Mechanized Proofs of Adversarial Complexity and Application to Universal Composability
Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Pierre-Yves Strub

To cite this version:

HAL Id: hal-03469015
https://hal.science/hal-03469015
Submitted on 7 Dec 2021

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Mechanized Proofs of Adversarial Complexity and Application to Universal Composability

(long version)

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

Gilles Barthe
MPI-SP & IMDEA Software Institute
Bochum, Germany
gjbarthe@gmail.com

Benjamin Grégoire
Inria
Sophia Antipolis, France
benjamin.gregoire@inria.fr

Adrien Koutsos
Inria
Paris, France
adrien.koutsos@inria.fr

Pierre-Yves Strub
École Polytechnique
Palaiseau, France
pierre-yves@strub.nu

ABSTRACT
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 [23]. 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, 9], which has been used to prove security of a diverse set of cryptographic constructions in the computational model of cryptography [2, 3]. 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 [16]. This is an important limitation, as compositional approaches are ideally suited for proving security of complex cryptographic designs involving many layers of simpler building blocks. This work overcomes this limitation and showcases the benefits of reasoning about computational complexity in EasyCrypt, through 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 for reductionist statements relating adversarial advantage...
and execution cost, for instance:

\[ \forall A, \exists B. \text{adv}_S(A) \leq \text{adv}_T(B) + \epsilon \land \text{cost}(B) \leq \text{cost}(A) + \delta \]

where typically \( \epsilon \) and \( \delta \) are polynomial expressions in the number of oracle calls. The above statement says that every adversary \( A \) can be turned into an adversary \( B \), with sensibly equivalent resources, such that the advantage of \( A \) against a cryptographic scheme \( S \) is upper bounded by the advantage of \( B \) against a hardness assumption \( H \). Note that the statement is only meaningful because the cost of \( B \) is conditioned on the cost of \( A \), 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 [8] 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 [31] and reduce formulæ about extended integers to integer formulæ 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. Implementation-wise, this extension required to add or rewrite around 8 kLoC of EasyCrypt.

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 open source [20].

**Case study: Universal Composability.** Universal Composability [15, 17] (UC) is a popular framework for reasoning about cryptographic systems. Its central notion, called UC-emulation, formalizes when a protocol \( \pi_1 \) can safely replace a protocol \( \pi_2 \). Informally, UC-emulation imposes that there exists a simulator \( S \) capable of fooling any environment \( Z \) by presenting to it a view that is fully consistent with an interaction with \( \pi_1 \), while it is in fact interacting with \( S(\pi_2) \). 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 [15, 16]. 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 [19], 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 [18]; 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 [19], where modular proofs for Diffie-Hellman key exchange and encryption over ideal authenticated channels are composed to construct a UC secure channel.

**Discussion.** The possibility to quantify over adversary using complexity claims introduces conceptual simplifications in layered proofs by i. supporting compositional reasoning and ii. avoiding the use of explicit cost accounting modeling. The downside is that it also introduces some additional burden on users, who now must prove complexity claims. However, we note that our extension does not invalidate existing EasyCrypt developments: complexity claims are optional, existing proofs have been left unchanged, and their type-checking remains as fast as before. Furthermore, it is possible to layer the complexity claims on top of standard EasyCrypt proofs that do not capture the complexity aspects — in effect, this is what we did in our example. We have also provided rudimentary support to automate proofs of complexity claims, and could enhance this support even further by adopting ideas from cost analysis. We think that the current tool is significantly more usable and scalable than prior versions without support for complexity reasoning.

To make this claim more concrete, let us consider the implications of refactoring an existing EasyCrypt development and extend it to take advantage of cost analysis for both dealing with query counts and to include complexity claims. Removing the layer of modular wrapping that explicitly keeps track of query counts leads to more readable code, and has essentially no impact on the proofs. However, when it comes to complexity claims, new specifications and proof scripts must be added to the development. The new specifications consist of the description of the cost model and the declarations of the types of the various algorithms, which include explicit cost expressions. The additional proof effort consists of applying our logic to prove complexity claims and discharging the relevant side-conditions. As a coarse metric on the additional proof and specification efforts required, we consider the ratio of the number of lines of codes related to the cost analysis over the total number of lines. For the example presented in the next section, this ratio is 117/495. For the Universal Composability example, the
We first define module types for random oracles \(\mathsf{RO}\), schemes \(\mathsf{Scheme}\), and adversaries \(\mathsf{Adv}\). 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 parameterized by a random oracle \(o\). No cost declaration is necessary. The module type for (chosen-plaintext) adversaries declares two procedures: \(\mathsf{choose}\) for choosing two plaintexts \(m_0\) and \(m_1\), and \(\mathsf{guess}\) 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 function 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 \(\mathsf{inv}\) for the one-way trapdoor permutation. It runs the adversary \(A\), keeping track of all the calls that \(A\) makes to \(H\) in a list \(qs\) (using the sub-module \(\mathsf{QH}\)), and then searches in the list \(qs\) for a pre-image of \(y\) under \(f\). 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 \(\tau_A\) such that:

\[
\text{cost } A.\mathsf{choose} \leq \text{comp}[\text{intr} : t_c, H.o : k_c] \\
\text{and cost } A.\mathsf{guess} \leq \text{comp}[\text{intr} : t_g, H.o : k_g] \\
\text{and fix } \tau_f \text{ such that:}
\]

\[
\text{cost } I.\mathsf{invert} \leq \text{comp}[\text{intr} : (5+t_f)\cdot(k_c+k_g)+4+t_o\cdot(k_c+k_g)+t_c+t_g].
\]

Then, \(\forall A \in \tau_A, \exists I \in \tau_f\), \(\text{adv}^{\text{BR93}}_{\text{IND-CPA}}(A) \leq \text{adv}^{H.o}(I)\).

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 \(\mathsf{lnw}(A)\) as a witness for the existential quantification, which creates two sub-goals. 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 required cost restrictions, and is proved using our Hoare logic for complexity. We declare the type of \(\mathsf{lnw}\) as:

\[
\text{cost } \mathsf{lnw} \mathsf{invert} \leq \text{comp}[\text{intr} : (5+t_f)\cdot(k_c+k_g)+4],
\]

\[
H.o = k_c + k_g, A.\mathsf{choose} = 1, A.\mathsf{guess} = 1
\]

and so we first must establish that \(\mathsf{lnw}\) belongs to this functor type. It is easy to show that \(A.\mathsf{choose}\) and \(A.\mathsf{guess}\) 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 \(\mathsf{lnw}\). 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,\(^2\) using the cost of each operator—fixed by choice in this particular example to 1, except for the operator \(t\). Our modeling of cost enforces useful

---

\(^1\)We use the following notation: \(\xleftarrow{\$}\) denotes a random sampling; \(||\) is bit-string concatenation; \([-\,]\) is the empty list; \(\cdot\) appends a to the list \(l\).

\(^2\)Notice that the condition of the loop is executed at most \(k_c + k_g\) time.
invariants that simplify reasoning. For instance, proving upper bounds on the execution cost of \( \text{inv} \) 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 \( \text{inv} \), by instantiating the complexity type of \( \text{inv} \) 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 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 \( I \), and users must analyze the complexity of \( I \) 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 formalization 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.

### 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 \( \mathcal{F}_E \) and a set of distribution operators \( \mathcal{F}_D \). For any \( g \in \mathcal{F}_E \cup \mathcal{F}_D \), we assume given its type: \( \text{type}(g) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau \) where \( \tau_1, \ldots, \tau_n, \tau \in \mathcal{B} \) with \( \mathcal{B} \) the set of base types. We require that bool is a base type, and otherwise leave \( \mathcal{B} \) unspecified.

We consider well-typed arity-respecting expressions built from \( \mathcal{F}_E \) and variables in \( \mathcal{V} \). Similarly, distribution expressions \( d \) are built upon \( \mathcal{F}_D \) and \( \mathcal{V} \). For any expression \( e \), we let \( \text{vars}(e) \) 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 \( s_1; s_2 \), an assignment \( x \leftarrow e \) of an expression to a variable, a random sampling \( x \leftarrow d \) from a distribution expression, a conditional, a while loop, or a procedure call \( x \leftarrow \text{call } F(\vec{e}) \).

### 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 (this is only here to improve readability, not to convey additional information). 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 \( \theta \) 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.
whose code is unknown), restrictions allow to constrain, through
restrictive memory restrictions
restricts the variables that
memory restriction
restriction
f
restrict
module-level restriction
the module
adv
against a hardware security module. In this sce-
cryptographic reductions.
our approach, which we exploit to prove complexity properties of
the type system, its possible instantiations. This is a key idea of
declaration time. In the case of an abstract module (i.e. a module
module or abstract module declarations.
module type HSM = { proc enc (x:msg) : cipher }.
module Hsm : HSM = { proc enc (x:msg) : cipher = [ . . . ] }.
module type Adv (H : HSM)㈠all mem, -Hsm) = { proc guess (): sk : key ⊑ compl [intr : k, x, f1 : k1, . . . , x1, f1 : k1] }

Figure 3: Module signatures and restrictions

module type HSM = { proc enc (x:msg) : cipher }.
module type Adv (H : HSM)㈠all mem, -Hsm) = { proc guess (): sk ⊑ compl [intr : k, H.enc : k] }

Figure 4: Example of adversary with restrictions.

Restrictions are useful for compositional reasoning, as they al-
allow stating and verifying properties of a module’s procedures at
declaration time. In the case of an abstract module (i.e. a module
whose code is unknown), 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 sce-
nario 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 restriction㈠all mem, -Hsm) states that such an adver-
sary can access all the memory, except for the memory used by
the module Hsm. The procedure-level restriction [intr : k0, H.enc : k]
attached to guess, states that guess execution time is at most k0 (ex-
cluding 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 T (no restriction), or the conjunction of a
memory restriction λm and a complexity restriction λc:

Memory. A memory restriction λm, attached to a procedure f,
restricts the variables that f can access directly. We allow for posi-
tive memory restrictions (v1, . . . , vl), which states that f can only
access the variables v1, . . . , vl and negative memory restrictions
㈠all mem, (v1, . . . , vl), which states that f can access any global
variables except the variables v1, . . . , vl.

Note that λm 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 λc attached to a procedure
f restricts its execution time and the number of calls that f can
make to its parameters: it is either T, i.e. no restriction; or the
restriction compl [intr : k, x1, f1 : k1, . . . , x1, f1 : k1], which states that:
i) f’s execution time (excluding calls to the parameters) must
be at most k; ii) f can call, for every i, the parameter’s procedure
x1, f1 at most k1 times. We require that all parameters’ procedures
appear in the restriction. This can be done w.l.o.g. by assuming
that any missing entry is zero (which is exactly what is done in our
EasyCrypt implementation).

3.3 Typing Enriched Module Restrictions

We check that modules verify their signatures through a type sys-
tem. 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 E. Essentially,
an environment is a list of declarations, which are either variable,
module or abstract module declarations.

E ::= e | E, var v : τ | E, module x = m : M | E, module x = absopen : M1
An abstract module declaration module x = absopen : M1 states that
x is a module with signature M1 whose code is unknown, and allows
to model open code. For any E, we let (E) = {x1, . . . , xn} be the
set of abstract module names declared in E.

Restrictions. The RestrMem rule checks that a procedure body
㈠s: return e㈠ (where s is the procedure’s instructions, and e the
returned expression) verifies a memory restriction through a fully
automatic syntactic check.

RestrMem
(mem_E(s) ⊔ vars(e)) ⊑ λm
E ⊢ (㈠s; return e) ⪯ λm
This syntactic check uses mem_E(s) and vars(e), which are sound
over-approximations of an instruction and expression memory
topology (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

5Actually, the type system in Appendix B uses more complex environment, called
typing environment, to account for sub-modules.
6Abstract 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 Ans (in Figure 6) to remain tractable.

5
Hoare logic for cost. These proof obligations are discharged interactively using the proof system we present later, in Section 4.

\[ \text{RestrCompl} \]

\[
E \vdash \{ \top \} s \ {\psi} \mid t \quad \vdash s \ {\psi} r \leq t_r \quad (t + t_r \cdot \# \text{conc}) \leq \text{compl} \ \lambda_c \quad E \vdash \{ \_ s; \text{return } r \} \triangleright \lambda_c
\]

Here, the proof obligation \( E \vdash \{ \top \} s \ {\psi} \mid t \) states that the execution of \( s \) in any memory has a complexity upper bounded by \( t \), and that the post-condition \( \psi \) holds after \( s \)'s execution. The proof obligation \( \vdash s \ {\psi} r \leq t_r \) 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 \( \lambda_c \) through the premise \( (t + t_r \cdot \# \text{conc}) \leq \text{compl} \ \lambda_c \). We leave the precise definition of \( \text{compl} \) to Section 4 (see Figure 27). Intuitively, \( t \) is a record of entries of the form \((x, f \mapsto t_f)\), each stating that the abstract module \( x \)'s procedure \( f \) has been called at most \( l_c \) times, plus a special entry \((\text{conc} \mapsto t_{\text{conc}})\) stating that \( s \) execution time, excluding abstract calls, is at most \( l_c \). Then, \( t_0 \leq \text{compl} \ \lambda_c \) checks that \( t_0[x, f] \leq \lambda_c[x, f] \) for every functor parameter \( x, f \), and that \( \lambda_c[\text{intr}] \) 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-bounds of programs. This logic manipulates judgment of the form \( E \vdash \{ \phi \} s \ {\psi} \mid t \), where \( s \) is a statement, \( \phi, \psi \) are assertions, and \( t \) is a cost. We leave the assertion language unspecified, and only require that the models of an assertion formula \( \phi \) are memories, and write \( v \in \phi \) whenever \( v \) satisfies \( \phi \).

Essentially, the judgment \( E \vdash \{ \phi \} s \ {\psi} \mid t \) states that \( s \) is a program well-typed in the environment \( E \) (e.g. this means that \( s \) can only call concrete or abstract procedures declared in \( E \)), and that: i) the execution of the program \( s \) on any initial memory \( v_i \) satisfying the precondition \( \phi \) (i.e. \( v_i \in \phi \)) terminates in time at most \( t \); and ii), the final memory \( v_f \) obtained by executing \( s \) starting from \( v_i \) satisfies the post-condition \( \psi \) (i.e. \( v_f \in \psi \)).

#### 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 between concrete and abstract cost, a cost \( t \) is a record of entries mapping each abstract procedure \( x, f \) to the number of times this procedure was called, and mapping a special element \( \text{conc} \) to the concrete execution time (i.e. excluding abstract procedure calls). Since the set of available abstract procedures (and consequently the number of entries in the cost \( t \)) depends on the current environment \( E \), we parameterize the notion of cost by the environment \( E \) considered.

**Definition 4.1.** A \( E \)-cost is an element of the form:

\[ t := [\text{conc} \mapsto k, x_1.f_1 \mapsto k_1, \ldots, x_i.f_i \mapsto k_i] \]

where \( E \) is an environment, \( k, k_1, \ldots, k_i \) are integers, and the \( x_i.f_i \) are all the abstract procedures declared in \( E \).

**Example 4.1.** Consider \( E \) with two abstract modules \( x \) and \( y \):

\[ E = (\text{module } x = \text{absopen} \cdot \text{sig (proc } f \_\text{ restr } \_\text{end}); \]

\[ (\text{module } y = \text{absopen} \cdot \text{sig (proc } h \_\text{ restr } \_\text{end}) \]

Then \([\text{conc} \mapsto 10; x, f \mapsto 0; y, h \mapsto 3]\) represents a concrete cost of 10, at most three calls to \( y, h \), and none to \( x, f \).

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

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:

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

is valid whenever \( t_{\text{conc}} \) upper-bounds 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 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 \( t_{\text{conc}} \).

