Université Paris VII — Denis Diderot

UFR d'Informatique

École doctorale de Sciences mathématiques de Paris Centre

Thèse

pour l'obtention du diplôme de

Docteur de l'Université Paris Diderot

Spécialité : **Informatique**

Sûreté des abstractions et Sessions sécurisées dans les langages distribués

Pierre-Malo Deniélou

Directeurs : James J. Leifer et Jean-Jacques Lévy

présentée et soutenue le ../01/2010

devant le Jury composé de :

M.	Giuseppe Castagna	Président
M.	Jean-Jacques Lévy	directeur
M.	James J. Leifer	directeur
M.	Joshua Guttman	rapporteur
M.	Cosimo Laneve	rapporteur
M.	Hubert Comon-Lundh	examinateur
M.	Cédric Fournet	examinateur
Mme.	Nobuko Yoshida	examinatrice

Résumé ...

Abstract ...

À mon grand-père, Guy Deniélou
14 Juin 1924 - 12 Décembre 2008

Remerciements

La thèse est un marathon (de 42,195 mois) qu'heureusement, on ne court pas seul. Je remercie donc chaleureusement mes entraîneurs James J. Leifer et Jean-Jacques Lévy pour leur amitié et leur expertise, tant du point de vue de la technique (de preuve) que de la tactique (administrative). Je salue évidemment mes camarades Cédric Fournet, Ricardo Corin et Karthikeyan Bhargavan, tous les coureurs et accompagnateurs (notamment Sylvie et Martine) des équipes Moscova et Gallium, du PPS et du laboratoire commun Microsoft Research-INRIA. Merci enfin à Hubert Comon, le coach de mon centre de formation à l'ENS Cachan.

Les coupeurs de citron de la première heure viennent évidemment de la famille : Yves-Pol, Claire, Gilbert, Geneviève, pour n'en citer que quelques uns, sont là depuis les premiers pas et les premières courses.

Je n'oublie pas enfin les encouragements ô combien importants venus de la foule en délire (ou pas) : un grand merci à Brice, Julien, Samuel, Julien, Nicolas, Nicolas et Alice, Sylvain et Fabienne, Christine et Alain, Pierre-Loïc et Pascaline, Sonia, Anna et Guillaume, Marie, Marie et Damien, Virginie, Caroline, Vincent, Benoît, Jean-Claude et Bénédicte, Armande et Florian, Rachel, Jennifer, Alain, Eric, Behzad, Srikumar, Emine, Eric, Greg, Mie, Björn, Sara, John, Gustavo.

Je ne pouvais terminer cette page de remerciements sans mentionner ceux qui m'accueillent si bien aujourd'hui à Londres : merci à Nobuko et Kohei, Sergio et Becky, Robin, Ray, Dimitris et Andi.

Cette thèse a été réalisée en partie lors de séjours et stages au laboratoire de Microsoft Research à Cambridge (Royaume-Uni). Je remercie chaleureusement Cédric et Karthik de m'y avoir invité.

Table des matières

Chapitre 1

Introduction

Programmation répartie

Un système réparti, ou système distribué, est un ensemble de programmes indépendants qui réalisent une tâche globale grâce à leurs interactions par le réseau. Ces systèmes sont aujourd'hui omniprésents, des serveurs web aux grappes de calcul, des jeux en réseau aux infrastructures de la finance mondiale. En dépit de leur importance croissante, la programmation de ces systèmes ne jouit pas encore d'un ensemble d'outils équivalent à ceux que la programmation usuelle (séquentielle) possède.

La programmation répartie donne en effet lieu à de nombreux défis pour le programmeur. La première difficulté est liée à l'incertitude sur l'environnement d'exécution : chaque programme indépendant est conçu avec une connaissance et un contrôle restreint des états que les autres programmes participants et le réseau vont pouvoir atteindre lors de l'exécution. Deuxièmement, les interactions possibles entre les programmes indépendants formant un système réparti sont extrêmement complexes et rendent la conception, l'analyse et la maintenance d'un tel système, difficiles : les actions de chacun des participants se répondent les unes aux autres et s'entrelacent de manière déstructurée. La troisième difficulté concerne les tentatives de vérification globale : le comportement que l'on souhaite obtenir de la part du système distribué repose sur des programmes indépendants, souvent inaccessibles et dont le comportement peut changer avec le temps.

Pour faire face à ces difficultés, les programmeurs ne disposent que de peu d'outils : les langages de programmation sont séquentiels avant tout, les bibliothèques standard ne concernent que les cas de communication les plus simples, les compilateurs manquent de moyens de vérification de propriété globale. Il est donc nécessaire d'aider le programmeur à réaliser plus facilement des logiciels distribués complexes, tout en assurant flexibilité, fiabilité et sécurité.

Les questions que nous nous posons sont donc les suivantes : comment concevoir des programmes distribués capables de s'accommoder de changements dans leur environnement d'exécution ? Comment programmer des systèmes répartis garantissant la réalisation d'une tâche globale envisagée ? Comment aider les programmeurs à obtenir des systèmes distribués flexibles, fiables et sécurisés ?

Langages de programmation, méthodes expérimentales et formelles

Pour aborder le problème de la programmation répartie, notre conviction est qu'il est nécessaire d'agir au niveau du langage de programmation et des outils de vérification automatiques. Toutes les tâches qui peuvent être épargnées au programmeur par des abstractions bien conçues, tous les bogues qui peut être évités dès la phase de typage ou de compilation, rendent les programmes répartis plus faciles à réaliser et plus fiables une fois

produits.

Nous proposons ainsi, dans cette thèse, de nouveaux langages et de nouvelles structures permettant au programmeur de faire face à certaines situations spécifiques aux systèmes répartis. Nous accompagnons ces langages d'outils (systèmes de types, compilateur) permettant au programmeur d'obtenir à partir de la programmation locale des garanties globales.

Pour s'assurer de la solidité de ces garanties, nous faisons appel à la fois à l'étude expérimentale et aux méthodes formelles[1]. Les programmes sont à la fois des objets industriels qui réclament une expertise technologique, c'est-à-dire une évaluation expérimentale, et des objets logiques, dont les mathématiques peuvent décrire, via des théorèmes, le comportement. C'est la spécificité de l'informatique comme technologie des objets logiques qui nous permet d'utiliser, de façon directe et complémentaire, ces deux méthodes.

Approches

Rendre la programmation répartie plus simple, c'est tout d'abord apporter de la structure. C'est elle seule qui permet la conception, l'analyse et la maintenance des projets complexes. Les méthodes usuelles (modules, objets, polymorphisme) ne sont pas adaptées au cadre de la programmation distribuée : ces dernières reposent sur des contraintes syntaxiques ou sémantiques locales. Une première idée est donc d'étendre la validité de ces structures à l'échelon global d'un système réparti. La seconde idée repose sur le fait que l'essence de la programmation distribuée est l'interaction et la coordination entre entités locales. Il est donc naturel de vouloir proposer aux programmeurs de nouvelles méthodes permettant de structurer ces communications.

De nouvelles structures pour programmer les systèmes répartis ne sont utiles que lorsque leur sémantique, leurs propriétés, les abstractions qu'elles représentent sont clairement définies. Inversement, pour élaborer ces structures, la question des propriétés que nous en attendons est cruciale. Nous partons de l'intuition suivante : partir d'un langage séquentiel pour en faire un langage distribué grâce à l'ajout de nouvelles structures n'a de sens que lorsque la sémantique obtenue est cohérente et fiable. Il s'agit de s'assurer d'une part que le langage ainsi créé est sûr, c'est-à-dire qu'il ne crée pas d'erreur par sa propre incohérence, et, d'autre part, que la sémantique résultante reste valide dans l'environnement incontrôlé d'une exécution répartie. En d'autres termes, nous nous intéressons au propriétés de sûreté (absence de danger) et de sécurité (absence d'exposition aux comportements malveillants).

C'est à partir de ces deux approches et de ces deux propriétés que cette thèse est organisée.

Le chapitre 2 étudie la généralisation des types abstraits de données à la programmation distribuée. Les types abstraits sont en effet des construction locales dont les garanties proviennent de critères syntaxiques qui ne sont plus valables dans un système réparti. Nous partons d'un système préexistant que nous rendons plus flexible grâce à l'introduction du sous-typage. Ceci permet de prendre en compte certains phénomènes des systèmes distribués, comme les mises à jour de programmes et de bibliothèques. La cohérence de notre langage est assurée dans un modèle où tous les participants sont bien typés et où le réseau est fiable : les propriétés de sûreté que sont l'aboutissement des calculs et l'absence d'erreur à l'exécution sont donc garanties et permettent de s'assurer que les propriétés locales des types abstraits sont préservées dans le contexte distribué.

Le chapitre 3 s'intéresse lui aux sessions, une construction permettant de chorégraphier

[1] Les méthodes formelles ne sont que l'autre nom des mathématiques lorsqu'elles ont pour objet des phénomènes technologiques, comme le sont les programmes.

les interactions entre de multiples participants. Nous nous occupons plus précisément de l'implémentation de cette abstraction dans la perspective d'obtenir des propriétés de sécurité. Notre implémentation sécurisée permet en effet à un programmeur d'une des parties indépendantes d'un système distribué, de ne pas avoir à se soucier du réseau ou de la corruption des autres parties : la propriété de sécurité de notre implémentation lui garantit que la session à laquelle il participe se déroulera toujours conformément à sa définition. Dans ce chapitre, nous présentons en détail le langage de description des sessions, notre compilateur, ainsi que le protocole cryptographique généré automatiquement pour chaque session.

Le chapitre 4 clôt notre thèse avec un résumé des principales contributions et un certain nombre de pistes pour les explorations futures.

Publications

Les résultats présentés dans le chapitre 2 ont fait l'objet d'une publication à la conférence ICFP en Septembre 2006 [22]. Les résultats présentés dans le chapitre 3 ont fait l'objet de publications aux conférences CSF'07 [6] et TGC'07 [19], ainsi que d'une publication dans le *Journal of Computer Security* [7]. Certains éléments ont aussi été présentés à la conférence CSF'09 [8].

Le code source et les exemples présentés peuvent être trouvés sur la page web de la thèse [23].

Chapitre 2

Sérialisation sûre de données abstraites

Les systèmes distribués reposent sur la coopération de programmes indépendants en vue de réaliser une tâche globale. Dans ce chapitre, nous nous intéressons à la sûreté de tels systèmes vis-à-vis de l'usage de certains éléments essentiels provenant de la programmation séquentielle : les types abstraits, pour la structure, et le sous-typage, pour la flexibilité.

2.1 Introduction

Nous commençons cette section par une description de quelques enjeux concernant la sûreté des langages distribués, les types abstraits et le sous-typage.

2.1.1 Types abstraits dans un environnement distribué

Les systèmes répartis fonctionnent grâce à l'échange, via le réseau, de valeurs qui ont été produites ou modifiées sur des machines distantes. Or les valeurs qui sont transmises possèdent des propriétés qui, dans la programmation séquentielle, peuvent être garanties localement grâce à la syntaxe ou au système de types. Dans le monde distribué, la situation se complique.

Sûreté de la sérialisation

La sérialisation est une opération qui a pour objet de convertir tout type de donnée en chaîne de caractères, dans le but ultérieur de la transmettre à un autre programme par le réseau. Ce dernier va alors essayer de recréer la valeur d'origine grâce à l'opération inverse de désérialisation.

Malheureusement, l'importation d'une valeur d'origine extérieure au sein d'un programme en cours d'exécution n'est pas sans risque : les deux programmes ne comportent pas nécessairement les mêmes modules, n'ont pas forcément accès aux mêmes versions de bibliothèques ou n'ont peut-être pas été compilés par le même compilateur. La désérialisation fait ainsi peser une menace importante sur la sûreté du langage car la sémantique d'une valeur sérialisée sur une machine n'est pas forcément identique à la sémantique de sa version désérialisée sur une autre.

Dans le cadre de notre étude, la programmation distribuée est modélisée à l'aide de deux fonctions send (de type `string` → `unit`) et receive (de type `unit` → `string`) qui représentent l'envoi et la réception de chaînes de caractères par le réseau. Ce dernier est simplement représenté comme un canal de communication unique, fiable et non-typé.

Ces simples fonctions d'envoi et de réception permettent de s'abstraire dans la suite des considérations relatives au réseau.

Nous notons (*Programme1* || *Programme2*) le fait de mettre, sur deux machines distinctes reliées par le réseau, les programmes *Programme1* et *Programme2*, qui auront alors la possibilité de s'échanger des chaînes de caractères grâce aux fonctions send et receive mentionnées précédemment.

Enfin, pour la sérialisation et la désérialisation, nous utilisons respectivement les fonctions `marshall` et `unmarshall` et requérons l'annotation explicite des types des valeurs sérialisées et désérialisées. La sérialisation de la valeur 3 est ainsi écrite

```
marshall( 3 : int )
```

alors que la désérialisation vers un entier d'une valeur contenue dans la variable x s'écrit

```
unmarshall (x) : int
```

Exemple 2.1 (Fonctions marshall et unmarshall) Voici un exemple d'envoi d'une valeur d'une machine vers une autre. Le *Programme1* va sérialiser, puis envoyer un entier. Le *Programme2* va désérialiser la chaîne reçue comme entier, puis utiliser cette valeur.

```
(* Programme1 : *)
      send ( marshall (3 : int))

(* Programme2 : *)
      print_int (unmarshall (receive ()) : int)
```

On met ensuite les deux programmes sur deux machines reliées par le réseau.

(*Programme1* || *Programme2*)

Le résultat sera l'impression sur l'écran de la seconde machine du chiffre `3`.

Nous illustrons maintenant quelques unes des difficultés qui surviennent lors de la désérialisation. Nous prenons pour exemple le langage Ocaml [48] qui ne fait que tester si les compilateurs de chacun des deux programmes ont le même numéro de version. Par conséquent, en Ocaml, seule la correspondance des fonctions de sérialisation et désérialisation est assurée, ce qui garantit l'absence d'erreur lors de la désérialisation elle-même. Les exemples suivants montrent que l'interprétation, c'est-à-dire la sémantique, de la valeur reçue joue aussi un rôle et peut causer des erreurs bien plus tard dans l'exécution.

Exemple 2.2 (Segfault en Ocaml) L'exemple suivant cause un `Segmentation fault` en Ocaml. Nous nous contentons de sérialiser (fonction `Marshal.to_string`) une valeur d'un type particulier (ici un entier), et de désérialiser (fonction `Marshal.from_string`) la chaîne obtenue en attendant un autre type (par exemple un flottant).

```
        Objective Caml version 3.11.0
# let s = Marshal.to_string 3 [ ] in
  1. +. (Marshal.from_string s 0 : float);;
Erreur de segmentation
```

Ici, le décodage de l'entier 3 se passe sans problème, jusqu'à l'interprétation de cette valeur comme une valeur flottante dont la représentation en mémoire est différente. Ce programme se termine donc par une erreur de segmentation.

La situation est plus complexe lorsque les représentations des valeurs envoyées et attendues sont compatibles, mais ne sont pas associées aux mêmes propriétés : les erreurs deviennent subtiles et difficiles à détecter et corriger.

Exemple 2.3 (true, false et 2) Via la désérialisation, il est possible en Ocaml de convertir un entier quelconque en booléen, puisqu'ils partagent la même représentation.

```
        Objective Caml version 3.11.0
# let s = Marshal.to_string 2 [] in
  let  b = (Marshal.from_string s 0 : bool) in
  (b && true, (not b) && true);;
- : bool * bool = (true, true)
```

Le booléen `b` obtenu ci-dessus à partir de l'entier `2` ne correspond cependant ni à `true`, ni à `false` : aucune des propriétés associées aux booléens n'est garantie et l'exécution ne suit plus la sémantique qui leur est associée.

La solution immédiate est de sérialiser, en même temps que la valeur, son type, puisque ce dernier est le garant habituel d'une sémantique spécifique. La désérialisation peut alors tester si le type attendu correspond au type envoyé et renvoyer une erreur lorsque ce n'est pas le cas : il vaut mieux une erreur dès la désérialisation. La question de la sûreté de la désérialisation se rapporte donc à la question de l'égalité entre types.

Tester cette dernière est simple lorsque le type de la donnée envoyée est une combinaison de types de base, qui ont une valeur universelle dans tous les programmes écrits dans le même langage. Malheureusement pour la vérification de l'égalité des types, le type de la valeur envoyée peut dépendre, via des abréviations ou des types abstraits, d'autres parties du programme qui ne sont pas partagées avec le programme destinataire : tester l'égalité des types n'est pas un problème simple dans le monde distribué.

Les stratégies pour résoudre ces difficultés sont diverses. Certains langages choisissent de laisser ce problème entre les mains du programmeur et vérifient simplement que les méthodes pour sérialiser et désérialiser correspondent, mais ne vérifient rien au niveau du typage (c'est le cas d'Ocaml comme nous l'avons vu plus haut).

D'autres langages font le choix de proposer des canaux qui abstraient à la fois la transmission par le réseau et le processus de sérialisation et désérialisation, tout se déroule de façon transparente et sûre. Cependant, la difficulté se porte alors sur l'établissement des canaux qui doivent s'accorder sur les noms à donner aux types. En Jocaml [32, 31] par exemple, l'établissement des canaux n'est sûr que s'il a lieu entre deux instances d'un même exécutable (deux programmes distincts doivent passer par un serveur de nom spécifique). La même restriction existe en C# [38] où les noms associés aux types dépendent de l'exécutable.

Une autre piste est d'abandonner la préservation du typage en ne se préoccupant que de l'absence d'erreurs fatales : l'objectif est de ne jamais obtenir d'`Erreur de segmentation` lors de la réception puis de l'utilisation d'une valeur. Java [44] envoie ainsi, en même temps que la valeur, le nom de la classe de l'objet, son interface et les types concrets des attributs. Il est possible de procéder d'une façon similaire en Ocaml [39].

Valeurs abstraites

Un type abstrait est un type déclaré dans la signature d'un certain module, mais dont la connaissance manifeste n'est réservée qu'à ce module : le reste du programme n'a pas accès à sa représentation interne et, pour le manipuler, peut seulement utiliser les fonctions qui sont exportées par l'interface.

Les types abstraits font partie des atouts des langages de programmation modernes. Ils sont utilisés lorsque l'implémentation d'un ensemble de types et de fonctions est cachée du reste du programme par une interface. Celle-ci impose alors une contrainte sur la création et la manipulation des valeurs dont le type est ainsi abstrait.

Ce mécanisme permet au programmeur de construire des types de données dont l'interface, par les limites qu'elle impose à la manipulation des données, garantit la continuelle validité des invariants internes. Le respect de cette propriété, que nous désignons par le terme *préservation des frontières de l'abstraction*, *préservation de l'abstraction* ou plus simplement *intégrité* est assuré par le système de types et la sémantique du langage qui vérifient que les accès aux types abstraits ne sont effectués qu'à l'aide de l'interface prévue.

La propriété de *confidentialité* est différente : celle-ci signifie que l'implémentation concrète des types abstraits est cachée et ne peut être observée par du code situé en dehors des frontières de l'abstraction. Plus précisément, la propriété demande que deux implémentations observationnellement équivalentes le restent, quel que soit le contexte dans lequel elles sont placées. On pourra se rapporter à l'introduction de la thèse de Gilles Peskine [57] pour une remarquable discussion des différentes propriétés et travaux relatifs aux types abstraits.

Exemple 2.4 (Compteur) Voici l'exemple d'un module définissant un type abstrait de compteur. Une structure définit un type (ici `int`) qui sera l'implémentation concrète du compteur. Elle définit aussi un compteur initial et quelques fonctions qui permettent d'incrémenter et d'accéder à la valeur d'un compteur. On adjoint à cette structure une signature. Celle-ci va déclarer le type `t` comme abstrait en omettant de donner son expression concrète : type `t`. Si la signature avait donné la valeur de l'implémentation de `t` avec la déclaration type `t = int`, on aurait pu accéder à un compteur *exactement* comme à un entier (type manifeste). La déclaration abstraite ne permet ici l'utilisation du compteur que par les éléments du module qui sont déclarés dans la signature. On ne peut donc que créer un compteur en appelant la valeur `initial`, l'incrémenter par la fonction `incremente` et en extraire la valeur sous forme d'entier en utilisant la fonction `valeur`.

```
module Compteur =
  struct                          sig
    type t = int                    type t
    let  initial = 0          :     val  initial : t
    let  incremente x = x + 1       val  incremente : t  → t
    let  valeur x = x               val  valeur : t  → int
  end                             end
```

On ne pourra donc pas diminuer la valeur d'un compteur décrit par ce module, faute d'accès à la représentation interne ou de fonction réalisant cette opération. Avoir un type abstrait représente ainsi une contrainte très forte sur les invariants associés à une donnée abstraite.

La propriété de confidentialité veut, elle, que pour le programmeur l'implémentation du compteur à l'aide par exemple d'un nombre négatif à la place d'un nombre positif ne soit pas détectable tant que l'interface et les invariants qui lui sont associés ne changent pas.

```
module Compteur =
  struct                          sig
    type t = int                    type t
    let  initial = 0          :     val  initial : t
```

```
  let incremente x = x - 1        val incremente : t  → t
  let valeur x = - x              val valeur : t  → int
end                             end
```

Quelles que soient les séquences d'opérations effectuées par l'utilisateur, le remplacement de la première implémentation du module `Compteur` par la seconde ne changera rien au résultat obtenu.

Notre objectif est d'assurer la sûreté du langage réparti, c'est-à-dire du respect de la sémantique telle que l'a spécifiée le programmeur via l'interface qu'il a donnée à son type abstrait. Nous nous intéressons donc dans la suite à la préservation des frontières de l'abstraction et laissons de côté la question complémentaire de la confidentialité.

Les langages de la famille de ML reposent sur un typage statique suffisant pour s'assurer localement de la préservation des abstractions. Lors de l'exécution, les informations concernant le typage peuvent alors être effacées du programme sans le moindre risque pour la sûreté. Dans un programme distribué, la situation est différente puisque l'échange de valeurs par le réseau requiert la transmission d'une certaine quantité d'information de typage pour garantir la sûreté de l'exécution.

Dans le cas des types abstraits, la connaissance des types concrets des valeurs échangées n'est pas suffisante pour s'assurer qu'une valeur ne sera pas reçue avec un type incorrect, ce qui invalide la préservation de l'abstraction. Les noms des types abstraits ne sont pas non plus suffisants car l'espace de nommage est purement local. En effet, si l'on veut envoyer un compteur après sérialisation, comment s'assurer que le correspondant va le récupérer comme un compteur avec toutes les contraintes qui s'y attachent et non comme un vulgaire entier ou comme un compteur aux propriétés différentes (mais qui peut avoir le même nom comme dans l'exemple ci-dessus).

La solution des empreintes et des couleurs

Leifer et coll. [46] proposent une solution à ce problème basée sur la création d'un espace global de nommage des types abstraits : le nom d'un type abstrait fait référence à son module d'origine via l'empreinte cryptographique de son code source. Cette méthode permet de savoir si deux types abstraits présents sur deux machines différentes font effectivement référence à une même implémentation, et donc à de mêmes invariants et à une même interface. Leur idée a été développée dans le prototype de langage de programmation Acute réalisé par Sewell et coll. [65].

Les identifiants des types abstraits (par exemple `Compteur.t`), qui faisaient référence au nom du module d'origine, sont ainsi remplacés à la compilation par une référence au résultat de l'empreinte de ce module d'origine (i.e. `h.t` si `h` est l'empreinte du module `Compteur`). L'ordre de déclaration des modules est respecté pour qu'au final toutes les références directes aux noms des modules aient été substituées par les empreintes correspondantes. On ne fait donc que l'empreinte d'un module lorsqu'il est *clos*, et comme toutes les références à celui-ci dans les modules suivants sont remplacées par des références à son empreinte, les dépendances entre modules sont prises en compte. Après sérialisation et désérialisation, il va donc être possible d'associer les types abstraits référant au même module en comparant les empreintes auxquels ils font référence. De plus, le nom du module est pris en compte dans l'empreinte. Cela permet ainsi au programmeur de forcer la distinction entre deux modules qui ont la même implémentation. Enfin, comme deux implémentations peuvent être identiques à α-renommage ou permutation près, l'empreinte ne peut se faire réellement sur le code source. Elle se fait donc plutôt sur son arbre de syntaxe abstraite normalisé.

Dans les langages habituels comme ML [50], les valeurs abstraites ne sont pas distinguées à l'exécution de leur forme concrète, ce qui nous empêche lors de la désérialisation de distinguer les abstractions. La solution envisagée fait ainsi intervenir des crochets colorés qui viennent protéger les valeurs abstraites d'une confusion avec les valeurs concrètes lors de l'exécution (on a donc $[e]_c^T$ au lieu de simplement e). Les crochets colorés sont annotés par une couleur, c'est-à-dire par un ensemble d'empreintes, précisément les empreintes des modules dont les types abstraits peuvent être révélés et manipulés à l'intérieur. La présentation originelle des crochets colorés est due à Grossman, Morrisett et Zdancewic [36].

Les empreintes ont lieu uniquement lors de la compilation. Les crochets colorés peuvent apparaître lors de la compilation comme de l'exécution du programme, mais l'utilisateur ne peut pas utiliser explicitement ni les uns ni les autres dans son programme : les crochets sont uniquement des outils sémantiques.

Nous reviendrons sur les empreintes et les crochets colorés de façon plus détaillée dans les sections suivantes. Un exposé détaillé sur l'utilisation des empreintes peut aussi être trouvé dans le papier décrivant Acute [65].

2.1.2 Sous-typage et enregistrements

Le sous-typage permet d'augmenter considérablement l'expressivité d'un langage en agissant comme une forme dérivée de polymorphisme. Nous utilisons les enregistrements comme prétexte à son introduction.

Enregistrements

Les enregistrements sont des n-uplets dont les éléments sont étiquetés plutôt qu'ordonnés. Il s'agit d'une structure de donnée simple qui a la faculté de donner une base de modélisation à de nombreuses autres structures de données comme les objets.

