal{E}',\rho'}(m(\vec{x}).f) + N \cdot \#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f)$$

From Equ. (20), we have:

$$\text{cost}_\nu^{\mathcal{E}',\rho'}(m(\vec{x}).f) + N \cdot \#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f) \le t_f[\text{conc}] + \sum_{\substack{A \in \text{abs}(\mathcal{E}') \\ h \in \text{procs}_{\mathcal{E}'}(A)}} t_f[A.h] \cdot \text{compl}_{A.h}^{\mathcal{E}',\rho'_N}$$

Using the inequality above, and Equ. (19), we have:

$$\begin{aligned} N \cdot \#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f) &\le \sum_{\substack{A \in \text{abs}(\mathcal{E}') \\ h \in \text{procs}_{\mathcal{E}'}(A)}} t_f[A.h] \cdot (\text{compl}_{A.h}^{\mathcal{E}',\rho'_N} - \text{compl}_{A.h}^{\mathcal{E}',\rho'}) \\ &\le t_f[y.g] \cdot (\text{compl}_{y.g}^{\mathcal{E}',\rho'_N} - \text{compl}_{y.g}^{\mathcal{E}',\rho'}) \\ &\qquad\qquad (\text{Since } \forall z \ne y, \rho'_N(z) = \rho'(z)) \\ &\le t_f[y.g] \cdot N \end{aligned}$$

We deduce that $\#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f) \le t_f[y.g]$. From Equ. (14), we get that:

$$\#\text{calls}_{y.g}^{\mathcal{E}',\rho'}(m(\vec{x}).f) \le \theta[f][y.g]$$

This, together with Equ. (18), proves Equ. (16)

Finally, to prove that Equ. (17) and Equ. 15 hold we do the exactly the same reasoning, this time by adding to the interpretation of x some code doing nothing and taking time N. □

We also prove the following weakening lemma for the intrinsic complexity of a procedure.

Lemma E.4. *For every well-typed environment $\mathcal{E}$, if:*

$$\mathcal{E} = \mathcal{E}_1; \mathcal{E}_2 \qquad \textit{where} \qquad \mathcal{E}_1 = \mathcal{E}_0; \text{module } x = \text{abs}_{open} : M_l$$

and $\mathcal{E}_2$ contains only module declarations. Then for every $f \in \text{procs}(M_l)$ and $\mathcal{E}$-interpretation ρ, we have $\text{compl}_{x.f}^{\mathcal{E},\rho} = \text{compl}_{x.f}^{\mathcal{E}_1,\rho_1}$ where ρ_1 is the restriction of ρ to $\mathcal{E}_1$'s abstract modules.

Proof sketch. Assume $M_l = (\text{func}(\vec{x} : \vec{M}) \text{ sig \_ end})$. Let $\mathcal{E}' = (\mathcal{E}, \text{module } \vec{x} = \text{abs}_{\text{open}} : \vec{M})$ and $\mathcal{E}'_1 = (\mathcal{E}_1, \text{module } \vec{x} = \text{abs}_{\text{open}} : \vec{M})$, we prove that $\text{compl}_{x.f}^{\mathcal{E},\rho} \le \text{compl}_{x.f}^{\mathcal{E}_1,\rho_1}$ and $\text{compl}_{x.f}^{\mathcal{E},\rho} \ge \text{compl}_{x.f}^{\mathcal{E}_1,\rho_1}$.

The latter inequality is straightforward to show, since any $\mathcal{E}'_1$-interpretation ρ_1 can be extended into an $\mathcal{E}'$-interpretation ρ that leaves the intrinsic cost of x unchanged, i.e. such that for any ν:

$$\text{i-cost}_\nu^{\mathcal{E}',\rho',\vec{x}}(x(\vec{x}).f) = \text{i-cost}_\nu^{\mathcal{E}'_1,\rho'_1,\vec{x}}(x(\vec{x}).f)$$

To prove the former inequality, we show that any $\mathcal{E}'$-interpretation ρ can be transformed into an $\mathcal{E}'_1$-interpretation ρ' such that Equ. (E.2) holds, by inlining all modules of $\mathcal{E}_2$ in ρ. We omit the details. □

Lemma E.5. *The rule* Instantiation *given in Figure 8 is sound.*

Proof. We consider an instance of the rule Instantiation. We want to prove that:

$$\mathcal{E}, \text{module } x = m : M_l \vdash \{\phi\}\ s\ \{\psi \mid T_{\text{ins}}\}$$

Let $\mathcal{E}_c = (\mathcal{E}, \text{module } x = m : M_l)$ and $\mathcal{E}_a = (\mathcal{E}, \text{module } x = \text{abs}_{\text{open}} : M_l)$. We know that:

$$M_l = \text{func}(\vec{y} : \vec{M}) \text{ sig } S_l \text{ restr } \theta \text{ end} \qquad \mathcal{E} \vdash_x m : \text{erase}_{\text{compl}}(M_l)$$

Let ρ be an $\mathcal{E}_c$-interpretation and $\nu \in \phi$, we need to show that for every memory ν:

$$\text{supp}(\pi_1^\#([\![s]\!]_\nu^{\mathcal{E}_c,\rho})) \subseteq \psi \tag{21}$$

$$\text{cost}_\nu^{\mathcal{E}_c,\rho}(s) \le T_{\text{ins}}[\text{conc}] + \sum_{\substack{A \in \text{abs}(\mathcal{E}_c) \\ g \in \text{procs}_{\mathcal{E}_c}(A)}} T_{\text{ins}}[A.g] \cdot \text{compl}_{A.g}^{\mathcal{E}_c,\rho} \tag{22}$$