**Expression cost.** We have a second kind of judgment \( \vdash \{ \phi \} e \leq t_e \), which states that the cost of evaluating \( e \) in any memory satisfying \( \phi \) is at most \( t_e \), where \( t_e \) is an integer, not a \( E \)-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 present our Hoare logic for cost, which allows to prove cost judgments of programs. Our logic has one rule for each possible program construct (assignment, loop,...), plus some structural rules (e.g. weakening). We only describe a simple Hoare rule for conditional construct, and then explain a core rule of our logic, which handles abstract calls. All other rules are given in Appendix F. 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 rule:

\[ \vdash \{ \phi \} e \leq t_e \quad E \vdash \{ \phi \wedge e \} s_1 \{ \psi \mid t \} \quad E \vdash \{ \phi \wedge \neg e \} s_2 \{ \psi \mid t \}
\]

\[ E \vdash \{ \phi \} \text{ if } e \text{ then } s_1 \text{ else } s_2 \{ \psi \mid t + t_e \}
\]

state that if: i) the evaluation of the condition \( e \) takes time at most \( t_e \); ii) the execution of the then branch program \( s_1 \), assuming pre-condition \( \phi \wedge e \), guarantees the post-condition \( \psi \) and takes
time at most \( t \); iii) and the execution of the else branch, assuming the pre-condition \( \phi \land -e \), guarantees the same post-condition \( \psi \), and also takes time at most \( t \); then the full conditional statement

**if** \( e \) **then** \( s_1 \) **else** \( s_2 \), assuming pre-condition \( \phi \), guarantees the post-condition \( \psi \) in time at most \( t + \epsilon \). Note that we use the same cost upper-bound \( t \) for both branches: essentially, \( t \) can be chosen to be the maximum of the execution times of the then and else branches.

Other rules are more involved, and require the user to show simultaneously invariants of the memory state of the program and cost upper-bounds.

**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**

\[
\begin{align*}
E(x) &= \text{abs-open } x \langle \overline{p} \rangle ; f \\
FV(I) \land \lambda_m = \emptyset \\
\theta[f] &= \lambda_m \land -
\end{align*}
\]

First, the function path \( F \) is resolved to \( \text{abs-open } x \langle \overline{p} \rangle ; f \), i.e. a call to the procedure \( f \) of an abstract functor \( x \) applied to the modules \( \overline{p} \) (the case where \( x \) is not a functor is handled by taking \( \overline{p} = e \)). Then, \( x \)'s module type is lookup in \( E \), and we retrieve the module restriction \( \theta \) 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 \( x.f \), excluding calls to the functor parameters. This is done by checking that \( x.f \) cannot access the variables used in \( I \), using its memory restriction \( \lambda_m \) (looked-up by the premise \( \theta[f] = \lambda_m \land - \) and requiring that \( FV(I) \land \lambda_m = \emptyset \).

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

**Abstract call.** We now present our **Abs** rule for cost judgments, which is given in Figure 6. Essentially, the cost of the call to \( x \langle \overline{p} \rangle ; f \) is decomposed between:

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

\[
\begin{align*}
\text{module type } &\text{T}_C = \{ \text{proc } c : \text{unit} \}; \\
\text{module C} &= \{ \text{proc } c t \} \times \{ s_0 \} \\
\text{module type } &\text{T}_B(C_0 : T_C) = \{ \text{proc } b : \text{unit comp[intr: } t_{ab}, C_0.b : 1, C_0.c : 1 \} \\
\text{module type } &\text{T}_A(B_0 \times T_C : C_0 : T_C) = \{ \\
\text{proc } a &() = \text{unit comp[intr: } t_{ab}, B_0.b : 1, C_0.b : 1 \} \\
\text{module A} &= (B_0 : T_B) (C_0 : T_C) : T_A = \{ \\
\text{proc } a &() = \{ s_1 : C_0.b(); s_2 : B_0.b(); s_3; \}
\end{align*}
\]

**Figure 5:** Graphical representation of the different cost measurements.

\[
\begin{align*}
\text{abs-partial} & \text{ F-Res} (F) = (\text{abs-open } x \langle \overline{p} \rangle ; f) \\
E(x) &= \text{abs-open } x \langle \overline{p} \rangle ; f \\
\forall p_0 \in \overline{p}, \forall g \in \text{proc}_{\overline{E}}(p_0), E \vdash \{ I \} p_0.g \{ I \}
\end{align*}
\]

**Abs-PARTIAL**

\[
\begin{align*}
E(x) &= \text{abs-open } x \langle \overline{p} \rangle ; f \\
FV(I) \land \lambda_m = \emptyset \\
\theta[f] &= \lambda_m \land -
\end{align*}
\]

We require, for every \( i \), a bound on the cost of the \( k \)-th call to the functor argument \( z_j \), \( f_i \), where \( k \) can range anywhere between \( 0 \) and the maximum number of calls \( x.f \) make to \( z_j \), which is \( K_j \). The cost of the \( k \)-th call to \( z_j.f_i \) is bounded by \( (t_k k) \) where \( k = \tilde{k}[i] \) and:

\[
E \vdash \{ I \} \tilde{k}[i] ; f_i \{ I \tilde{k} + 1 \} | t_k k
\]

To improve precision, we let the invariant \( I \) depend on the number of calls to the functor parameters through the integer vector \( \tilde{k} \). After calling \( \tilde{k}[i] ; f_i \), we update \( \tilde{k} \) by adding one to its \( i \)-th entry \( (1 \leq i \leq n) \) and all other entries are zero.

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

**4.3 Soundness**

We define a formal denotational semantics of our language and module system, and use it to prove the soundness of our rules. For space reasons, we omit the details here (they can be found in Appendix E and F), and only state the main soundness theorem.

**Theorem 4.1.** The proof rules in Figures 6, 26 and 27 are sound.
5 EXAMPLE: UNIVERSAL COMPOSABILITY

UC security guarantees that a protocol $\pi_1$ can safely replace a protocol $\pi_2$ while preserving both the functionality and the security of the overall system. The most common application of this framework is to set $\pi_2$ 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 $F$. An ideal functionality $F$ defines what protocol $\pi_1$ should achieve both in terms of correctness and security to securely replace the TTP. Moreover, $F$ can be used as an ideal sub-component when designing higher-level protocols, which then can be instantiated with protocol $\pi_1$ 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 [15, 16]. 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 [8], 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 [19, 22, 26]. 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.

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 [18]. As in [18], 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 [18], 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 of [18] by explicitly defining a hybrid real-world, in which concrete modules for ideal authenticated channels are available to the communicating parties. We discuss the trade-offs associated with our approach more in depth at the end of this section, drawing a parallel to the work in [19].

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 formalization 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 7. 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 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) I/O 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).
Figure 8: Execution model for real/ideal worlds (top) and composition of functionality with a simulator (bottom).

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 8). 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.\footnote{Real-world settings using ideal functionalities as sub-components are called hybrid.} 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 $\pi_1$ and $\pi_2$: if $\pi_1$ UC-emulates $\pi_2$ with small advantage $\epsilon$ then $\pi_1$ can replace $\pi_2$ in any context (within a complexity class).

Definition 5.1 (UC emulation). Protocol $\pi_1$ UC-emulates $\pi_2$ under complexity restrictions $c_{\text{sim}}$ and $c_{\text{env}}$ and advantage bound $\epsilon$ if

$$\exists S \in \pi_{\text{sim}}, \forall Z \in \pi_{\text{env}}, \text{ Pr}[Z(\pi_1) : T] - \text{Pr}[Z((\pi_2 \parallel S(\pi_2)) : T)] \leq \epsilon$$

We write this as $\text{Adv}_{\text{uc}}(\pi_{\text{sim}},\pi_{\text{env}})(\pi_1,\pi_2) \leq \epsilon$.

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$_{\text{emul}}$ parameterized with $\text{ENV} = Z$ 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 9 (right), $\pi_2$ is typically an ideal functionality; this corresponds to game UC$_{\text{emul}}$ parameterized with $\text{ENV} = Z$ and a protocol $P$ that results from attaching $S$ to the BACKDOORS interface of $\pi_2$. We denote this ideal-world $P$ by $(\pi_2 \parallel S(\pi_2))$, corresponding to the EasyCrypt functor $\text{CompS}$ also shown in Figure 8.

UC-emulation imposes that a simulator $S$ 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 $\pi_2$ allows. If such a simulator exists, then clearly $\pi_2$ cannot be worse than $\pi_1$ in the information it reveals to the environment via its BACKDOORS interface.\footnote{The emulation notions in [15, 16] quantify only 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 [15, Section 4.4] for a discussion of quantitative UC definitions such as the one we adopt. For this reason, as we show below, we also do not need to keep the dummy adversary explicitly in the real world.} 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 9, 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 $Z$ and $S$ in more detail. We first note that the definition of emulation is parametric in the resource restrictions $c_{\text{sim}}$ and $c_{\text{env}}$. Clearly the IO interface of $\pi_2$ must match the type of the IO interface of $\pi_1$, which is consistent with the goal that $\pi_1$ can replace $\pi_2$ 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 $\pi_2$ 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 $S$ is given by $\tau_{\pi_1, \pi_2, c_{\text{sim}}}$, which defines the type of modules that have access to the BACKDOORS interface of $\pi_2$; exposes the BACKDOORS interface of $\pi_1$ and is restricted memory-wise to exclude the memory of $\pi_2$ and resource-wise by $c_{\text{sim}}$. Note that, if $S$ 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 $\tau_{\pi_1, \pi_2, \Sigma, c_{\text{env}}}$, the type of modules that have oracle access to the IO and BACKDOORS interfaces of $\pi_1$, and are restricted memory-wise to exclude the memories of $\pi_1$, $\pi_2$ and $S$, and resource-wise by $c_{\text{env}}$. In this case, if the environment could look inside $\pi_1$, $\pi_2$ or $S$ 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 $c_{\text{env}}$ is of the form:

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

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 $\pi_1$ UC-emulates $\pi_2$ and this, in turn, UC-emulates $\pi_3$, then $\pi_1$ UC-emulates $\pi_3$. When stating this lemma in EasyCrypt 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 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 $\text{Adv}^{uc, S}(\cdot, \cdot)$ notation by indicating the universally quantified simulator $S$ in superscript).

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

$$\text{Adv}^{uc, S_{1, 2}}_{\text{env}(1, 2), \text{env}(1, 3)}(\pi_1, \pi_2) \leq \epsilon_{1, 2} \Rightarrow \text{Adv}^{uc, S_{2, 3}}_{\text{env}(2, 3), \text{env}(1, 3)}(\pi_2, \pi_3) \leq \epsilon_{2, 3} \Rightarrow \text{Adv}^{uc, S_{1, 2}}_{\text{env}(1, 2), \text{env}(1, 3)}(\pi_1, \pi_3) \leq \epsilon_{1, 2} + \epsilon_{2, 3}$$

where $\epsilon_{1, 2}$ and $\epsilon_{2, 3}$ are chosen to preserve security and $\epsilon_{1, 2} + \epsilon_{2, 3}$ is the total additional cost. This result is immediate from the definitions; it states that UC-emulation is a (binary) relation on protocols. The transitivity of the relation is preserved in the definition as well as the cost restrictions on the simulators. In particular, $\epsilon_{1, 2}$ and $\epsilon_{2, 3}$ need not be the same for the two calls to the $\text{Adv}^{uc, S}(\cdot, \cdot)$ function; only their sum needs to be bounded by $\epsilon_{1, 2} + \epsilon_{2, 3}$.
that UC-emulates \( f \). The proof first shows that the simulator \( S_\pi \) that exists by hypothesis can be converted into a simulator that justifies that \( \rho(\pi) \) UC-emulates \( \rho(f) \): intuitively this new simulator uses \( S_\pi \) when interacting with the backdoors of \( f \) and just passes along the environment’s interactions with the backdoors of \( \rho \). This part of the proof combines any successful environment \( Z \) against the composed protocol into a successful environment that absorbs \( \rho \) and breaks \( \pi \). This justifies the cost restriction on \( c_{\text{env}} \). Then, we know by hypothesis that \( \rho(f) \) UC emulates \( F_\pi \), 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 10. 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 \( F_{\text{Auth}} \) exposing IO interface \( \Pi_{\text{REAL}}(\text{IO}) \)). The full real-world configuration is obtained by applying a functor \( \text{CompRF} \) that composes this protocol with \( F_{\text{Auth}} \) and exposes the backdoors of both \( \text{DHKE} \) and \( F_{\text{Auth}} \) in a combined \( \text{BACKDOORS} \) 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.

The initiator code is shown in Figure 11. On initialization it samples its ephemeral key pair and resets the derived key. When the environment provides input, which includes the identities of the parties that will take part in the key exchange, the ephemeral public key is transmitted via one of the ideal authenticated channels. The party then returns control to the environment (note that delivering a message to the authenticated channel does not pass control to the authenticated channel). When the environment calls step, the initiator checks the incoming ideal channel to see if it received a message. At any point the environment can check the initiator output using output. The backdoor interface provides no information, since all communications go through the authenticated channels. The responder code is symmetric.

In the middle code-snippet of Figure 10 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 backdoor procedure to observe the state of the channel, including the transmitted message and the party identities and it can use the step procedure to control when the message is delivered (the unlock operator changes the state so that, if a message is buffered, then it is made available at the output procedure) to the receiving party (get_message is checking for identity consistency, which models authentication).

The example starts with a proof that the Diffie-Hellman protocol on the left of Figure 10 UC-emulates the ideal functionality for key exchange shown on the right of Figure 10 in a 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 \( c_{\text{ddh}} \in \mathbb{R}^+ \) and let \( \epsilon_{\text{DDH}} \) be the maximum advantage of any DDH attacker against the group over which we implement DHKE. Then, we have that

\[
\text{Adv}_{\text{env}(\text{DHKE})}^{\text{uc}}(\text{DHKE}(\text{FKE}), \text{FKE}) \leq \epsilon_{\text{DDH}}
\]

where \( \text{Adv}_{\text{env}(\text{DHKE})}^{\text{uc}} \) is the cost of a concrete simulator \( S_{\text{DHKE}} \) that just samples random group elements as the protocol messages and mimics the states of the real-world parties and \( \text{FKE} \); \( \epsilon_{\text{env}(\text{DHKE})} \) must be such that \( \epsilon_{\text{ddh}} \) 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 \( c_{\text{env}(\text{OTP})} \). Then we have

\[
\text{Adv}_{\text{env}(\text{OTP})}^{\text{uc}}(\text{OTP}(\text{FKE}, \text{FAuth}), \text{FSC}) = 0
\]

where \( \text{Adv}_{\text{env}(\text{OTP})}^{\text{uc}} \) is the cost of a concrete simulator \( S_{\text{OTP}} \) 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 a 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 \( c_{\text{ddh}} \in \mathbb{R}^+ \) and let \( \epsilon_{\text{DDH}} \) be the maximum advantage of any DDH attacker against the group over which we implement DHKE. Then

\[
\text{Adv}_{\text{env}(\text{OTP})}^{\text{uc}}(\text{OTP}(\text{DHKE}, \text{FAuth}), \text{FSC}) \leq \epsilon_{\text{DDH}}
\]

where \( \epsilon_{\text{env}} \) is constrained so that \( \epsilon_{\text{env}(\text{DHKE})} \) accommodates an environment that internally uses \( \epsilon_{\text{env}} \) resources and additionally runs OTP, and \( \epsilon_{\text{sim}} \) corresponds to the cost of composing \( S_{\text{OTP}} \) and \( S_{\text{DHKE}} \).