Exemple 2.5 (Enregistrements) Nous écrivons un enregistrement dans Ocaml à l'aide d'accolades, au sein desquelles des valeurs sont associées à des étiquettes. L'enregistrement ci-dessous associe respectivement aux étiquettes `nom`, `prenom` et `age`, des valeurs de types `string`, `string` et `int`. On note un type enregistrement de manière similaire. Le type correspondant à l'enregistrement précédent sera ainsi :

```
{ nom : string ; prenom : string ; age : int }
```

L'accès aux éléments d'un enregistrement se fait par un point '.' suivi par le nom du champ. Par exemple :

```
print_string ( { nom = "Dupont" ; prenom = "Jean" ; age = 33 }.nom )
```

va imprimer sur l'écran la chaîne `"Dupont"`.

Les enregistrements peuvent aussi être modifiables en place (à l'image de tableaux) ou peuvent être sujet à des opérations plus complexes comme l'ajout d'un nouveau champ ou la concaténation de deux enregistrements. Nous nous contenterons ici d'enregistrements statiques, à valeurs non-mutables. Les enregistrements ne servent en pratique dans notre langage que de prétexte minimal au sous-typage.

Sous-typage

Le sous-typage consiste à considérer certains types comme des restrictions (ou des extensions) de certains autres, via une relation d'ordre. On s'autorise alors, à chaque fois

qu'une valeur d'un certain type est attendue, à donner à la place une valeur d'un type plus petit. Cela correspond de fait à une forme de polymorphisme et enrichit considérablement l'expressivité d'un langage.

Lorsqu'un type est plus grand qu'un autre dans la relation de sous-typage, on pourra le qualifier de « plus général », de « moins précis » ou de « super-type ». Symétriquement, on peut parler du type plus petit comme d'un type « plus précis » ou « sous-type ».

Le sous-typage peut être explicite ou implicite, c'est-à-dire qu'il peut y avoir des annotations à chaque fois que le sous-typage doit être utilisé, ou alors que le système de type est capable d'inférer les éventuelles conversions de typage d'une expression. La présence d'un opérateur de « cast » est très souvent l'expression d'un sous-typage explicite.

On peut relier les types enregistrement par une relation de sous-typage. Celle-ci peut se définir de plusieurs façons. Nous avons choisi ici la façon suivante : un type enregistrement est un sous-type d'un autre si ses étiquettes comprennent celles du premier et que les types des étiquettes communes sont identiques. C'est le sous-typage dit « en largeur ». Nous notons le sous-typage $T <: T'$ lorsque T est un sous-type de T'.

$$\frac{}{\vdash \{l_1 : T_1, ..., l_j : T_j, ..., l_k : T_k\} <: \{l_1 : T_1, ..., l_j : T_j\}} \text{ Largeur}$$

On aurait pu aussi demander à ce que les types correspondants soient dans la même relation de sous-typage. C'est alors le sous-typage dit « en profondeur ».

$$\frac{\vdash T_i <: T'_i \quad 1 \leqslant i \leqslant k}{\vdash \{l_1 : T_1, ..., l_k : T_k\} <: \{l_1 : T'_1, ..., l_k : T'_k\}} \text{ Profondeur}$$

Exemple 2.6 (Sous-typage et enregistrements) Voici un exemple de deux types enregistrements qui sont sous-types l'un de l'autre. Le sous-typage est ici, comme dans toute la suite, simplement en largeur.

```
{ nom : string ; prenom : string ; age : int } <: { nom : string }
```

Le type enregistrement de gauche possède des champs supplémentaires par rapport au type enregistrement de droite, tandis que le champ commun `nom` possède le même type de chaque côté : les deux types sont donc en relation de sous-typage. Cela signifie que toutes les fois où une expression de type `{nom: string}` est requise, il est possible de fournir une expression de type `{nom: string ; prenom: string ; age: int}` à la place.

Pour illustrer ce fait, nous présentons ci-dessous la définition d'un enregistrement de type `{nom: string ; prenom: string ; age: int}`, puis d'une fonction qui affiche le champ `nom` d'un enregistrement dont le type doit être un sous-type de `{nom: string}`. Nous appliquons enfin l'un à l'autre, le résultat devant être l'affichage de `"Dupont"` à l'écran.

```
let jean = { nom = "Dupont" ; prenom = "Jean" ; age = 33 } in
let donne_nom (x : {nom : string}) = print_string (x.nom) in
donne_nom jean
```

Les langages objets (Java [44], Ocaml [48], ...) possèdent très souvent une notion de sous-typage ou une notion similaire. Les langages traitant nativement des schémas XML l'utilisent aussi (Cduce [3], Xtatic [33]).

On peut cependant noter qu'en Ocaml, le langage dont on a approximativement suivi la syntaxe jusqu'ici, il n'y a pas de sous-typage entre les enregistrements simples. Ocaml possède cependant des enregistrements extensibles (les objets) qui utilisent une forme de polymorphisme, le ρ-polymorphisme [63], et sont en relation de sous-typage.[1]

Abstraction partielle

Le sous-typage peut permettre la déclaration de types partiellement abstraits, c'est-à-dire de types pour lesquels il est donné une information partielle permettant sa déconstruction.

Concrètement, la déclaration se fait dans la signature en associant au type abstrait un type plus général que son implémentation réelle. Les autres modules pourront ainsi supposer que le type abstrait est un sous-type du type déclaré. Cela a particulièrement une utilité pour les objets qui peuvent ainsi ne montrer qu'un certain nombre de leurs méthodes.

Exemple 2.7 (Abstraction partielle) Donnons un exemple à l'aide d'un module schématisant la situation bancaire d'un contribuable malhonnête. La structure propose simplement un type enregistrement donnant les valeurs déposées en France et en Suisse. La signature va abstraire partiellement ce type implémentation en ne donnant comme information que la présence d'un champ `france`.

```
module Compte =
  struct
    type t = {france : int ; suisse : int }
    let  v = {france = 1 ; suisse = 10000000 }
  end :
  sig
    type t <: { france : int }
    val  v : t
  end
```

Les autres programmes ne peuvent donc pas connaître l'existence du champ `suisse` et ne peuvent accéder directement qu'au champ `france`. On peut imaginer (elle ne sont pas écrites dans cet exemple) des fonctions d'accès à la partie suisse qui vérifient un certain mot de passe avant d'effectuer leur tâche.

Cette notion d'abstraction partielle a été présentée notamment par Cardelli et Wegner [14] et a été implémentée dans certains langages objets, comme Modula-3 [15] ou Scala [56]. Ocaml en possède une version pour les objets et variants déclarée à l'aide du mot clé `private` [34].

Problème et solution proposée

Les systèmes distribués concernent non seulement du code qui est exécuté sur différentes machines, mais qui est aussi modifié et déployé de façon asynchrone. Il devient alors important d'envisager, lors de la programmation d'un tel système, que les messages échangés vont faire référence à des versions différentes des types de données.

Le langage Acute prend en compte deux aspects du phénomène de changement de version. Tout d'abord, le changement de version d'une bibliothèque est permis via une réédition des liens que l'utilisateur peut contrôler, notamment pour décider si les bibliothèques présentes sur un site distant satisfont certaines contraintes de version. D'autre

[1] Ocaml utilise la notation $T :> T'$ lorsque T est un sous-type de T'.

part, Acute permet de sérialiser une valeur abstraite d'un certain type et de la désérialiser sous un autre type (correspondant à une version différente du même module). Cette fonctionnalité passe par une conversion explicite qui indique dans quels cas les égalités de types sont acceptables. Une difficulté de cette approche est qu'elle est forcément bidirectionnelle, ce qui restreint les contextes possibles d'utilisation.

Nous proposons dans ce chapitre de rendre plus précise la relation de compatibilité entre types qui est vérifiée au moment de la désérialisation grâce au sous-typage.

Le sous-typage que nous ajoutons peut provenir de trois sources différentes. Premièrement, nous ajoutons des enregistrements statiques, c'est-à-dire la structure la plus simple permettant un sous-typage structurel. Deuxièmement, pour modéliser le changement de version d'un module, nous introduisons un sous-typage explicite entre types abstraits. Troisièmement, nous permettons la déclaration de types abstraits partiels.

Pour définir une sémantique opérationnelle tenant compte de ces ajouts conséquents, des changement importants ont été réalisés par rapport aux travaux antérieurs de Leifer et coll. [46]. Notre deuxième ajout, le sous-typage entre types abstraits, nécessite notamment l'introduction d'une relation d'ordre partielle entre empreintes de modules, calculée au moment de la compilation et propagée par le réseau au cours de l'exécution. Les types abstraits partiels requièrent enfin un sous-typage explicite et changent le traitement sémantique des valeurs abstraites pour garantir la sûreté du langage. Notre théorème d'effacement montre finalement que le sous-typage explicite ainsi que les crochets colorés peuvent être éliminés sans dommage lors de la compilation.

Plan

L'objectif de notre travail dans ce chapitre de la thèse est d'obtenir un calcul typé préservant les abstractions et comprenant un système simple de module avec types abstraits, primitives de sérialisation et de communication, empreintes et crochets colorés. Notre exposé s'articulera en trois étapes.

Tout d'abord, la section 2.2 (page 24) présente HAT+S, un calcul dérivé de celui présenté par Leifer et coll. [46], avec deux changements principaux : l'ajout d'enregistrements et de sous-typage structurel, et le remplacement du test d'égalité effectué au moment de la désérialisation par un test de sous-typage. Cette section servira de base pour les ajouts successifs.

Dans la section 2.3 (page 33), nous ajoutons des déclarations de compatibilité entre modules qui induisent une relation de sous-typage entre types abstraits. Cet ajout conduit à changer la sémantique opérationnelle pour permettre la propagation de ces déclarations au moment des communications.

La section 2.4 (page 37) conclut nos extensions par l'ajout dans les signatures de modules de la possibilité de déclarer des types partiellement abstraits. La sémantique opérationnelle doit alors s'accommoder de sous-typage explicite et de crochets colorés additifs.

Nous énonçons dans la section 2.5 (page 42) les principaux théorèmes caractérisant le comportement du langage final (baptisé HATS). La définition complète de ce langage est en annexe A.

La section 2.6 (page 43) discute enfin des conditions d'implémentation de HATS et énonce un théorème d'effacement.

2.2 Une base : HAT+S

Dans cette section nous décrivons le cœur d'un λ-calcul distribué avec types abstraits, enregistrements et sous-typage. Il ne s'agit que d'une variation marginale du calcul utilisé par Leifer et coll.[46].

L'intérêt principal de ce calcul par rapport à un langage bien plus proche de ML réside dans sa concision : il nous permet de décrire simplement les sémantiques statiques et dynamiques du langage. Par souci de clarté, nos exemples restent dans une syntaxe proche de celle d'Ocaml. Par la suite, nous utiliserons ce calcul comme point de départ pour l'ajout de nouveautés relatives aux types abstraits et au sous-typage.

2.2.1 Syntaxe

Nous décrivons tout d'abord la syntaxe de HAT+S, notre calcul de départ. Les éléments de syntaxe situés en dessous d'une ligne pointillée ne peuvent être écrits par le programmeur. Ils sont produits au moment de la compilation ou de l'exécution du programme. Les couleurs ou les empreintes font partie de cette catégorie, et n'entraînent de fait aucune charge de travail spécifique pour le programmeur.

Nous utiliserons les identifiants x, X et U pour désigner respectivement des variables d'expressions, de types et de modules.

Réseau

Un réseau est constitué de machines distinctes capables de communiquer entre elles.

$n ::= \mathbf{0}$	réseau vide
$\mid m$	machine
$\mid (n \mid n)$	deux réseaux mis en communication

La syntaxe $(m_1 | m_2)$ correspond à la mise en parallèle de deux machines que nous écrivions (*Programme1* || *Programme2*) dans les exemples plus haut.

Machines (programmes complets)

Une machine est une suite de définitions de modules suivie par une expression qui constitue le programme proprement dit. Chaque module est défini par un identifiant N utilisé dans l'empreinte, un nom de variable de module U (susceptible d'α-renommage), une structure M et une signature S.

$m ::= e$	expression
$\mid$ module $NU = M : S$ in m	déclaration de module (U lié dans m)

Dans les exemples, aucune distinction n'est faite entre les noms N et U des modules.

Modules (structures et signatures)

Pour simplifier nos preuves et nos raisonnements, nous limitons les modules à la déclaration d'un seul type et d'une seule valeur. Ainsi une structure M ne va définir qu'un type T et une valeur $v^\bullet$ et une signature S ne va donner qu'une sorte K (*kind*), celle du type déclaré par la structure, et le type T de la valeur. Retrouver la généralité des modules permettant la déclaration de plusieurs éléments requiert un certain travail, principalement à cause des dépendances entre types ou valeurs (cf. Acute [65]). Nous utiliserons cependant dans nos exemples, par souci de clarté, des modules à plusieurs champs pour lesquels nous nous

sommes assurés que le codage se passait sans difficulté.

$M ::= [T, v^\bullet]$	structure ($v^\bullet$ est une valeur, c'est-à-dire une classe particulière d'expression : voir ci-dessous)
$S ::= [X : K, T]$	signature (X lié dans T)

Une structure $[T, v^\bullet]$ correspond à la déclaration struct type t = T val v = $v^\bullet$ end. Les signatures $[X : K, T]$ (où la variable de type est X) sont écrites dans les exemples sig type t K val v : T end (où la variable de type est habituellement t). Nous décrivons les sortes K plus bas.

Types

Les types de notre langage comportent un cœur de types usuels à la ML : unit, string, flèche $\rightarrow$, produit *, enregistrement. On y ajoute un type $\top$ dont tous les autres types sont des sous-types. $\top$ nous permet de déclarer des types abstraits et de modéliser certains jugements de correction. Nous notons de plus U.type la référence à la déclaration de type du module correspondant à la variable U. Le type abstrait h.type est lui un type qui fait référence au module dont l'empreinte est h. Rappelons que les éléments situés sous la ligne pointillée ne peuvent être directement écrits par le programmeur dans son programme.

$T ::=$ unit | int | bytes		types de base
| X | $T \rightarrow T$		variable, fonction
| $T * ... * T$		produit
| $\{l_1 : T; ...; l_j : T\}$		enregistrement ($j > 0$)
| U.type		type déclaré par le module U
| $\top$		type le plus général
.....		
| h.type		type empreinte

Sortes (kinds)

La sorte $\mathbf{Le}(T)$ dans une signature permet de modéliser l'abstraction partielle ou totale (en prenant $T = \top$). La sorte singleton $\mathbf{Eq}(T)$ est utilisée pour les types manifestes, c'est-à-dire les types dont l'implémentation est exposée dans la signature. La sorte **Le** est dérivée des travaux de Cardelli et Wegner [14]. La sorte singleton **Eq** provient, quant à elle, des résultats de Leroy [47], Harper et Lillibridge [37], Stone et Harper [66].

$K ::= \mathbf{Le}(T)$	sorte des sous-types de T
| $\mathbf{Eq}(T)$	sorte des types égaux à T

Plus concrètement, la signature abstraite de nos exemples sig type t val v : T end est écrite $[X : \mathbf{Le}(\top), T]$ dans notre calcul. Une signature partiellement abstraite sig type t <: T' val v : T end est, elle, écrite $[X : \mathbf{Le}(T'), T]$. Les types manifestes déclarés par sig type t = T' val v : T end utilisent la sorte $\mathbf{Eq}(T')$ dans leur représentation $[X : \mathbf{Eq}(T'), T]$.

Empreintes et Couleurs

L'empreinte d'un module a lieu lorsque celui-ci déclare un type au moins partiellement abstrait. Nous ne précisons pas ici la méthode utilisée pour l'empreinte (cf. Acute [65]), mais une empreinte $\mathbf{hash}(N, M : [X : \mathbf{Le}(T), T])$ doit porter sur le nom déclaré du module N, sa structure M et sa signature $[X : \mathbf{Le}(T), T]$. Le résultat de l'empreinte se voit adjoint,

en clair, le type concret défini dans la structure ainsi que la sorte qui lui correspond. Ces deux informations sont utiles pour l'élimination des crochets colorés et pour le typage. Nous rediscuterons de cet élément dans la section 2.6.

Une couleur c est définie comme une simple collection d'empreintes de modules. Ces modules correspondront aux modules pour lesquels on pourra associer au type abstrait son implémentation réelle.

$h ::= \mathbf{hash}(N, M : [X : \mathbf{Le}(T), T])$	empreinte
$c ::= \bullet$	couleur vide
$\mid \quad \{h\}$	couleur avec une seule empreinte
$\mid \quad c \cup c$	union de couleurs

Expressions

Comme le langage a pour base un cœur de λ-calcul avec produits et enregistrements, on retrouve ces éléments de façon classique. De façon similaire à $U.\texttt{type}$, l'expression $U.\texttt{term}$ renvoie à la valeur définie par le module que la variable U représente. Les opérations de sérialisation et de désérialisation sont annotées par le type des données envoyées ou attendues. On utilise la syntaxe $\mathbf{marshalled}\,(e : T)$ pour représenter l'objet intermédiaire qu'est une valeur sérialisée. Enfin, les crochets colorés $[e]_c^T$ délimitent les espaces dans lesquels on peut associer un module au résultat de son empreinte : c'est uniquement à l'intérieur des crochets que les valeurs abstraites dont les empreintes sont dans c peuvent être révélées. Le type T représente alors le type qu'a l'expression e vue de l'extérieur.

$e ::= () \mid 0 \mid 1 \mid \ldots$	unit, entiers
$\mid \quad (e, ..., e)$	n-uplet
$\mid \quad \mathbf{proj}_i\, e$	projection
$\mid \quad \{l_1 = e, ..., l_j = e\}$	enregistrement
$\mid \quad e.l_i$	accès à un champ
$\mid \quad x$	variable
$\mid \quad \lambda x : T.e$	fonction (x lié dans e)
$\mid \quad e\, e$	application
$\mid \quad U.\texttt{term}$	valeur déclarée par le module U
$\mid \quad \mathbf{mar}\,(e : T)$	sérialisation
$\mid \quad \mathbf{unmar}\, e : T$	désérialisation
$\mid \quad !\, e$	envoi
$\mid \quad ?$	réception
.....	
$\mid \quad \mathbf{marshalled}\,(e : T)$	résultat de la sérialisation
$\mid \quad \mathbf{Unmarfailure}^T$	erreur causée par **unmar** lorsque les types ne correspondent pas
$\mid \quad [e]_c^T$	crochets colorés

L'erreur $\mathbf{Unmarfailure}^T$ est l'erreur légitime levée lorsque la désérialisation échoue.

Valeurs

Parmi les expressions on distingue les valeurs, notées v^c, qui correspondent aux expressions que l'on ne peut pas évaluer plus avant avec la sémantique. Elles sont annotées par une couleur c qui donne la liste des modules connus et ainsi, par contraste, les types abstraits qui ne peuvent être révélés. La définition des valeurs permet principalement d'assurer le déterminisme de notre sémantique opérationnelle : on comprendra donc mieux les détails

de cette définition après la lecture de la section 2.2.3.

$v^{c_0} ::= ()$	$\overline{n}$	unit, entiers
	$(v_1^{c_0}, ..., v_j^{c_0})$	produit ($j \geqslant 2$)
	$\{l_1 = v_1^{c_0}, ..., l_j = v_j^{c_0}\}$	enregistrement ($j \geqslant 1$)
	$\lambda x : T.e$	fonction (x lié dans e)
	$\mathbf{marshalled}\,(v^\bullet : T)$	valeurs sérialisées
	$[v^{c_1}]_{c_1}^{h_1.\texttt{type}}$	crochets colorés où $h_1 \notin c_0 \cap c_1$

2.2.2 Typage

Jugements

Nous utilisons des jugements $\vdash$ pour exprimer les propriétés de typage. Ceux-ci sont accompagnés d'un environnement, désigné par la méta-variable E, qui peut lier des variables d'expression, de type ou de module. Les jugements sont par ailleurs annotés par une couleur qui permet d'exprimer la connaissance que l'on a de l'implémentation réelle des types abstraits.

$\vdash h\,\text{ok}$	Correction d'une empreinte
$\vdash c\,\text{ok}$	Correction de couleur
$E \vdash_c \text{ok}$	Correction d'environnement
$E \vdash_c K\,\text{ok}$	Correction de sorte
$E \vdash_c K == K'$	Equivalence de sorte
$E \vdash_c K <: K'$	Sous-sortage
$E \vdash_c T : K$	Sortage
$E \vdash_c T == T'$	Equivalence de type
$E \vdash_c T <: T'$	Sous-typage
$E \vdash_c S\,\text{ok}$	Correction de signature
$E \vdash_c S == S'$	Equivalence de signatures
$E \vdash_c S <: S'$	Sous-signaturage
$E \vdash_c M : S$	Signature de module
$E \vdash_c U : S$	Signature d'une variable de module
$E \vdash_c e : T$	Typage d'expression
$E \vdash_\bullet m : T$	Typage de machine
$\vdash n\,\text{ok}$	Correction de réseau

FIG. 2.1 – Forme des différents jugements utilisés

Le système de types définit des jugements pour la correction des empreintes, couleurs, environnements, sortes, signatures et réseaux, pour le sous-typage, le sous-sortage (*sub-kinding*), le sous-signaturage (*sub-signaturing*) ainsi que les trois relations d'équivalence associées. Enfin, il existe des jugements pour exprimer le sortage d'un type (*kinding*), le typage d'une expression ou d'une machine et le signaturage (*signaturing*) d'un module ou d'une variable de module. La forme des différents jugements que notre système de types va utiliser est détaillée dans la figure 2.1.

Nous allons donner quelques explications sur quelques unes des règles d'inférence qui permettent de prouver ces jugements. Le lecteur pourra se référer aux annexes pour avoir le système de types définitif.

Correction

La correction des éléments d'un jugement est essentielle à sa prouvabilité. Voici donc quelques règles qui permettront de prouver la correction d'une empreinte, d'une couleur, d'un environnement, d'une sorte, d'un type, d'une signature et d'un réseau.

La correction d'une couleur est donnée par la correction des empreintes qui la composent. La correction d'un environnement se fait en décomposant celui-ci élément par élément. L'environnement vide est noté **nil**.

$$\frac{\vdash h\,\mathrm{ok}}{\vdash \{h\}\,\mathrm{ok}} \qquad \frac{\vdash c_0\,\mathrm{ok} \quad \vdash c_1\,\mathrm{ok}}{\vdash c_0 \cup c_1\,\mathrm{ok}} \qquad \vdash \bullet\,\mathrm{ok}$$

$$\frac{\vdash c\,\mathrm{ok}}{\mathbf{nil} \vdash_c \mathrm{ok}} \qquad \frac{E \vdash_c T : \mathbf{Le}(\top) \qquad x \notin \mathrm{dom}\, E}{E, x : T \vdash_c \mathrm{ok}}$$

$$\frac{E \vdash_c K\,\mathrm{ok} \quad X \notin \mathrm{dom}\, E}{E, X : K \vdash_c \mathrm{ok}} \qquad \frac{E \vdash_c S\,\mathrm{ok} \quad U \notin \mathrm{dom}\, E}{E, U : S \vdash_c \mathrm{ok}}$$

La correction des sortes dépend de la correction des types qui les composent. Notons que la correction d'un type T s'exprime par un jugement de sortage $E \vdash_c T : \mathbf{Le}(\top)$.

$$\frac{E \vdash_c T : \mathbf{Le}(\top)}{E \vdash_c \mathbf{Le}(T)\,\mathrm{ok}} \qquad \frac{E \vdash_c T : \mathbf{Le}(\top)}{E \vdash_c \mathbf{Eq}(T)\,\mathrm{ok}}$$

La correction d'un type T s'obtient en prouvant qu'il est un sous-type de $\top$, puis en utilisant la règle permettant de passer du sous-typage au sortage (voir plus bas).

Nous concluons par la correction d'une signature (correction de la sorte et du type) et d'un réseau (correction des machines qui le composent).

$$\frac{E, X : K \vdash_c T : \mathbf{Le}(\top)}{E \vdash_c [X : K, T]\,\mathrm{ok}}$$

$$\frac{\vdash n_i\,\mathrm{ok} \quad i = 1, 2}{\vdash n_1 \mid n_2\,\mathrm{ok}} \qquad \vdash \mathbf{0}\,\mathrm{ok} \qquad \frac{\mathbf{nil} \vdash_\bullet m : \mathrm{unit}}{\vdash m\,\mathrm{ok}}$$

Equivalences de sortes et de types

Ces équivalences naissent tout d'abord de la révélation des types abstraits. Celle-ci ne peut se faire que lorsque la couleur du jugement contient l'empreinte correspondante. On a donc un ensemble différent d'équivalences qui sont prouvables suivant la couleur du jugement, celle-ci changeant à chaque fois que l'on veut émettre un jugement concernant l'intérieur de crochets colorés.

$$\frac{E \vdash_c \mathrm{ok}}{E \vdash_c h.\mathtt{type} == T} \qquad \text{où } h = \mathbf{hash}(N, [T, v^\bullet] : [X : \mathbf{Le}(T'), T'']) \in c$$

Les équivalences peuvent aussi venir de la déclaration d'un type manifeste : sa sorte dans la signature est alors $\mathbf{Eq}(T)$ pour un certain T, puis nous utilisons la règle de passage entre la sorte et l'équivalence.

$$\frac{E \vdash_c U : [X : K, T]}{E \vdash_c U.\texttt{type} : K} \qquad \frac{E \vdash_c T : \mathbf{Eq}(T')}{E \vdash_c T == T'}$$

Toutes les autres équivalences de types, puis de sortes sont immédiates (car structurelles) et ne sont pas reproduites ici.