We know that $\mathsf{M_I} = \mathsf{func}(\vec{\mathsf{y}} : \vec{\mathsf{M}})\ \mathsf{sig}\ \mathsf{S_I}\ \mathsf{restr}\ \theta\ \mathsf{end}$ and:

$$\mathcal{E}_a \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t_s\} \tag{23}$$

Using Lemma E.3, we know that $\rho_a = \rho, (\mathsf{x} \mapsto \mathsf{m})$ is an $\mathcal{E}_a$-interpretation. Hence, using Equ. (23), we deduce that:

$$2\,\mathrm{supp}(\pi_1^{\#}(\llbracket \mathsf{s} \rrbracket_\nu^{\mathcal{E}_a, \rho_a})) \subseteq \psi \tag{24}$$

$$\mathsf{cost}_\nu^{\mathcal{E}_a, \rho_a}(\mathsf{s}) \leq t_s[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}_a) \\ g \in \mathsf{procs}_{\mathcal{E}_a}(\mathsf{A})}} t_s[\mathsf{A}.g] \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_a, \rho_a} \tag{25}$$

Using the fact that $\rho_a(\mathsf{x}) = \mathsf{m}$, we can check (by induction over $\#\mathsf{size}_{\mathcal{E}_a}^{\rho_a}(\mathsf{s})$) that:

$$\llbracket \mathsf{s} \rrbracket_\nu^{\mathcal{E}_a, \rho_a} = \llbracket \mathsf{s} \rrbracket_\nu^{\mathcal{E}_c, \rho} \qquad \text{and} \qquad \mathsf{cost}_\nu^{\mathcal{E}_a, \rho_a}(\mathsf{s}) = \mathsf{cost}_\nu^{\mathcal{E}_c, \rho}(\mathsf{s})$$

From the left equality above and Equ. (24), we know that Equ. (21) holds. It remains to prove Equ. (22).

From the right equality above and Equ. (25):

$$\begin{aligned}
\mathsf{cost}_\nu^{\mathcal{E}_c, \rho}(\mathsf{s}) \ &\leq\ t_s[\mathsf{conc}] \ + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}_a) \\ g \in \mathsf{procs}_{\mathcal{E}_a}(\mathsf{A})}} t_s[\mathsf{A}.g] \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_a, \rho_a} \\
&\leq\ t_s[\mathsf{conc}] \ + \sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot \mathsf{compl}_{\mathsf{x}.f}^{\mathcal{E}_a, \rho_a} \\
&\qquad + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}_c) \\ g \in \mathsf{procs}_{\mathcal{E}_c}(\mathsf{A})}} t_s[\mathsf{A}.g] \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_a, \rho_a}
\end{aligned}$$

We replace $\mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_a, \rho_a}$ by $\mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_c, \rho}$ for every $\mathsf{A} \in \mathsf{abs}(\mathcal{E}_c)$ and $g \in \mathsf{procs}_{\mathcal{E}_c}(\mathsf{A})$ using Lemma E.4:

$$\begin{aligned}
\mathsf{cost}_\nu^{\mathcal{E}_c, \rho}(\mathsf{s}) \ &\leq\ t_s[\mathsf{conc}] \ + \sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot \mathsf{compl}_{\mathsf{x}.f}^{\mathcal{E}_a, \rho_a} \\
&\qquad + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}_c) \\ g \in \mathsf{procs}_{\mathcal{E}_c}(\mathsf{A})}} t_s[\mathsf{A}.g] \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_c, \rho}
\end{aligned} \tag{26}$$

Using Lemma E.3, we upper-bound $\mathsf{compl}_{\mathsf{x}.f}^{\mathcal{E}_a, \rho_a}$ for every $f \in \mathsf{procs}(\mathsf{S_I})$:

$$\mathsf{compl}_{\mathsf{x}.f}^{\mathcal{E}_a, \rho_a} \leq t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ g \in \mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_f[\mathsf{A}.g] \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}, \rho}$$

We check that the quantities above are identical when evaluated in $\mathcal{E}_c$, hence:

$$\mathsf{compl}_{\mathsf{x}.f}^{\mathcal{E}_a, \rho_a} \leq t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}_c) \\ g \in \mathsf{procs}_{\mathcal{E}_c}(\mathsf{A})}} t_f[\mathsf{A}.g] \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_c, \rho}$$

Hence, re-organizing the terms in the sum in Equ. (26):

$$\begin{aligned}
&\mathsf{cost}_\nu^{\mathcal{E}_c, \rho}(\mathsf{s}) \leq t_s[\mathsf{conc}] + \sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot t_f[\mathsf{conc}] \ + \\
&\sum_{\mathsf{A} \in \mathsf{abs}(\mathcal{E}_c)} \sum_{g \in \mathsf{procs}_{\mathcal{E}_c}(\mathsf{A})} \Big(t_s[\mathsf{A}.g] + \sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot t_f[\mathsf{A}.g]\Big) \cdot \mathsf{compl}_{\mathsf{A}.g}^{\mathcal{E}_c, \rho}
\end{aligned}$$

Which, by definition of T_{ins}, is exactly Equ. (22). □