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 \( c_{\text{ddh}} \), as otherwise the reduction to DDH would be meaningless. Indeed, the class of DDH attackers must allow for the
Our formalization vs EasyUC. Our Diffie-Hellman example is an alternative formalization of the example given by Canetti, Stoughton and Varia [19] for the EasyUC framework. We borrow it because, as in [19], it is a good toy example with which to validate and demonstrate our formalization. This example is also convenient to show that the approach in this paper and EasyUC in effect complement each other. An important design goal of EasyUC is to follow the UC execution model as closely as possible; this allows a more direct translation of protocols and ideal functionalities.

In contrast, our goal is to take advantage of the EasyCrypt machinery to reduce proof effort and development size: our development (including complexity) takes 2300 lines of code and it includes general UC theorems that can be reused in future work; this compares to 18K lines of code for EasyUC.9 The downside of our approach is the impact in the way one specifies protocols and ideal functionalities: message passing corresponds to procedure calls, and these must adhere to the EasyCrypt tree-based procedure call semantics. For example, we do not allow an execution environment where a party communicates with an ideal functionality arbitrarily without relying on the environment for scheduling; one could of course formalize a message passing mechanism on top of EasyCrypt as in [19] to allow for this, but this would then fall out of the scope of our general composition theorems. Moreover, it would lead to larger developments and increased proof effort, which would defeat our original purpose.

In short, 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 direction for future work to develop a sound translation between these two approaches to modeling UC for a representative class of protocols such as those considered in [18]. Another interesting direction for future work is to identify UC security proofs that cannot be naturally expressed using our approach to formalizing UC and to investigate how it can be extended to deal with these examples.

6 RELATED WORK

Cost analysis. There is a very large body of work that uses program logics for cost analysis of imperative programs. [28] uses Hoare logic for proving upper bounds on execution time of deterministic programs. In the probabilistic setting, [24] uses a preexpectation calculus inspired from Kozen [25] and Morgan, McIver and Seidel [27] to compute upper bounds on the expected cost of probabilistic programs. In contrast, cryptography primarily considers worst-case execution times. In addition, there is a long line of work on automating cost analysis, both for deterministic and for probabilistic programs, see e.g. [1, 13, 21]. These techniques could be helpful to alleviate users efforts, and connecting with tools that support them is an important direction for future work.

9The count excludes general purpose libraries, but we should note that the exact numbers are not important, as the size of a development varies significantly with style of coding and the use of automation.

---

Figure 10: Examples of real-world (left) and ideal-world protocols (middle and right). Left: structure of a Diffie-Hellman protocol relying on FAuth for authenticated communication (one shot each way). Middle: ideal functionality for one-shot authenticated channel FAuth. Right: ideal functionality for key exchange.

Figure 11: Diffie-Hellman Initiator.
We have developed an extension of the shown how to formalize key results from Universal Composability, and supports compositional reasoning. As a main example, we have support reasoning complexity claims. The extension captures reduc-

7 CONCLUSION

Our module system is inspired from EasyCrypt [6, 9]. 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 [14] 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. [30] introduces the notion of interface, which is similar to module, for formalizing cryptography.

We have developed an extension of the EasyCrypt proof assistant to support reasoning complexity claims. The extension captures reduction 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

Consider the functor in Figure 12, which extends any real-world protocol with abstract adversary $\mathcal{A}$ (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 quantify over such adversaries, we quantify also over the potential forms of environment-to-adversary information exchange. The following theorem shows 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 execution with an explicit real-world adversary. More precisely:

- Emulation with an implicit dummy adversary implies emulation with an explicit arbitrary adversary: For all $\epsilon \in \mathbb{R}^+$, all protocols $\pi_1$ and $\pi_2$ with $iO$ interfaces of the same type, all complexity restrictions $\epsilon_{\text{sim}}, \epsilon_{\text{env}}$, and all simulators $S \in \mathcal{P}_{\mathcal{A}}^{\pi_1, \pi_2, \epsilon_{\text{sim}}}$, we have

\[
\text{Adv}^{\text{uc}, S}_{\text{sim}, \epsilon_{\text{env}}}(\pi_1, \pi_2) \leq \epsilon \implies \forall S \in \tau_{\text{adv}}, \text{Adv}^{\text{uc}, S}_{\epsilon_{\text{sim}}, \epsilon_{\text{env}}}(\langle \pi_1 \parallel \mathcal{A}(\pi_1) \rangle, \pi_2) \leq \epsilon
\]

where $\epsilon_{\text{sim}}$ allows for a simulator $S'$ that combines adversary $\mathcal{A}$ and simulator $S$.

- Emulation with an explicit dummy adversary is implied by emulation with an explicit arbitrary adversary: For all $\epsilon \in \mathbb{R}^+$, all protocols $\pi_1$ and $\pi_2$ with $iO$ interfaces of the same type, all complexity restrictions $\epsilon_{\text{sim}}, \epsilon_{\text{env}}$, and all simulator memory spaces $M$, we have

\[
\forall S \in \tau_{\text{adv}}, \text{Adv}^{\text{uc}, \mathcal{M}, S}_{\epsilon_{\text{sim}}, \epsilon_{\text{env}}}(\langle \pi_1 \parallel \mathcal{A}(\pi_1) \rangle, \pi_2) \leq \epsilon \implies \text{Adv}^{\text{uc}, \mathcal{M}}_{\epsilon_{\text{sim}}, \epsilon_{\text{env}}}(\pi_1, \pi_2) \leq \epsilon
\]

where $\tau_{\text{adv}}$ accommodates the dummy adversary.

Our proof gives a simulator $S'$ for the first part of the theorem that joins together simulator $S$ and adversary $\mathcal{A}$: intuitively the new simulator uses the existing one to foil 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 $S'$ are those required to run the composition of $S$ 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’

---

**Figure 12: Real-world protocol with adversary.**

---

**Appendix Outline.** We quickly outline the structure of the appendix. We show that the standard dummy adversary theorem holds in our UC modeling in Appendix A. We present the type system for our programming language and module system in Appendix B. Then, we present the semantics of our module system. This module semantics takes the form of a module resolution mechanism, which describe how module expressions are evaluated, and is given in Appendix C. Appendix D proves that modules have the subject reduction property: essentially, evaluating a module expression preserves its type. This is a crucial property, as it proves that module restrictions are preserved by the evaluation of a module using the module resolution mechanism. Then, we define the semantics of our programming language and of the cost judgment in Appendix E. Finally, we present the full set of rules of our Hoare logic for cost and prove their soundness in Appendix F.

**A. THE DUMMY ADVERSARY IN UC**

The standard notion of UC emulation 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. 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 $S$ such that, for all environments $E$, the real- and ideal- worlds are indistinguishable. We now show that the same result holds in our setting.

---

[For this reason the simulator is often called an ideal world adversary; we do not adopt this terminology here to avoid confusion.]
We now present the core rules of our module type system, which we quantify existentially in the hypothesis. We note a technicality in 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 $M$, restrict simulators in the hypothesis to only use $M$, and prevent other algorithms to interfere with this memory space where appropriate (we abuse notation by indicating $M$ in $\text{Adv}^{\text{uc}}$ 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 13.

#### B.1 Program and Module Typing

We now present the core rules of our module type system, which are summarized in Figure 14 and Figure 15. The rest of the rules are postponed to Appendix B.2. For clarity of presentation, our module type system requires module paths to always be long module paths, from the root of the program to the sub-module called (we give a simple example in Figure 16). 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 $\Gamma$ is a list of typing declarations. A typing declaration, denoted $\delta$, is either a variable, module, abstract module or procedure declaration, with a type.

$$\delta ::= \text{var } v : \tau \mid \text{module } p = m : M \mid \text{module } x = \text{abs}_x : M$$

$$\mid \text{proc } p, f(\vec{v} : \vec{\tau}) \mapsto \tau_r = \text{body}$$

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

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

An abstract module declaration module $x = \text{abs}_x : M$ 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: a memory restriction ignores the memory footprint of a functor parameter, and a complexity restriction restricts the number of calls that can be made by parameters’ procedures. 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 $p, x = m : M$) 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 $\Gamma$ as a partial function from variable names $v$, module paths $p$ or procedure paths $p, f$ to (base, module, abstract modules or procedure) values and their types, defined as follows:

$$\Gamma(v) = \tau \quad \text{(if } \Gamma = \Gamma_1; \text{ var } v : \tau; \Gamma_2)$$

$$\Gamma(p) = m : M \quad \text{(if } \Gamma = \Gamma_1; \text{ module } p = m : M; \Gamma_2)$$

$$\Gamma(x) = \text{abs}_x : M \quad \text{(if } \Gamma = \Gamma_1; \text{ module } x = \text{abs}_x : M; \Gamma_2)$$

$$\Gamma(p, f) = \text{proc } f(\vec{v} : \vec{\tau}) \mapsto \tau_r = \text{body}$$

(\text{if } \Gamma = \Gamma_1; \text{ proc } p, f(\vec{v} : \vec{\tau}) \mapsto \tau_r = \text{body}; \Gamma_2)

and $\Gamma(\vec{z}) = \text{undef}$ otherwise. Also, we write $\Gamma(\vec{z})\not\in_{\text{undef}}$ when $\Gamma(\vec{z}) = \text{undef}$ for any prefix $\vec{z}'$ of $\vec{z}$.

#### Abstract Modules

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

$$M_l ::= \text{sig } S_1 \text{ restr } \theta \text{ end } \mid \text{func}(x : \text{sig } S_1 \text{ restr } \theta \text{ end}) \ M_l$$

$$S_l ::= D_{l_1} ; \ldots ; D_{l_n} \quad D_l ::= \text{proc } f(\vec{v} : \vec{\tau}) \mapsto \tau_r$$

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 $\text{Abs}$ presented in Figure 6 to remain tractable.

For any $M_l$, we let $\text{procs}(M_l) = \{ f_1, \ldots, f_n \}$ be the set of procedure names declared in $M_l$.

#### Typing Module Paths

The typing judgment $\Gamma \vdash p : M$ states that the module path $p$ refers to a module with type $M$. Its typing rules, which are given in Figure 14, are standard, except for the functor application typing rule $\text{FuncApp}$:

$$\text{FuncApp}$$

$$\Gamma \vdash p : \text{func}(x : M') \ M \quad \Gamma \vdash p' : M'$$

$$\Gamma \vdash p(p') : \text{M}[x \mapsto \text{mem}_p(p')]$$

A key point here is that we need to substitute $x$ in the module signature. The substitution function is standard (see Figure 20), 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 $\text{mem}_p(p')$. This is handled when substituting $x$ in a memory restriction:

$$\lambda_m[x \mapsto \text{mem}_p(p')] = \lambda_m \cup \text{mem}_p(p')$$

- 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.

$$\text{comp}[\text{intr } k, y_1, f_1 : k_1, \ldots, y_1, f_1 : k_1][x \mapsto \_1] = \text{comp}[\text{intr } k, (y_1, f_1 : k_1)[x \mapsto \_1], \ldots, (y_1, f_1 : k_1)[x \mapsto \_1]]$$

where $\{ y, f : k \}[x \mapsto \_] = \{ \epsilon \text{ if } y = x \}$

$\{ y, f : k \}$ otherwise

\footnote{Meaning that the (variable, module or procedure) path $z$ is not declared by $\Gamma$, even through a sub-module or functor application.}
Expression (distribution expressions are similar):

e := v ∈ V  (variable)  
| f(e₁, . . . , eₙ) (if fᵣ ∈ Tₑ)

Statements:

s := abort  (abort)  
| skip  (skip)  
| s₁; s₂  (sequence)  
| x ← e  (assignment)  
| x ← d  (sampling)  
| x ← call f(ℓ)  (procedure call)  
| if e then s₁ else s₂  (conditional)  
| while e do s  (loop)

Procedure body:

body := { var (ℓ : τ); s; return e }

Module paths:

p := x  (mod. ident.)  
p; x  (mod. comp.)  
p(p)  (func. app.)

Function paths:

F := p, f  (proc. lookup)

Module expressions:

m := p  (mod. path)  
| struct st end  (structure)  
| func(x : M) m  (functor)

Module structures:

st := d₁; . . . ; dₙ  (for any n ∈ ℕ)

Module declarations:

d := proc f(ℓ : τ) → τᵣ = body  (proc.)  
| module x = m  (module)

Signature structures:

S := D₁; . . . ; Dₙ  (for any n ∈ ℕ)

Module signature declarations:

D := proc f(ℓ : τ) → τᵣ  (procedure decl.)  
| module x : M  (mod. decl.)

Module signature:

M := sig S restr θ end  (restr. sig. struct.)
| func(x : M) M'  (functor)

Typing environment declarations:

δ := var v : τ  (variable decl.)  
| module p = m : M  (module decl.)  
| module x = absk : M  (abs. mod.)  
| proc p,f(ℓ : τ) → τᵣ = body  (proc. decl.)

Abstract module kind:

K := open  (open module)  
| param  (module parameter)

Typing Environment:

Γ := e | Γ, δ

Module restrictions:

θ := e | θ, (f : λ)  
λ := τ | λᵣ + λᵦ

Memory restrictions:

λᵣ := +all mem\{v₁, . . . , vₙ\}  (for any l ∈ ℕ)  
| {v₁, . . . , vₙ}  (for any l ∈ ℕ)

Complexity restrictions:

λᵦ := τ | compl[intr : k, x₁.f₁, . . . , xₙ.fₙ, k₁]  (for any l, k, k₁ ∈ ℕ)

Low-order module signature:

M₁ := sig Sᵣ restr θ end  (restr. sig. struct.)  
| func(x : sig Sᵣ restr θ end) M₁  (low-order func.)

Low-order signature structures:

S₁ := D₁; . . . ; Dₙᵣ  (for any n ∈ ℕ)

Low-order module signature declarations:

D₁ := proc f(ℓ : τ) → τᵣ  (procedure decl.)

Extended module expressions:

m := m | absk x

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 Γ,p : 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 Γ,p,φ st : S is annotated by both the module path of the structure being typed, and the module restriction φ that the structure must verify. Remark that when we type a procedure using PROCDECL, we check that the procedure f body satisfies the module restriction φ[f] by requiring that the restriction checking judgment Γ ⊢ φ[f] holds.
Module path typing $\Gamma \vdash p : M$.

<table>
<thead>
<tr>
<th>NAME</th>
<th>COMPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma(p) = _{-} : M$</td>
<td>$\Gamma \vdash p : \text{sig } S_1; \text{module } x : S_2$ restr $\theta$ end</td>
</tr>
<tr>
<td>$\Gamma \vdash p : M$</td>
<td></td>
</tr>
</tbody>
</table>

$\text{FUNCAPP}$

$\Gamma \vdash p : \text{func}(M') M \quad \Gamma \vdash p' : M'$

$\Gamma \vdash p\{p'\} : [M[x \mapsto \text{mem}_M(p')]]$

Module expression typing $\Gamma \vdash p \cdot M : M$.

We omit the rules $\Gamma \vdash M$ to check that a module signature $M$ is well-formed.