Sous-sortage, sous-typage, sous-signaturage

Ces relations prennent leur naissance à trois moments distincts. Tout d'abord, le sous-typage peut être le résultat de la déclaration d'abstraction partielle. Cela se fait via la règle donnant la sorte associée à un type abstrait, et la règle permettant le passage du sortage vers le sous-typage.

$$\frac{E \vdash_c \text{ ok} \quad \vdash h \,\text{ok}}{E \vdash_c h.\texttt{type} : K} \quad \text{où } h = \mathbf{hash}(N, [T, v^\bullet] : [X : K, T']) \qquad \frac{E \vdash_c T : \mathbf{Le}(T')}{E \vdash_c T <: T'}$$

Le sous-typage peut aussi provenir de sa construction explicite à partir des enregistrements et des constructeurs du langage. On notera la contravariance du premier élément du type $\rightarrow$ et le sous-typage en profondeur appliqué aux n-uplets.

$$\frac{E \vdash_c T_i : \mathbf{Le}(\top) \quad 1 \leqslant i \leqslant k}{E \vdash_c \{l_1 : T_1, ..., l_j : T_j, ..., l_k : T_k\} <: \{l_1 : T_1, ..., l_j : T_j\}}$$

$$\frac{E \vdash_c T_0' <: T_0 \quad E \vdash_c T_1 <: T_1'}{E \vdash_c T_0 \rightarrow T_1 <: T_0' \rightarrow T_1'} \qquad \frac{E \vdash_c T_i <: T_i' \quad 1 \leqslant i \leqslant j}{E \vdash_c T_1 * ... * T_j <: T_1' * ... * T_j'}$$

Enfin, de manière moins marquante, le sous-typage peut provenir de l'équivalence des types.

$$\frac{E \vdash_c T == T'}{E \vdash_c T <: T'}$$

Les règles de sous-sortages sont assez évidentes, et le lecteur pourra se référer à l'annexe A.2.9. Nous concluons par la règle de sous-signaturage.

$$\frac{E \vdash_c K <: K' \quad E, X : K \vdash_c T <: T'}{E \vdash_c [X : K, T] <: [X : K', T']}$$

Typage des expressions

Les règles de typage comportent tout d'abord des éléments permettant d'utiliser le sous-typage et les équivalences de type. Ces règles sont suivies par celles qui permettent de typer une variable ou une valeur de module.

$$\frac{E \vdash_c e : T \quad E \vdash_c T <: T'}{E \vdash_c e : T'} \qquad \frac{E \vdash_c e : T \quad E \vdash_c T == T'}{E \vdash_c e : T'}$$

$$\frac{E, x : T, E' \vdash_c \text{ ok}}{E, x : T, E' \vdash_c x : T} \qquad \frac{E \vdash_c U : [X : K, T] \quad E \vdash_c T : \mathbf{Le}(\top)}{E \vdash_c U.\texttt{term} : T}$$

Dans la quatrième règle ci-dessus, on impose que le type T ne fasse pas appel à la variable X, mais plutôt à $U.\texttt{type}$. Cette contrainte est résolue par une règle de signaturage des variables de module que nous verrons plus loin dans la section 2.2.2.

Nous avons bien entendu les règles habituelles traitant des expressions du λ-calcul et des différents constructeurs et destructeurs. Le lecteur peut les trouver en annexe.

Maintenant viennent les jugements de typage de la sérialisation. Nous voyons ici la particularité de **marshalled** $(e : T)$ dont l'expression e doit être typable par T dans la couleur vide pour que la comparaison des type puisse avoir lieu lors de la désérialisation sur une machine quelconque.

$$\frac{E \vdash_c e : T}{E \vdash_c \mathbf{mar}\,(e : T) : \text{bytes}} \qquad \frac{E \vdash_c \text{ok} \quad \mathbf{nil} \vdash_\bullet e : T}{E \vdash_c \mathbf{marshalled}\,(e : T) : \text{bytes}}$$

$$\frac{E \vdash_c T : \mathbf{Le}(\top) \quad E \vdash_c e : \text{bytes}}{E \vdash_c (\mathbf{unmar}\, e : T) : T} \qquad \frac{E \vdash_c T : \mathbf{Le}(\top)}{E \vdash_c \mathbf{Unmarfailure}^T : T}$$

La désérialisation peut produire une erreur lorsque le type d'envoi et celui attendu ne correspondent pas. L'expression $\mathbf{Unmarfailure}^T$ remplace alors la valeur désérialisée et bloque ainsi le reste de la réduction (dernière règle ci-dessus).

Il nous reste enfin les crochets colorés à typer.

$$\frac{E \vdash_c T : \mathbf{Le}(\top) \quad E \vdash_{c'} e : T}{E \vdash_c [e]^T_{c'} : T}$$

Notons que e est typé dans la couleur c'. Les crochets forment ainsi la frontière de l'abstraction : à l'intérieur des crochets, une couleur différente est à l'œuvre, et celle-ci permet alors, nous l'avons vu dans la section 2.2.2, d'avoir de nouvelles équivalences pour former des jugements, tout en perdant celles qui étaient disponibles à l'extérieur.

Sortage

Concernant le sortage, nous avons déjà présenté les règles les plus importantes. Nous les récapitulons ci-dessous.

$$\frac{E \vdash_c T : K \quad E \vdash_c K <: K'}{E \vdash_c T : K'} \qquad \frac{E \vdash_c T == T'}{E \vdash_c T : \mathbf{Eq}(T')} \qquad \frac{E \vdash_c T <: T'}{E \vdash_c T : \mathbf{Le}(T')}$$

$$\frac{E, X : K, E' \vdash_c \text{ok}}{E, X : K, E' \vdash_c X : K} \qquad \frac{E \vdash_c U : [X : K, T]}{E \vdash_c U.\texttt{type} : K}$$

$$\frac{E \vdash_c \text{ok} \qquad \vdash h\,\text{ok}}{E \vdash_c h.\texttt{type} : K} \quad \text{où } h = \mathbf{hash}(N, [T, v^\bullet] : [X : K, T'])$$

Modules

Pour typer une machine, il est tout d'abord nécessaire de pouvoir vérifier l'adéquation d'une structure $[T, v^c]$ avec une signature $[X : K, T']$. Il s'agit en fait de s'assurer de la correspondance entre T et K d'une part, et entre v^c et T' d'autre part.

$$\frac{\begin{array}{l}E \vdash_c T : K \\ E, X : K \vdash_c T' : \mathbf{Le}(\top) \\ E, X : \mathbf{Eq}(T) \vdash_c T'' <: T' \\ E \vdash_c v^c : T''\end{array}}{E \vdash_c [T, v^c] : [X : K, T']}$$

Diverses règles donnent d'autres manières de donner une signature à un module ou à une variable de module. On peut les retrouver en annexe. Une règle a cependant un comportement intéressant puisqu'elle permet d'obtenir l'équivalence entre X et $U.\texttt{type}$ lorsque l'on a pu associer U à $[X : K, T]$.

$$\frac{E \vdash_c U : [X : K, T]}{E \vdash_c U : [X : \mathbf{Eq}(U.\texttt{type}), T]}$$

Machines

Il nous reste enfin à donner un type à une machine, c'est-à-dire à une succession de déclarations de modules qui précède une expression. Il s'agit en fait de vérifier l'adéquation de la structure avec la signature et d'enrichir l'environnement avec la nouvelle déclaration de module. Le cas de l'expression est, quant à lui, trivial (la prémisse étant un jugement de typage d'expression et la conclusion un jugement de typage de machine). On peut noter aussi que ces jugements se font dans la couleur vide.

$$\frac{\begin{array}{c}E \vdash_\bullet T : \mathbf{Le}(\top) \\ E \vdash_\bullet [T_0, v^\bullet] : S \\ E, U : S \vdash_\bullet m : T\end{array}}{E \vdash_\bullet (\text{ module } NU = [T_0, v^\bullet] : S \text{ in } m) : T} \qquad \frac{E \vdash_\bullet e : T}{E \vdash_\bullet e : T}$$

2.2.3 Sémantique

La sémantique opérationnelle à petit pas est divisée en trois relations de réduction :
- $m \to_m m'$ pour la réduction des machines, c'est-à-dire la compilation ;
- $e \to_c e'$ et $n \to_n n'$ pour la réduction des expressions et des réseaux qui a lieu à l'exécution.

Nous allons détailler ces trois relations de réduction dans les deux sections suivantes, en séparant la phase de compilation de l'exécution.

Compilation

La compilation des modules diffère selon qu'ils déclarent un type manifeste ou qu'ils déclarent un type partiellement ou complètement abstrait.

Dans le cas d'un type manifeste, la réduction est simple : il suffit de substituer le type et la valeur par leur implémentation.

Si le type du module U est abstrait (au moins partiellement), il est nécessaire de prendre l'empreinte du module pour remplacer les références au type du module $U.\texttt{type}$ par le type empreinte qui, lui, sera clos. La partie terme du module utilise parfois l'équivalence entre le type abstrait et son implémentation réelle T : il faut donc enfermer ce terme dans des crochets colorés pour que l'information nécessaire lui soit disponible. Ces crochets délimitent les frontières de l'abstraction.

$$\begin{array}{ll} \text{module } NU = & [T, v^\bullet] : [X : \mathbf{Eq}(T''), T'] \text{ in } m \rightarrow_m \{U.\texttt{type} \leftarrow T'', U.\texttt{term} \leftarrow v^\bullet\} m \\ \\ \text{module } NU = & [T, v^\bullet] : [X : \mathbf{Le}(T''), T'] \text{ in } m \rightarrow_m \\ & \quad \{U.\texttt{type} \leftarrow h.\texttt{type}, U.\texttt{term} \leftarrow [v^\bullet]_{\{h\}}^{\{X \leftarrow h.\texttt{type}\} T'}\} m \\ & \text{où} \quad h = \mathbf{hash}(N, [T, v^\bullet] : [X : \mathbf{Le}(T''), T']) \end{array}$$

Exécution

Les crochets colorés servent à protéger une valeur dont le type est abstrait d'un accès intempestif. Or le caractère licite ou non d'un tel accès dépend de la connaissance que l'on a des modules, connaissance qui change à chaque fois que l'on traverse de nouveaux crochets colorés. Pour refléter ces changements, il est nécessaire d'annoter la relation de réduction par la couleur courante de la réduction. C'est pourquoi la relation réduction des expressions est notée $\rightarrow_c$.

Les réductions des expressions seront discutées de façon plus approfondie dans la section 2.4. Nous n'en présenterons donc ici que quelques unes.

Les valeurs v^c sont conçues pour être les expressions irréductibles par ces règles. Leur utilisation dans les définitions qui suivent permet d'avoir une sémantique déterministe.

λ-calcul

Dans le cadre de notre λ-calcul en appel par valeur, certaines des règles sont simples (comme la projection) mais la β-réduction ne l'est pas puisqu'il lui faut ajouter des crochets interdisant à l'argument d'être placé dans une couleur différente de la couleur ambiante.

$$\begin{array}{rcll} \mathbf{proj}_i\,(v_1^c, ..., v_j^c) & \rightarrow_c & v_i^c & \text{si } 1 \leqslant i \leqslant j \\ \\ \{l_1 = v_1^c, ..., l_j = v_j^c\}.l_i & \rightarrow_c & v_i^c & \text{si } 1 \leqslant i \leqslant j \\ \\ (\lambda x : T.e)\, v^c & \rightarrow_c & \{x \leftarrow [v^c]_c^T\} e & \end{array}$$

La sérialisation, elle aussi, fait intervenir des crochets colorés, puisque la valeur actuellement envoyée doit être typable dans la couleur vide. On lui adjoint donc des crochets portant la couleur requise. La désérialisation teste, elle, la correspondance entre les types envoyés et attendus. Cette correspondance est ici donnée par la relation de sous-typage.

$$\mathbf{mar}\,(v^c : T) \rightarrow_c \mathbf{marshalled}\,([v^c]_c^T : T)$$

$$\mathbf{unmar}\,(\mathbf{marshalled}\,(v^\bullet : T) : T') \rightarrow_c \begin{cases} v^\bullet & \text{si } \mathbf{nil} \vdash_c T <: T' \\ \mathbf{Unmarfailure}^{T'} & \text{sinon} \end{cases}$$

Crochets colorés

Comment se comportent les crochets colorés dans la sémantique ? La stratégie, lorsque les crochets colorés ne protègent pas un type abstrait, est de les pousser vers les feuilles des arbres syntaxiques pour qu'ils ne puissent pas interférer avec les interactions entre constructeurs et destructeurs.

$$
\begin{array}{rcl}
[()]^{\text{unit}}_{c'} & \to_c & () \\
[(v_1^{c'}, ..., v_j^{c'})]^{T_1 * ... * T_j}_{c'} & \to_c & ([v_1^{c'}]^{T_1}_{c'}, ..., [v_j^{c'}]^{T_j}_{c'}) \\
[\{l_1 = v_1^{c'}, .., l_j = v_j^{c'}\}]^{\{l_1:T_1,...,l_j:T_j\}}_{c'} & \to_c & \{l_1 = [v_1^{c'}]^{T_1}_{c'}, .., l_j = [v_j^{c'}]^{T_j}_{c'}\} \\
[\lambda x : T.e]^{T' \to T''}_{c'} & \to_c & (\lambda x : T'.[\{x \leftarrow [x]^{T}_{c'}\} e]^{T''}_{c'}) \\
[\textbf{marshalled}\,(v^\bullet : T)]^{\text{bytes}}_{c'} & \to_c & \textbf{marshalled}\,(v^\bullet : T)
\end{array}
$$

Il reste la question de la révélation d'un type abstrait. Voici la règle permettant de l'effectuer. Nous notons $\text{impl}\,(h)$ le type concret dont $h.\texttt{type}$ est l'abstraction. Remarquons la condition $h \in c \cap c'$: elle impose que l'empreinte h soit connue à l'intérieur et à l'extérieur des crochets colorés, ce qui illustre bien le fait que ces crochets soient devenus inutiles. Ils ne sont cependant pas supprimés, mais laissés à la disposition des règles ci-dessus qui permettent de les déconstruire à partir du type révélé T.

$$
[v^{c'}]^{h.\texttt{type}}_{c'} \to_c [v^{c'}]^{T}_{c'} \qquad \text{lorsque } h \in c \cap c' \text{ et } \text{impl}(h){=}T
$$

Nous finissons cette présentation des règles de réduction des expressions par la réduction au sein d'un contexte qui permet, notamment, de réduire à l'intérieur des crochets colorés. La définition exacte des contextes, notés $C^{c'}_c$, peut être trouvée en annexe A.1.

$$
\frac{e \to_c e'}{C^{c'}_c.e \to_{c'} C^{c'}_c.e'}
$$

Un certain nombre de ces règles seront modifiées dans les sections suivantes pour atteindre le système final.

Communication

Nous concluons par la règle essentielle de réduction des réseaux. Il s'agit, bien entendu, de la communication entre deux machines d'une valeur que le système de types impose comme une chaîne de caractères (le résultat de la sérialisation d'une valeur arbitraire).

$$
CC^\bullet_c.!\,v^c \mid CC^\bullet_{c'}.? \to_n CC^\bullet_c.() \mid CC^\bullet_{c'}.v^c
$$

Maintenant armés d'un langage de base, nous allons pouvoir étudier plus en détail les interactions entre le sous-typage, les abstractions totales et partielles, les empreintes et les couleurs.

2.3 Abstraction et sous-typage

Durant la vie d'un programme distribué, certaines de ses parties sont modifiées ou réécrites pour corriger un bogue ou ajouter de nouvelles fonctionnalités. Si le déploiement est décentralisé, les mises à jour ne s'effectuent pas nécessairement de façon simultanée sur toutes les machines. La désérialisation des valeurs abstraites a alors tous les risques d'échouer si l'on s'en tient à la définition initiale des types empreintes : ceux-ci sont différents dès que la définition d'un module dont ils dépendent est modifiée.

Cette façon de faire est logique car il n'est pas possible de déterminer automatiquement si les modifications effectuées sur un module vont préserver les invariants des types abstraits les utilisant. Il est cependant possible pour le programmeur de s'assurer des éventuelles compatibilités résultant d'une mise à jour.

Nous enrichissons donc notre calcul avec des déclarations de compatibilités entre types abstraits. Nous nous interrogeons sur les vérifications nécessaires à effectuer par le compilateur pour s'assurer de la sûreté de la déclaration (il faut notamment s'assurer que les types concrets sont compatibles).

2.3.1 Exemple initial

Avant d'introduire la nouvelle déclaration de compatibilité dans notre calcul, l'exemple suivant illustre un cas de compatibilité souhaitable entre types abstraits. Le nouveau mot-clé restricts y est en particulier présenté.

Exemple 2.8 (restricts) Définissons tout d'abord un module déclarant un type abstrait de compteur similaire à celui présenté dans l'exemple 2.4.

```
module CompteurA =
struct  type t = int
        let  v = { init = 0 ;
                   incr = fun (x:int) →  x+1;
                   valeur = fun (x:int) →  x }
end :
sig  type t
     val  v : { init  : t ;
                incr  : t  → t ;
                valeur : t  → int }
end
```

On peut décider d'écrire une nouvelle version de ce module en ajoutant une fonction `decr`.

```
module CompteurB restricts CompteurA =
struct  type t = int
        let  v  = { init = 0;
                    incr = fun (x:int) →  x+1;
                    decr = fun (x:int) →  x-1;
                    valeur = fun (x:int) →  x }
end :
sig  type t
     val  v : { init  : t ;
                incr  : t  → t ;
                decr  : t  → t ;
                valeur : t  → int }
end
```

L'invariant des valeurs de type `CompteurA.t` (être un entier positif) est plus strict que le nouvel invariant (être entier) des valeurs de type `CompteurB.t`.

Le programmeur peut alors garantir lui-même la compatibilité entre les deux types abstraits en écrivant restricts `CompteurA`. Le système de type en déduit alors que `CompteurA.t` est un sous-type de `CompteurB.t`.

Il est important de remarquer que la compatibilité inverse n'est pas souvent désirable : une valeur de type `CompteurB.t` peut être négative, ce qui n'est pas le cas des valeurs de type `CompteurA.t`. La compatibilité n'est intéressante que si l'invariant que le programmeur attend est maintenu dans le second module.

Le mot-clé restricts introduit ainsi une relation de sous-typage entre un type abstrait précédemment défini qui devient sous-type d'un nouveau type abstrait. Le mot-clé extends déclare symétriquement qu'un type abstrait existant est un super-type d'un nouveau type abstrait.

Si le programmeur souhaite une compatibilité dans les deux sens, les mots-clés restricts et extends peuvent être utilisés conjointement. Pour ne pas surcharger la présentation, nous nous limiterons dans ce chapitre au mot-clé restricts. L'annexe A contient les deux mots-clés et leur permet d'accepter des arguments multiples.

La précision du sous-typage constitue une amélioration par rapport au mécanisme utilisé dans Acute. Le mot-clé `with!` y permet en effet d'introduire une égalité de type entre deux types abstraits. Le programmeur a donc seulement le choix entre l'absence de compatibilité et la compatibilité bidirectionnelle. L'avantage de `with!` repose sur sa facilité d'implémentation : lorsqu'un nouveau type abstrait est déclaré comme égal à un précédent type abstrait, le compilateur peut lui assigner exactement la même empreinte. Le choix du sous-typage et de la compatibilité unidirectionnelle induit un coût d'implémentation plus grand en échange d'une plus grande flexibilité.

2.3.2 Syntaxe, typage et sémantique

Nous étendons la syntaxe de notre langage pour y inclure la nouvelle déclaration suivante (κ est une métavariable désignant le nom ou l'empreinte d'un module précédemment défini).

$m ::= e$	expression
$\mid$ module NU[restricts κ] $= M : S$ in m	déclaration de module

La déclaration module NU restricts κ permet d'enrichir la relation de sous-typage avec le nouveau couple $U.\texttt{type} <: \kappa.\texttt{type}$. Cette relation de sous-typage étendue (que nous notons H) doit être alors ajoutée en annotation sur les jugements de typage pour que ceux-ci puissent en faire usage. Les jugements de typage deviennent alors de la forme suivante $E \vdash_c^H e : T$.

La relation de sous-typage étendue intervient dans le système de type par la règle suivante :

$$\frac{E \vdash_c^H \text{ok} \qquad \kappa <: \kappa' \in H}{E \vdash_c^H \kappa.\texttt{type} <: \kappa'.\texttt{type}}$$

L'ajout de cette annotation ne modifie pas la plupart des règles de typage de HAT+S : les prémisses et la conclusion des règles sont annotées par la même métavariable H. Nous ne détaillons que les règles dont les changements sont les plus conséquents.

La première règle à être modifiée est la règle de typage des déclarations de module.

$$\frac{E \vdash^{H}_{\bullet} T : \mathbf{Le}(\top) \quad E \vdash^{H}_{\bullet} [T_0, v^\bullet] : S \quad E, U(T_0) : S \vdash^{H \cup \kappa <: U}_{\bullet} m : T}{E \vdash^{H}_{\bullet} (\text{ module } NU \text{ restricts } \kappa = [T_0, v^\bullet] : S \text{ in } m) : T}$$

Par rapport à la première version de cette règle présentée dans la section 2.2.2, deux changements ont été effectués. Tout d'abord, l'annotation de sous-typage H qui apparaît maintenant dans tous les jugements est enrichie par la nouvelle déclaration donnée par restricts κ. Cela signifie que le reste du programme (m) sera typé en utilisant cette déclaration. Deuxièmement, pour pouvoir vérifier la correction de $\kappa <: U$ et des déclarations potentiellement présentes dans m, il est nécessaire de se souvenir du type concret T_0 dans l'environnement : on y garde donc $U(T_0) : S$.

Remarquons que les modules déclarés dans m peuvent maintenant utiliser le sous-typage enrichi dans leur définition et que donc les empreintes de modules en dépendent pour leur correction. Notre règle de correction des empreintes suppose l'existence d'une telle relation sans utiliser une relation propagée dans les termes qui aurait le désavantage de polluer l'égalité et autres opérations sur les empreintes et couleurs.

Il est cependant à noter qu'au sein d'une implémentation où les types ont été effacés, il n'est pas nécessaire de vérifier la correction des empreintes.

Comme les jugements de typage prennent un nouveau paramètre H, les règles de réduction doivent s'adapter. La forme de la réduction des modules, $H, m \rightarrow_m H', m'$, et de la réduction des expressions, $H, e \rightarrow_c H', e'$, est ainsi modifiée.

La réduction des modules lorsque le type déclaré est abstrait prend maintenant en compte la déclaration restricts pour enrichir la relation de sous-typage courante.

$$H, \text{ module } NU \text{ restricts } h_0 = [T, v^\bullet] : [X : \mathbf{Le}(T''), T'] \text{ in } m \rightarrow_m \{U.\mathtt{type} \leftarrow h.\mathtt{type}, U.\mathtt{term} \leftarrow [v^\bullet]^{\{X \leftarrow h.\mathtt{type}\}T'}_{\{h\}}\} m$$

$$\text{avec} \quad \begin{cases} h & = \mathbf{hash}(N, [T, v^\bullet] : [X : \mathbf{Le}(T''), T']) \\ H' & = H \cup h_0 <: h \end{cases}$$

2.3.3 Diffusion de H par le réseau

Comme le typage dépend maintenant de la relation de sous-typage H, il est nécessaire que celle-ci soit propagée par le réseau pour que le typage réussisse à la réception d'une valeur distante. Nous annotons donc le constructeur **marshalled** avec la relation courante H (en plus de la couleur ambiante).

$e ::= \ldots \mid \mathbf{marshalled}_{c,H}(e : T)$ valeur sérialisée

La règle de typage et la règle de réduction deviennent les suivantes :

$$\frac{E \vdash^{H_0}_{c} \text{ ok} \qquad \mathbf{nil} \vdash^{H_1}_{c'} e : T}{E \vdash^{H_0}_{c} \mathbf{marshalled}_{c',H_1}(e : T) : \text{bytes}}$$

$$H, \mathbf{mar}\,(v^c : T) \rightarrow_c H, \mathbf{marshalled}_{c,H}([v^c]^T_c : T)$$

Au moment de désérialiser, la machine réceptrice doit maintenant enrichir sa propre relation de sous-typage avec la relation reçue de sorte que le terme reçu soit bien typé. La règle de réduction de la désérialisation devient donc :

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{c',H'}(v^\bullet : T) : T') \rightarrow_c \begin{cases} H \cup H', [v^\bullet]^T_{c'} & \text{si } \mathbf{nil} \vdash^{H \cup H'}_\bullet T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{sinon} \end{cases}$$

Remarquons que la condition $\mathbf{nil} \vdash^{H \cup H'}_\bullet T <: T'$ ne vérifie pas seulement la relation de sous-typage entre T et T', mais aussi la correction de $H \cup H'$ et donc de H', ce qui permet de s'assurer de la sûreté de la désérialisation.

Nous observons de plus que la machine réceptrice fait implicitement confiance à la machine émettrice pour propager une relation de sous-typage correcte. Même si les questions de sécurité sont orthogonales à notre objectif ici (la sûreté du langage), la machine réceptrice est libre de choisir la politique de confiance qu'elle souhaite : elle peut faire confiance par défaut et accepter toutes les déclarations reçues (comme dans la règle ci-dessus) ; il est aussi possible de définir une règle plus restrictive qui génère des exceptions $\mathbf{Unmarfailure}^{T'}$ supplémentaires lorsque la relation de sous-typage reçue ou la valeur sérialisée ne sont plus acceptables.

2.4 Couleurs et sous-typage

Nous ajoutons maintenant de l'expressivité au calcul en autorisant les types abstraits à être partiellement révélés. Nous permettons ainsi aux interfaces de modules de déclarer un super-type pour les types abstraits. Ce mécanisme, appelé *types partiellement abstraits* ou *types existentiels bornés*, a été étudié d'un point de vue théorique par Broy et Wirsing [11], puis par Cardelli et coll.[14, 62], et enfin implémenté dans les langages Oberon [69] et Modula-3 [15].

Il s'agit du dernier ajout à notre calcul, qui permet maintenant de nombreuses et intéressantes instances de relation de sous-typage entre types abstraits et concrets. La complexité des interactions entre sous-typage et frontières d'abstraction entraîne alors deux changements significatifs à notre calcul. Tout d'abord, les crochets colorés deviennent additifs : plutôt que de masquer la couleur située à l'extérieur, ils ajoutent leur couleur à cette dernière. Deuxièmement, nous éliminons le sous-typage implicite et propageons explicitement les annotations de sous-typage. Ces annotations de sous-typage peuvent être effacées en même temps que les types avant l'exécution.

2.4.1 Signatures étendues pour les types partiellement abstraits

Un type partiellement abstrait est obtenu lorsqu'à la place de donner une signature opaque comme `sig type t val ... end`, le programmeur indique un super-type de la manière suivante : `sig type t <: T val ... end`. Les types partiellement abstraits peuvent être vus comme une manière de cacher une partie d'un type de donnée. Par exemple, certains champs d'un enregistrement peuvent être public tandis que les autres restent cachés.

Précédemment, les signatures étaient de la forme suivante $[X : K, T]$ avec la sorte K restreinte aux cas $K = \mathbf{Eq}(T)$ (type concret) et $K = \mathbf{Le}(\top)$ (type abstrait). La déclaration

des types partiellement abstraits s'effectue maintenant en spécifiant un super-type arbitraire, $K = \mathbf{Le}(T')$ (le super-type peut toujours être $\top$).