<table>
<thead>
<tr>
<th>ALIAS</th>
<th>STRUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash p_a : M$</td>
<td>$\Gamma \vdash p_a \cdot M; S$</td>
</tr>
<tr>
<td>$\Gamma \vdash p \cdot M$</td>
<td>$\Gamma \vdash p : \text{struct } S \vdash : \text{sig } S$ restr $\theta$ end</td>
</tr>
</tbody>
</table>

$\text{PROCDECL}$

$\text{body} = \{ \text{var } (\xi_1 : \xi_1); \text{ return } r \}$

$\xi_1, \xi_1 \text{ fresh in } \Gamma \quad \Gamma(f) = \Gamma, \text{ var } \xi_1 : \xi_1, \text{ var } \xi_1 : \xi_1$

$\Gamma \vdash s \quad \Gamma(f) = \Gamma, \text{ var } \xi_1 : \xi_1, \text{ var } \xi_1 : \xi_1$

$\Gamma(p,f) \{\text{undet} \} \quad \Gamma, \text{ proc } p(f(\xi_1 : \xi_1) : \xi_1) \rightarrow r_1 = \text{ body } \Gamma, \text{ st } : S$

$\Gamma \vdash p(f_1(\xi_1 : \xi_1) : \xi_1) \rightarrow r_1$ (body $\Gamma$, st $S$) (proc $f_1(\xi_1 : \xi_1) : \xi_1) \rightarrow r_2$)

$\text{MODDECL}$

$\Gamma \vdash p_x : M \quad \Gamma(p_x) \{\text{undet} \} \quad \Gamma, \text{ module } p_x : m = M \vdash p, \text{ st } : (\text{module } x : M; S)$

$\Gamma \vdash p, \text{ st } : (\text{module } x : M; S)$

$\text{STRUCTEMP}$

$\Gamma \vdash \epsilon : \epsilon$

Environments typing $\vdash E$

<table>
<thead>
<tr>
<th>ENVEMP</th>
<th>ENVSEQ</th>
<th>ENVVAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash \epsilon$</td>
<td>$\vdash \epsilon; \delta$</td>
<td>$\vdash (\epsilon {\text{v} : \tau} {\text{undet} })$</td>
</tr>
</tbody>
</table>

$\text{ENVMOD}$

$\vdash (\text{module } x : M)\{\text{undet} \} \quad \vdash (\text{module } x : \text{absx} : M)$

Figure 14: Core typing rules.

The rule $\text{RestrMemExt}$ in Figure 15 is more general than the $\text{RestrMem}$ rule presented in the body, as it allows typing a memory restriction in any typing environment, not only in an environment $E$. Crucially, the complexity checking rule $\text{RestrCompl}$ is not extended to typing environment, because the cost Hoare judgment $E \vdash \{\text{T} \} \epsilon \{\text{v} \} \{\text{t} \}$ is not defined for typing environment.

Remark B.1. While we could probably extend $\text{RestrCompl}$ to allow typing in a typing environment $E$, 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 $\text{absx} : M$ by a concrete module $m$ of the same type $M$ (because an environment $E$ cannot contain a declaration of an abstract module of kind $\text{param}$, only of open modules of kind $\text{open}$, which are never substituted, only instantiated). Instead, we only need to show closure under such substitution for typing judgment (not Hoare logic derivations), which makes the proof simpler.

B.2 Additional Typing Rules

The memory restriction union $\sqcup$, intersection $\cap$ and the memory restriction subset $\subseteq$ operations are defined in Figure 18. In Figure 19, we present our sub-typing rules, our typing rules for statements and expressions, and the definition of the function mem$_M(p)$ which computes the memory footprint of $p$ in $\Gamma$. Note that we need two different rules to type function paths: $\text{T-Proc1}$ does a lookup of the procedure as a component of an already typed module; and $\text{T-Proc2}$ 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 C.1. Consider the modules and procedures given in
Figure 17. When typing h, the typing environment contains one
module declaration and one procedure declaration:

\[ \Gamma = (\text{module A.B = _; struct proc f() \rightarrow unit end}; \]
\[ (\text{proc A.C.g() \rightarrow unit = \ldots}) \]

Here, the call to f in h is typed using the T-Procid rule, while the
call to g is typed using T-Proce2.

C  MODULE RESOLUTION

Our module resolution mechanism, given in Figure 21, allows to
evaluate any module expression \( m \) in a typing environment \( \Gamma \)
(mostly, it takes care of functor applications). Essentially, this de-
defines the semantics of our module system, and will be used to give
the semantics of our programming language in Appendix E.

Extended module resolution. Because a module expression \( m \) is
evaluated in a typing environment \( \Gamma \) that can contain abstract mod-
ules (representing open code or functor parameters), the resolved
module \( \text{res}_T(\_m) \) may not be a module expression according to the
syntactic categories defined in Figure 13. We let extended module expressions be the elements of the form:

\[ \bar{m} ::= m | \text{abs}_z x \]

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 \text{param} are reserved to the type system;
and \text{open} modules must be introduced at the logical level (in the
ambient higher-order logic).

Module resolution. The resolution function \( \text{res}_T(\_m) \) evaluates a
module path, in \( \Gamma \), into a (resolved) extended module expression,
which can be a module structure, a functor, or an (potentially ap-
plied) abstract module. Mostly, \( \text{res}_{T}(\_m) \) take care of functor application
through the rules:

\[ \text{res}(p(p')) = \text{res}(\text{m}_0[x \mapsto p']) \quad (\text{if } \text{res}_{T}(\text{p}) = \text{func}(x : M) \text{m}_0) \]
\[ \text{res}(p(p')) = (\text{abs}_{x}(\text{p})(p')) \quad (\text{if } \text{res}_{T}(\text{p}) = (\text{abs}_{x}(\text{m}))(\text{p}')) \]

(the full definition is in Figure 21). In the concrete functor case, we
must substitute the module identifier \( x \) into a path \( p' \) in a module
expression \( m_0 \).

Example C.1. Consider a typing environment \( \Gamma \), and the path
\( x.y(z)(v)(w) \), which must be read as \(((x.y(z))(v))(w)) \). Then, assum-
ing that \( \Gamma(z) = \text{abs}_{open}(z) \), \( \Gamma(v) = m_1 \), \( \Gamma(w) = \text{abs}_{param} w \) and:

\[ \Gamma(x) = \text{struct module y = func(u : \ldots) u end} \]

where \( m_1 \) is some module expression, then \( \text{res}_{T}(x.y(z))(v)(w)) = \)

\[ (\text{abs}_{open} z)(v, w) \].

We define the module procedure resolution function \( f \cdot \text{res}_T(m, f) \).
A resolved module procedure \( f \cdot \text{res}_T(m, f) \) is: i) either a concrete
procedure declaration (\( \text{proc } f : (\bar{v} : \mathcal{T}) \rightarrow \tau_r = \text{body} \); ii) or
the procedure component \( f \) of a resolved (potentially applied) abstract
module (\( \text{abs}_{x}(\text{m})(\text{p}) \)).

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 C.2. Consider a well-typed typing environment \( \vdash \Gamma \),
and a module path \( p \) where:

\[ \Gamma \vdash p : \text{sig} S_1 ; \text{module x : M}_1 \_ \text{restr} \_ \text{end} \]

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

\[ f \cdot \text{res}_T(p) = (\text{struct st}_1 ; \text{module x = m}_1 \_ \text{end}) \]
Module signature and structure sub-typing $\vdash M_1 <: M_2$ and $\vdash S_1 <: S_2$.

We omit the reflexivity and transitivity rules.

<table>
<thead>
<tr>
<th>SubSig</th>
<th>SubFunc</th>
<th>SubStruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash S_1 &lt;: S_2$</td>
<td>$\vdash \theta_1 &lt;: \theta_2$</td>
<td>$\vdash M_0' &lt;: M_0$</td>
</tr>
<tr>
<td>$\vdash \text{sig } S_1 \text{ restr } \theta_1 \text{ end } &lt;: \text{sig } S_2 \text{ restr } \theta_2 \text{ end}$</td>
<td>$\vdash \text{func}(x : M_0) M &lt;: \text{func}(x : M_0') M'$</td>
<td>$\forall i \in {1; \ldots; n}, \vdash D_i &lt;: D_i'$</td>
</tr>
<tr>
<td>$\vdash \text{proc } \theta$</td>
<td>$\vdash M &lt;: M'$</td>
<td>$\vdash D_1; \ldots; D_n &lt;: D_i'; \ldots; D_n'$</td>
</tr>
</tbody>
</table>

Statements and function paths typing $\Gamma \vdash s$ and $\Gamma \vdash f : _\_$.  

<table>
<thead>
<tr>
<th>T-Abort</th>
<th>T-Skip</th>
<th>T-Seq</th>
<th>T-Assign</th>
<th>T-Rand</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \text{abort}$</td>
<td>$\Gamma \vdash \text{skip}$</td>
<td>$\Gamma \vdash s$; $\Gamma \vdash s_2$</td>
<td>$\Gamma \vdash \text{x} : \tau$; $\Gamma \vdash e : \tau$</td>
<td>$\Gamma \vdash \text{x} : \tau$; $\Gamma \vdash d : \tau$</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{call } f(\overline{x})$</td>
<td>$\Gamma \vdash p.f : (\text{proc } f(\overline{x}) : \tau)$</td>
<td>$\Gamma \vdash \text{p; sig } S_1; \text{ proc } f(\overline{x}) : \tau; S_2 \text{ restr } \theta \text{ end}$</td>
<td>$\Gamma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2$</td>
<td>$\Gamma \vdash \text{while } e \text{ do } s$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ExprApp</th>
<th>ExprVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{type}(f) = \tau_1 \times \cdots \times \tau_n : \tau$</td>
<td>$\Gamma \vdash \text{v} : \tau$</td>
</tr>
<tr>
<td>$\forall i \in {1; \ldots; n}, \Gamma \vdash e_i : \tau_i$</td>
<td>$\Gamma \vdash \text{f}(e_1, \ldots, e_n) : \tau$</td>
</tr>
</tbody>
</table>

Restriction entailment $\vdash \theta : 0'$.  

We omit the transitivity and reflexivity rules for $\vdash \theta : 0'$.

<table>
<thead>
<tr>
<th>$\theta : 0'$</th>
<th>$\lambda_m \leq \lambda_m'$</th>
<th>$\lambda_e \leq \lambda_e'$</th>
<th>$\lambda_m \leq \lambda_m'$</th>
<th>$\lambda_m \leq \lambda_m'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall f \in \text{dom}(\theta, 0'), \vdash \theta[f] : 0'[f]$</td>
<td>$\lambda_\theta \leq \lambda_\theta'$</td>
<td>$\lambda_e \leq \lambda_e'$</td>
<td>$\lambda_m \leq \lambda_m'$</td>
<td>$\lambda_e \leq \lambda_e'$</td>
</tr>
<tr>
<td>$\vdash \theta : 0'$</td>
<td>$\lambda_m \leq \lambda_m'$</td>
<td>$\lambda_e \leq \lambda_e'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory restriction.

<table>
<thead>
<tr>
<th>mem$^\tau$(abort)</th>
<th>mem$^\tau$(skip)</th>
<th>mem$^\tau$(if e then s1 else s2)</th>
<th>mem$^\tau$(x ← e)</th>
<th>mem$^\tau$(x ← d)</th>
<th>mem$^\tau$(while e do s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${x} \cup \text{vars}(e)$</td>
<td>${x} \cup \text{mem}_1(s_1) \cup \text{mem}_1(s_2)$</td>
<td>${x}$</td>
<td>$\text{vars}(e) \cup \text{mem}_1(s)$</td>
</tr>
</tbody>
</table>

When f-res$^\tau$(p.f) = (proc $f(\overline{x} : \tau)$ : $\tau_r = \{ \text{var}(\overline{v}_i : \tau); s; \text{return } r \}$):

<table>
<thead>
<tr>
<th>mem$^\tau$(p.f)</th>
<th>mem$^\tau$(x ← call p.f(\overline{e}))</th>
<th>mem$^\tau$(p.f)</th>
<th>mem$^\tau$(p.f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${x} \cup \text{mem}_1(s) \cup \text{vars}(\overline{e})$</td>
<td>${x} \cup \text{mem}_1(p.f) \cup \text{vars}(\overline{e})$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>mem$^\tau$(p.f)</th>
<th>mem$^\tau$(p.f)</th>
<th>mem$^\tau$(p.f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta[f]$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

When f-res$^\tau$(p.f) = (abs$_\text{param}$ x)(\overline{p}_0).f:

<table>
<thead>
<tr>
<th>mem$^\tau$(p.f)</th>
<th>mem$^\tau$(p.f)</th>
<th>mem$\overline{p}_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 19: Additional typing rules and operations.
Substitution in module signatures and declarations

\[
(func(y : M_0)) M_1[x \mapsto \lambda_m] = \begin{cases} 
  (func(y : M_0)) M_1[x \mapsto \lambda_m] & \text{when } y \neq x \\
  \{func(y : M_0) M_1 & \text{otherwise}
\end{cases}
\]

\[\sig S \ restr \theta \ end[x \mapsto \lambda_m] = \sig S(x \mapsto \lambda_m) \ restr \theta[x \mapsto \lambda_m] \end{cases}\]

\[(D_1, \ldots; D_n)[x \mapsto \lambda_m] = D_1[x \mapsto \lambda_m]; \ldots; D_n[x \mapsto \lambda_m] \end{cases}\]

\[\proc f(\tilde{\varphi} : \tau) \rightarrow \tau_s[x \mapsto \lambda_m] = \proc f(\tilde{\varphi} : \tau) \rightarrow \tau_s \end{cases}\]

\[\module y : M[x \mapsto \lambda_m] = \module y : (M[x \mapsto \lambda_m]) \end{cases}\]

Substitution in module restriction

\[(\lambda^m_\alpha \land \lambda_\xi)[x \mapsto \lambda_m] = \lambda^m_\alpha[x \mapsto \lambda_m] \land \lambda_\xi[x \mapsto \lambda_m] \end{cases}\]

\[\tau[x \mapsto \lambda_m] = \tau \end{cases}\]

Substitution in memory restriction

\[\lambda^m_\alpha[x \mapsto \lambda_m] = \lambda^m_\alpha \cup \lambda_m \end{cases}\]

Substitution in complexity restriction

\[\comp[intr : k, y_1, f_1 : k_1, \ldots, y_1, f_1 : k_1][x \mapsto \lambda_m] = \comp[intr : k, (y_1, f_1 : k_1)[x \mapsto \lambda_m], \ldots, (y_1, f_1 : k_1)[x \mapsto \lambda_m]] \end{cases}\]

where \((y, f : k)[x \mapsto \lambda_m] = \begin{cases} 
  e & \text{if } y = x \\
  y, f : k & \text{otherwise}
\end{cases}\]

Substitution in module paths

\[\gamma[x \mapsto p] = \begin{cases} 
  p & \text{if } y = x \\
  \gamma & \text{otherwise}
\end{cases}\]

\[(p' \cdot y)[x \mapsto p] = \begin{cases} 
  (p'[x \mapsto p]), y & \text{if } y \neq x \\
  (p'[x \mapsto p]) & \text{otherwise}
\end{cases}\]

Substitution in module expressions

\[(func(y : M)[x \mapsto p] = func(y : M)[x \mapsto p]) \end{cases}\]

\[(\struct st)[x \mapsto p] = \struct st \end{cases}\]

Substitution in statements

\[\begin{array}{ll}
  \text{abort}[x \mapsto p] = \text{abort} & \text{skip}[x \mapsto p] = \text{skip} \\
  x \leftarrow e[x \mapsto p] = x \leftarrow e & x \leftarrow d[x \mapsto p] = x \leftarrow d \\
  (s_1 ; s_2)[x \mapsto p] = s_1[x \mapsto p] ; s_2[x \mapsto p] & (s_1 \leftarrow m_1)[x \mapsto p] = s_1 \leftarrow m_1[x \mapsto p] \\
  (if \ e \ then \ s_1 \ else \ s_2)[x \mapsto p] = (if \ e \ then \ s_1[x \mapsto p] \ else \ s_2[x \mapsto p]) & (while \ e \ do \ s)[x \mapsto p] = (while \ e \ do \ s)[x \mapsto p]
\end{array}\]

Figure 20: Substitution functions.

and that we have a derivation:

\[
\begin{array}{l}
  \text{struct } st_1; \text{module } x = m; \text{ _end } : \\
  \Gamma \vdash p \quad \text{sig } S_1; \text{module } x : M_1; \text{ restr _end}
\end{array}
\]

Then, we know that \(p \cdot x\) resolves to \(m\), i.e. \(f \cdot \text{res}_F(p) = m\). But we do not have:

\[
\Gamma \vdash p, x : M
\]

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

\[
st_1 = \text{(module } z = m_0) \quad m = z
\]

Therefore, we cannot state a subject reduction property for the module resolution function w.r.t. the typing judgment \(\Gamma \vdash p, m : M\).