En conséquence, toutes les empreintes peuvent à tout moment être comparées à leur super-type déclaré.

$$\frac{E \vdash_c^H \text{ok} \qquad \vdash h \,\text{ok}}{E \vdash_c^H h.\texttt{type} <: T_0'}$$

$$\text{où } h = \mathbf{hash}(N, [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1'])$$

Dans la suite, nous examinons les difficultés soulevées dans notre langage par l'ajout des types partiellement abstraits.

2.4.2 Conséquences sur la sémantique

Considérons l'expression constituée d'un destructeur appliqué à une valeur abstraite. Avant l'introduction des types partiellement abstraits, cette valeur abstraite pouvait être révélée (sinon l'expression aurait été mal typée). L'évaluation de notre expression aurait alors poussé les crochets colorés de l'abstraction vers l'intérieur du terme pour permettre au destructeur d'agir. Maintenant, il existe la possibilité que la valeur ne soit que partiellement abstraite et qu'il ne soit donc pas possible de la révéler dans la couleur ambiante.

La propriété principale des valeurs partiellement abstraites est ainsi la possibilité de leur appliquer certains destructeurs. Nous cherchons donc une sémantique déterministe et préservant le typage qui permette la destruction partielle de valeurs abstraites.

L'exemple suivant illustre les étapes de raisonnements poussant aux changements décrits plus loin dans cette section.

Exemple 2.9 (Types partiellement abstraits et abstractions) Soit un type partiellement abstrait $h.\texttt{type}$ dont on peut établir que dans la couleur c_0 et la relation de sous-typage H il possède comme super-type le couple $T_1 * T_2$. La première projection appliquée à une valeur abstraite de ce type (en toute généralité $[v^c]_c^{h.\texttt{type}}$) se doit d'être bien typée et de se réduire. Une situation résumée par la formule suivante.

$$H, \mathbf{proj}_1\, [v^c]_c^{h.\texttt{type}} \rightarrow_{c_0}^* \ldots \qquad \text{lorsque } \vdash_{c_0}^H h.\texttt{type} <: T_1 * T_2$$

Il est maintenant nécessaire de décider d'une stratégie de réduction qui mène à un résultat bien typé.

On peut tout d'abord distinguer deux cas : soit $[v^c]_c^{h.\texttt{type}}$ est une valeur lorsque $\vdash_{c_0}^H h.\texttt{type} <: T_1 * T_2$, soit il est possible de réduire cette expression. Il nous faut écarter cette deuxième possibilité du fait qu'il n'est pas possible de déterminer à l'avance l'usage qui sera fait d'une valeur partiellement abstraite : une valeur partiellement abstraite $[v^c]_c^{h.\texttt{type}}$ peut par exemple être l'argument de deux fonctions qui attendent des valeurs de types $h.\texttt{type}$ et $T_1 * T_2$, nécessitant un usage du sous-typage seulement dans le second cas.

En supposant donc que $[v^c]_c^{h.\texttt{type}}$ reste une valeur même si $h.\texttt{type}$ est un type abstrait, la réduction éventuelle s'effectue en présence du destructeur. Le principe de la réduction est alors de permuter le destructeur avec les crochets colorés.

Cette permutation pose cependant un problème de typage : $h.\texttt{type} <: T_1 * T_2$ est prouvable dans la couleur c_0 mais pas nécessairement dans la couleur c présente à l'intérieur des crochets. La solution à ce problème est de passer à un système où les crochets sont additifs : la couleur à l'intérieur des crochets est dans notre exemple $c \cup c_0$ plutôt que c.

$$H, \mathbf{proj}_1\ [v^c]_c^{h.\texttt{type}} \rightarrow_{c_0} [\mathbf{proj}_1\ v^c]_c^{\cdots} \qquad \text{lorsque } \vdash_{c_0}^H h.\texttt{type} <: T_1 * T_2$$

Une fois le destructeur à l'intérieur des crochets, il reste la question de l'annotation de type à mettre sur les nouveaux crochets permutés. Il est tentant de mettre T_1, mais le choix d'un type plus général que $h.\texttt{type}$ n'est pas unique et il n'existe pas de notion de plus petit super-type dans notre système. La solution pour contourner cette situation est de passer à un système à sous-typage explicite.

Les deux changements justifiés par l'exemple précédent mènent à des conséquences que nous présentons dans les deux sections suivantes.

2.4.3 Crochets additifs

Notre premier changement majeur est donc de rendre les crochets colorés additifs, c'est-à-dire de rendre les empreintes extérieures accessibles à l'intérieur. La règle de typage des crochets colorés est modifiée de la façon suivante :

$$\frac{E \vdash_c^H T : \mathbf{Le}(\top) \qquad E \vdash_{c\cup c'}^H e : T}{E \vdash_c^H [e]_{c'}^T : T}$$

L'expression r est maintenant typée dans la couleur $c \cup c'$ plutôt que dans la couleur c' précédemment utilisée.

Un certain nombre de règles de réduction sont adaptées à la nouvelle sémantique additive des crochets colorés. L'exemple le plus emblématique en est la règle de β-réduction qui réapparaît dans sa version habituelle, puisque tous les endroits où la variable x apparaît sont maintenant typés dans une couleur contenant au moins la couleur courante c :

$$(\lambda x : T.e)\ v^c \rightarrow_c \{x \leftarrow v^c\}e$$

Les règles gouvernant les crochets colorés sont aussi adaptées, comme le montre la nouvelle règle de révélation suivante :

$$H, [\widehat{v}^{c'\cup c}]_{c'}^{h.\texttt{type}} \rightarrow_c H, [\widehat{v}^{c'\cup c}]_{c'}^{T} \qquad \text{si } h \in c \text{ et } \mathrm{impl}(h){=}T$$

Remarquons que la définition des valeurs doit être changée pour s'adapter aux nouveaux contextes colorés. Nous notons $\widehat{v}$ une valeur dont le constructeur extérieur n'est pas un crochet coloré. Les conditions d'application des règles sont aussi modifiées, comme c'est le cas dans la version définitive de la règle de fusion des crochets :

$$H, [[\widehat{v}^{c\cup c_0\cup c_1}]_{c_0}^{h_0.\texttt{type}}]_{c_1}^{h_1.\texttt{type}} \rightarrow_c H, [\widehat{v}^{c\cup c_0\cup c_1}]_{c_0\cup c_1}^{h_1.\texttt{type}} \qquad \text{si } h_0 \notin c_1 \cup c \text{ et } h_1 \notin c$$

La question cruciale reste cependant celle de la préservation des frontières de l'abstraction. Le passage aux crochets additifs ne change rien, puisque les règles de typage et de réductions continuent de s'assurer qu'un type abstrait n'est révélé que lorsque son empreinte est connue, c'est-à-dire que son module est ouvert. Il est intéressante de remarquer que, jusqu'à l'ajout des types partiellement abstraits, le choix entre les versions additives ou non-additives est arbitraire : il n'y a aucune différence en terme de préservation des abstractions.

2.4.4 Sous-typage explicite

L'ajout du sous-typage est la solution que nous retenons face à la complexité de la relation de sous-typage. Nous montrons plus loin que ces annotations peuvent être effacées à l'exécution tandis que nous espérons qu'une inférence des types (peut-être seulement partielle) permette de s'en passer.

L'avantage principal du sous-typage explicite est de simplifier le système de types et la sémantique de notre langage : la syntaxe des termes à typer ou à réduire donne désormais de façon unique la règle à appliquer.

Nous ajoutons tout d'abord les annotations de sous-typage explicite à notre grammaire des expressions.

$e ::= \ldots \quad \mid \quad {}_{(T_1 <: T_2)}e$ sous-typage explicite

La règle de typage modélisant le sous-typage implicite est remplacée par sa version explicite.

$$\frac{E \vdash_c^H e : T \qquad E \vdash_c^H T <: T'}{E \vdash_c^H {}_{(T<:T')}e : T'}$$

La plupart des règles de typage ne sont pas modifiées puisque le sous-typage explicite est seulement un témoin syntaxique du sous-typage implicite qui était déjà présent. Seules les quelques règles qui utilisaient le sous-typage implicite ont donc besoin d'être adaptées. C'est tout d'abord le cas de la désérialisation dans laquelle le type de sérialisation doit être un sous-type du type de désérialisation : nous introduisons donc l'annotation correspondante dans le membre droit.

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{\,c',H'}(v^\bullet : T) : T') \rightarrow_c \begin{cases} H \cup H',\ {}_{(T<:T')}[v^\bullet]_{c'}^{T} & \text{si } \mathbf{nil} \vdash_c^{H \cup H'} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{sinon} \end{cases}$$

Lors du typage d'un module, il y a de même sous-typage implicite entre le type implicite de la valeur définie dans le module et le type déclaré dans la signature. Nous devons donc rendre explicite le type des valeurs définies dans les structures.

$M ::= [T, v^\bullet : T']$ structure

La réduction de la définition d'un module peut alors générer l'annotation de sous-typage correcte.

$$H,\ \text{module } NU = [T_0, v^\bullet : T_1] : [X : \mathbf{Eq}(T'_0), T'_1] \text{ in } m \\ \to_m H, \{U.\texttt{type} \leftarrow T'_0, U.\texttt{term} \leftarrow {}_{(T_1 <: \{X \leftarrow T'_0\} T'_1)} v^\bullet\} m$$

$$H,\ \text{module } NU \text{ restricts } \ h_1\ = [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T'_0), T'_1] \text{ in } m \\ \to_m H \cup h_1 <: h,\ \sigma\, m$$

$$\text{où } \begin{cases} h = & \mathbf{hash}(N, [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T'_0), T'_1]) \\ \sigma = & \{U \leftarrow h, \\ & U.\texttt{type} \leftarrow h.\texttt{type}, \\ & U.\texttt{term} \leftarrow [{}_{(T_1 <: \{X \leftarrow h.\texttt{type}\} T'_1)} v^\bullet]_{\{h\}}^{\{X \leftarrow h.\texttt{type}\} T'_1} \} \end{cases}$$

Dans ces deux règles de réduction des définitions de modules, les annotations de sous-typage sont introduites pour permettre à la valeur $v^\bullet$ définie par la structure d'avoir le type déclaré par la signature.

Enfin, il reste à traiter le cas de l'interaction du sous-typage explicite et des crochets colorés : notre règle les fait permuter.

$$H,\ {}_{(T' <: T'')}([\widehat{v}^{c' \cup c}]_{c'}^{h.\texttt{type}}) \to_c H, [{}_{(T' <: T'')} \widehat{v}^{c' \cup c}]_{c'}^{T''} \text{ où } h \notin c$$

Remarquons que déplacer l'annotation de sous-typage à l'intérieur des crochets colorés dépend de l'additivité des crochets colorés : sans celle-ci, la relation de sous-typage qui est valide dans la couleur c ne serait pas nécessairement valide dans la couleur c'. Grâce à l'additivité des crochets colorés, la couleur à l'intérieur est maintenant $c \cup c'$ ce qui est suffisant pour prouver la validité du jugement de sous-typage.

Il est aussi important de noter que l'introduction du sous-typage explicite et de la stratégie de réduction associée modifient à nouveau la définition de la grammaire des valeurs. Celle-ci est maintenant divisée en trois étages : v^c est une valeur quelconque dans la couleur c ; $\widehat{v}^c$ est une valeur qui n'est pas de la forme $[v]_{c'}^T$; $\widehat{\widehat{v}}^c$ est la sous-catégorie des valeurs $\widehat{v}^c$ qui ne sont pas de la forme ${}_{(T <: T')} v$.

Lorsque le sous-typage explicite se déplace à l'intérieur des crochets colorés, certains types abstraits peuvent être révélés du fait de la présence de couleurs supplémentaires. Un petit nombre de règles de réduction gèrent ces réécritures d'annotation de sous-typage explicite. L'annexe A contient le reste des définitions.

$$H,\ {}_{(T_0 <: T_1)} \widehat{v}^c \to_c H,\ {}_{(TV_2^c <: T_1)} \widehat{\widehat{v}}^c \qquad \text{where } \widehat{v}^c = {}_{(TV_2^c <: TV_3^c)} \widehat{\widehat{v}}^c$$

$$H,\ {}_{(T_0 <: h_0.\texttt{type})} \widehat{\widehat{v}}^c \to_c H,\ {}_{(T_0 <: T_1)} \widehat{\widehat{v}}^c \qquad \text{when } h_0 \in c \text{ and } \mathrm{impl}\,(h_0) = T_1$$

$$H,\ {}_{(h_0.\texttt{type} <: TV^c)} \widehat{\widehat{v}}^c \to_c H,\ {}_{(T_0 <: TV^c)} \widehat{\widehat{v}}^c \qquad \text{when } h_0 \in c \text{ and } \mathrm{impl}\,(h_0) = T_0$$

$$H,\ {}_{((T_1 * \ldots * T_j) <: (T'_1 * \ldots * T'_j))}(v_1^c, \ldots, v_j^c) \to_c H, ({}_{(T_1 <: T'_1)} v_1^c, \ldots, {}_{(T_j <: T'_j)} v_j^c)$$

$$H,\ {}_{(\{l_1 : T'_1; \ldots; l_i : T'_i\} <: \{l_1 : T_1; \ldots; l_j : T_j\})}\{l_1 = v_1^c; \ldots; l_j = v_j^c\} \to_c H, \{l_1 = v_1^c; \ldots; l_i = v_i^c\}$$

$$H,\ {}_{((T_1 \to T_2) <: (T'_1 \to T'_2))}(\lambda x : T_0.e) \to_c H, (\lambda x : T'_1.\ {}_{(T_2 <: T'_2)}\{x \leftarrow {}_{(T'_1 <: T_1)} x\} e)$$

$$H,\ {}_{(\text{bytes} <: \text{bytes})}\mathbf{marshalled}_{\,c', H'}(e : T) \to_c H, \mathbf{marshalled}_{\,c', H'}(e : T)$$

$$H,\ {}_{(\text{unit} <: \text{unit})}() \to_c H, ()$$

2.5 HATS : résultats

Dans cette partie, nous résumons les principaux résultats obtenus concernant la sûreté du typage et de la sémantique de notre langage. L'ensemble des définitions de syntaxe, typage et sémantique de notre langage final HATS sont données dans l'annexe A. Tous les résultats théoriques ainsi que les démonstrations détaillées sont dans l'annexe B.

Dans notre système, les valeurs abstraites sont représentées par des crochets colorés : ces crochets matérialisent les frontières de l'abstraction que nous nous somme efforcés de préserver dans un environnement distribué avec sous-typage. C'est ainsi le choix des règles de propagation des crochets colorés que nous avons présentées tout au long de ce chapitre qui, lorsqu'elles sont justifiées par le théorème de préservation du typage, garantissent la préservation des abstractions.

Les énoncés suivants concernent les propriétés de préservation du typage par les réductions d'expressions, de machines et de réseau dans le système HATS.

Theorem 2.1 (Typage)
Si $\vdash^{H}_{\bullet} m : T$ et $H, m \rightarrow_m H', m'$ alors $\vdash^{H'}_{\bullet} m' : T$.
Si $\vdash^{H}_{c} e : T$ et $H, e \rightarrow_c H', e'$ alors $\vdash^{H'}_{c} e' : T$.
Si $\vdash n\,\text{ok}$ et $n \rightarrow_n n'$ alors $\vdash n'\,\text{ok}$.

Comme la préservation du typage n'a de sens que si les réductions ont effectivement lieu, nous avons aussi démontré des propriétés de progrès pour la réduction des machines, des expressions et des réseaux.

Theorem 2.2 (Avancement de la compilation) Si $\vdash^{H}_{\bullet} m : T$ alors soit m est une expression soit il se réduit par $\rightarrow_m$. De plus la compilation $(\rightarrow_m)$ termine.

Theorem 2.3 (Avancement du calcul) Si $\vdash^{H}_{c} e : T$ alors de quatre choses l'une :
- e est une valeur, i.e. il existe une valeur v^c telle que $e = v^c$
- e peut se réduire, i.e. il existe e' et H' tels que $H, e \rightarrow_c H', e'$
- e est en attente d'émission ou de réception, i.e. il existe un contexte $CC^c_{c_2}$ et une valeur v^{c_2} tels que $e = CC^c_{c_2}.!\,v^{c_2}$ ou $e = CC^c_{c_2}.?$
- e a lancé une exception, i.e. il existe un contexte $CC^c_{c_2}$ et un type T' tels que $e = CC^c_{c_2}.\mathbf{Unmarfailure}^{T'}$

Concernant l'avancement des réseaux, le théorème doit prendre en compte toutes les possibilités créées par l'état des communications. Nous définissons une congruence struc-

turelle standard $\equiv$ sur les réseaux et les classes de réseau suivantes :

$n_{()}$ ::=	$\mathbf{0}$	vide
	$n_{()} \mid n_{()}$	composition parallèle
	$()$	unit
n_{fail} ::=	$\mathbf{0}$	vide
	$n_{\text{fail}} \mid n_{\text{fail}}$	composition parallèle
	$CC^{\bullet}_{c}.\mathbf{Unmarfailure}^{T}$	arrêt
$n_{?}$ ::=	$n_{?} \mid n_{?}$	composition parallèle
	$CC^{\bullet}_{c}.?$	en attente de réception
$n_{!}$ ::=	$n_{!} \mid n_{!}$	composition parallèle
	$CC^{\bullet}_{c}.!\, v$	en attente d'émission

Le théorème de progrès garantit que tout réseau bien-typé peut se réduire lorsque la communication est possible.

Theorem 2.4 (Avancement des communications) Si $\vdash n$ ok alors l'un des cas suivants est vérifié :
– n est à l'arrêt, i.e. il existe $n_{()}$ et n_{fail} tels que $n \equiv n_{()} \mid n_{\text{fail}}$.
– n est en attente de réception, i.e. il existe $n_{()}$, n_{fail} et $n_{?}$ tels que $n \equiv n_{()} \mid n_{\text{fail}} \mid n_{?}$
– n est en attente d'émission, i.e. il existe $n_{()}$, n_{fail} et $n_{!}$ tels que $n \equiv n_{()} \mid n_{\text{fail}} \mid n_{!}$
– n peut se réduire, i.e. il existe n' tel que $n \rightarrow_{n} n'$

Le principe de notre sémantique est de séparer les valeurs abstraites des valeurs concrètes grâce aux crochets colorés. Cette séparation n'est effective que si le typage est suffisamment précis. Le théorème suivant garantit que le typage est unique à équivalence près.

Theorem 2.5 (Typage propre) Si $\vdash^{H}_{c} e : T$ et $\vdash^{H}_{c} e : T'$ alors $\vdash^{H}_{c} T == T'$.

Ces théorèmes rassemblés organisent l'exécution des programmes distribués dans le respect d'un système de types conçu pour assurer l'intégrité des valeurs abstraites. Démontrer la propriété de confidentialité réclame une approche différente [60] fondée sur une notion d'équivalence observationnelle que nous n'abordons pas dans cette thèse.

2.6 Implémentation

Après la présentation de notre langage et de ses propriétés, nous évoquons les questions relatives à la réalisation d'une implémentation.

2.6.1 Effacement

Dans la plupart des langages de la famille ML, les annotations de typage sont effacées à l'exécution pour une efficacité plus grande. Ceci est permis par un théorème stipulant que les comportements des programmes dans le langage originel et dans leurs versions effacées sont identiques. Notre langage vérifie un tel théorème.

Décrivons tout d'abord l'opération d'effacement. Un *terme effacé* est un terme dans lequel toutes les annotations de typage et sous-typage ainsi que les crochets colorés ont

été enlevés, avec l'exception notable des annotations sur **mar** , **marshalled** et **unmar** . Les règles de réduction sur ces termes nettoyés sont les dérivées des règles actuelles une fois l'opération d'effacement appliquée aux membres gauche et droit.

Une règle cependant ne se comporte pas de façon satisfaisante dans le langage effacé : il s'agit de la suppression des champs inutilisés d'un enregistrement, une opération similaire à l'action d'un ramasse-miettes.

$$H,\ _{(\{l_1:T_1;\ldots;l_j:T_j\}<:\{l_1:T'_1;\ldots;l_i:T'_i\})}\{l_1 = v_1^c; \ldots; l_j = v_j^c\} \rightarrow_c H, \{l_1 = v_1^c; \ldots; l_i = v_i^c\}$$

La version effacée de cette règle effectue un effacement complètement non-déterministe des champs des enregistrements.

Une solution pour corriger ce problème pourrait être d'ajouter explicitement des nœuds d'effacement de champs à notre langage effacé pour remplacer les annotations de sous-typage. Cela complique cependant la correspondance entre les réductions des deux langages.

Nous choisissons donc une solution différente qui consiste à affaiblir notre fonction d'effacement en une relation d'effacement $\text{erase}\,(e, \underline{e})$ qui associe une expression e à une de ses versions effacées $\underline{e}$ dans laquelle chaque enregistrement $\{l_1 = e_1; \ldots; l_j = e_j\}$ de e est en relation avec un enregistrement de $\underline{e}$ qui peut contenir des champs supplémentaires avec des valeurs arbitraires $\{l_1 = \underline{e}_1; \ldots; l_j = \underline{e}_j; \ldots; l_k = \underline{e}_k\}$. Nous notons $\rightarrow_{\text{nb}}$ la réduction dans le langage effacé.

Le théorème d'effacement porte alors sur la correspondance des réductions entre les deux langages : chaque réduction dans le langage originel correspond à au plus une réduction dans le langage effacé, alors que chaque réduction dans le langage effacé est reflétée par une séquence d'au moins une réduction dans le langage annoté.

Theorem 2.6 (Effacement) Si $\vdash_c^H e : T$ et $H, e \rightarrow_c H', e'$ et l'expression du langage effacé $\underline{e}$ est telle que $\text{erase}\,(e, \underline{e})$, alors il existe $\underline{e}'$ tel que $H, \underline{e} \rightarrow_{\text{nb}}^? H', \underline{e}'$ et $\text{erase}\,(e', \underline{e}')$. Si $\vdash_c^H e : T$ et il existe $\underline{e}$ tel que $\text{erase}\,(e, \underline{e})$ et si $H, \underline{e} \rightarrow_{\text{nb}} H', \underline{e}'$ alors il existe e' tel que $\text{erase}\,(e', \underline{e}')$ et $H, e \rightarrow_c^+ H', e'$.

Remarquons enfin que la présence des champs supplémentaires dans le langage effacé n'est pas gênante : le typage réalisé à la compilation permet de savoir quels champs sont utiles pour l'exécution. Il semble donc possible d'imaginer une stratégie d'effacement des champs superflus à l'exécution.

2.6.2 Empreintes

Jusqu'à présent notre présentation a considéré les empreintes comme $\mathbf{hash}(N, [T_0, v^\bullet] : [X : \mathbf{Le}(T), T'])$ comme des valeurs algébriques dont on peut extraire chacun des constituants, comme la structure ou le type concret T_0. Dans une implémentation, les empreintes peuvent cependant reposer sur un mécanisme plus efficace puisque la seule opération réalisée sur les empreintes est le test d'égalité. Des fonctions pseudo-injectives, comme SHA-1, permettent alors de calculer de façon efficace une empreinte concise.

Nous examinons donc la sémantique du langage effacé pour montrer que cette façon d'implémenter les empreintes est sûre, modulo un petit changement. Dans la sémantique effacée, une seule règle fait intervenir les parties d'une empreinte : la règle de désérialisation requiert en effet un test de sous-typage qui repose sur la possibilité de déconstruire une empreinte.

– Dans les arbres de preuve de jugements de sous-typage, toutes les empreintes sont déconstruites pour être prouvées correctes. Notre hypothèse de correction des empreintes (le réseau et les participants étant fiables) nous dispense de cette opération. La seconde conséquence est que les preuves de sous-typage ne font donc pas intervenir de preuve de typage d'expressions.
  Pour la même raison, la vérification de la correction de la relation de sous-typage entre types abstraits n'est pas nécessaire.
– Le test de sous-typage réalisé à la désérialisation est effectué dans la couleur vide. Observons que toutes les règles de typage utilisent la même couleur dans leurs prémisses que dans la conclusion (en excluant les règles de typage des expressions, voir ci-dessus). La révélation du type concret correspondant à une empreinte n'est donc jamais effectuée.
– Enfin, les types abstraits sont des sous-types de leur super-type déclaré : ce super-type doit donc être accessible à partir d'une empreinte donnée.

Nous pouvons conclure que les empreintes $h = \mathbf{hash}(N, M : [X : \mathbf{Le}(T), T'])$ peuvent être implémentées avec un couple constitué d'une empreinte cryptographique des éléments $N, (M : [X : \mathbf{Le}(T), T'])$ et du super-type déclaré T. L'implémentation est donc concise et efficace, en étant notamment indépendante de la taille du code source déclarant le type abstrait.

2.6.3 Décidabilité du système de types

Notre sémantique repose sur un test de sous-typage pour comparer le type reçu au type attendu au moment de la désérialisation. L'existence d'un algorithme n'est pas évidente du fait de la diversité des moyens qui aboutissent à une relation de sous-typage. Grâce à la transitivité, le sous-typage peut provenir d'une combinaison du sous-typage structurel entre types enregistrements, du sous-typage entre types abstraits présents dans la relation H, ou de la relation entre un type partiellement abstrait et son super-type. La question de la décidabilité du système reste donc ouverte bien que l'existence d'un algorithme soit quand même probable (voir l'annexe B.8.2).

Chapitre 3

Compilation de sessions sécurisées

Les applications réparties permettent de réaliser des tâches globales grâce aux interactions de systèmes indépendants. La difficulté principale de la programmation répartie vient du manque de contrôle que le programmeur a sur l'environnement d'exécution de son programme : non seulement le réseau est une possible source d'erreurs, mais on ne peut pas toujours faire confiance aux programmes distants pour respecter le protocole souhaité. Nous avons vu dans le chapitre précédent une approche souple permettant d'obtenir des garanties de sûreté pour les valeurs de types abstraits de données partagées entre machines bien typées. Nous nous tournons maintenant vers la question du respect d'un protocole par des machines possiblement compromises.