Instead, we introduce another typing judgment, noted \(\Gamma \vdash \cdot m : M\), which is similar to the typing judgment \(\Gamma \vdash p, m : M\) of Figure 14, but is used to type a module expression in an environment which has already been typed, while \(\Gamma \vdash p, m : M\) 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 D.1. Using this alternative typing judgment notion, we can state the subject reduction property we want (the proof is postponed to Appendix D).

**Lemma C.1 (Subject reduction).** If \(\Gamma \vdash E \ and \ \Gamma \vdash m : M\) then \(\Gamma \vdash \text{res}_E(m) : M\) whenever \(\text{res}_E(m)\) is well-defined.

**D Subject Reduction Property of Module Resolution**

The goal of this section is to prove that the module resolution mechanism of Figure 21 has the subject reduction property (Lemma C.1), which we recall below:
Module path resolution \( \text{res}_\tau(p) \) to module expression

\[
\text{res}_\tau(p) = \text{res}_\tau(m) \quad \text{(if } \Gamma(p) = \bar{m}: \_ \text{)}
\]

\[
\text{res}_\tau(p.x) = \text{res}_\tau(m) \quad \text{(if } \text{res}_\tau(p) = \text{struct } \theta_1; \text{module } x = m : M, \theta_2 \text{ end)}
\]

\[
\text{res}_\tau(p(p')) = \text{res}_\tau(\text{func}(x : M) m) \quad \text{(if } \text{res}_\tau(p) = \text{func}(x : M) m_0 \text{)}
\]

\[
\text{res}_\tau(p(p')) = (\text{abs}_K x)(\bar{p}_0, p') \quad \text{(if } \text{res}_\tau(p) = (\text{abs}_K x)(\bar{p}))
\]

Module expression resolution \(\text{res}_\tau(m)\)

\[
\text{res}_\tau(\text{struct } \theta_1) = \text{struct } \theta_1
\]

\[
\text{res}_\tau(\text{func}(x : M) m) = \text{func}(x : M) m
\]

\[
\text{res}_\tau((\text{abs}_K x)(\bar{p})) = (\text{abs}_K x)(\bar{p})
\]

Module procedure resolution \( f \cdot \text{res}_\tau(m, f) \)

\[
(f \cdot \text{res}_\tau(p, f) = (\text{proc } f(\bar{\vec{r}} : \bar{\tau}) \rightarrow \bar{\tau} = \text{body}) \quad \text{(if } \Gamma(p, f) = (\text{proc } f(\bar{\vec{r}} : \bar{\tau}) \rightarrow \bar{\tau} = \text{body}))
\]

\[
(f \cdot \text{res}_\tau(m, f) = (\text{proc } f(\bar{\vec{r}} : \bar{\tau}) \rightarrow \bar{\tau} = \text{body}) \quad \text{(if } \text{res}_\tau(m) = \text{struct } \theta_1; \text{proc } f(\bar{\vec{r}} : \bar{\tau}) \rightarrow \bar{\tau} = \text{body}; \theta_2 \text{ end})
\]

\[
(f \cdot \text{res}_\tau(m, f) = (\text{abs}_K x)(\bar{p})). \quad f \quad \text{(if } \text{res}_\tau(m) = (\text{abs}_K x)(\bar{p}))
\]

Figure 21: Resolution functions for paths, module expressions and module procedure.

Lemma (Subject reduction). If \( \Gamma \vdash E \) and \( \Gamma \vdash m : M \) then \( \Gamma \vdash \text{res}_\tau(m) : M \) whenever \( \text{res}_\tau(m) \) is well-defined.

Essentially, this lemma states that evaluating a module expression preserves its type. This is a crucial property that allows to show that memory and complexity restrictions are preserved by the evaluation of a module using the module resolution mechanism. That is, the module expression resulting from the evaluation of another module expression has the same memory footprint, and satisfies the same complexity restrictions.

Proof outline. The proof is essentially the proof that simply typed \( \lambda \)-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 \( \Gamma \vdash m : M \) in Appendix D.1, and prove the link between \( \Gamma \vdash m : M \) and \( \Gamma \vdash p, m : M \); we present an alternative, more computational, way of defining the module resolution procedure in Appendix D.2; finally, we prove that the module resolution procedure has the subject reduction property in Appendix D.3.

D.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 done by first evaluating \( p \) into a functor or an abstract module, and then performing 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 \( m.x \).
We introduce a new notion of lookup in a typing environment, then:

For example, assume that:

if it contains no duplicate declarations, and it contains only well-formed environments. Essentially, a typing environment is not extended with the inferred type, whenever \( \Gamma \) is well-formed. 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 type.

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

Definition D.2. A typing environment \( \Gamma \) is well-formed iff:

- whenever \( \Gamma = (\Gamma_0; \text{Decl}_z; \Gamma_1) \), then \((\Gamma_0; \Gamma_1)(z) \) undefined, where:

\( \text{Decl}_z \) \in \{ module \ z = m : M; \; \text{var} \; z : \tau; \; \text{proc} \; z : \tilde{\tau}; \tilde{\tau} \rightarrow \tau \}

- whenever \( \Gamma = \Gamma_0; \text{module} \; x = m : M; \tilde{\Gamma}_1 \) then \( M \) is well-formed in \( \Gamma_0 \), i.e. \( \tilde{\Gamma}_0 \vdash m : M \).

It is straightforward to check that if \( \Gamma \) is well-typed, then \( \Gamma \) is well-formed.

Proposition D.1. If \( \Gamma \vdash \text{then} \Gamma \) then \( \text{well-formed} \).

Proof. This is by induction over \( \vdash \Gamma \).

We recall that given a module path \( p \), \( \Gamma(p) \) returns the typing declaration corresponding to \( p \) in \( \Gamma \), if it exists. Note that this does not descend in sub-module declarations to retrieve the type of \( p \).

For example, assume that:

\( \Gamma = (\text{module} \; p = \text{struct} \; \text{module} \; y = m : \text{end}) \)

then \( \Gamma(p) = _{\text{end}}(\text{sig module} \; y : M \; \text{restr} \; \text{end}) \), but \( \Gamma(p.y) \) is not defined.

We introduce a new notion of lookup in a typing environment, \( \Gamma[p] \), which descends in sub-module declarations. It also descends below functor definitions, in the case where the functor argument is a module identifier. Continuing the example above, we have \( \Gamma[p.y] = m : M \). Formally:

Definition D.3. For every well-formed typing environment \( \Gamma \) and module path \( p \), we let:

\[ \Gamma[p] = m : M \]

(\( \text{if} \) \( \Gamma(p) = m : M \))

\[ \Gamma[p(x)] = m : M \quad (\text{if} \; \Gamma[p] = \text{func}(x : M_0) : m : \text{func}(x : M_0) \; M) \]

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

\[ \Gamma[p] = (\text{struct} \; \_ : \text{module} \; x = m ; \; \_ \; \text{end}) \]

then:

\[ \Gamma[p.x] = m : M \]

The following proposition allows to replace a typing environment \( \Gamma_0 \) by \( \Gamma_1 \), as long as they coincide w.r.t. \( [p] \) on every \( p \) such that \( \Gamma_0[p] \) is well-defined.

Proposition D.2. For every well-formed environments \( \Gamma_0 \) and \( \Gamma_1 \), if \( \Gamma_0[p] = \Gamma_1[p] \) for every \( p \) such that \( \Gamma_0[p] \) is well-defined, then for any \( m \) and \( M \), if \( \Gamma_0 \vdash m : M \) then \( \Gamma_1 \vdash m : M \).

Proof. This is immediate by induction over the typing derivations \( \Gamma_0 \vdash m : M \).

Example D.2. The following two typing environments \( \Gamma_0 \) and \( \Gamma_1 \) verify the proposition hypothesis:

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

and:

\[ \Gamma_1 = (\text{module} \; p = (\text{struct} \; \text{module} \; x = m ; \; \text{module} \; y = m' \; \text{end}) \; : \; (\text{sig module} \; x : M ; \; \text{module} \; y : M' \; \text{restr} \; \text{end}) \) \]

We can strengthen the context.

Proposition D.3. For any well-formed environment \( \Gamma \), if \( (\Gamma, \Gamma_1) \) is well-formed then \( \Gamma, \Gamma_1 \vdash m : M \) whenever \( \Gamma \vdash m : M \).

Proof. This is straightforward by induction over the typing derivation \( \Gamma \vdash m : M \).

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

Proposition D.4. For any well-formed environment \( \Gamma \), if

\[ \Gamma, \text{module} \; p = m : M ; \tilde{\Gamma}_1 \vdash m_0 : M_0 \]

then for any \( \vdash \Gamma' \vdash M' < : M \), we have:

\[ \Gamma, \text{module} \; p = m : M' ; \tilde{\Gamma}_1 \vdash m_0 : M_0 \]

Proof. We show this by induction over the typing derivation \( \Gamma, \text{module} \; p : M, \tilde{\Gamma}_1 \vdash \Gamma' \vdash p(p_0, \ldots, p_n).p' : M' \), by replacing any application of the D-VAR typing rule on a path starting by \( p \), i.e. of the form \( p(p_0, \ldots, p_n).p' \), by an application of D-VAR on \( p \), followed by D-SUB and by the applications of D-FUNCT and D-COMPNT to replay the path typing derivation of \( \Gamma \vdash p(p_0, \ldots, p_n).p' : M' \) as a module expression typing derivation of \( \Gamma \vdash p(p_0, \ldots, p_n).p' : M' \).
Lemma D.5. For every well-formed typing environment $\Gamma$ such that $\Gamma(p) \nvdash \text{undef}$:

- if $\Gamma \vdash p \Rightarrow m : M$ then there exists $M'$ well-formed in $\Gamma$ such that $\vdash M' \prec M$ and:
  
  $\Gamma; \text{module } p = m : M' \vdash m : M'$

- if $\Gamma \vdash p, \theta ((d_1, \ldots, d_n) : (D_1, \ldots, D_n))$ then there exists $D'_1, \ldots, D'_n$ well-formed such that $\vdash D'_i < D_i$ for every $i$ and:

$\Gamma; \delta_1, \ldots, \delta_n \vdash (d_1, \ldots, d_n) : (D_1, \ldots, D_n)$

where for every $i$:

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

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

$\delta_i = \text{proc } p.f(\bar{\vec{v}} : \bar{\vec{t}}) \to \tau_r = \text{body}$

if $d_i = (\text{proc } f(\bar{\vec{v}} : \bar{\vec{t}}) \to \tau_r = \text{body})$.

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 $\Gamma(p) \nvdash \text{undef}$ in the Alias case, to ensure that the path lookup in $\Gamma$ is still well-formed.

- we show the STRUCT case by applying the induction hypothesis, and using Proposition D.2.

- for MODDECL we have $st = (\text{module } x = m_0; s_t_0)$, and $S = (\text{module } x : M_0; S_0)$, and a derivation of the form:

$\Gamma(p.x) \nvdash m_0 : M_0$

We know that $s_t_0 = (\text{struct } d_1, \ldots, d_n \text{ end})$ and:

$S_0 = (\text{sig } D_1, \ldots, D_n \text{ restr } \_ \text{ end})$

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

$\Gamma, \text{module } p.x = m_0 : M_0 \vdash m_0 : M_0'$

and $\Gamma, \text{module } p.x = m_0 : M_0 \vdash \theta (d_1, \ldots, d_n) : (D'_1, \ldots, D'_n)$

where $\vdash M_0' \prec M_0$, and $\Gamma'' = (D''_1, \ldots, D''_n)$ where for every $i$:

$\vdash D''_i < D_i$

and:

$\Gamma'' = \begin{cases} 
\text{module } p.y = m : M \\
\quad \text{if } D'_i = (\text{module } y : M) \text{ and } d_i = (\text{module } y = m)) \\
\quad \text{proc } p.f(\bar{\vec{v}} : \bar{\vec{t}}) \to \tau_r = \text{body} \\
\quad \text{if } d_i = (\text{proc } f(\bar{\vec{v}} : \bar{\vec{t}}) \to \tau_r = \text{body})) 
\end{cases}$

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

$\Gamma, \text{module } p.x = m_0 : M_0', \Gamma'' \vdash m_0 : M_0'$

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

$\Gamma, \text{module } p.x = m_0 : M_0', \Gamma'' \vdash (d_1, \ldots, d_n) : (D'_1, \ldots, D'_n)$

We conclude by applying the D-MODDECL typing rule.

- PROCDECL is similar to the MODDECL case. We omit the details.

Figure 23: The relation $\Rightarrow r$.  

- for SUB, we have:

$\text{SUB} \quad \Gamma \vdash p : \text{M}_0 \vdash \text{M}_0' \prec \text{M}$

$\Gamma \vdash p : \text{M} \vdash : \text{M}$

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

- for FUNC we have $m = \text{func}(x : \text{M}_0) \text{ m'}$ and $M = \text{func}(x : \text{M}_0) M'$, and the derivation:

$\text{FUNC} \quad \Gamma \vdash m_0 : M_0$

$\Gamma \vdash \text{func}(x : \text{M}_0) \text{ m'} : \text{func}(x : \text{M}_0) M'$

By induction hypothesis, we have a derivation of $\Gamma, \text{module } p : \text{M}_0 \vdash m' : \text{M}'$ and:

$\Gamma, \text{module } x = \text{absparam} : \text{M}_0 \vdash \text{M}_0 < \text{M}_0'$. Using the sub-typing transitivity rule, we have that $\vdash \text{M}_0' < \text{M}$. This concludes this case.

We extend sub-typing judgments to typing environments, by re-ordering the declarations in the context, to obtain a derivation of:

$\Gamma, \text{module } p(x) = m' : M', \text{module } x = \text{absparam} : M_0 \vdash m' : M''$

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

$\Gamma, \text{module } p(x) = m' : M' \vdash \text{func}(x : M_0) M''$ by Proposition D.2, there is a derivation of:

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

We extend sub-typing judgments to typing environments, by re-ordering that $\vdash I_0 < I_1$ whenever $I_0$ and $I_1$ are of the same length, and any declaration in $I_0$ is a sub-type of the corresponding declaration in $I_1$.

Lemma D.6. If $\vdash I_0 < I_1$ then there exists $I' \prec I_1$ such that $\vdash I' \prec I_1$.

Proof. We use Lemma D.5 to replace the ENVMOD rules by D-ENVMOD rules. We omit the details.  

D.2 Module Resolution as a Rewrite Relation

We define in Figure 23 the relation $\Rightarrow r$ on partially resolved module expressions. This relation is exactly the evaluation strategy used by the module resolution procedure defined in Figure 21.

We let $\bar{m} : \Gamma$ be the normal form of $m$ w.r.t. the reflexive and transitive closure of $\Rightarrow r$, if it exists.

We now state a property of $\Rightarrow r$ in the case where $\Gamma$ is an environment, i.e. top-level module declarations are not module aliases.
we state the following substitution lemmas.

PROPOSITION D.7. The relation \( \rightarrow_{\Gamma} \) is deterministic on well-formed module expression: for every well-formed \( \bar{m} \), if \( \bar{m} \rightarrow_{\Gamma} \bar{m}_0 \) and \( \bar{m} \rightarrow_{\Gamma} \bar{m}_1 \) then \( \bar{m}_0 = \bar{m}_1 \).

Moreover, for any well-formed \( \mathcal{E} \) and \( \bar{m} \), if \( \text{res}^\mathcal{E}(\bar{m}) \) is well-defined then \( \text{res}^\mathcal{E}(\bar{m}) = \bar{m} \downarrow^{\mathcal{E}} \).

PROOF. The fact that \( \rightarrow_{\Gamma} \) is deterministic is straightforward from the definition of \( \rightarrow_{\Gamma} \) in Figure 23; indeed, we only need to observe that well-formed module expressions cannot have two module declarations with the same name.

Moreover, for any \( \bar{m}_1 \) and \( \bar{m}_2 \), if \( \bar{m}_1 \rightarrow_{\Gamma} \bar{m}_2 \) and \( \bar{m}_1 \) is well-formed then so is \( \bar{m}_2 \). Hence if \( \rightarrow_{\Gamma} \) terminates on a well-formed \( \bar{m} \) then \( \bar{m} \downarrow^{\Gamma} \) exists.

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

D.3 Subject Reduction

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

LEMMA D.8 (Substitution 1). We have the following substitution properties:

- if \( \theta_1 \downarrow \theta_2 \) then \( \theta_1[x \mapsto \lambda_m] \downarrow \theta_2[x \mapsto \lambda_m] \).
- if \( \lambda_1 \downarrow \lambda_2 \) then \( \lambda_1[x \mapsto \lambda_m] \downarrow \lambda_2[x \mapsto \lambda_m] \).
- if \( M_0 \downarrow M_0' \) then \( M_0[x \mapsto \lambda_m] \downarrow M_0'[x \mapsto \lambda_m] \).

PROOF. The properties above are shown by induction over their respective typing derivation. \( \square \)

LEMMA D.9 (Substitution 2). For every:

\[ \Gamma_1; \text{module } x = \text{absparam } : M; \Gamma_2 \]

which is a well-formed typing environment, if:

\[ \Gamma_1; \text{module } x = \text{absparam } : M; \Gamma_2 \vdash e : \tau \]

then:

\[ \Gamma_1; \Gamma_2[x \mapsto \lambda_m] \vdash e : \tau \]

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

\( \square \)

PROPOSITION D.10. For any well-formed typing environment \( \Gamma \), if \( \Gamma \vdash p : M \) and \( \text{mem}_\Gamma(p) = \lambda_m \) then:

\[ \text{mem}_\Gamma(F(x \mapsto p)) \subseteq (\text{mem}_\Gamma, \text{module } x = \text{absparam } : M(F))(x \mapsto \lambda_m) \]

LEMMA D.11 (Substitution 3). For every well-formed environment \( \Gamma \) of the form:

\[ \Gamma_1; \text{module } x = \text{absparam } : M; \Gamma_2 \]

for every module path \( p \) such that \( \Gamma_1 \vdash p : M \), if \( \text{mem}_\Gamma(p) = \lambda_m \) then:

- if \( \Gamma \vdash m : M_0 \) then:
  \[ \Gamma_1; \Gamma_2[x \mapsto \lambda_m] \vdash m[x \mapsto p] : M_0[x \mapsto \lambda_m] \]

- if \( \Gamma \vdash \{ \text{var}(\bar{c}_i : \bar{f}_i); s; \text{return } e \} \eta \) then:
  \[ \Gamma_1; \Gamma_2[x \mapsto \lambda_m] \vdash \{ \text{var}(\bar{c}_i : \bar{f}_i); s[x \mapsto p]; \text{return } e \} \eta[x \mapsto \lambda_m] \]

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).

\( \square \)

LEMMA D.12. If \( \Gamma \vdash \mathcal{E} \) and \( \Gamma \vdash m : M \) then if \( m \rightarrow_{\Gamma} m' \) then \( \Gamma \vdash m' : M \).

PROOF. W.l.o.g., we assume that the typing derivations \( \Gamma \vdash \mathcal{E} \) and \( \Gamma \vdash m : M \) 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 \( m \rightarrow_{\Gamma} m' \):

- if \( x \rightarrow_{\Gamma} \mathcal{E}(x) \). We check that we must have a derivation of the following form:
  \[
  \begin{array}{c}
  \Gamma \vdash m : \text{sig } S_1; \text{module } x : M_0; S_2 \text{ restr } \theta \end{array} \]

Using the fact that \( \Gamma \vdash \mathcal{E} \), we know that \( \mathcal{E} = (\mathcal{E}_0; \text{module } x = m : M_0; \mathcal{E}_1) \) and \( \Gamma \vdash m : M_0 \). We conclude by applying D-Sub.

- if \( m.x \rightarrow_{\Gamma} m' \) where \( m = \text{struct } S_1; \text{module } x = m' ; S_2 \text{ end} \). We check that we must have a derivation of the following form:

\[
\begin{array}{c}
\Gamma \vdash m : \text{sig } S_1; \text{module } x : M_0; S_2 \text{ restr } \theta \end{array} \]

Hence we have a derivation of \( \Gamma \vdash m' : M' \). We conclude by sub-typing, using the fact that we can derive \( M' \downarrow M \).

- if \( m.x \rightarrow_{\Gamma} m', x \) where \( m \rightarrow_{\Gamma} m' \), we know that our derivation is of the form:

\[
\begin{array}{c}
\Gamma \vdash m : \text{sig } S_1; \text{module } x : M_0; S_2 \text{ restr } \theta \end{array} \]

By induction hypothesis, we have a derivation of:

\[
\Gamma \vdash m' : \text{sig } S_1; \text{module } x : M_0; S_2 \text{ restr } \theta \end{array} \]

We conclude immediately using D-Compnt and D-Sub.

\- the case where \( m(p) \rightarrow_{\Gamma} m'(p) \) with \( m \rightarrow_{\Gamma} m' \) is the same.
\- if \( m(p) \rightarrow_{\Gamma} m_0[x \mapsto p] \) where \( m = \text{func}(x : M_0) m_0 \), we have a derivation of the form:

\[
\begin{array}{c}
\Gamma \vdash \text{func}(x : M_0) \text{ M' }; \Gamma \vdash p : M_0' \end{array} \]

\[
\begin{array}{c}
\Gamma \vdash m(p) : M' \downarrow \lambda_m \end{array} \]

\[
\begin{array}{c}
\Gamma \vdash m(p) \downarrow \lambda_m \end{array} \]

\[
\begin{array}{c}
\Gamma \vdash m(p) : M \end{array} \]
where $\lambda_m = \text{mem}_F(p')$. Since $m = \text{func}(x : M_p) \cdot m_0$, we must have a derivation of the form:

\[
\begin{align*}
\Gamma &\vdash M_p \\
\Gamma(x) &\downarrow\text{undef} \\
\text{D-Func} &\quad \text{module } x = \text{abs}_{\text{param}} : M_p \cdot m_0 : M'' \\
\Gamma &\vdash \text{func}(x : M_p) \cdot m_0 : \text{func}(x : M_p) \cdot M'' \\
&\quad \vdash M'_p \cdot M'' \cdot M' \\
\text{D-Sub} &\quad \text{func}(x : M_p) \cdot m_0 : \text{func}(x : M'_p) \cdot M'
\end{align*}
\]

Since $\Gamma \vdash p : M'_p$ and $\vdash M'' : M_p$, we know that we have a derivation of $\Gamma \vdash p : M_p$. Since:

\[
\Gamma, \text{module } x = \text{abs}_{\text{param}} : M_p \cdot m_0 : M''
\]

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

\[
\Gamma \vdash m_0[x \mapsto p] : M''[x \mapsto \lambda_m]
\]

Using Lemma D.8, we know that since $\vdash M'' : M'_p$, we have a derivation of:

\[
\Gamma \vdash m_0[x \mapsto p] : M'_p[x \mapsto \lambda_m]
\]

- the case where $n(p) \mapsto^1 \text{abs}_N x(\rho_0, p)$ and $n = \text{abs}_N x(\rho_0)$ is immediate.

We recall and prove Lemma C.1:

**Lemma (Subject Reduction).** If $\Gamma \vdash E$ and $\Gamma \vdash m : M$ then $\Gamma \vdash \text{res}_{E}(m) : M$ whenever $\text{res}_{E}(m)$ is well-defined.

**Proof.** From Proposition D.7, we know that $\text{res}_{E}(m) = m \downarrow E$. From Lemma D.12, we know that the type of the module expression is preserved. This concludes this proof.

**E INSTRUMENTED SEMANTICS**

We now define the denotational semantics of our programming language and cost judgments. We quickly introduce the main aspects of our semantics below, before defining it formally in the rest of the section. We use this semantics to state and prove our main soundness theorem in Appendix F.

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

\[
\text{cost}_{E, \rho}(s) = \inf \left\{ c' \mid \Pr((\_, c) \leftarrow [s]^E_{\nu, \rho} : c \leq c') = 1 \right\}
\]

**Judgments semantics.** Basically, the judgment $E \vdash \{\phi\} s (\psi \mid t)$ states that: i) the memory $\nu$ obtained after executing $s$ in an initial memory $\nu \in \phi$ must satisfy $\psi$; 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 $A.f$ of the number of calls to $A.f$ times the intrinsic complexity of $A.f$. Formally:

\[
\text{cost}_{E, \rho}(s) \leq \text{conc} + \sum_{f \in \text{procs}(E(A))} t[A.f] \cdot \text{compl}_{A.f}
\]

where $\text{compl}_{A.f}$ is the intrinsic complexity of the procedure $A.f$, i.e. its complexity excluding calls to $A.$’s functor parameters.

**Outline of this Section.** We present the semantics of our programs in Appendix E.1. Then, we define the semantics of our cost judgments. This requires two additional complexity measures: 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). These additional complexity measures are defined in Appendix E.2. Finally, we give the semantics of our cost judgment in Appendix E.3.

**E.1 Semantics**

For any set $\mathcal{A}$, we denote by $D(\mathcal{A})$ the set of discrete sub-distributions over $A$—i.e. the set of function $\mu : A \rightarrow [0, 1]$ with discrete support s.t. $\mu$ is summable and $|\mu| = \sum_{x} \mu(x) \leq 1$. For $x \in A$, the Dirac distribution at $x$ is written $\delta_{\nu}^0$ or $\delta_{\nu}$ when $\nu$ is clear from the context. If $\mu \in D(\mathcal{A})$ and $\nu \in D(\mathcal{B})$, the expected distribution of $\mu' \in D(\mathcal{B})$ when ranging over $\mu$, written $E_\nu[\mu'[\nu]]$ or $E_\mu[\nu']$, is defined as $E_\mu[\nu'] = b \in B \mapsto \sum_{a \in A} \mu(a) \cdot \nu'(a)(b)$. For $\mu' \in D(\mathcal{A})$ and $f : A \rightarrow B$, the marginal of $\mu'$ w.r.t. $f$, written $f^*(\mu') \in D(\mathcal{B})$, is defined as $f^*(\mu') = b \mapsto \sum_{a \in A} f(a)(b)$. We write $\pi_1^\nu$ (resp. $\pi_2^\nu$) for the first and second marginal — i.e. when $\nu$ is resp. the first and second projection. For any base type $\tau \in \mathbb{B}$, we assume an interpretation domain $\nu_\tau$. We let $\mathcal{V}$ be the set of all possible values $\tau_r \in \mathcal{V}_\tau$. A memory $\nu \in \mathcal{M}$ is a function from $\mathcal{V}$ to $\mathcal{V}$. We write $\nu(x)$ for $\nu(x)$. For $\nu \in \mathcal{M}$ and $v \in \mathcal{V}$, we write $\nu(x) \mapsto v$ for the memory that maps $x$ to $v$ and $y$ to $v(y)$ for $y \neq x$.

**Expressions semantics.** For any operator $f \in \mathcal{F}_E$ with type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, we assume its semantics $[f] : \mathcal{V}_{\tau_1} \times \cdots \times \mathcal{V}_{\tau_n} \rightarrow \mathcal{V}_\tau$ and the cost of its evaluation $c_E(f, \cdot) : \mathcal{V}_{\tau_1} \times \cdots \times \mathcal{V}_{\tau_n} \mapsto \mathbb{N}$. The semantics $[e]_{\nu} : \mathcal{M} \mapsto \mathcal{V}$ of a well-typed expression $e$ in a memory $\nu$ is defined inductively by:

\[
\llbracket e \rrbracket_{\nu} = \begin{cases} \nu(x) & \text{if } e = x \in \mathcal{V} \\ [f](\llbracket \epsilon_1 \rrbracket_{\nu}, \ldots, \llbracket \epsilon_n \rrbracket_{\nu}) & \text{if } e = f(\epsilon_1, \ldots, \epsilon_t) \end{cases}
\]

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

\[
\begin{cases} 1 & \text{if } e = x \in \mathcal{V} \\ c_f + \sum_{1 \leq i \leq n} c_E(e_i, \nu) & \text{if } e = f(\epsilon_1, \ldots, \epsilon_n) \\ c_f \leq c_f(e, \lfloor \epsilon_1 \rrbracket_{\nu}, \ldots, \lfloor \epsilon_n \rrbracket_{\nu}) & \text{and } \nu \in \mathcal{M} \end{cases}
\]

For technical reasons, we assume that there exists one operator with a non-zero cost.\footnote{Some of our lemmas do not hold if all programs have a cost of zero.}

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

**Environment and $E$-pre-interpretation.** To give the semantics of a program in an environment $E$, we need an interpretation of $E$’s abstract modules. A $E$-pre-interpretation is a function $\rho$ from $E$’s abstract modules to module expressions, with the correct types,
well-typed environment, and where every complexity restriction f-res procedure module expression. Note that this subsumes the syntax of state-x under the memory such that $E \vdash f$ for the distribution $x$ use any module or abstract module declared before it by except for complexity restrictions. We will specify what it means for $f$-res (M $\times$ $x$)

**Definition E.1.** Let erase_{comp}(M) be the module signature M where every complexity restriction $\lambda$ has been erased, by replacing it by $\top$. Then $\rho$ is a $\mathcal{E}$-pre-interpretation if and only if for every $x$ such that $\mathcal{E} \in \mathcal{E}_1$; module $x = \text{abs}_\text{open} : M_1; \mathcal{E}_2$, we have $\mathcal{E}_1 \vdash_{\mathcal{E}} \rho(x) : \text{erase}_{\text{comp}}(M_1)$.

Note that we type $\rho(x)$ 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 $\mu \in \mathcal{D}(M \times N)$ and $n \in N$, 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 $\mathcal{E} \vdash s$. The $\mathcal{E}$-denotational semantics of an instruction $s$ under the memory $\nu$ and $\mathcal{E}$-pre-interpretation $\rho$, written $\llbracket s \rrbracket_{\mathcal{E}, \rho} \in \mathcal{D}(M \times N)$, is defined in Figure 24.

We give the semantics for an extended syntax, which allows procedure calls to be of the form $x \leftarrow \text{call } m.f(\bar{x})$ 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 (abs_open $x)(\bar{\rho}).f$ as the semantics of a call to $\rho(x)(\bar{\rho}).f$.

**Definition E.2 Instrumented Semantics**

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

**Function call counting**

The function call counting semantics $\gamma{\cdot}(\llbracket s \rrbracket_{\mathcal{E}, \rho})$, given in Figure 25, evaluates the instruction $s$ under the memory $\nu$ and $\mathcal{E}$-pre-interpretation $\rho$, counting the number of calls to the abstract procedure $\gamma_{\cdot}$.