3.1 Introduction

Un premier pas vers la simplification de la programmation distribuée est de proposer au programmeur, via des primitives de langage de programmation ou des fonctions de bibliothèque, des abstractions pour les moyens de communications les plus communs. Le « Remote Procedure Call » (RPC) est par exemple le schéma de communication où l'envoi d'une valeur est immédiatement suivi d'une réponse : le programmeur d'un client RPC n'a ainsi pas à se soucier de programmer lui-même successivement l'envoi et la réception. Les canaux privés de communication sont un autre exemple de structure. L'utilisation de protocoles cryptographiques que permet une bibliothèque comme SSL [20] permet en effet de créer un canal binaire de communication dont le comportement pour le programmeur est similaire à celui d'un canal habituel, mais dont les garanties en matière de sécurité incluent authentification, confidentialité et intégrité des données transmises.

Au delà de ces façons simples de communiquer entre deux machines, les applications réparties concernent généralement un grand nombre d'entités qui s'échangent des messages selon des séquences préétablies, que l'on nomme protocoles, contrats [16, 17], conversations [12] ou *sessions* [41]. La description d'une session correspond alors à la spécification du comportement de chacune des entités impliquées, nommées *rôles*. Ces rôles sont joués dans chaque instance du protocole par les entités présentes sur le réseau, les *principaux*.

L'existence d'une spécification globale des interactions entre rôles est intéressante a priori pour guider l'implémentation, et essentielle a posteriori pour s'assurer de sa correction.

3.1.1 Contexte scientifique

L'étude des phénomènes de concurrence et des programmes distribués s'est d'abord effectuée dans des calculs synchrones où les communications consistent en l'envoi d'un unique message sur un canal donné. Les principaux formalismes en sont CCS [49], le π-calcul [51], la machine abstraite chimique [4], le join-calcul [30], le Kell-calcul [64]. Le passage à des versions asynchrones ou à des primitives de communication basées sur des canaux s'est imposé pour la plupart des implémentations concrètes du fait du coût en pratique élevé de la synchronisation.

Si l'étude des protocoles de communication est relativement ancienne, l'utilisation d'une spécification définie par des séquences préétablies de messages dans le contexte de la programmation distribuée ou concurrente est récente [40, 41]. Ces efforts ont porté principalement sur l'utilisation des sessions comme type pour les canaux dans le π-calcul : la conformité d'un code donné par rapport à une spécification de session peut être vérifié statiquement et garantit qu'un programme bien typé va se comporter comme la session l'exige [35, 18, 13, 42]. Ainsi, sous l'hypothèse que tous les participants sont bien typés, le programmeur a la garantie d'une exécution conforme à la spécification.

De nombreux travaux essayent d'intégrer les sessions à des systèmes existants : langages objets [24, 25, 43] ou fonctionnels [54, 68], systèmes d'exploitation [29], services web [16, 13]. Dans chacun de ces cas, les sessions sont des spécifications qui sont respectées dès lors que tous les participants sont bien typés.

3.1.2 Objectifs

Les deux motivations principales de ce travail sont les suivantes : d'une part nous voulons utiliser la spécification de session comme source pour réaliser automatiquement une implémentation correcte à la place de l'habituelle vérification a posteriori ; d'autre part nous souhaitons considérer le cas réaliste où les participants distants ne sont pas nécessairement bien typés.

Dans la situation dans laquelle nous nous plaçons, il n'est plus possible de s'assurer à l'avance que les différents principaux vont jouer correctement leurs rôles dans la session. Certains d'entre eux peuvent contrôler tout ou partie du réseau, ce qui leur permet d'écouter, d'intercepter et de modifier les messages échangés. Il est même possible qu'ils forment une coalition pour perturber l'exécution de la session et tromper des participants honnêtes.

Il est ainsi nécessaire, dans le cadre d'une implémentation sécurisée, que chacun des participants surveille le comportement des autres pour détecter toute déviation par rapport à la spécification. Réaliser une telle implémentation à la main est difficile pour les programmeurs puisque les langages de programmation ou bibliothèques disponibles n'offrent aucune aide dans le domaine de la sécurisation de protocoles impliquant plusieurs participants.

3.1.3 Compilation sécurisée de sessions : architecture

Nous proposons donc un schéma d'implémentation sécurisée des sessions sous la forme d'un compilateur pour un langage étendant ML. Ce compilateur, que nous nommons `s2ml` est, à notre connaissance, le premier qui compile les déclarations de sessions vers un protocole cryptographique spécifique. Ce dernier apporte de solides garanties en termes de sécurité de l'exécution de la session en dépit de la présence d'un attaquant pouvant contrôler le réseau et prendre le contrôle des participants.

La base de notre travail est la conception d'un langage de description de sessions. La section 3.2 (page 49) définit ainsi les deux représentations duales des sessions que nous utilisons : la représentation locale décrit une session comme collection de rôles, et la représentation globale d'une session est donnée sous la forme d'un graphe. Ces définitions sont accompagnées de nombreux exemples inspirés des services web.

Ce langage de session est ensuite intégré au langage ML grâce à une interface simple. La section 3.3 (page 58) décrit cette interface en détail et l'illustre avec quelques exemples de code utilisateur. Le langage ML a été choisi pour plusieurs raisons. Tout d'abord, nous utilisons le système de types de ML pour garantir que le code utilisateur respecte localement la session. Ensuite, le pilotage de la session est effectué par une série de continuations qui sont plus naturelles à exprimer dans un langage fonctionnel. Enfin, les types algébriques et le filtrage permettent une interface concise et expressive. En pratique, nous utilisons la syntaxe d'Ocaml [48] et bénéficions de la possibilité d'utiliser le compilateur d'Ocaml ou celui de F# [67].

La section 3.4 (page 68) formalise l'intégration des sessions à ML et donne une sémantique au comportement que le programmeur est en droit d'attendre des sessions. De cette sémantique découle l'énoncé d'un théorème de sécurité se rapportant au respect par l'implémentation sécurisée du flot de messages spécifié par la session, et ceci même en présence d'un attaquant contrôlant le réseau et certains des participants.

La section 3.5 (page 76) traite de la conception du protocole cryptographique que l'implémentation sécurisée va devoir utiliser pour satisfaire le théorème de sécurité. Nous apportons un grand soin à obtenir un protocole efficace en minimisant la taille des messages et le nombre des vérifications cryptographiques qu'il est nécessaire d'effectuer lors d'une réception.

La section 3.6 (page 84) décrit les bibliothèques sur lesquelles notre implémentation sécurisée s'appuie. Ces bibliothèques permettent de manipuler certains types de données, d'effectuer les principales opérations cryptographiques et d'accéder au réseau. Pour une même interface, nous fournissons plusieurs implémentations de ces bibliothèques : une version concrète s'appuyant sur OpenSSL ou .Net, et une version symbolique remplaçant les opérations cryptographiques par l'application de constructeurs algébriques (similaire à la cryptographie dans le modèle de Dolev-Yao [26]). En liant une implémentation avec les versions concrètes des bibliothèques, on peut obtenir un programme exécutable dans un environnement distribué, alors que la version symbolique permet d'établir les résultats de correction et sert au débogage.

La section 3.7 (page 88) donne une description détaillée de notre compilateur. Celui-ci prend en entrée une définition de session et génère une implémentation sécurisée d'un protocole adapté (celui décrit à la section 3.5) pour chacun des rôles.

La stratégie mise en place pour prouver notre théorème de sécurité nécessite une sémantique spécifique à l'adversaire. Cette dernière est donnée dans la section 3.8 (page 103). Les théorèmes et lemmes intermédiaires y sont également discutés.

La question de l'évaluation des résultats de nos travaux fait l'objet de la section 3.9 (page 107). Nous y donnons quelques exemples de taille plus importante qui permettent de juger de la résistance de notre stratégie et de notre compilateur aux changements d'échelle. La performance de l'implémentation que nous générons y est aussi discutée.

3.2 Sessions

Une session est une description statique de l'ensemble des séquences valides de messages qui peuvent être échangés entre un nombre fixe de rôles. Elle correspond non seulement à

la description des alternances d'envois et de réceptions que chaque rôle doit effectuer, mais est aussi garante de la cohérence et de la séquentialité globale de ces communications.

Rôles

Les rôles d'une session sont précisés par un ensemble fini $\mathcal{R} = \{r_0, ..., r_{n-1}\}$ pour $n \geqslant 2$. Par convention le premier rôle r_0 envoie le message qui démarre la session. Chaque rôle représente un participant à la session avec sa propre implémentation locale de la session. Nous ne considérons pas le cas où le nombre de participants n'est pas pré-établi.

Messages

Les messages sont de la forme $f(v)$, où f est la description du message, ou *étiquette*, et v est le contenu du message. L'étiquette d'un message reflète l'intention de l'utilisateur à son égard dans la session et est utilisée comme identifiant unique d'un message donné au sein d'une session.

Nous utiliserons à la fois les notations $\widetilde{v}$ et $(v_i)_{i<n}$ pour dénoter une liste de valeurs séparées par des virgules $v_0, ..., v_{n-1}$; lorsqu'il est nécessaire de mentionner explicitement les indices, nous privilégions la notation $(v_i)_{i<n}$ plutôt que $\widetilde{v}$.

Dans le message $f(v)$, v représente une valeur dont le type est fixé à l'avance pour chaque message et dont le contenu est déterminé à l'exécution par l'utilisateur de la session.

Exécution

À tout moment de l'exécution de la session, au plus un rôle peut envoyer le message suivant : au départ r_0 envoie le premier message, puis le rôle qui reçoit ce message envoie le suivant, ... etc. La session décrit quelles peuvent être les étiquettes pour ce message. Le choix d'un message particulier (lorsqu'il est offert par la session) ainsi que de son contenu est laissé au programme contrôlant chaque rôle. À chaque fois qu'un rôle reçoit un message, il prend la main dans la session et ne la relâche qu'au moment de l'envoi du message suivant.

Nous définissons deux représentations interconvertibles des sessions. La première est globale : un graphe précise, à la manière d'un automate à états finis, les séquences valides de messages échangés entre rôles. La seconde est locale : pour chaque rôle, sa description syntaxique détermine l'alternance autorisée de réceptions et d'émissions des messages. Le graphe est essentiel pour tout raisonnement global, notamment pour discuter des propriétés de sécurité et des conditions d'implémentation sécurisée. Les descriptions locales sont à la base de l'implémentation et fournissent une interface claire pour la programmation indépendante de chacun des rôles.

Dans cette section, nous commençons par décrire de façon informelle ces deux représentations par des exemples, puis nous en donnons une définition formelle.

3.2.1 Exemples de sessions

Nous présentons quelques exemples de sessions dont nous donnons les représentations globales et locales. Ces exemples seront réutilisés tout au long de ce chapitre.

Les sessions peuvent être décrites par des graphes connexes orientés, dont un nœud est marqué (par un contour double) comme *initial*. Chaque arête représente un message de la session et est annotée par un identifiant (l'étiquette) du message. Les nœuds sont alors les états de la session et sont étiquetés par le rôle actif, c'est-à-dire le rôle amené à envoyer le message suivant. Autrement dit, une arête qui relie un nœud étiqueté par le rôle r_1 à un nœud étiqueté par le rôle r_2 représente un message envoyé par r_1 à destination de r_2.

Exemple 3.1 (Single) Un premier exemple de session est illustré par le graphe de la figure 3.1.

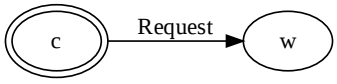


FIG. 3.1 – Une session simple à un seul message : Single

Cette session concerne deux rôles : un client (noté `c`) envoie un unique message à un service web (`w`). L'arête de ce graphe comporte l'étiquette du message ; ici, `c` envoie le message d'étiquette `Request` à `w`.

La seconde façon de décrire une session repose sur la représentation séparée des actions de chacun des rôles. Une session est introduite par le mot-clé session suivie du nom choisi. Chaque rôle est ensuite décrit successivement par le type de retour de l'exécution complète de la session et une représentation spécifiant les alternances d'envois et de réceptions qui concernent le rôle. Les envois utilisent le mot-clé send et les réceptions le mot-clé recv. Chaque rôle précise un type correspondant à la valeur que son exécution calcule. Chaque message doit de la même manière spécifier le type de la valeur qu'il transporte. Remarquons que cette information, pour des raisons de lisibilité, n'est pas reproduite dans les graphes (elle est cependant présente dans la définition formelle de la section 3.2.3).

Exemple 3.2 (Single - Code source) Le code ci-dessous correspond à la session `Single` :

```
session Single =
  role c : unit =
    send Request : string
  role w : string =
    recv Request : string
```

Ici, le rôle `c` a un type de retour `unit` et son comportement est réduit à l'envoi d'un message `Request` dont la valeur est de type `string`. Le rôle `w` a pour type de retour `string` et son comportement est dual à celui de `c`, c'est-à-dire qu'il reçoit le message d'étiquette `Request` de valeur de type `string`.

Dans la syntaxe locale, les symboles `;` et $\rightarrow$ expriment le caractère consécutif des actions. Le choix d'une syntaxe différente pour exprimer la suite d'un envoi ou d'une réception est expliqué plus bas (exemple 3.5).

Exemple 3.3 (Rpc) Il est possible de décrire une session dans laquelle une réponse est attendue de la part du service web. Le graphe de la figure 3.2 correspond alors au traditionnel « Remote Procedure Call » (RPC).

Dans cette session, `c` envoie le premier message d'étiquette `Request` au rôle `w`. Celui-ci prend alors la main et envoie le second message `Reply` à `c`. La session s'interrompt alors puisque le troisième nœud n'a pas d'arête sortante. Au final, cette session n'accepte comme chemin que la séquence `Request - Reply`.

Le code décrivant la session `Rpc` est le suivant :

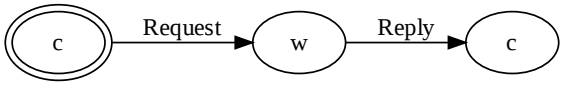


FIG. 3.2 – Une session simple à deux messages : Rpc

```
session Rpc =
  role c : int =
    send Request : string ;
    recv Reply : int
  role w : unit =
    recv Request : string →
    send Reply : int
```

Le comportement du rôle `c` comprend maintenant un envoi suivi d'une réception. Le message `Request` a toujours pour contenu une valeur de type `string`, mais le message `Reply` transmet une valeur de type `int`. De façon duale, le rôle `w` comporte la réception de `Request` puis l'envoi de `Reply`.

Le nombre de rôles dans une session est fixé au départ mais n'est pas limité à 2. Lorsque la session comporte trois rôles ou plus, on remarque que la dualité simple qui était présente entre les processus décrivant les deux rôles n'existe plus : la vérification de la cohérence des ensembles de processus locaux est discutée dans la section 3.2.5.

Exemple 3.4 (Forward) Nous décrivons ici une session similaire à la session `Rpc` à l'exception du fait que la communication entre client `c` et service web `w` passe par un proxy `p`. Un message `Forward` et un nœud annoté par `p` sont simplement ajoutés au graphe de la session (représenté dans la figure 3.3).

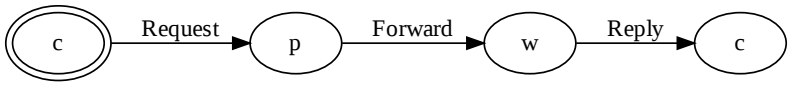


FIG. 3.3 – Une session à trois participants : Forward

Dans cette session, le client `c` envoie le premier message d'étiquette `Request` au proxy `p`. Celui-ci prend alors la main et envoie le second message `Forward` au service web `w`. La session s'achève par la réponse `Reply` que `w` envoie au client `c`. Cette session n'accepte comme chemin que la séquence `Request - Forward - Reply`.

Le code décrivant la session `Forward` est le suivant :

```
session Forward =
  role c : int =
    send Request : string ;
    recv Reply : int
  role p : unit =
    recv Request : string →
    send Forward : string
  role w : unit =
    recv Forward : string →
    send Reply : int
```

Les rôles y sont décris successivement en commençant par `c`, l'initiateur de la session. Chacun des rôles participe à exactement un envoi et une réception. Notons que la spécification de la session ne précise pas si la valeur du message `Forward` est la même que celle du message `Request`. Cette relation est de la responsabilité du programmeur du rôle `p` (l'extension des session avec des relations entre valeurs est discutée dans la section 4.2).

Le choix entre plusieurs chemins est représenté dans le graphe par des arêtes partant d'un même nœud. La syntaxe locale utilise respectivement les signes `+` et `|` lorsque le choix est interne ou externe.

Exemple 3.5 (Ws) Cet exemple illustre la possibilité d'effectuer un choix de chemin au moment de l'exécution de la session.

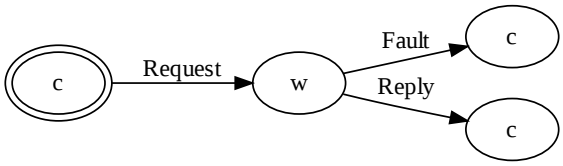


FIG. 3.4 – Une session avec choix : Ws

Lorsque le rôle `w` reçoit le message `Request` de la part de `c`, l'utilisateur pilotant `w` peut décider d'envoyer à `c` soit le message `Fault`, soit le message `Reply`. La décision peut être dynamique et dépendre notamment de la valeur du message `Request`. Les séquences valides de cette session sont `Request - Reply` et `Request - Fault`.

Le code décrivant la session `Ws` est le suivant :

```
session Ws =
  role c : unit =
    send Request : string ;
    recv [ Reply : int | Fault ]
  role w : string =
    recv Request : string →
    send ( Reply : int + Fault )
```

Le choix externe (c'est-à-dire en réception) utilise l'opérateur `|` ainsi que les crochets. Le choix interne est noté à l'aide de `+` et des parenthèses. Le choix d'une syntaxe différente dans ces deux cas est une aide visuelle au programmeur.

Il est possible de décrire des séquences non-bornées de messages à l'aide de cycles dans les graphes de sessions. Les processus locaux utilisent des points de récursion pour décrire ces boucles.

Exemple 3.6 (Wsn) La figure 3.5 ajoute une boucle au graphe de la figure 3.4.

Ici, lorsque le rôle `c` reçoit le message `Reply`, il doit envoyer un message `Extra` pour recommencer une itération de la boucle, que seul le rôle `w` peut briser.

Le code décrivant la session `Wsn` est le suivant :

```
session Wsn =
  role c : unit =
```

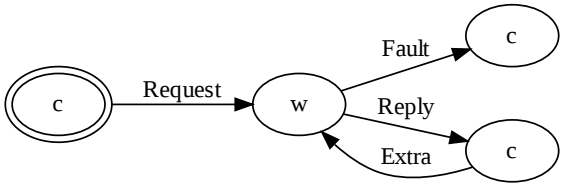


FIG. 3.5 – Une session avec boucle : Wsn

```
    send Request : string;
    loop:
    recv
    [ Reply : int  →
      send Extra : string ; loop
    | Fault ]
  role  w : string =
    recv Request : string  →
    loop:
    send
    ( Reply : int ;
      recv Extra : string  → loop
    + Fault )
```

Notons l'utilisation des points de récursion `loop:` pour modéliser les boucles.

Pour illustrer la flexibilité des descriptions de session, la figure 3.6 illustre l'ajout d'un message `Exit` pour permettre au rôle `c` d'interrompre lui aussi la répétition de la boucle.

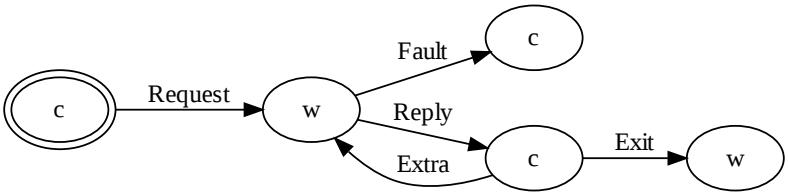


FIG. 3.6 – Une session avec boucle et message de sortie : Wsne

Le rôle `c` a ainsi le choix à chaque tour de boucle soit de demander une nouvelle itération par le message `Extra`, soit de signaler la fin de la session par le message `Exit`.

Nous présentons maintenant notre exemple principal.

Exemple 3.7 (Shopping) Le graphe de la figure 3.7 illustre l'ajout d'un troisième rôle aux sessions précédentes. La session peut maintenant modéliser un protocole d'e-commerce où le client `c` négocie la livraison d'un objet avec un magasin `w`, le tout sous l'œil attentif d'un observateur `o`.

Alors que les rôles `c` et `w` peuvent s'échanger un nombre illimité de messages dans une instance de la session `Shopping`, l'implication du rôle `o` est limitée à la réception de deux messages et l'envoi du message `Contract`.

Le code décrivant la session `Shopping` est le suivant :

```
session  Shopping =
```

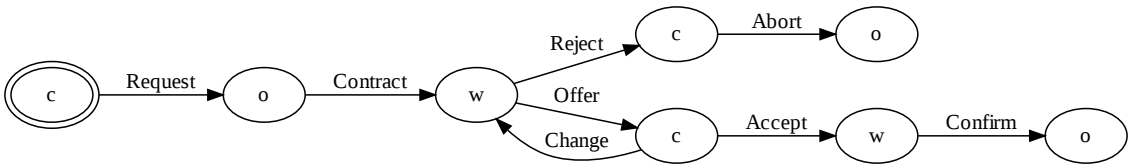


FIG. 3.7 – Une session avec trois participants : Shopping

```
role c:unit =
 send Request:string;
 start:
 recv
 [ Offer:string  →
    send ( Change:string; start
         + Accept:unit )
 |  Reject:string  → send Abort:unit
 ]

role o:unit =
  recv Request:string  →
  send Contract:string ;
  recv
  [ Confirm:unit
  | Abort:unit
  ]

role w:string =
  recv Contract:string  →
  loop:
  send
  ( Offer:string ;
    recv
    [ Change:string  → loop
    |  Accept:unit  → send Confirm:unit
    ]
  +  Reject:string )
```

Notons ici que les points de récursion sont locaux à chaque rôle et peuvent être notés avec n'importe quel identifiant (ici `start:` et `loop:`). En pratique, ces points de récursion permettent aussi de spécifier lorsque deux branches se rejoignent.

Trois exemples supplémentaires (les exemples 3.32, 3.33 et 3.34) sont également utilisés dans la suite.

3.2.2 Modèle d'exécution d'une session

Une fois définie, une session s'exécute en faisant tourner chacun des processus des rôles sur des machines connectées entre elles par un réseau. Chaque rôle d'une session est joué par un principal qui a pour fonction de donner les valeurs successives des messages de la

session et d'éventuellement effectuer les choix de chemin dans le graphe. Il va de soi qu'un principal dans notre cas est un programme utilisateur.

L'exécution d'une session commence lorsqu'un principal a_0 choisit le rôle initial r_0, sélectionne les principaux qui vont jouer les autres rôles et détermine le premier message envoyé. Si a_0 choisit le principal a_i pour jouer le rôle r_i, alors a_i rejoint l'exécution de la session en tant que r_i seulement lorsqu'un message lui est envoyé. L'exécution de la session consiste alors en l'échange de messages entre les principaux jouant les différents rôles jusqu'à ce que le processus correspondant au rôle actif s'arrête. Une session est donc dans notre cadre une abstraction portant sur la réalisation d'un certain motif de communications entre plusieurs participants.

Exemple 3.8 (Session Ws) Prenons l'exemple de la session `Ws` décrite dans l'exemple 3.5.

Supposons que le principal `Alice` décide d'exécuter le rôle du client `c`. `Alice` sélectionne le principal `Bob` qui joue le rôle du service web `w`, et envoie à `Bob` le premier message `Request` contenant une certaine chaîne de caractères.

À la réception, `Bob` rejoint l'exécution de la session `Ws` en tant que `w`, et choisit d'envoyer soit un message `Reply` soit un message `Fault` à `Alice`, ce qui dans tous les cas conclut son rôle dans la session. Après avoir reçu ce message, `Alice` en a fini avec son rôle et l'exécution est donc terminée.

Nous expliquons maintenant de façon plus formelle quels sont les graphes et collections de processus qui peuvent être utilisés pour représenter les sessions.

3.2.3 Graphes de sessions

Formellement, un graphe de session $\mathcal{G} = \langle \mathcal{R}, \mathcal{V}, \mathcal{L}, m_0 \in \mathcal{V}, \mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times V, R : \mathcal{V} \rightarrow \mathcal{R} \rangle$ regroupe un ensemble fini de rôles $\mathcal{R} = \{r_0, ..., r_{n-1}\}$, un ensemble de nœuds $m, m', m_i \in \mathcal{V}$, un ensemble d'étiquettes $f, g, l \in \mathcal{L}$, avec un nœud initial m_0, des arêtes étiquetées $(m, f, m') \in \mathcal{E}$ et une fonction R depuis les nœuds vers les rôles telle que $R(m_0) = r_0 \in \mathcal{R}$.

Nous demandons de plus que les graphes de sessions vérifient les propriétés suivantes.

1. Les arêtes relient des nœuds aux rôles distincts : si $(m, f, m') \in \mathcal{E}$ alors $R(m) \neq R(m')$.
2. Deux arêtes différentes ont des étiquettes distinctes : si $(m_1, f, m'_1) \in \mathcal{E}$ et $(m_2, f, m'_2) \in \mathcal{E}$ alors $m_1 = m_2$ et $m'_1 = m'_2$.

La propriété 1 interdit à un rôle de s'envoyer un message à lui-même : ce message serait alors invisible aux autres rôles et n'a rien à faire dans la description d'une session. La propriété 2 permet de s'assurer que chaque message possède une étiquette non-ambiguë, en particulier une étiquette détermine les nœuds d'envoi et de réception. Remarquons enfin qu'il est toujours possible de transformer un graphe donné de manière à ce qu'il satisfasse la condition 2 par renommage de certaines étiquettes.

Un *chemin* est une séquence d'arêtes connexes. Par la condition 2, une séquence d'étiquettes représente de façon unique un chemin. Nous désignons donc simplement par $\widetilde{f}$ le chemin suivant les arêtes étiquetées par $\widetilde{f}$. Un chemin est *initial* lorsqu'il débute au nœud initial m_0. Un rôle r est dit *actif* sur un chemin $\widetilde{f}$ lorsqu'il est le rôle d'un nœud d'origine d'une des arêtes du chemin $\widetilde{f}$.

3.2.4 Description locale des sessions

Une session peut aussi être représentée par la donnée des interactions que chacun des rôles a avec l'ensemble des autres. Chaque rôle est alors décrit par un processus spécifiant les alternances d'envoi et de réception lui correspondant. C'est cette description locale que nous utilisons comme syntaxe.

La figure 3.8 présente la syntaxe des sessions.

$\tau ::=$	unit | int | string	types de base
$p ::=$	0	fin de processus
	$\mathrm{send}(f_i : \tau_i\,;\, p_i)_{i<k}$	envoi
	$\mathrm{recv}[f_i : \tau_i \to p_i]_{i<k}$	réception
	$\chi : p$	nommage d'un sous-processus
	χ	continuation vers un sous-processus
$\Sigma ::=$	$(r_i : \tau_i = p_i)_{i<n}$	session (processus initiaux pour chaque rôle)

FIG. 3.8 – Syntaxe pour les descriptions locales de sessions

Les processus (et les rôles qu'ils décrivent) peuvent effectuer deux opérations de communication : l'envoi send et la réception recv. L'envoi comporte un choix interne entre les différents messages à envoyer, représentés par les étiquettes f_i pour $0 \leqslant i < k$. Le contenu de chaque message est une valeur de type τ_i. De façon symétrique, la réception consiste en un choix externe entre les messages f_i. L'élément de syntaxe $\chi : p$ permet de définir un point de récursion que l'on peut atteindre par χ : les boucles sont possibles de cette façon. Enfin, 0 représente la fin de l'exécution d'un rôle (cet élément de syntaxe est cependant fréquemment omis lorsqu'il n'y a pas d'ambiguïté). Un rôle terminant une session retourne une valeur du type τ_i donné par la définition du processus $r_i : \tau_i = p_i$. Remarquons que les annotations de type sont en général omises lorsque le type est `unit`. Notons enfin que, comme nous l'avons vu dans les exemples précédents, notre syntaxe concrète utilise les mots-clés session et role devant les définitions de sessions et rôles.

Une collection de processus locaux arbitraires ne forme pas automatiquement une session. Pour cela, certaines propriétés de cohérence sont nécessaires. Ces propriétés sont vérifiées lors de la reconstruction du graphe de la session.

3.2.5 Reconstruction des sessions globales

Pour permettre la reconstruction du graphe de la session, la collection de processus locaux doit au minimum vérifier les propriétés suivantes :

- chaque étiquette n'apparaît que dans un seul envoi et dans une seule réception, et ceci dans des rôles différents ;
- chaque χ se rapporte à un sous-processus $\chi : p$ du même rôle ;
- chaque processus consiste en une alternance stricte d'envois et de réceptions ;
- p_0 commence par un envoi.

Ces propriétés sont les équivalentes pour les processus locaux des propriétés 1-2.

Étant donnés les processus décrivant les rôles d'une session $\Sigma = (r_i : \tau_i = p_i)_{i<n}$, et supposant que les propriétés ci-dessus soient vérifiées, il est possible de construire un graphe $\mathcal{G}(\Sigma) = \langle \mathcal{R}, \mathcal{V}, \mathcal{L}, m_0, \mathcal{E}, r \rangle$ par la méthode suivante.

- Pour chaque sous-processus $\mathrm{send}(f_i : \tau_i\,;\, p_i)_{i<k}$ d'un rôle r de Σ, un nœud $m_{(f_i)_{i<k}}$ est créé, et $R(m_{(f_i)_{i<k}})$ est le rôle r. Le nœud m_0 correspond au processus p_0. Pour

chaque sous-processus recv$[f_i : \tau_i \to 0]_{i<k}$ d'un rôle r de Σ, un nœud $m^0_{(f_i)_{i<k}}$ est aussi créé et $R(m^0_{(f_i)_{i<k}})$ est le rôle r. L'ensemble de ces nœuds forme l'ensemble $\mathcal{V}$.
- Pour chaque étiquette f_k, il existe un unique sous-processus send$(f_i : \tau_i\,;\, p_i)_{i\in I}$ de Σ tel que $k \in I$ et une unique réception recv$[f_j : \tau_j \to q_j]_{j\in J}$ telle que $k \in J$: nous pouvons donc créer les arêtes $(m_{(f_i)_{i<k}}, f_k, m)$ où m est le nœud correspondant à q_k. Si q_k n'est ni un envoi, ni le processus nul, les cas suivants s'appliquent : si q_j est de la forme $\mu\chi.q$, on prend le nœud correspondant à q ; si $q_j = \chi$, alors il existe un processus $\mu\chi.q$ dont nous prenons le nœud . Ces arêtes définissent l'ensemble $\mathcal{E}$.
- $\mathcal{R} = \{r_i\}_{i<n}$ est le même ensemble de rôles et $\mathcal{L}$ l'ensemble des étiquettes rencontrées.

Si le graphe obtenu est connexe et si chaque nœud y est accessible à partir du nœud initial, alors les processus locaux forment une définition correcte d'une session. De façon symétrique, étant donné le graphe global d'une session, une projection standard [42] permet de retrouver les processus locaux.

Après avoir expliqué comment spécifier les sessions, la section suivante détaille la manière dont les programmeurs peuvent les utiliser dans leurs programmes ML.

3.3 Programmer avec des sessions

Le point de départ de la programmation avec les sessions est la description d'une session. Cette description est contenue dans un fichier dont l'extension est `.session`. Nous avons vu plusieurs exemples de tels fichiers dans la section précédente.

Le fichier de session est compilé en utilisant notre compilateur `s2ml`, dont le résultat est la production de plusieurs fichiers. Pour une session `S`, sont créés les fichiers suivants :
- `S.ps` est un fichier généré avec `dot` [28] qui contient l'image du graphe global de la session. Ce graphe permet à l'utilisateur de vérifier si la programmation de la session par processus locaux correspond à l'idée initiale qu'il en avait.
- `S.mli` contient l'interface d'un module ML et représente l'API que le programmeur peut utiliser pour démarrer ou rejoindre une session.
- `S.ml` contient l'implémentation sécurisée de la session. Celle-ci fait appel à un certain nombre de librairies pour le réseau, la cryptographie et les principaux. Leur description aura lieu en section 3.6.

La façon dont le compilateur `s2ml` génère ces fichiers est détaillé dans la section 3.7.

Exemple 3.9 (Compilation de Rpc en ligne de commande) Partons du fichier `Rpc.session` dont le contenu est présenté dans l'exemple 3.3.

Le fichier est compilé par la commande :

```
$> s2ml Rpc.session
```

Trois fichiers, appelés `Rpc.ml` et `Rpc.mli`, sont créés. Le premier correspond à l'implémentation sécurisée de la session `Rpc` et le second à l'API que l'utilisateur est invité à utiliser.

Le troisième fichier, `Rpc.ps`, est une représentation du graphe réalisée à l'aide de l'utilitaire `dot` [28]. C'est cette représentation qui est utilisée dans la figure 3.2 pour illustrer l'exemple 3.3.

La seconde phase de la programmation avec les sessions consiste à écrire en ML un programme appelant l'API ainsi générée. Ce programme sera ensuite compilé de façon usuelle, puis lié au module généré ainsi qu'aux bibliothèques utilisées pour le réseau et la cryptographie.

Cette section est dédiée à cette seconde phase : l'écriture de programmes faisant appel aux sessions. Nous commençons par la gestion des principaux, puis nous décrivons l'interface du module généré, et nous terminons par des exemples complets d'utilisation de session.

3.3.1 Principaux

Les principaux sont les entités élémentaires du réseau qui jouent les rôles des sessions. Pour permettre au protocole généré de fonctionner, les principaux doivent pouvoir être associés à leur adresse (IP, port) et clés (publiques et privées). C'est pour cette raison que l'utilisateur d'une session doit déclarer les principaux concernés avant de commencer l'exécution.

Nous mettons à la disposition de l'utilisateur une bibliothèque, nommée `Prins`, pour gérer les principaux (voir section 3.6). Celle-ci contient notamment une fonction `register` dont le type est le suivant :

```
val register : string → string → string → int → unit
```

Un appel à `register id filename inet port` enregistre un principal de nom `id`, dont les informations cryptographiques sont dans le fichier `filename`, dont l'adresse IP et le port sont respectivement `inet` et `port`. Un alias est créé pour pouvoir utiliser les noms de principaux sous forme de chaîne de caractères comme identifiants :

```
type principal = string
```

Exemple 3.10 (Principaux) L'enregistrement de trois principaux `"alice"`, `"bob"` et `"charlie"` pour exécuter la session `Shopping` de l'exemple 3.7 peut s'effectuer de la façon suivante :

```
Prins.register "alice" "alice.key" "193.55.250.70" 8765 ;
Prins.register "bob" "bob.cer" "193.55.250.71" 8765 ;
Prins.register "charlie" "charlie.cer" "193.55.250.72" 8766
```

Le programmeur précise ici qu'il a connaissance de la clé privée (`alice.key`) d'`"alice"` alors qu'il n'a connaissance que des clés publiques de `"bob"` et `"charlie"` (données sous la forme de certificats X.509 [53]).

Les fichiers contenant les informations cryptographiques peuvent être de plusieurs sortes. Il peut s'agir d'un fichier donnant la clé privée (pour les principaux dont le programmeur veut assumer l'identité) ou d'un fichier donnant seulement la clé publique (pour les principaux distants). Les clés publiques peuvent être fournies sous la forme d'un certificat X.509 [53] ou d'un simple fichier `.pub` [20].

Nous décrivons maintenant l'interface de programmation des sessions, avant de donner quelques exemples d'utilisation.

3.3.2 Interface générée

Nous détaillons ici l'interface ainsi que le schéma général de sa génération. Notons tout d'abord qu'il s'agit de l'interface d'un seul module, dont le nom est le même que celui de la session déclarée. Une session dont la déclaration commence par « session `S =` » donnera lieu à la génération d'un module ML appelé `S`, indépendamment du nom du fichier contenant la déclaration de session. Un fichier de session peut plus généralement contenir plusieurs déclarations de session.

Nous examinons l'interface de ce module généré. La génération du code source de ce module est présentée dans la section 3.7.

Principaux

Le premier type déclaré dans chacune des interfaces générées est le type `prins`. Ce type est un enregistrement spécifique à chaque déclaration de session qui associe chaque rôle à un principal.

Exemple 3.11 (Principaux - Rpc, Shopping) La déclaration du type `prins` pour la session `Rpc` est la suivante :

```
type prins = {
  prins_c : principal ;
  prins_w : principal }
```

Dans le cas de la session `Shopping`, la déclaration est simplement :

```
type prins = {
  prins_c : principal ;
  prins_o : principal ;
  prins_w : principal }
```

Le schéma général de la génération de ce type est le suivant :

```
type prins = {
  Pour chaque rôle r ∈ R, [ prins_r : principal ; ]
}
```

Ce type permet de collecter l'ensemble des principaux participant à une instance donnée d'une session. Nous verrons son utilité par la suite.

Rôles

Le reste de l'interface comprend la déclaration d'un ensemble de types et d'une fonction pour chaque rôle. La fonction permet de jouer le rôle en question, tandis que le pilotage de la session à l'exécution est réalisé par une série de continuations contraintes par les types déclarés.

Le principe du pilotage de la session est ainsi de demander à l'utilisateur des continuations successives pour analyser le contenu des messages reçus et choisir le message à envoyer en réponse. Nous commençons notre présentation par quelques exemples.

Exemple 3.12 (Rpc - Interface) La session `Rpc` ne contient que deux messages. Le rôle `w` reçoit un message `Request` et renvoie un message `Reply`. L'interface générée pour ce rôle est la suivante :

```
(* Function for role w *)
type result_w = unit
type msg3 = {
   hRequest : (prins * string  → msg4)}
and msg4 =
   Reply of (int * result_w)
val w : principal  → msg3  → result_w
```

Le type `result_w` est un alias pour le type de retour de la fonction `w` (le type déclaré par le rôle : `role w : unit = ...`). Celle-ci attend deux arguments : le premier est le nom du principal souhaitant jouer ce rôle, le second est une valeur de type `msg3`. Le type `msg3` est un type enregistrement à un seul champ `hRequest` dont le type est `(prins * string → msg4)`. Il s'agit de la continuation, fournie par l'utilisateur, qui sera appelée une fois le message `Request` reçu avec comme arguments la liste des principaux jouant les rôles et la chaîne de caractères correspondant au contenu du message. Le type `msg4`, quant à lui, est un type algébrique à un constructeur, `Reply`, dont les arguments sont un entier (le contenu du message `Reply`) et la valeur de retour du rôle `w`.

Examinons maintenant l'interface correspondant au rôle `c`.

```
(* Function for role c *)
type result_c = int
type msg0 =
  Request of (string * msg1)
and msg1 = {
  hReply : (prins * int → result_c)}
val c : prins → msg0 → result_c
```

Le type `result_c` est maintenant un alias pour `int`, le type de retour de la fonction `c` (et donc du rôle `c`). Pour jouer ce rôle, initiateur de la session, la fonction `c` demande comme premier argument une valeur de type `prins`, c'est-à-dire un enregistrement donnant les identités de tous les participants à la session. Le deuxième argument est de type `msg0` : comme le rôle envoie le premier message `Request`, l'utilisateur doit fournir, à l'aide du constructeur `Request`, la valeur de type `string` qu'il veut transmettre. Comme la session attend une réponse de la part du rôle `w`, l'utilisateur doit aussi fournir une continuation de type `msg1`. Cette continuation sera appelée à la réception du message `Reply` avec la valeur transmise par `w` comme argument. Cette continuation a pour objet de calculer la valeur de retour de la session pour le rôle `c`.

Trois éléments sont générés pour chaque rôle : un alias pour le type de retour, un ensemble de types mutuellement récursifs spécifiant les types des continuations et contenus des messages, et une fonction permettant à l'utilisateur de jouer le rôle donné.

Le type de retour est le type mentionné dans la déclaration d'un rôle. Le schéma général pour une session Σ est le suivant :

Pour chaque rôle $r \in \mathcal{R}$, *tel que* $(r : \tau = p) \in \Sigma$
[type `result_`r `=` τ]

Les types des continuations et contenus des messages sont numérotés d'une façon qui ne se déduit pas immédiatement de la structure de la session : ce code numérique sans signification particulière est utilisé par le compilateur lors de la génération du code et ne transparaît pour l'utilisateur que lors de la définition des types dits de *pilotage*. Ils sont de la forme `msg`n avec n un entier. Nous notons n_r le numéro du type du premier ensemble de continuations à fournir pour le rôle r. Nous pouvons maintenant donner la forme générale des déclarations des fonctions permettant de jouer les rôles d'une session.

val r_0 `: prins → msg0 → result_`r_0
Pour chaque rôle $r \neq r_0 \in \mathcal{R}$
[val r `: principal → msg`n_r `→ result_`r]

Le premier rôle a pour tâche de désigner les autres participants : la fonction r_0 réclame donc une valeur de type `prins` alors que les autres rôle se contentent de préciser leur propre identité de type `principal`. Le second argument de ces fonctions est d'un type de pilotage unique `msg`n_r à chaque rôle. Comme nous l'avons vu, cet argument est un ensemble de continuations effectuant le choix des messages et de leurs contenus (dans les limites de la spécification de la session). Nous examinons maintenant la structure des types des continuations (de la forme `msg`n). Nous commençons à en illustrer la génération par un exemple.

Exemple 3.13 (Ws - Interface) Nous avons vu dans l'exemple précédent que, pour la session à deux messages `Rpc`, deux types de pilotage étaient déclarés pour chaque rôle. La structure de ces types suit les actions que chacun des rôles doit effectuer : les envois deviennent des types algébriques, les réception des ensembles de continuations. Examinons le cas du rôle `w` de la session avec choix `Ws`.

```
(* Function for role w *)
type result_w = string
type msg4 = {
   hRequest : (prins * string  → msg5)}
and msg5 =
   Reply of (int * result_w)
 | Fault of (unit * result_w)
val w : principal  → msg4  → result_w
```

Le rôle `w` a pour tâche dans cette session de recevoir le message `Request` puis de choisir entre renvoyer un message `Reply` ou un message `Fault`. C'est la continuation de type `msg4` de la fonction `w` qui détermine ce choix. Le type `msg4` est ainsi un enregistrement ne contenant qu'une fonction (la continuation en question) de type `prins * string  → msg5`. La signification de ce type est que la continuation donnée par l'utilisateur de la session se voit fournir les identités de tous les participants à la session (de type `prins`) ainsi que le contenu (de type `string`) du message `Request`, et doit produire une valeur de type `msg5`. Ce type est un type somme dont les constructeurs sont `Reply` et `Fault`. De cette manière, la spécification de la session est traduite en terme de types qui forment autant de contraintes que l'utilisateur de la session doit respecter.

Examinons les différences avec l'interface produite pour le rôle dual `c`.

```
(* Function for role c *)
type result_c = unit
type msg0 =
   Request of (string * msg1)
and msg1 = {
   hReply : (prins * int  → result_c) ;
   hFault : (prins * unit  → result_c)}
val c : prins  → msg0  → result_c
```

Là où `w` doit faire un choix interne entre deux messages, `c` doit, lui, être prêt à recevoir les deux. L'utilisateur doit ainsi fournir deux continuations : le type `msg1` est un enregistrement à deux champs, `hReply` et `hFault`. Ces champs contiennent chacun une fonction dont l'objet est de traiter les données reçues dans le message correspondant.

Notons que la liste des principaux est donnée comme argument à chaque continuation. L'utilisateur n'est bien sûr pas obligé d'en tenir compte.

Concernant les envois et réceptions, la structure des types de pilotage est donc, dans le cas général, la suivante : lorsque le processus d'un rôle est face à un choix interne (envoi de différents messages), le type en question est un type somme dont les constructeurs sont les étiquettes des messages ; lorsqu'il s'agit d'un choix externe (réception de différents messages), le type est un enregistrement contenant une fonction par message qu'il est possible de recevoir. Chacun de ces types contient (dans le cas d'un envoi) ou fournit (dans le cas d'une réception) le type continuation correspondant à la suite de l'exécution du processus ou le type de retour du rôle lorsque la session s'achève.

Examinons maintenant le cas des boucles. Nous commençons avec un exemple.

Exemple 3.14 (Wsn - Interface) Dans la session `Wsn` de l'exemple 3.6, le rôle `c`, après avoir reçu un message `Reply` de la part de `w`, envoie un message `Extra` qui laisse la possibilité à `w` de choisir à nouveau l'envoi d'un message `Reply`.

Les types générés pour le rôle `c` sont les suivants :

```
(* Function for role c *)
type result_c = unit
type msg0 =
   Request of (string * msg1)
and msg1 = {
   hReply : (prins * int  → msg2) ;
   hFault : (prins * unit  → result_c)}
and msg2 =
   Extra of (string * msg1)
val c : prins  → msg0  → result_c
```

Nous observons que les types `msg1` et `msg2` sont mutuellement récursifs. Le type `msg2` correspondant au message `Extra` a pour continuation une valeur de type `msg1`, c'est-à-dire un enregistrement dont la composante `hReply` retourne une expression de type `msg2`.

Le point de récursion présent dans la session donne ainsi des types de pilotage mutuellement récursifs. La façon concrète de convertir le processus de `w` vers son type est la suivante : au point de récursion `loop:`, on associe un entier $n_{\text{loop}} = 1$ correspondant au type `msg1` du processus `send ( Reply : int ; recv Extra : string  → loop + Fault )` qu'il dénote. Au moment de l'appel récursif utilisant `loop`, le type associé au processus est alors `msg1`.

Les points de récursion dans les spécifications de session se traduisent ainsi par des types de pilotage mutuellement récursifs. Nous présentons maintenant la formalisation de la génération des types de pilotage. Cette génération se fait en deux étapes.

Syntaxe des processus nommés

L'étape initiale est d'associer un entier χ à chaque point de l'arbre de syntaxe de chacun des rôles, les points de récursion étant substitués de façon cohérente. La syntaxe des processus obtenus devient alors :

$\chi ::= 0, 1, 2, ...$		noms de sous-processus
$\tau ::=$ unit $\mid$ int $\mid$ string		types de base
$p' ::= 0$		fin de processus
$\mid$	$\chi : \text{send}(f_i : \tau_i \,;\, p'_i)_{i<k}$	envoi
$\mid$	$\chi : \text{recv}[f_i : \tau_i \rightarrow p'_i]_{i<k}$	réception
$\mid$	χ	continuation vers un sous-processus
$\Sigma ::= (r_i : \tau_i = p'_i)_{i<n}$		session (processus initiaux pour chaque rôle)

Illustrons cette manipulation sur un exemple.

Exemple 3.15 (Wsn - Renommage) Dans la session `Wsn` de l'exemple 3.6, le rôle `c` est décrit de la manière suivante :

```
send Request : string;
loop:
recv
[ Reply : int  →
  send Extra : string ; loop
| Fault ]
```

La numérotation de chaque point de l'arbre de syntaxe donne le processus nommé suivant :

```
0 : send Request : string;
1 : recv
    [ Reply : int  →
      2 : send Extra : string ; 1
    | Fault ]
```

Nous observons que le point de récursion `loop:` a été substitué par le numéro `1`, de même que l'appel `loop` situé après l'envoi du message `Extra`.

Fonction msg_r

Définissons ensuite la fonction msg_r qui à un processus nommé p' d'un rôle r donné associe un nom de type. Formellement, msg_r se définit par :

$$\begin{aligned}
\text{msg}_r(\chi : \text{send}(_)) &= \text{msg}\chi \\
\text{msg}_r(\chi : \text{recv}[_]) &= \text{msg}\chi \\
\text{msg}_r(\chi) &= \text{msg}\chi \\
\text{msg}_r(0) &= \texttt{return_}r
\end{aligned}$$

La seconde étape de la génération des types de pilotage consiste en l'application pour chaque rôle d'une fonction parcourant récursivement la structure de son processus nommé. Cette fonction de génération est désignée, lorsqu'elle s'applique au rôle r, par des chevrons $\langle \_ \rangle^r$, et produit un ensemble de déclarations de type.

Génération des types $\text{msg}n$

Pour chaque rôle $r \in \mathcal{R}$, *tel que* $(r : \tau = p') \in \Sigma$

$$\langle \chi : \text{send}(f_i : \tau_i \,;\, p_i)_{i<k} \rangle^r = \{\texttt{type msg}\chi \texttt{ = } \left[\texttt{|}\, f_i \texttt{ of } \tau_i \texttt{ * msg}_r(p_i)\right]_{i<k}\} \cup \langle p_i \rangle^r_{i<k}$$

$$\langle \chi : \text{recv}[f_i : \tau_i \rightarrow p_i]_{i<k} \rangle^r = \{\texttt{type msg}\chi \texttt{ = } \texttt{\{h}f_i \texttt{ = prins*}\tau_i \texttt{ → msg}_r(p_i)\texttt{;\}}_{i<k}\} \cup \langle p_i \rangle^r_{i<k}$$

$$\langle \chi \rangle^r = \langle 0 \rangle^r = \varnothing$$

Le processus χ : send$(f_i : \tau_i\,;\, p_i)_{i<k}$ donne ainsi lieu à la génération d'un type somme `msg`χ `= [ |`f_i `of` τ_i `*` `msg`$_r(p_i)$`]`$_{i<k}$ suivi de la génération des types associés aux sous-processus : $\langle p_i \rangle^r_{i<k}$. De la même façon, le processus χ : recv$[f_i : \tau_i \rightarrow p_i]_{i<k}$ permet de générer le type enregistrement `msg`χ `= {h`f_i`= prins *` τ_i $\rightarrow$ `msg`$_r(p_i)$`;}`$_{i<k}$ suivi par les types associés aux sous-processus : $\langle p_i \rangle^r_{i<k}$. Aucune définition de type n'est générée pour les sous-processus χ et 0.

Les figures 3.22 et 3.23 de la section 3.7 récapitulent les différents éléments de la génération de l'interface.

3.3.3 Exemples de code utilisateur

Donnons maintenant quelques exemples simples d'utilisation de cette interface. Commençons par un simple appel à la session `Rpc`.

Exemple 3.16 (Rpc - Pilotage) La session `Rpc` de l'exemple 3.3 ne comprend que deux messages `Request` et `Reply`. Le pilotage du rôle `w` ne s'effectue qu'à l'aide d'une seule continuation, dont voici un exemple très simple :

```
Rpc.w "Bob" {hRequest = function (_,_)  → Reply(42,())}
```

Une fois l'enregistrement du principal `"Bob"` effectué (voir section 3.3.1), une seule ligne suffit à l'utilisateur pour jouer le rôle de `w` dans une instance de la session `Rpc`. La fonction `Rpc.w` est appelée avec comme arguments le nom du principal jouant `w` et l'enregistrement contenant la continuation : celle-ci est une fonction prenant un couple en argument (dont la première composante correspond à la liste des principaux participant à la session et la deuxième au contenu du message `Request`) et retournant d'une part le contenu du message `Reply` (ici la constante `42`) et le résultat de la fonction `w` (ici `()` de type `unit`).

Donnons maintenant un exemple mettant en jeu l'utilisation du contenu transmis dans les messages.

Exemple 3.17 (Ws - Pilotage) La session `Ws` de l'exemple 3.5 a pour principale caractéristique le fait que le rôle `w` doit choisir entre répondre `Fault` ou `Reply` au message `Request`. La continuation que le programmeur qui souhaite jouer `w` doit utiliser peut alors déterminer son chemin en fonction du contenu du message `Request`.

```
let  res = Ws.w "Bob"
             {hRequest = function (_,q)  →
                      if  q = "Answer?"
                      then Reply(42,"Sent.")
                      else  Fault((),"Wrong question.")} in
print_string res
```

Ici, la valeur transmise par le rôle `c` dans le message `Request` est lue par la continuation (dans la variable `q`) et le choix du message suivant `Reply` ou `Fault` est alors basé sur un calcul arbitrairement complexe (pouvant utiliser `q`). Notons aussi dans l'exemple, la valeur de retour de type `string` (ici pouvant être `"Sent"` ou `"Wrong question."`) et son utilisation.

Examinons maintenant la programmation des sessions avec boucles.

Exemple 3.18 (Wsn - Pilotage) La session `Wsn` de l'exemple 3.6 comprend une boucle d'interaction lorsque le message `Extra` est envoyé par le client `c`. Les types de pilotage attendus par les fonctions `c` et `w` sont récursifs et si le programmeur souhaite effectuer plusieurs tours de boucle, il est tenu de fournir des fonctions récursives.

```
let rec handler_Extra query =
  if query = "Answer?"
  then Reply(42, {hExtra = function (_,query) → handler_Extra query})
  else Fault ((),"Wrong question.")
in
let res =
  Wsn.w "Bob"
    {hRequest = function (_,query) → handler_Extra query}
in print_string res
```

Le pilotage du rôle `w` s'effectue ainsi avec la fonction récursive `handler_Extra` qui examine le contenu de la requête (dans la variable query) et détermine si un message `Fault` (la session s'arrête) ou un message `Reply` (la session repart pour un tour) est envoyé.