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

$$\# \text{calls}_{\gamma_{\cdot}, \mathcal{E}, \rho}(v) = \sup(\text{supp}(\pi_{\mathcal{E}}(\gamma_{\cdot}(\llbracket s \rrbracket_{\mathcal{E}, \rho})))$$ (in memory $v$)

$$\# \text{calls}_{\gamma_{\cdot}, \mathcal{E}, \rho}(v) = \max(\# \text{calls}_{\gamma_{\cdot}, \mathcal{E}, \rho}(v))$$ (in any memory)
And the intrinsic cost of every instruction \( s \) with intrinsic cost under memory \( \nu \) parameters \( \bar{x} \), written \( \llbracket s \rrbracket^{E,\rho}_\nu \bar{x} \) is the cost of the execution of \( s \) under \( \nu \) in \( \rho \), without counting the costs of function calls to the parameters \( \bar{x} \). Formally, \( \llbracket s \rrbracket^{E,\rho}_\nu \bar{x} \) is defined exactly like \( \llbracket s \rrbracket^{E,\rho}_\nu \bar{x} \) in Figure 24, except for the abstract procedure call case, which is replaced by:

\[
\llbracket x \leftarrow \text{call } m(f(x)) \rrbracket^{E,\rho}_\nu \bar{x} = \begin{cases} E(\nu', \bar{e}') \llbracket x \leftarrow \text{call } \rho(x) \bar{y}, f(x) \rrbracket^{E,\rho}_\nu \bar{y} & \text{if } z \in \bar{x} \\ \llbracket x \leftarrow \text{call } \rho(x) \bar{y}, f(x) \rrbracket^{E,\rho}_\nu \bar{y} & \text{if } z \notin \bar{x} \end{cases}
\]

where \( f \text{-res}_{E}(m, f) = (\text{abs}_{open} x(p), f) \).

Remark that both semantics coincide on their first component. Indeed, for any \( E, \rho \)-pre-interpretation \( \rho \):

\[
\forall v \ s. \pi^{E,\rho}_v(\llbracket s \rrbracket^{E,\rho}_v \bar{x}) = \pi^{E,\rho}_v(\llbracket s \rrbracket^{E,\rho}_v \bar{x})
\]

The \( (E, \rho) \)-intrinsic cost \( i \text{-cost}_{E,\rho}^{E,\rho}_s \) of an instruction \( s \) is:

\[
i \text{-cost}_{E,\rho}^{E,\rho}_s(\bar{x}) = \text{sup}(\text{sup}(\pi^{E,\rho}_v(\llbracket s \rrbracket^{E,\rho}_v \bar{x})))
\]

The intrinsic cost of a procedure \( m, f \) with parameters \( \bar{x} \) is:

- If \( f \text{-res}_{E}(m, f) = (\text{proc } f(\bar{v} : \bar{e}) \rightarrow \tau_r = \{<; \_ ; \_ \} \)) then:
  \[
i \text{-cost}_{E,\rho}^{E,\rho}_s(\bar{x}) = i \text{-cost}_{E,\rho}^{E,\rho}_s(\bar{x}) + c_f(r, v)
\]

- If \( f \text{-res}_{E}(m, f) = (\text{abs}_{open} x(p), f) \) then:
  \[
i \text{-cost}_{E,\rho}^{E,\rho}_s(\bar{x}) = i \text{-cost}_{E,\rho}^{E,\rho}_s(\rho(x)p, f)
\]

The intrinsic cost in any memory of an instruction \( s \) or a module procedure \( m, f \) is:

\[
i \text{-cost}_{E,\rho}^{E,\rho}_s(\bar{x}) = \max_{v \in M} i \text{-cost}_{E,\rho}^{E,\rho}_s(\bar{x})
\]

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

\[
\text{Definition E.2. Let } E \text{ be an well-typed environment. A } E \text{-pre-interpretation } \rho \text{ is an } E \text{-interpretation if for every module identifier } x \text{ such that } E = E_1; \text{module } \bar{x} = \text{abs}_{open} : M_1; E_2 \text{ where:}
\]

\[
M_1 = \text{func}(\bar{z} : \bar{M}) \text{ sig } S_1 \text{ restr } \theta \text{ end}
\]

and for every procedure \( f \in \text{procs}(S) \), for every valuation \( \bar{m} \) of the function’s parameters such that, for every \( 1 \leq i \leq |\bar{z}| \), if we let

\[
z_i = \bar{z}[i], m_i = \bar{m}[i] \text{ and } m_i = \bar{M}[i] = \text{sig restr } \lambda_i \text{ \& end, if:}
\]

\[
E \vdash (\text{module } z_i = m_i : \text{erase}_\text{comp}(M_i))
\]

and

\[
\forall g \in \text{procs}(M_i), \text{cost}_{\nu}^{E}(m_i, g) \leq \lambda_i
\]

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

\[
\theta[f] = _\_ \& \lambda_c = \text{comp}[\text{intr : } k, y_1.f_1 : k_1, \ldots , y_l.f_l : k_l]
\]

Then for every \( 1 \leq j \leq l \),

\[
\text{#calls}^{E',\rho'}_{y_j.f_j}(x(\bar{z}), f) \leq k_j \text{ and } i \text{-cost}_{E',\rho'}^{E',\rho'}(x(\bar{z}), f) \leq k
\]

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

\[
\text{cost}_{E,\rho}^{E,\rho}(x.f) = \text{sup } i \text{-cost}_{E,\rho}^{E,\rho}(x(\bar{z}), f)
\]

E.3 Soundness of our Proof System

We now have all the tools to define the semantics of our expression and program cost judgments.

\[
\text{Definition E.3. the judgment } E \vdash \{ \phi \} e \leq t_e \text{ stands for:}
\]

\[
\forall v \in \phi, E(\nu, v) \leq t_e
\]

\[
\text{Definition E.4. The judgment } E \vdash \{ \phi \} \nu \{ \psi \mid t \} \text{ means that for any } E\text{-interpretation } \rho \text{ and } v \in \phi:
\]

\[
\text{sup}(\pi^{E,\rho}_v(\llbracket s \rrbracket^{E,\rho}_v \bar{x})) \subseteq \psi \land \text{cost}_{E,\rho}^{E,\rho}(x) \leq t[\text{conc}] + \sum_{f \in \text{procs}(E(\bar{A}))} t[\bar{A}.f] \cdot \text{comp}_{E,\rho}^{E,\rho}(x.f)
\]

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 \( A.f \) of the number of calls to \( A.f \) times the intrinsic complexity of \( A.f \).

F HOARE LOGIC FOR COST

We present the full set of rules of our Hoare logic for cost and state their soundness.

\[\text{Hoare logic rules.} \text{ Our Hoare logic for cost comprises the basic rules in Figure 26, the abstract call rule in Figure 6, and the instantiation rule in Figure 27.}\]

The basic rules in Figure 26 are essentially Hoare logic rules with some additional components to handle the cost aspects of the logic. Some examples of basic rules: the \( \text{if} \) rule handles the conditional program construct, and has already been presented in Section 4.2;
Weakening rule. The Instantiation rule, given in Figure 27, allows to instantiate an abstract module \( x \) by a concrete module \( m \). Assume that we can upper-bound the cost of a statement \( s \) by \( t_s \), when \( x \) is abstract:

\[ E \vdash \{ s \} x : \{ \psi \mid t \}, t' \leq t \]

Then we can instantiate \( x \) by a concrete module \( m \) as long as \( m \) complies with the module signature \( M \), which is checked through two conditions.

First, we check that \( m \) has the correct module type, except for complexity restrictions, through the premise \( E \vdash _{m} m \text{ erase}_\text{compl}(M) \).

Then, we check that \( m \) satisfies the complexity restriction \( \theta \) in \( M \), by requiring that for any procedure \( f \) of \( x \):

\[ E \vdash \{ s \} x : \{ \psi \mid t \}, t' \leq t \]

where \( t' \) must respect \( \theta[f] \), which is guaranteed by \( t_f \leq \text{compl}_\text{in} t(f) \), which does two checks:

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

\[ t_f[\text{con}] + \sum_{h \in \text{procs}(A)} t_f[H] \cdot \text{intr}_\text{in}(A.h) \leq \theta[f]|\text{intr} \]

where \( \text{intr}_\text{in}(A.h) \) is the upper-bound on \( A.h \) intrinsic cost declared in \( E \) (if \( A.h \) declares no intrinsic bound in \( E \), then \( \text{intr}_\text{in}(A.h) \) is undefined (hence \( A.h \) execution time can be arbitrarily large), and the Instantiation rule cannot be applied). In other words, the concrete execution time \( t_f[\text{con}] \) of \( x.f \), plus the abstract execution time of \( x.f \) (excluding function parameters, already accounted for), must be bounded by \( \theta[f]|\text{intr} \).

The final cost \( T_m \) (in Figure 27) 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 \( x.f \) (which is \( t_s[x.f] \)), times the cost of \( x.f \) (which is \( t_f \)).

Soundness. We now prove the soundness of our Hoare logic rules. We recall Theorem 4.1.

THEOREM. The proof rules in Figures 6, 26 and 27 are sound.

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

First, some notation. We extend the program semantics \( L_{\text{comp}}^P \) to function paths, by letting \( [F]^E_{\theta} \) be the execution of \( F \)'s body without the procedure's arguments evaluation phase:

\[ [F]^E_{\theta} = E_{\theta} \cdot \text{pr} \cdot \text{pr} \cdot \text{pr} \cdot \text{pr} \]

The cost of \( F \) is defined as expected, by taking the maximum over all memories of the support of the second marginal of \( [F]^E_{\theta} \):

\[ \operatorname{cost}_{\theta}^E_{\text{pr}}(F) = \sup \{ \text{supp}(x^2_{[F]^E_{\theta}}(F)) \} \]
Finally, cost judgment $E \vdash \{\phi\} F \{\psi \mid t\}$ on function path have the same semantics than cost judgment on statements, i.e. for every well-typed environment $E$, for every $E$-interpretation $\rho$ and $\nu \in \phi$:

$$ \text{supp}(\pi^1_{\nu}([F]^{\text{E}\rho}{\nu})) \subseteq \psi \land \text{cost}^E_{\nu}(F) \leq t[\text{conc}] + \sum_{A \in (E)} \sum_{f \in \text{proc}(E(A))} t[A.f] \cdot \text{compl}^A_{A.f} \quad (4) $$

### F.1 Abstract Call Rule Soundness

We now prove that the abstract call rule $\text{Abs}$ in Figure 6 is sound.

First, remark that since the rule $\text{Abs}$ uses a different upper-bound $t_{H.g}(k)$ on the cost of the $k$-th call to the oracle $H.g$, 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 of $E$ of the procedure $\rho(x).f$ called, where procedure calls are inlined. Formally, for every environment $E$ and $E$-pre-interpretation $\rho$, we define:

- $\text{size}^E_{\rho}(\text{abort}) = 1$
- $\text{size}^E_{\rho}(\text{skip}) = 1$
- $\text{size}^E_{\rho}(\text{call } F(f)) = 1 + \text{size}^E_{\rho}(\text{call } \rho(x)(\bar{\rho}).f(\bar{f}))$
- $\text{size}^E_{\rho}(\text{if } e \text{ then } s_1 \text{ else } s_2) = 1 + \text{size}^E_{\rho}(s_1) + \text{size}^E_{\rho}(s_2)$
- $\text{size}^E_{\rho}(\text{while } e \text{ do } s) = 1 + \text{size}^E_{\rho}(s)$

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 soundness of $\text{Abs}$ after.

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

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

$$ \Gamma \vdash s \quad \Gamma \vdash \lambda^s_m \quad \text{where} \quad \Gamma = E, \text{module} \bar{x} = \text{abs}_{\text{param}} \bar{x} : \bar{M} $$

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

$$ \{ y.h \mid y \in \bar{x} \land h \in \text{proc}(\bar{M}[y]) \} $$

Then if:

- the memory of the statement $s$ is independent of $I$, except for calls to the parameters $\bar{x}$, i.e.:
  $$ \lambda^s_m \cap \text{FV}(I) = \emptyset $$

- for every parameters’ procedure $H.g \in \text{proc}_{E} (\bar{x})$, the $k$-th call to $H.g$ (when $H$ is instantiated by $\bar{\rho}[H]$) preserves the invariant and has a cost upper-bounded by $t_{H.g}(k)$ for $k \leq \lambda_c[H.g]$:  
  $$ \forall k \leq \lambda_c[\bar{\rho}], \bar{k}[H.g] < \lambda_c[H.g] \rightarrow $$
  $$ E \vdash \begin{cases} \text{if } \bar{k} \vdash H.f \{ I \bar{k} + 1 \text{H.g} \mid t_{1H.g}(\bar{k}[H.g]) \} \end{cases} $$

- the number of calls from $s$ to the parameters’ procedure is upper-bounded by $\lambda_c[\bar{\rho}] - \bar{k}$ (for memory satisfying the invariant $I \bar{k}$):

$$ \forall k \leq \lambda_c[\bar{\rho}], \forall \nu \in \bar{I} \rightarrow \forall H.g \in \text{proc}_{E}(\bar{x}) $$

$$ \text{#calls}_{\text{H.g},\nu}(s) \leq \lambda_c[H.g] - \bar{k} $$

For every $H.g \in \text{proc}_{E}(\bar{x})$ and $j < \lambda_c[H.g]$, we define:

$$ \text{call-cost}^E_{H.g}(j) = t_{H.g}(j)[\text{conc}] + \sum_{A \in (E)} \sum_{f \in \text{proc}(E(A))} t_{H.g}(j[A.f]) \cdot \text{compl}^A_{A.f} $$

Let $\rho' = (x : \bar{x})$. We have that for every $k \leq \lambda_c[\bar{\rho}]$ and $\nu \in \bar{I}$:

$$ \text{supp}(\pi^1_{\nu}([s_1]^E \rho'_{\nu})) \subseteq I(\bar{k} + \text{#calls}_{\text{H.g},\nu}(s)) $$

and:

$$ \text{cost}^E_{\nu}(s) \leq \text{i-cost}^E_{\nu}(s) + \sum_{\nu \in \text{proc}_{E}(\bar{x})} \sum_{j=\bar{k}[H.g]}^{\text{H.g}+\text{#calls}_{\text{H.g},\nu}(s)-1} \text{call-cost}^E_{\text{H.g},\nu}(j) $$

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

**Proof.** We prove this induction over $\text{size}^E_{\rho}(s)$. We do a case analysis on $s$:

- if $s$ is $\text{abort}$ or $\text{skip}$, this is immediate.
- if $s = s_{1:2}$. Let $\bar{k} \leq \lambda_c[\bar{\rho}]$ and $\nu \in \bar{I}$. We know that:

$$ \text{supp}(\pi^1_{\nu}([s_1]^E \rho'_{\nu})) = \bigcup_{\nu' \in \text{supp}(\pi^1_{\nu}([s_2]^E \rho'_{\nu}))} \text{supp}(\pi^1_{\nu'}([s_2]^E \rho'_{\nu})) $$