Le code pilotant le rôle `c` fonctionne d'une manière similaire.

```
let rec handler_Reply sum counter =
  {hReply =
      (function (_,x) →
        if counter < 42
        then Extra ("Answer?",handler_Reply (sum+x) (counter+1))
        else Extra ("Question?",handler_Reply (sum+x) (counter+1)));
   hFault =
      (function _ → printf "The sum of my knowledge is %i.\n" sum)
  } in
Wsn.c
  {prins_c = "Alice"; prins_w = "Bob"}
  (Request ("Answer?",handler_Reply 0 0))
```

La continuation `handler_Reply` reçoit ici des paramètres supplémentaires qui lui permettent d'accumuler des résultats. Le même effet est réalisable à l'aide de références : le programmeur est libre de choisir le style de programmation qui lui convient.

En même temps que la valeur transmise par les messages, les continuations prennent toujours comme argument un enregistrement contenant les noms des principaux participants à la session. Cet argument est souvent redondant, mais il est indispensable lorsqu'il s'agit de rejoindre une session.

Exemple 3.19 (Shopping - Pilotage) Le rôle `o` dans la session `Shopping` représente un observateur qui peut certifier que la session s'est déroulée comme prévu. Le principal jouant le rôle `o` est cependant choisi par le client `c` au début de la session. Le rôle `w` peut ainsi décider de refuser de procéder si le choix de `o` ne lui convient pas.

```
let offer loc = List.assoc loc
  [ "Paris", "Paris, 9am-10am";
    "Orsay", "Orsay, lunchtime";
    "Cambridge", "Cambridge, 6pm-7pm" ] in
let server (peers,req) =
  if peers.prins_o <> "Bob"
  then failwith "I only like Bob."
  else
    let rec new_offer (_,loc) =
    try
      let appointment = offer loc in
      Offer(appointment, {
          hChange = new_offer;
          hAccept = (fun _ → Confirm((),"in "^appointment)); })
    with _ → Reject("No offer available","No offer available") in
    new_offer (prins,"Paris")
in
Shopping.w "Bob"
  { hContract = server; }
```

Dans le code ci-dessus, le rôle `w` traite le message `Contract` grâce à la continuation `server`. Cette fonction commence par examiner le nom du principal jouant le rôle `o`. Si celui-ci n'est pas `"Bob"`, une exception (`failwith "I only like Bob."`) est lancée et la session s'arrête. Sinon, la session continue avec une boucle de négociation permettant d'associer à un lieu (par exemple `"Paris"`), une heure de livraison (par exemple `"9am-10am"`).

En annexe C.4, un exemple détaillé d'exécution de session (code source, code généré, traces) est présenté.

3.3.4 Rôle du typage

L'interface générée à partir des descriptions locales des sessions repose sur le système de types usuel de ML. Elle fait notamment appel aux fonctions d'ordre supérieur et aux types algébriques (somme et produit). Le compilateur traduit la structure du processus local de la session vers le langage des types de ML.

Nous utilisons ce système de types de manière à faciliter la tâche de l'utilisateur des sessions : les types agissent à la fois comme documentation et comme contrainte statique qui interdit à l'utilisateur de ne pas respecter le rôle qu'il souhaite jouer. Le système de types de ML est amplement suffisant pour réaliser ces deux buts.

Comme le système de types permet de contraindre le code de l'utilisateur à respecter localement la session, nous avons la propriété (informelle) suivante.

Conjecture 3.1 (Correction locale par typage) Lorsque tous les participants à une session sont bien typés, les traces observées des messages échangés respectent la spécification de la session.

Cette conjecture correspond au résultat usuel obtenu par les types de session appliqués à certains calculs de processus. Nous avons cependant choisi de focaliser notre effort vers la situation non-idéale, c'est-à-dire celle où certains participants ne sont pas dignes de

confiance. Lorsque des rôles peuvent être compromis, l'hypothèse que l'ensemble des participants est bien typé ne tient plus et c'est pourquoi, dans la suite, la sémantique que nous utilisons pour décrire le comportement des sessions et du langage ML sous-jacent n'est pas typée.

3.4 Intégrité des sessions

Nous avons vu précédemment qu'une session était la description des séquences acceptables de messages échangés entre plusieurs rôles. Nous avons aussi examiné la façon dont les programmeurs peuvent tirer parti de l'abstraction que les sessions fournissent. Nous nous interrogeons maintenant sur le comportement de la session auquel les participants honnêtes s'attendent lorsque certains rôles ne sont plus dignes de confiance. Il s'agit de définir précisément quelle propriété l'implémentation sécurisée doit satisfaire.

3.4.1 Vers une propriété d'intégrité

Une session décrit un ensemble de séquences acceptables de messages que les participants s'engagent à respecter. Cela signifie que toutes les traces qu'un observateur peut observer sont censées s'accorder à la spécification. Cette idée sert de base à notre réflexion sur la sécurisation des sessions : l'exécution d'une implémentation sécurisée d'une session garantit que les seules traces observées sont conformes et, par contraposée, que toute déviation est détectée. La question de l'observation des messages échangés devient alors cruciale mais se heurte à des considérations pratiques : une implémentation locale ne peut s'attendre à observer tous les messages qui circulent sur un réseau (même si l'adversaire en a, lui, le pouvoir).

Plaçons-nous donc dans le cadre où l'observation des messages sur le réseau n'est pas possible : un des rôles participant à la session ne peut alors observer que les messages auxquels il prend directement part, c'est-à-dire les messages qu'il envoie et ceux qu'il reçoit. Dans une session binaire (à deux participants), cela suffit pour s'assurer dynamiquement que l'exécution de la session est conforme car les messages ne sont échangés qu'entre les deux rôles. Dans les autres cas, seule une vue de la session restreinte aux messages reçus et envoyés par le rôle en question est directement accessible à celui-ci. Une implémentation sécurisée des sessions ne peut ainsi dans un premier temps que s'assurer de la compatibilité des messages envoyés et reçus par rapport à la spécification de la session projetée pour le rôle en question, c'est-à-dire par rapport au processus local.

Cette propriété d'*intégrité locale* n'est cependant qu'une première étape car les sessions sont une spécification globale : les processus locaux correspondent les uns aux autres. Deux rôles honnêtes observant la même exécution de la session à laquelle ils participent s'attendent à obtenir, d'une part, des résultats cohérents avec le graphe global de la session et, d'autre part, des résultats cohérents entre les visions partielles qu'il en ont. C'est cette propriété d'*intégrité globale* ou plus simplement d'*intégrité* que l'implémentation sécurisée a la responsabilité de réaliser. L'objet de cette section est de définir formellement cette notion.

La stratégie que nous adoptons consiste à définir deux sémantiques pour un langage de programmation distribué avec sessions. La première sémantique impose aux sessions de respecter leur spécification et de ce fait garantit par définition leur intégrité. La seconde sémantique traite les sessions comme des appels à leur implémentation sous-jacente. L'objet de notre théorème d'intégrité est alors de lier les traces observables à haut-niveau (avec sessions) aux traces de bas-niveau (avec l'implémentation sécurisée).

Commençons par décrire précisément dans quel cadre formel nous nous plaçons. Nous utilisons le langage ML à la fois comme base pour y ajouter les sessions (langage de haut-niveau) et comme langage de bas-niveau pour les y implémenter. Nous formalisons donc le cœur de ML auquel nous ajoutons des déclarations et appels de sessions. Nous donnons une sémantique étiquetée à ce langage, ce qui permet de formaliser le comportement des sessions et de préciser la notion de traces (pour les sessions et les communications ordinaires). Nous utilisons cette sémantique pour énoncer précisément la propriété d'intégrité que nous attendons d'une implémentation sécurisée des sessions.

3.4.2 Une syntaxe formelle pour les sessions en ML

Nous incorporons notre langage de session à ML. Plus précisément, nous utilisons (un sous-ensemble de) la syntaxe concrète d'Ocaml et F#, à laquelle nous ajoutons la possibilité de définir des sessions. Nous appelons ce langage F+S : il comprend à la fois des fonctions de communication par canaux et des primitives d'ouverture de session. Lorsque les sessions sont absentes (les éléments à enlever sont marqués d'une dague †), le langage est appelé F (cf [9]). La figure 3.9 donne la syntaxe des langages F et F+S.

Les types T comprennent des variables de type t (représentant les types algébriques), des types de base int , string , unit, des types de canaux T chan (où T est le type des données échangées sur le canal), et des types fonctionnels $T_1 \to T_2$. Les types canaux ne sont présents que pour des raisons de compatibilité avec les bibliothèques ML que nous utilisons. Notre sémantique n'est en fait pas typée, permettant en cela la modélisation d'attaquants arbitraires.

Les valeurs v sont formées des constantes, fonctions et termes formés par les constructeurs des types algébriques. Nous supposons données un certain nombre de constantes de principaux, comme `Alice` et `Bob`.

Les expressions comprennent une expression inerte (ne se réduisant pas) **0** représentant la terminaison d'un thread. Notre langage possède de plus quatre primitives de communication similaires à celles du π-calcul : new, ! (envoi), ? (réception) et fork ; dont nous donnons la sémantique spécifique dans la section 3.4.3. Les structures de données usuelles (booléens, n-uplets, listes, enregistrements) sont accessibles via des bibliothèques. Leur définition est standard et donc omise ici.

Les sessions (décrites par le langage présenté dans la figure 3.8) sont utilisées de la façon suivante dans F+S : la déclaration session $S = \Sigma$ in permet de définir une session, puis l'expression $S.r^b\ \widetilde{a}\ (e)$ permet de la démarrer.

Dans le cas où r est le rôle initial r_0 de la session, le premier argument, $\widetilde{a}$, est un n-uplet de principaux (en pratique, l'enregistrement de type `prins` dans les exemples que nous avons vus) qui détermine qui joue chacun des rôles de la session. Si r n'est pas le premier rôle de la session, $\widetilde{a}$ est juste le nom du principal jouant le rôle r.

Dans les deux cas, e est une expression spécifiant l'exécution de la session, notamment le choix et le contenu des messages. Si le rôle doit recevoir un message, e est un ensemble de continuations, c'est-à-dire un n-uplet (en pratique, un enregistrement) comprenant une continuation pour chaque message que le rôle peut recevoir à ce point d'exécution de la session. Si le rôle est censé envoyer un message, e est une valeur d'un type algébrique dont le constructeur correspond à l'étiquette du message et dont l'argument est un couple composée du contenu du message et d'un éventuel ensemble de continuations si des réponses sont attendues. Cette syntaxe concise est très similaire à la syntaxe concrète utilisée à la section 3.3.

Notre syntaxe pour l'appel d'une session $S.r^b\ \widetilde{a}\ (e)$ dans F+S coïncide avec la syntaxe d'un simple appel de fonction de ML dans F, où S est le nom du module implémentant

$T ::=$		Types
	unit $\mid$ int $\mid$ string	types de base
	t	variable de type
	T chan	type de canal
	$T_1 \to T_2$	type fonctionnel
$v ::=$		Valeurs (aussi utilisées comme motifs)
	x	variable
	`0`, `1`, …, `Alice`, `Bob`, …, ()	constantes pour les types de base
	$l, c, n, \ldots$	noms de fonctions, canaux, nonces
	$f(v_1, \ldots, v_k)$	constructeur (d'arité k)
$e ::=$		Expressions
	v	valeur
	$l\, v_1 \ldots v_k$	application d'une fonction
	match v with $(\mid v_i \to e_i)_{i<k}$	filtrage
	let $x = e_1$ in e_2	définition de valeur
	let $(l_i\, x_0 \ldots x_{k_i} = e_i)_{i<k}$ in e	définition de fonctions
	type $(t_i = (\mid f_{j_i}$ of $T_{j_i})_{j_i<k_i})_{i<k}$ in e	définition de types
	session $S = \Sigma$ in e	définition de session †
	$S.r^b\, \widetilde{v}\, (v)$	ouverture de session †
	$s.p(e)$	rôle d'une session $^\dagger, \star$
	0	expression inerte $\star$
$E[\cdot] ::=$		Contextes d'évaluation
	$[\cdot]$	Contexte global
	let $x = E[\cdot]$ in e	évaluation séquentielle
	$s.p(E[\cdot])$	évaluation au sein d'une session $^\dagger, \star$
$P ::=$		Processus
	e	thread
	$P \mid P$	composition parallèle

(les expressions et contextes marqués d'une dague † sont exclusif à F+S, ceux marqués d'une étoile $\star$ ne sont présents qu'à l'exécution)

FIG. 3.9 – Syntaxe d'F et F+S

la session, r le nom d'une fonction de ce module, et e une expression usuelle. La marque b de $S.r^b$ est une indication qui est mis à $\bullet$ lorsque le rôle r est joué par l'adversaire ; ce marqueur est uniquement utilisé pour énoncer les propriétés d'intégrité de session dans le cas où certains principaux sont corrompus ; il s'agit d'un artifice de la sémantique.

À l'exécution, l'appel d'une session se réduit vers l'expression correspondant à un rôle actif $s.p(e')$, dans laquelle s est un identifiant unique de session, p est le processus courant du rôle (dont nous avons donné la syntaxe à la figure 3.8) et e' est l'expression gouvernant la prochaine action de la session : choix et contenu du message si le rôle envoie le prochain message ; continuation s'il attend d'être contacté.

3.4.3 Sémantique

Dans cette section, nous décrivons formellement le comportement des langages F+S et F à l'aide d'une sémantique étiquetée.

Processus locaux

Le comportement des processus décrivant les rôles d'une session est donné par une sémantique étiquetée de la forme $p \xrightarrow{\eta}_{\mathsf{r}} p'$, où les étiquettes η sont soit de la forme f (pour une réception), soit $\overline{f}$ (pour un envoi), avec f l'étiquette du message. Nous prenons le point de vue équi-récursif pour traiter la récursion : les processus sont développés en un arbre infini de manière à n'avoir que deux règles pour l'envoi et la réception [59, § 21.8].

$$\text{(Send)}\quad \mathrm{send}(f_i : \tau_i\,;\, p_i)_{i<k} \xrightarrow{\overline{x} f_i}_{\mathsf{r}} p_i \qquad \text{(Receive)}\quad \mathrm{recv}[f_i : \tau_i \to p_i]_{i<k} \xrightarrow{f_i}_{\mathsf{r}} p_i$$

FIG. 3.10 – Sémantique des processus locaux

Les traces de cette sémantique étiquetée représentent les séquences d'actions possibles pour chacun des rôles.

Exemple 3.20 (Traces de Ws) Le rôle `c` de la session `Ws` (voir exemple 3.5) peut effectuer les deux séquences suivantes :

$$\mathrm{send}\ \mathrm{Request{:}string};\ \mathrm{recv}[\,\mathrm{Reply{:}int}\,|\,\mathrm{Fault}\,] \xrightarrow{\overline{\mathrm{Request}}}_{\mathsf{r}} \mathrm{recv}[\,\mathrm{Reply{:}int}\,|\,\mathrm{Fault}\,] \xrightarrow{\mathrm{Reply}}_{\mathsf{r}} 0$$
$$\mathrm{send}\ \mathrm{Request{:}string};\ \mathrm{recv}[\,\mathrm{Reply{:}int}\,|\,\mathrm{Fault}\,] \xrightarrow{\overline{\mathrm{Request}}}_{\mathsf{r}} \mathrm{recv}[\,\mathrm{Reply{:}int}\,|\,\mathrm{Fault}\,] \xrightarrow{\mathrm{Fault}}_{\mathsf{r}} 0$$

Les traces $\overline{\mathrm{Request}}\mathrm{Reply}$ et $\overline{\mathrm{Request}}\mathrm{Fault}$ correspondent aux actions que le rôle `c` peut effectuer.

Une fois la sémantique des processus définie, nous pouvons en déduire la sémantique des sessions.

Evaluation des expressions de session

Nous définissons ici les transitions permettant de gérer les sessions. Ces transitions sont de la forme $\rho, \sigma \xrightarrow{\eta}_{\mathsf{s}} \rho', s.p$. Nous désignons par la métavariable σ les expressions de session suivantes.

$\sigma ::=$		Sessions
	$\mid S.r^b\,\widetilde{a}\,(e)$	appel de session
	$\mid s.p(e)$	session en cours d'exécution

σ désigne des expressions partielles de la forme $S.r^b\ \widetilde{a}$ ou $s.p$, c'est-à-dire des appels de sessions et des sessions en cours d'exécution.

Notre sémantique comporte un environnement global explicite, désigné par la métavariable ρ, qui garde la trace des noms générés, des définitions de fonction et de type et, dans F+S, des définitions de session ainsi que les informations correspondant aux sessions en cours d'exécution. L'environnement ρ est défini de la manière suivante.

$\rho ::=$	Environnement d'exécution
$\mid \{n\}$	noms
$\mid \{(t_i = (\mid f_{j_i} \text{ of } \widetilde{T_{j_i}})_{j_i<k_i})_{i<k}\}$	déclaration de type
$\mid \{(l_i\, x_0 \ldots x_{k_i} = e_i)_{i<k}\}$	déclaration de fonction
$\mid \{S = \Sigma\}$	déclaration de session
$\mid \{s\, \widetilde{a}\, (\delta) : S\}$	session
$\mid \rho \uplus \rho'$	union disjointe

Concrètement, ρ comprend des noms n, des définitions de type $(t_i = (\mid f_{j_i} \text{ of } \widetilde{T_{j_i}})_{j_i<k_i})_{i<k}$, des définitions de fonction $(l_i\, x_0 \ldots x_{k_i} = e_i)_{i<k}$, des définitions de session $S = \Sigma$, et les sessions en cours d'exécution $s\, \widetilde{a}\, (\delta) : S$ où s est l'identifiant de la session, $\widetilde{a}$ les principaux de chacun des rôles, δ est l'ensemble des rôles impliqués dans la session à ce point de l'exécution, et S est le nom de la session. Nous utilisons $\uplus$ pour désigner l'union disjointe lors d'extension de la mémoire ρ (si les domaines ne sont pas disjoints, nous procédons à un renommage partiel).

$$(\textsc{Init})\ \frac{p_0 \xrightarrow{\overline{g}}_{\mathsf{r}} p' \quad S = (r_i : \widetilde{\tau}_i = p_i)_{i<n} \in \rho \quad s \text{ fresh}}{\rho, S.r_0^b\, (a_i)_{i<n} \xrightarrow{\overline{g}}_{\mathsf{s}} \rho \uplus \{s\, (a_i)_{i<n}\, \{r_0\} : S\}, s.p'} \qquad (\textsc{Step})\ \frac{p \xrightarrow{\eta}_{\mathsf{r}} p'}{\rho, s.p \xrightarrow{\eta}_{\mathsf{s}} \rho, s.p'}$$

$$(\textsc{Join})\ \frac{p_j \xrightarrow{f}_{\mathsf{r}} p' \quad S = (r_i : \widetilde{\tau}_i = p_i)_{i<n} \in \rho \quad \rho' = \rho \uplus \{s\, (a_i)_{i<n}\, \delta : S\}}{\rho', S.r_j^b\, a_j \xrightarrow{f}_{\mathsf{s}} \rho \uplus \{s\, (a_i)_{i<n}\, (\delta \uplus \{r_j\}) : S\}, s.p'}$$

FIG. 3.11 – Sémantique des expressions de session

Les règles de réduction des expressions de session sont décrites dans la figure 3.11. La règle (INIT) démarre une session, ajoutant une nouvelle entrée $s\, (a_i)_{i<n}\, \{r_0\} : S$ dans ρ avec s un identifiant de session fraîchement généré. La règle (JOIN) requiert que (1) r_j soit un rôle de la session S pour un $j < n$; (2) S soit le nom de la définition de session instantiée par s ; (3) l'ensemble δ des rôles ayant déjà rejoint la session s ne contienne pas r_j ; et (4) le principal a_j rejoignant la session soit le même principal correspondant au rôle r_j dans s. L'étiquette f enregistre une des étiquettes pouvant être reçues par p_j en suivant la session S.

Nous plongeons maintenant ces règles au sein des règles de réduction des expressions.

Réduction des expressions

Notre présentation des transitions étiquetées commence par les réductions de F (figures 3.12 et 3.13) auxquelles les règles additionnelles de F+S font suite (figure 3.14).

Les transitions sont soit vierges de toute annotation (action silencieuse) soit étiquetées par une entrée zv ou une sortie $\overline{z}v$, où z est soit un nom de canal (e.g. $c, \overline{c}$), soit un nom de session concaténé avec une étiquette de message(e.g. $sf, s\overline{f}$), et v est une valeur. α et β sont des métavariables d'étiquettes, et φ, ψ représentent des séquences d'étiquettes.

La sémantique des expressions du langage F, notée $\rightarrow_{\mathsf{e}}$, est présentée dans la figure 3.12. Cette sémantique à petit-pas est standard : seules les applications des fonctions d'envoi ! et de réception ? sont reflétées par des étiquettes. La règle (APPLY) correspond à la β-réduction. Les règles (MATCH) et (MISMATCH) gouvernent le filtrage. La règle (LETVAL) réduit des déclarations de variables locales. Les règles (LETFUN) et (TYPE) enrichissent l'environnement d'exécution avec respectivement des déclarations de fonctions et de types.

$$
\begin{array}{rl}
(\textsc{Apply}) & \rho, l_i\, v_0 \ldots v_k \to_{\mathsf{e}} \rho, e_i\{x_0 = v_0; \ldots; x_k = v_k\} \\
& \quad \text{lorsque } (l_i\, x_0 \ldots x_{k_i} = e_i)_{i<k} \in \rho \\
(\textsc{Match}) & \rho, \mathtt{match}\ v\ \mathtt{with}\ (|\ v_i \to e_i)_{i<k} \to_{\mathsf{e}} \rho, e_0\gamma \\
& \quad \text{lorsqu'il existe une substitution } \gamma \text{ telle que } v = v_0\gamma \\
(\textsc{Mismatch}) & \rho, \mathtt{match}\ v\ \mathtt{with}\ (|\ v_i \to e_i)_{i<k} \to_{\mathsf{e}} \rho, \mathtt{match}\ v\ \mathtt{with}\ (|\ v_i \to e_i)_{0<i<k} \\
& \quad \text{sinon} \\
(\textsc{LetVal}) & \rho, \mathtt{let}\ x = v\ \mathtt{in}\ e \to_{\mathsf{e}} \rho, e\{x = v\} \\
(\textsc{LetFun}) & \rho, \mathtt{let}\ (l_i\, x_0 \ldots x_{k_i} = e_i)_{i<k}\ \mathtt{in}\ e \to_{\mathsf{e}} \rho \uplus \{(l_i\, x_0 \ldots x_{k_i} = e_i)_{i<k}\}, e \\
(\textsc{Type}) & \rho, (t_i = (|\ f_{j_i}\ \mathtt{of}\ \widetilde{T_{j_i}})_{j_i<k_i})_{i<k}\ \mathtt{in}\ e \to_{\mathsf{e}} \rho \uplus \{(t_i = (|\ f_{j_i}\ \mathtt{of}\ \widetilde{T_{j_i}})_{j_i<k_i})_{i<k}\}, e \\
(\textsc{Fresh}) & \rho, \mathtt{new}() \to_{\mathsf{e}} \rho \uplus \{n\}, n \\
(\textsc{Send}) & \rho, !\, c\, v \xrightarrow{\overline{c}\, v}_{\mathsf{e}} \rho, () \text{ lorsque } c \in \rho \\
(\textsc{Recv}) & \rho, ?c \xrightarrow{c\, v}_{\mathsf{e}} \rho, v \text{ lorsque } c \in \rho
\end{array}
$$

FIG. 3.12 – Sémantique des expressions de F

La règle (FRESH) permet de générer un nom frais (pouvant servir de canal ou de nonce). Finalement, les règles (SEND) et (RECV) réalisent les envois et réceptions. Leurs étiquettes enregistrent le canal utilisé et la valeur transmise.

$$
\begin{array}{rlrl}
(\textsc{Eval}) & \dfrac{\rho, e \xrightarrow{\alpha}_{\mathsf{e}} \rho', e'}{\rho, E[e] \xrightarrow{\alpha}_{\mathsf{P}} \rho', E[e']} & (\textsc{Fork}) & \rho, E[\mathtt{fork}\, l] \to_{\mathsf{P}} \rho, E[()] \mid l() \\
(\textsc{CommR}) & \dfrac{\rho, P \xrightarrow{\overline{z}\, v}_{\mathsf{P}} \rho', P' \quad \rho', Q \xrightarrow{z\, v}_{\mathsf{P}} \rho'', Q'}{\rho, P \mid Q \to_{\mathsf{P}} P' \mid Q'} & (\textsc{ParR}) & \dfrac{\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', P'}{\rho, Q \mid P \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q \mid P'} \\
(\textsc{CommL}) & \dfrac{\rho, P \xrightarrow{z\, v}_{\mathsf{P}} \rho', P' \quad \rho', Q \xrightarrow{\overline{z}\, v}_{\mathsf{P}} \rho'', Q'}{\rho, P \mid Q \to_{\mathsf{P}} P' \mid Q'} & (\textsc{ParL}) & \dfrac{\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', P'}{\rho, P \mid Q \xrightarrow{\alpha}_{\mathsf{P}} \rho', P' \mid Q}
\end{array}
$$

FIG. 3.13 – Sémantique des processus de F

La sémantique des processus de F, notée $\to_{\mathsf{P}}$, est décrite par la figure 3.13. La règle de réduction des contextes (EVAL) fait le lien avec la relation de réduction des expressions $\to_{\mathsf{e}}$. La règle (FORK) permet de créer un nouveau processus en parallèle. Les règles de communication (COMMR) et (COMML) font correspondre les actions d'envoi et de réception. Les règles (PARR) et (PARL) permettent de réduire un processus parmi d'autres placés en parallèle.

Sémantique de F+S

La sémantique de F+S est une extension de la sémantique de F permettant de réduire les expressions de session.