Let $\bar{k}_1 = \bar{k} + \text{#calls}_{\text{H.g},\nu}(s_1)$. By induction hypothesis applied on $s_1$:

$$ \text{supp}(\pi^1_{\nu'}([s_1]^E \rho'_{\nu})) \subseteq I \bar{k}_1 $$

Let $\nu' \in \text{supp}(\pi^1_{\nu'}([s_2]^E \rho'_{\nu}))$, we know that $\bar{k}_1 \leq \lambda_c[\bar{\rho}]$. Hence, by induction hypothesis on $s_2$, we deduce that:

$$ \text{supp}(\pi^1_{\nu'}([s_2]^E \rho'_{\nu})) \subseteq I \bar{k}_2 $$
where \( \hat{k}(v') = \hat{k} + \text{#calls}_{\nu}^{E', \rho'}(s_1) + \text{#calls}_{\nu}^{E', \rho'}(s_2) \). Since:
\[
\text{#calls}_{\nu}^{E', \rho'}(s_1) + \text{#calls}_{\nu}^{E', \rho'}(s_2) \leq \text{#calls}_{\nu}^{E', \rho'}(s_1; s_2)
\] (10)
we deduce:
\[
\text{supp}(\pi_{\nu}^+([s_2 z_{E', \rho'}])) \subseteq I(\hat{k}) + \text{#calls}_{\nu}^{E', \rho'}(s_1; s_2)
\]
which concludes the proofs of the first point. It remains to prove that the complexity of \( s_1; s_2 \) is upper-bounded by the wanted quantity. First, we have:
\[
\text{cost}_{\nu}^{E', \rho'}(s_1; s_2) = \text{cost}_{\nu}^{E', \rho'}(s_1) + \max_{v' \in \text{supp}(\pi_{\nu}^+([s_1 z_{E', \rho'}]))} \text{cost}_{\nu}^{E', \rho'}(s_2)
\]
By applying the induction hypothesis on \( s_1 \) and \( s_2 \) with, respectively, \( \hat{k} \) and \( \hat{k}_1 \), we get:
\[
\text{cost}_{\nu}^{E', \rho'}(s_1) \leq \text{i-cost}_{\nu}^{E', \rho', \hat{k}}(s_1) + \sum_{H, g \notin \text{proc}_{\nu}(\hat{k})} \sum_{j = \hat{k}[H, g]}^{\hat{k}[H, g]} \text{call-cost}_{H, g}(j)
\]
and:
\[
\text{cost}_{\nu}^{E', \rho'}(s_2) \leq \text{i-cost}_{\nu}^{E', \rho', \hat{k}}(s_2) + \sum_{H, g \notin \text{proc}_{\nu}(\hat{k})} \sum_{j = \hat{k}[H, g]}^{\hat{k}[H, g]} \text{call-cost}_{H, g}(j)
\]
Since \( \text{i-cost}_{\nu}^{E', \rho'}(s_1; s_2) \) is equal to:
\[
\text{i-cost}_{\nu}^{E', \rho'}(s_1) + \max_{v' \in \text{supp}(\pi_{\nu}^+([s_1 z_{E', \rho'}])))} \text{i-cost}_{\nu}^{E', \rho'}(s_2)
\]
and using Eq. (10), we deduce that:
\[
\text{cost}_{\nu}^{E', \rho'}(s_1) + \text{cost}_{\nu}^{E', \rho'}(s_2) \leq \text{i-cost}_{\nu}^{E', \rho', \hat{k}}(s_1; s_2) + \sum_{H, g \notin \text{proc}_{\nu}(\hat{k})} \sum_{j = \hat{k}[H, g]}^{\hat{k}[H, g]} \text{call-cost}_{H, g}(j)
\]
We conclude by taking the max over \( v' \) and using Eq. 9.

- if \( s = x \leftarrow e \). Let \( \hat{k} \leq \lambda_c[0] \) and \( v \in I(\hat{k}) \). Since \( \text{mem}_v(s) \cap \text{FV}(I) = \emptyset \), we know that \( x \not\in \text{FV}(I) \). Hence the invariant is preserved, i.e.:
\[
\forall v' \in \text{supp}(\pi_{\nu}^+([s_2 z_{E', \rho'}])), v' \in I(\hat{k})
\]
Since \( \text{#calls}_{\nu}^{E', \rho'}(s) = 0 \), this proves the first point. Moreover:
\[
\text{cost}_{\nu}^{E', \rho'}(x \leftarrow e, v') = \text{i-cost}_{\nu}^{E', \rho', \hat{k}}(x \leftarrow e)
\]
which concludes the proof.

- if \( s = e \in \text{d} \), conditional if \( e \) then \( s_1 \) else \( s_2 \) and while loop while \( e \) do \( s \) cases are similar. We omit the details.
Since \( \bar{p} \) is well-typed in \( E \), we know that \( \text{cost}^{E',\rho'}(\bar{p}[H],g) \geq \text{cost}^{E,\rho}((\bar{p})[H],g) \). Hence:

\[
\text{cost}^{E',\rho'}(x \leftarrow \text{call } f(\bar{v})) = i \cdot \text{cost}^{E',\rho'}(x \leftarrow \text{call } f(\bar{v})) + \text{cost}^{E,\rho}((\bar{p})[H],g)
\]

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

**F.2 Instantiation Rule Soundness**

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

i) \( m \) has type \( \text{erase}	ext{ compi}(M_1) \);

ii) and we can show that \( m \) verifies \( M_1 \)'s complexity restriction by proving that:

\[
\forall f \in \text{procs}(M_1), \quad E, \text{module } \bar{x} = \text{abs}(\text{open}; M) + \{ \tau \} m(\bar{x}), f(\tau \mid t_f) \quad \text{where } \bar{x} \text{ are } M_1 \text{'s functor parameters, and } M \text{ their types.}
\]

**Lemma F.3.** Let \( E \) be a well-typed environment, \( M_1 \) be low-order module signature s.t.:

\[
M_1 = \text{func}(\bar{x} : \bar{M}) \text{ sig } \_ \text{ restr } \theta \text{ end}
\]

and \( m \) be a module expression s.t. \( E \vdash x : \text{erase compi}(M_1) \). Let \( E_a = E, \text{module } x = \text{abs}(\text{open}; M_1) \) and \( \rho \) be an \( E_a \)-interpretation. Moreover, for any \( f \in \text{procs}(M_1) \), we have:

\[
E, \text{module } \bar{x} = \text{abs}(\text{open}; \bar{M}) + \{ \tau \} m(\bar{x}), f(\tau \mid t_f) \quad \text{such that } \bar{x} \text{ are } M_1 \text{’s functor parameters, and } \bar{M} \text{ their types.}
\]

**Lemma F.4.** Let \( E = E' \); module \( \bar{x} = \text{abs}(\text{open}; \bar{M}) \); and \( \rho' = \rho, (\bar{x} : \bar{m}) \).

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

\[
\forall v, \text{cost}^{E',\rho'}(m(\bar{x}),f) \leq t_f[\text{conc}] + \sum_{A \in E'} \text{cost}^{E,\rho}(A, h) \cdot \text{comp}^{E,\rho}(A, h)
\]

Let \( N > 0 \) be an non-zero positive integer. We are going to change the interpretation of \( \tau \) in \( \rho' \) by adding some code doing nothing and taking time \( N \). Let \( \text{st}_1 \) be such that \( \rho'(\tau) = \text{struct } \text{st}_1 \). By typing hypothesis, we know that \( \text{st}_1 \) is of the shape:

\[
\text{st}_1 = \text{st}_1; \text{ proc } g(\bar{v} : \bar{\tau}) \to \tau_r = \{ \text{ var } (\bar{g}_1 : \bar{h}_1); \text{ sg}; \text{ return } e_g \}; \text{ st}_2
\]

Then, let \( \text{st}'_N \) be the module structure:

\[
\text{st}'_1; \text{ proc } g(\bar{v} : \bar{\tau}) \to \tau_r = \{ \text{ var } (\bar{g}_1 : \bar{h}_1); (\text{ tic}^N; \text{ sg}); \text{ return } e_g \}; \text{ st}_2
\]

and let \( \rho_N' \) be the interpretation with the same domain as \( \rho' \) s.t.:

\[
\forall w \in \text{dom}(\rho'), \rho_N'(w) = \begin{cases} \rho'(w) & \text{if } w \neq y \\ \text{struct } \text{st}_N' \text{ end} & \text{if } w = y \end{cases}
\]

Let \( v \) be some arbitrary memory. Since \( \rho' \) is an \( E' \)-interpretation, then so is \( \rho_N' \). Using Eqn. (14), we get:

\[
\text{cost}^{E',\rho_N'}(m(\bar{x}),f) \leq t_f[\text{conc}] + \sum_{A \in E'} \text{cost}^{E,\rho}(A, h) \cdot \text{comp}^{E,\rho}(A, h)
\]

Moreover, we can easily check that:

\[
\#\text{calls}_{\bar{y}, \bar{g}}(m(\bar{x}), f) = \#\text{calls}_{\bar{y}, \bar{g}}(m(\bar{x}), f)
\]
and:
\[ \text{cost}_{E', \rho'}_{\text{module}}(m(x), f) = \text{cost}_{E', \rho'}(m(x), f) + N \cdot \text{#calls}_{y, g}(m(x), f) \]

From Eq. (20), we have:
\[ \text{cost}_{E', \rho'}_{\text{module}}(m(x), f) + N \cdot \text{#calls}_{y, g}(m(x), f) \leq \]
\[ \sum_{h \in \text{procs}_{\text{inst}}(A)} t_f[A, h] \cdot \text{compl}_{A,h}^{E', \rho'}_{\text{module}} \]

Using the inequality above, and Eq. (19), we have:
\[ N \cdot \sum_{h \in \text{procs}_{\text{inst}}(A)} t_f[A, h] \cdot (\text{compl}_{A,h}^{E', \rho'} - \text{compl}_{A,h}^{E, \rho'} ) \leq \]
\[ t_f[y, g] \cdot (\text{compl}_{y, g}^{E', \rho'} - \text{compl}_{y, g}^{E, \rho'} ) \]

(Since \( \forall z \neq y, \rho'_N(z) = \rho'(z) \))
\[ \leq t_f[y, g] \cdot N \]

We deduce that \#calls_{y, g}(m(x), f) \leq t_f[y, g]. From Eq. (14), we get that:
\[ \#calls_{y, g}(m(x), f) \leq \| f \|[y, g] \]

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

Finally, to prove that Eq. (17) and Eq. (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 F.4.** For every well-typed environment \( \mathcal{E}_c \), if:
\[ \mathcal{E}_c = \mathcal{E}_1; \mathcal{E}_2 \quad \text{where} \quad \mathcal{E}_1 = \mathcal{E}_0; \text{module } x = \text{absopen} : M_1 \]
and \( \mathcal{E}_2 \) contains only module declarations. Then for every \( f \in \text{procs}(M_1) \) and \( \mathcal{E}_c \)-interpretation \( \rho \), we have \( \text{compl}_{\mathcal{E}_1; \mathcal{E}_2, f} = \text{compl}_{\mathcal{E}_1, f} \), where \( \rho_1 \) is the restriction of \( \rho \) to \( \mathcal{E}_1 \)'s abstract modules.

**Proof Sketch.** Assume \( M_1 = (\text{func}(\bar{x} : \bar{\mathcal{M}}) \text{ sig } \text{ end}) \). Let \( \mathcal{E}' \) be \( (\mathcal{E}_c, \text{module } \bar{x} = \text{absopen} : \bar{\mathcal{M}}) \). We prove that \( \text{compl}_{\mathcal{E}', f} = \text{compl}_{\mathcal{E}_1, f} \), for every \( f \) in \( \text{procs}(S_1) \) and \( \mathcal{E}_1 \)-interpretation \( \rho_1 \) that leaves the intrinsic cost of \( x \) unchanged, i.e., such that for any \( v \):
\[ i \text{-cost}_{\mathcal{E}', f, \bar{x}}(x, \bar{x}, f) = i \text{-cost}_{\mathcal{E}, f, \bar{x}}(x, \bar{x}, f) \]

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

**Lemma F.5.** The rule **INSTANTIATION** given in Figure 27 is sound.

**Proof.** We consider an instance of the rule **INSTANTIATION**. We want to prove that:
\[ \mathcal{E}_c, \text{module } x = m : M_1 \vdash \{ \varphi \} s \{ \psi | T_{\text{ins}} \} \]

Let \( \mathcal{E}_c = (\mathcal{E}_c, \text{module } x = m : M_1) \) and \( \mathcal{E}_a = (\mathcal{E}_a, \text{module } x = \text{absopen} : M_1) \). We know that:
\[ M_1 = \text{func}(\bar{y} : \bar{\mathcal{M}}) \text{ sig } S_1 \text{ restr } \theta \text{ end} \]
\[ \mathcal{E}_a \vdash x : \text{erase} \cdot \text{compl}(M_1) \]

Let \( \rho \) be an \( \mathcal{E}_c \)-interpretation and \( \nu \in \phi \), we need to show that for every memory \( v \):
\[ \text{supp}(\pi^\nu(\llbracket s \rrbracket_{\mathcal{E}_c, \rho})) \subseteq \psi \]
\[ \text{cost}_{\mathcal{E}_c, \rho}(s) \leq T_{\text{ins}}[\text{conc}] + \sum_{A \in \text{procs}(S_1)} T_{\text{ins}}[A, g] \cdot \text{compl}_{A,g}^{\mathcal{E}_c, \rho} \]

We know that \( M_1 = \text{func}(\bar{y} : \bar{\mathcal{M}}) \text{ sig } S_1 \text{ restr } \theta \text{ end} \) and:
\[ \mathcal{E}_a \vdash \{ \varphi \} s \{ \psi | T_s \} \]

Using Lemma F.3, we know that \( \rho_a = \rho, (x \mapsto m) \) is an \( \mathcal{E}_a \)-interpretation. Hence, using Eq. (23), we deduce that:
\[ \text{supp}(\pi^\nu(\llbracket s \rrbracket_{\mathcal{E}_a, \rho_a})) \subseteq \psi \]
\[ \text{cost}_{\mathcal{E}_a, \rho_a}(s) \leq T_s[\text{conc}] + \sum_{A \in \text{procs}(S_1)} T_s[A, g] \cdot \text{compl}_{A,g}^{\mathcal{E}_a, \rho_a} \]

We replace \( \text{compl}_{A,g}^{\mathcal{E}_a, \rho_a} \) by \( \text{compl}_{A,g}^{\mathcal{E}_c, \rho} \) for every \( A \in \mathcal{E}_c \) and \( g \in \text{procs}_{\mathcal{E}_c}(A) \) using Lemma F.4:
\[ \text{cost}_{\mathcal{E}_c, \rho}(s) \leq T_s[\text{conc}] + \sum_{f \in \text{procs}(S_1)} T_s[f, x] \cdot \text{compl}_{\mathcal{E}_a, \rho_a}^{\mathcal{E}_c, \rho} \]

Using Lemma F.3, we upper-bound \( \text{compl}_{\mathcal{E}_a, \rho_a}^{\mathcal{E}_c, \rho} \) for every \( f \in \text{procs}(S_1) \):
\[ \text{compl}_{\mathcal{E}_a, \rho_a}^{\mathcal{E}_c, \rho} \leq T_f[\text{conc}] + \sum_{A \in \text{procs}(\mathcal{E}_c)} T_f[A, g] \cdot \text{compl}_{A,g}^{\mathcal{E}_c, \rho} \]
We check that the quantities above are identical when evaluated in \( \mathcal{E}_c \), hence:

\[
\text{compl}_{x,f}^{E_a^\rho} \leq t_f[\text{conc}] + \sum_{A \in (\mathcal{E}_c)} \sum_{g \in \text{procs}_{E_c}(A)} t_f[A.g] \cdot \text{compl}_{A.g}^{E_a^\rho}
\]

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

\[
\text{cost}_{\nu_{E_a^\rho}}(s) \leq t_s[\text{conc}] + \sum_{f \in \text{procs}(S)} t_s[x.f] \cdot t_f[\text{conc}] + \\
\sum_{A \in (\mathcal{E}_c)} \sum_{g \in \text{procs}_{E_c}(A)} (t_s[A.g] + \sum_{f \in \text{procs}(S)} t_s[x.f] \cdot t_f[A.g]) \cdot \text{compl}_{A.g}^{E_a^\rho}
\]

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