Les réductions des expressions comportant des sessions dans F+S sont gouvernées par les règles décrites dans la figure 3.14. Le prédicat `safe` σ (défini plus loin dans la section 3.6) dépend de l'honnêteté du principal participant à la session. La règle (SESSION)

$$\text{(Session)}\quad \rho, \text{session } S = \Sigma \text{ in } e \to_{\mathsf{e}} \rho \uplus \{S = \Sigma\}, e \qquad \text{up to renamings of } S$$

$$\text{(SendS)}\quad \frac{\rho, \sigma \xrightarrow{\overline{g}}_{\mathsf{s}} \rho', s.p \quad \mathtt{safe}\ \sigma}{\rho, \sigma\,(g(v), w) \xrightarrow{s\overline{g}\,v}_{\mathsf{e}} \rho', s.p\,(w)}$$

$$\text{(RecvS)}\quad \frac{\rho, \sigma \xrightarrow{g}_{\mathsf{s}} \rho', s.p \quad s\,\widetilde{a}\,\delta : S \in \rho \quad \mathtt{safe}\ \sigma}{\rho, \sigma\,(w) \xrightarrow{sg\,v}_{\mathsf{e}} \rho', s.p\,(w.g\,(\widetilde{a}, v))}$$

$$\text{(EndS)}\quad \rho, s.0\,(v) \to_{\mathsf{e}} \rho, v$$

FIG. 3.14 – Sémantique des sessions dans F+S

enrichit ρ d'une définition de session ; les règles (SendS) et (RecvS) permettent aux processus-rôles d'envoyer et de recevoir des messages au sein d'une session ; la règle (EndS) permet de conclure l'exécution d'une session et d'obtenir son résultat.

Ces règles s'appuient sur la sémantique $\to_{\mathsf{s}}$ des expressions de session décrite à la figure 3.11, qui elles-mêmes dépendent la réduction $\to_{\mathsf{r}}$ des processus locaux décrite à la figure 3.10. Démarrer une session dans F+S garantit sémantiquement l'intégrité globale : seules les interactions spécifiées par la définition de la session peuvent avoir lieu.

3.4.4 Sémantique par test

Avant de présenter dans les sections qui suivent les détails de l'implémentation, il est utile de définir précisément le théorème que celle-ci doit satisfaire.

Dans cette section, nous avons donné les détails de la sémantique de deux langages. L'un, F, est un langage fonctionnel distribué relativement standard. L'autre, F+S, est une extension de F avec des sessions dont le comportement, pour chaque rôle, suit scrupuleusement la spécification donnée par les processus locaux. Notre objectif est tout d'abord d'atteindre pour une implémentation des sessions dans F les mêmes garanties que les sessions natives de F+S apportent. Nous souhaitons de plus que ces garanties restent vraies en présence d'un adversaire contrôlant le réseau et une partie des principaux.

Définissons maintenant le théorème que l'implémentation des sessions dans F doit satisfaire. Nous commençons par quelques notations et remarques.

- Soit L l'ensemble des bibliothèques `Data`, `Net`, `Crypto` et `Prins` sur lesquelles repose l'implémentation sécurisée. Parmi celles-ci, la bibliothèque `Prins` gère les principaux et les informations cryptographiques les concernant. Toutes ces bibliothèques sont écrites dans F et ne sont considérées formellement que dans leur version symbolique (voir section 3.6 pour les définitions des bibliothèques) ;
- Soit $\widetilde{S}$ un ensemble de déclarations de sessions et soit $M_{\widetilde{S}}$ leur implémentation (définie formellement dans la section 3.7) ;
- Soit U le code utilisateur. Il a le droit d'appeler des sessions, mais pas la bibliothèque `Prins`. U est écrit dans F+S, mais peut être ramené dans F en interprétant les appels de sessions de $\widetilde{S}$ comme des appels de fonction de $M_{\widetilde{S}}$;
- Soit O le code de l'adversaire dans F+S, qui a accès à l'interface spécifique de `Prins` (voir section 3.6), aux autres bibliothèques et aux appels de sessions ;
- Soit enfin O' un adversaire de F avec des restrictions identiques à celles de O. Nous supposons sans perte de généralité que O' ne fait pas appel à $M_{\widetilde{S}}$ (puisque O' peut posséder sa propre copie du code généré).

Notons que O, O' et U peuvent, indépendamment des sessions, échanger des messages sur des canaux partagés et utiliser l'ensemble des bibliothèques (notamment de cryptogra-

phie) à l'exception de la bibliothèque `Prins` dont l'accès est restreint pour ne pas divulguer par défaut l'ensemble des clés privées des principaux honnêtes.

Nous utilisons la notation $M\,M'$ pour $M[M'[\_]]$. Un *système de haut-niveau* est un processus de F+S de la forme $L\,\widetilde{S}\,U\,O$. Un *système de bas-niveau* est un processus de F de la forme $L\,M_{\widetilde{S}}\,U\,O'$. Une *configuration* est la donnée d'un environnement ρ et d'un processus P. Les *configurations initiales* sont les configurations dont l'environnement $\rho = \varnothing$ est vide.

Notre théorème principal est énoncé sous la forme d'une propriété de test. Les sémantiques par test datent originellement de l'équivalence de Morris pour le λ-calcul [52]. Dans les calculs de processus, De Nicola and Hennessy [21] ont par exemple étudié les équivalences par test dans CCS. De façon informelle, le code utilisateur U a la possibilité de réaliser des tests arbitraires sur le comportement des sessions en présence de l'adversaire O ou O' et de décider à quel moment rapporter une erreur. Nous montrons de cette manière que les configurations initiales de bas-niveau (dans F) utilisant l'implémentation des sessions ne permettent pas au code utilisateur de relever plus d'erreurs que les configurations initiales de haut-niveau (dans F+S) qui utilisent les sessions natives.

L'erreur est par convention rapportée sur un canal spécial ω. Une configuration $\emptyset, P$ *échoue* lorsqu'il existe une réduction $\emptyset, P \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$.

Theorem 3.2 (Correction par test) Si $L\,M_{\widetilde{S}}\,U\,O'$ peut échouer dans F avec un adversaire O' dans lequel ω n'est pas présent, alors $L\,\widetilde{S}\,U\,O$ peut aussi échouer dans F+S avec un adversaire O dans lequel ω n'est pas présent.

L'énoncé du théorème 3.2 dit que les configurations de bas-niveau dans F, dans lesquelles les sessions sont implémentées par le code généré par le compilateur décrit à la section 3.7, ne peuvent avoir un comportement déviant de façon observable (par les principaux honnêtes) de celui des configurations de haut-niveau dans F+S, dans lesquelles les sessions sont automatiquement respectées. La quantification porte sur le code utilisateur U (identique dans F et F+S) et sur le code de l'adversaire O et O'. La restriction sur la présence de ω dans O et O' vient du fait que c'est l'utilisateur qui ne doit pas être capable (avec le même code U) de distinguer si la sémantique est celle de F ou celle de F+S.

L'objet de la suite de notre chapitre est d'obtenir un compilateur `s2ml` qui prend en entrée une définition de session S et produit un ensemble de définitions M_S tel que le théorème 3.2 s'applique à chaque fois.

L'intégrité des sessions correspond par essence au respect des relations de causalité entre messages. De telles relations sont généralement décrites comme des correspondances injectives [70] entre évènements signalés par différents participants. Intuitivement, chaque graphe de session est la somme des correspondances qui sont vérifiées dans toute exécution. Le théorème 3.2 garantit que ces correspondances sont satisfaites pour les participants honnêtes. Les correspondances entre évènements sont de plus au cœur de la technique de preuve utilisée dans [8].

Rappelons que la section suivante (section 3.5) donne les éléments de conception du protocole utilisé par l'implémentation sécurisée des sessions. La section 3.6 décrit ensuite les bibliothèques L sur lesquelles repose cette implémentation. La section 3.7 détaille alors la génération automatique par notre compilateur de l'implémentation M_S spécifique de chaque session S. La section 3.8 donne enfin les étapes principales de la démonstration que les implémentations générées satisfont le théorème ci-dessus. La modélisation du comportement de l'adversaire y est notamment décrite. Les détails complets de la preuve sont donnés dans l'annexe D.7.

3.5 Conception du protocole

Nous décrivons dans cette section le protocole générique que l'implémentation générée utilise. Nous commençons par donner les principes qui sous-tendent l'élaboration de ce protocole, puis nous décrivons quelques catégories d'attaque que ce dernier doit être en mesure de déjouer. Nous continuons par la description de la troisième condition d'implémentation, suivie par la définition de la notion d'état et du concept de visibilité. Nous concluons par un résumé des différents concepts utilisés.

3.5.1 Principes

Les deux principes que nous suivons lors de la génération du protocole pour une session donnée sont les suivants :

1. Un message de la session est implémenté par un unique message concret : la description de la session rend compte exactement des messages échangés entre les rôles.

2. Le nombre d'opérations effectuées lors de l'exécution du protocole est le plus réduit possible : l'objectif est d'avoir un envoi et un traitement des messages lors de l'exécution avec une complexité constante en la taille du graphe (ne dépendant par exemple que du nombre de rôles) avec une taille de message raisonnable.

Le principe 1 permet au programmeur d'avoir une idée claire de l'abstraction que les sessions constituent. Si certaines sessions nécessitent des messages supplémentaires pour être sécurisées, le compilateur les détecte et suggère des corrections que l'utilisateur est libre de suivre. La section 3.5.3 définit exactement quelles sessions sont exclues et précise la façon dont le programmeur peut contourner le problème.

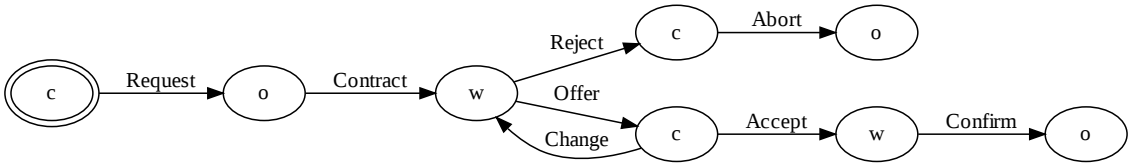
Le principe 2 garantit une efficacité maximale du protocole généré. Les opérations algorithmiques sur le graphe de la session sont en particulier restreintes à la phase de compilation.

3.5.2 Attaques

Avant de discuter les propriétés du protocole qui doit être généré pour chaque session, nous commençons par donner une idée des comportements indésirables du réseau ou de participants corrompus qui se coordonnent pour faire dérailler une session.

Nous illustrons sur l'exemple suivant les principales classes d'attaques.

Exemple 3.21 (Attaques) Nous supposons ici l'absence de toute protection cryptographique lors de l'exécution de la session `Shopping` dont nous reproduisons le graphe ci-dessous (tiré de l'exemple 3.7 de la session `Shopping`) : chaque arête du graphe est traduite par un simple message ne contenant que l'étiquette et la valeur transmise.



1. Attaques contre les messages :
   (a) Modification d'un message échangé entre deux principaux honnêtes : l'adversaire peut intercepter et changer l'étiquette du message (remplacer `Offer` par `Reject`).

   (b) Répétition d'un message précédemment échangé : l'adversaire peut rejouer un message `Offer` qu'il a espionné sur le réseau pour forcer une nouvelle itération de la boucle. L'adversaire peut aussi rejouer un message `Request` d'une instance précédente de la session de manière à se passer du principal jouant le rôle `c`.

2. Attaques contre la session :
   (a) Envoi d'un message non-autorisé par le flot : après avoir reçu un message `Reject`, un principal corrompu jouant le rôle `c` peut décider d'envoyer un message `Change` au rôle `w`.
   (b) Envoi d'un message à la place d'un rôle honnête : après avoir reçu un message `Offer` et envoyé un message `Accept` à `w`, un principal corrompu jouant le rôle `c` peut décider d'envoyer un message `Confirm` au rôle `o` en se faisant passer pour `w`.
   (c) Envoi de plusieurs messages suivant chacun une branche de la session : un principal corrompu jouant le rôle `w` peut envoyer à la fois des messages `Reject` et `Offer`.
   (d) Envoi d'un message sans attendre la succession des messages précédents : un principal corrompu jouant le rôle `w` peut envoyer directement un `Confirm` au rôle `o` sans passer par l'interaction `Offer - Accept`.

Intuitivement, chaque fois qu'un participant honnête prend part à l'exécution d'une session en envoyant ou recevant un message, celui-ci doit être autorisé par le graphe de la session. À l'inverse, chaque fois qu'un participant malhonnête essaye de dévier de l'exécution conforme d'une session en envoyant ou rejouant un message incorrect, ce message doit être détecté ou ignoré (c'est-à-dire qu'il ne doit pas être présenté à l'utilisateur comme étant valide).

Même si tous les principaux sont honnêtes, l'adversaire contrôlant le réseau peut enfin essayer de les tromper en mélangeant les messages provenant d'exécution distinctes d'une même session, ou en rejouant des messages au sein d'une même exécution.

Nous considérons ainsi comme attaque contre l'intégrité des sessions le fait qu'au cours de l'exécution d'une session certains principaux malhonnêtes, parfois en collaboration avec l'adversaire contrôlant le réseau, réussissent à faire accepter un message déviant de la spécification de la session à un principal honnête. Faire accepter à deux participants honnêtes deux issues différentes d'une même session est aussi une attaque par cette définition.

Les attaques présentées ci-dessus, ainsi que d'autres attaques contre le contenu des messages, sont similaires aux attaques classiques contre les protocoles cryptographiques (incorrects) décrits dans le style de Dolev et Yao [26].

Pour résumer, les attaques qu'une implémentation non sécurisée peut subir sont de plusieurs sortes :

1. Attaques contre les messages :
   (a) Modification d'un message entre deux principaux honnêtes
   (b) Répétition d'un message précédemment échangé
2. Attaques contre la session :
   (a) Envoi d'un message non-autorisé par le flot
   (b) Envoi d'un message à la place d'un rôle honnête
   (c) Envoi de plusieurs messages suivant chacun une branche de la session

(d) Envoi d'un message sans attendre la succession des messages précédents

Notons que les attaques que nous considérons ne concernent pas la valeur que chaque message transmet. Des techniques usuelles (chiffrement, signatures) permettent d'obtenir pour ces valeurs des propriétés de secret ou d'authenticité. La propriété d'intégrité des sessions ne le nécessite cependant pas. La section 4.2 évoque l'inclusion de la gestion des valeurs échangées dans la spécification de la session.

3.5.3 Une condition d'implémentation sur les sessions

Dans certains graphes de session, certaines attaques sont inévitables sans introduire automatiquement des messages supplémentaires et contrevenir à notre principe 1. La solution est de demander à l'utilisateur de le faire dès l'écriture de la session. Une autre solution est d'introduire une relation de confiance entre principaux jouant certains rôles : cette relation restreindrait les ensembles possibles de rôles corrompus. La question de la relation de confiance est discutée dans la section 4.2 mais n'a pas été retenue ici.

Exemple 3.22 (Choix aveugle - contre-exemple) Dans l'exemple de la session décrite par le graphe de la figure 3.15(a), où `S` peut envoyer soit un message `Reject` au rôle `C`, soit un message `Accept` au rôle `O`. Sauf communication entre `C` et `O`, un principal malhonnête jouant `S` peut envoyer les deux messages sans que les destinataires s'en rendent compte immédiatement (sans qu'ils communiquent). Une telle attaque remet en cause l'intégrité de la session puisqu'une séquence de message illicite est acceptée par les participants honnêtes.

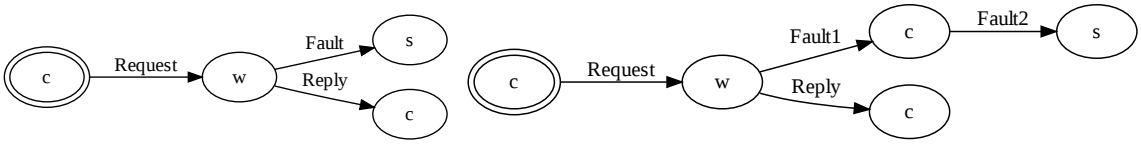


FIG. 3.15 – (a) Un graphe de session vulnérable et (b) une version corrigée.

Pour éviter les graphes de sessions se prêtant à de telles attaques, nous imposons une troisième condition d'implémentation, en plus des conditions 1 et 2 décrites ci-dessus.

3. Pour tous chemins $\widetilde{f_1}$ et $\widetilde{f_2}$ débutant au même nœud et s'achevant avec les rôles r_1 et r_2, si ni r_1, ni r_2 ne sont parmi les rôles actifs de $\widetilde{f_1}$ ou $\widetilde{f_2}$, alors $r_1 = r_2$.

(Un rôle est dit actif sur un chemin lorsqu'il est l'expéditeur d'un des messages du chemin) Cette condition 3, dite du choix aveugle, est automatiquement remplie par les sessions qui ne possèdent que deux rôles. Seules sont exclues les sessions où le choix entre deux branches n'est pas partagé par l'ensemble des rôles impliqués jusque-là.

Il est cependant toujours possible de transformer une session ne respectant par la condition 3 en ajoutant des messages supplémentaires.

Exemple 3.23 (Choix aveugle - correction par ajout de messages) La session de la figure 3.15(a) ne remplit par la condition 3 : le principal jouant le rôle `S` actif sur les chemins `(Reject)` et `(Accept)` peut tromper les rôles r_1 =`C` et r_2 =`O` en envoyant les deux messages simultanément.

Il reste cependant possible de transformer cette session (comme les autres ne respectant pas la condition 3) en une session équivalente au prix de l'ajout d'un message supplémentaire.

La figure 3.15(b) illustre une version sûre de la session décrite par le graphe de la figure 3.15(a) : le message `Accept` est décomposé en deux messages, `Accept1` et `Accept2` de manière à ce que `C` soit informé dans tous les cas du choix de la branche.

Nous montrons maintenant de façon explicite une attaque contre la session vulnérable décrite plus haut. Nous décrivons précisément en quoi cette attaque viole le théorème 3.2.

Exemple 3.24 (Choix aveugle - attaque) Prenons l'exemple de la session de la figure 3.15(a) qui ne satisfait pas la propriété 3. Supposons que les principaux jouant les rôles `c` et `s` soient honnêtes et participent à une session dans laquelle `w` est joué par l'adversaire. Comme expliqué plus haut, l'adversaire peut attaquer la session en envoyant les deux messages `Reply` et `Fault`. Les codes utilisateurs des rôles `c` et `s` peuvent alors communiquer sur un canal séparé, détecter s'ils ont été dupés et éventuellement envoyer ω ! D un autre côté, l'adversaire de haut-niveau ne peut manipuler l'exécution de la session pour permettre l'envoi par `c` ou `s` d'ω.

Nous donnons ci-dessous le code permettant à `c` et `s` de relever l'existence d'une attaque dans cette session `S`.

```
let pr = { prins_c = "Alice"; prins_s = "Bob"; prins_w = "Charlie"; } in
let x = new() in
let y = new() in
let alice_plays_c () =
  S.c pr (Request (42,{hReply = (fun _  → ! x "OK")})) in
let bob_plays_s () =
  S.s "Bob"
    {hFault = fun (p,_)  →
                if p = pr
                then let _ = ? x
                     in ! ω () } in
fork bob_plays_s; alice_plays_c()
```

Dans le code ci-dessus, `Alice` joue le rôle `c` et `Bob` celui du rôle `s` dans une exécution unique de la session. Ces principaux communiquent sur un canal `x` lorsqu'ils reçoivent à la fois `Reply` et `Fault`. Dans ce cas, `Bob` peut lever ω.

Il est possible de montrer en général la nécessité de la condition 3 pour prouver le théorème 3.2. Dans la suite, nous ne considerons que les sessions remplissant les trois conditions 1–3.

3.5.4 Se protéger des répétitions de messages

Confusion entre sessions

Pour éviter que l'adversaire ne réutilise certains messages provenant d'instances passées de la même session, le protocole que nous proposons repose sur la génération d'un identifiant de session unique à chaque instance. Cet identifiant est calculé de la façon suivante : $s = \mathbf{hash}(D\,\widetilde{a}\,N)$, où $D\,\widetilde{a}\,N$ est la concaténation de $D = \mathbf{hash}(S = \Sigma)$, l'empreinte de

l'intégralité de la définition de la session, les principaux $\widetilde{a}$ jouant les rôles de la session, et un nonce N généré par l'initiateur de la session. Notre implémentation des sessions est alors à même de distinguer les messages provenant de différentes sessions et d'instances différentes d'une même session.

Confusion au sein d'une même instance d'une session

Lorsqu'une session comporte une boucle, l'adversaire peut essayer de rejouer un message provenant d'une itération précédente de la boucle. Cet écueil peut être évité grâce à la présence d'une horloge logique qui numérote les messages. Ainsi, tout message rejoué d'une boucle aura un numéro qui n'est pas strictement supérieur aux numéros des messages précédents et pourra de ce fait être détecté.

Initialisation de la session

Puisqu'il est de la responsabilité du premier rôle de créer pour chaque instance un identifiant de session unique, un premier rôle corrompu, ou le réseau rejouant un des premiers messages, peut proposer à un principal de rejoindre une session avec un identifiant déjà utilisé. Chaque principal peut se protéger de ce comportement grâce à un cache contenant les identifiants de session déjà utilisés. Pour permettre à un principal de participer à une même session en jouant plusieurs rôles, le cache doit associer à chaque identifiant les rôles que le principal en question joue.

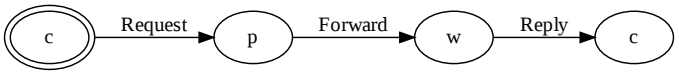
3.5.5 Protéger l'intégrité de la session

Protéger l'intégrité des sessions consiste à s'assurer que, du point de vue des participants honnêtes, les instances des sessions se déroulent de façon cohérente et conforme à la spécification.

Signer l'histoire

Commençons par étudier le problème sur un cas simple.

Exemple 3.25 (Signatures) Rappelons tout d'abord le graphe de la session `Forward` (exemple 3.4).



Commençons par examiner le rôle du proxy `p`. Recevoir le message `Request` de la part du client `c` lui suffit pour s'assurer du bon déroulement de la session jusque-là. Il est cependant nécessaire de s'assurer que `c` est bien à l'origine du message. Cette authentification peut être obtenue par l'utilisation d'un canal privé préétabli ou par la transmission avec le message de la signature cryptographique de son étiquette. Dans tous les cas, nous supposons l'existence d'une identité cryptographique initiale (paire clé privée, clé publique) pour chacun des principaux.

Examinons maintenant le cas du rôle du service web `w`. Celui-ci reçoit un message `Forward` qui doit le convaincre que la session s'est déroulée sans accroc jusque-là. Le rôle `w` doit donc non seulement recevoir une preuve de la participation de `c` et de `p` mais aussi une preuve de l'accord entre leurs deux visions de l'exécution de la session. Comme `w` ne reçoit un message que de la part de `p`, l'authentification implicite apportée par le canal privé entre `c` et `p` ne peut convaincre `w` de la participation de `c`, puisque `p` peut être corrompu et mentir. Une signature par `c` de l'étiquette `Request` (ainsi que de l'identifiant de la session) a, elle, une validité universelle : à la fois `p` et `w` peuvent la vérifier. Reste à `p` de convaincre

`w` qu'il partage la même vision de la session que `c`, c'est-à-dire qu'il a vérifié les preuves apportées par `c` et les a acceptées. Le moyen le plus simple est de contresigner la signature apportée par `c` à laquelle est ajoutée l'étiquette du message `Forward`.

La façon de procéder avec le message `Reply` est similaire : `w` contresigne ce qu'il convient d'appeler l'histoire de la session et transmet le résultat à `c`. Celui-ci peut alors vérifier les signatures successives et s'assurer que le chemin suivi par la session est un chemin légal et conforme à la connaissance partielle qu'il en a.

En généralisant le raisonnement employé pour cet exemple, une façon d'empêcher toute déviation par rapport à la spécification de la session de la part de n'importe lequel des principaux participant, est d'inclure dans chaque message une trace signée de l'intégralité de l'histoire de la session. Celle-ci est obtenue à chaque étape en contresignant l'histoire reçue dans le message précédent après y avoir ajouté l'étiquette du message envoyé. Chacun des rôles recevant un message peut alors vérifier que les signatures successives sont correctes et que la suite des messages forme un chemin acceptable sur le graphe de la session.

La validité de cette solution est intuitive. Quelle que soit la partition entre participants honnêtes et malhonnêtes, tous les rôles signent leur accord sur une histoire commune. Ainsi, tant que les clés privées des principaux honnêtes restent secrètes, l'intégrité de la session est préservée, c'est-à-dire que tout message illicite reçu par un participant honnête sera détecté et ignoré.

Cette solution est malheureusement inefficace : à la réception, le nombre et la taille des signatures à vérifier peuvent être arbitrairement longs (les sessions avec cycles peuvent produire un nombre arbitraire de signatures à vérifier).

Premières optimisations

Contresigner, puis vérifier une trace complète est extrêmement coûteux. Le but de la contresignature est de vérifier d'une part si la succession des différents messages forme une trace valide de la session et de s'assurer d'autre part de la cohérence de la vision qu'ont les principaux honnêtes de l'exécution de la session. La succession des messages peut être retrouvée en incorporant dans les signatures la valeur de l'horloge logique mentionnée dans la section précédente. Cette solution remplace la contre-signature par une simple juxtaposition des signatures de chacun des messages, chaque rôle se contentant de transmettre les signatures qu'il a lui-même reçu.

L'accord entre les participants honnêtes au sujet de la vue qu'ils ont chacun de la session est garanti par la vérification que chacun fait des signatures transmises par le protocole : les signatures réalisées par les principaux honnêtes jalonnent la trace qui ne peut diverger à ces endroits puisqu'un principal honnête ne signerait pas plusieurs messages possédant la même horloge logique et le même identifiant de session.

Le protocole que nous obtenons alors, bien que plus simple (la taille des signatures est restreinte), ne limite toujours pas le nombre des signatures transmises. Une optimisation simple consiste à ne pas transmettre les signatures que le destinataire du message a déjà reçues. Illustrons cette version du protocole dans quelques exemples.

Exemple 3.26 (Se souvenir de l'histoire) Reprenons le graphe de la session `Shopping` (exemple 3.7) rappelé à la figure 3.26.

Lors de l'exécution de la séquence de messages `Request-Contract-Reject-Abort`, la version précédente du protocole demandait que le client `c` transmette à `o` dans le message `Abort` les signatures des messages `Request`, `Contract`, `Reject` et `Abort`. L'optimisation

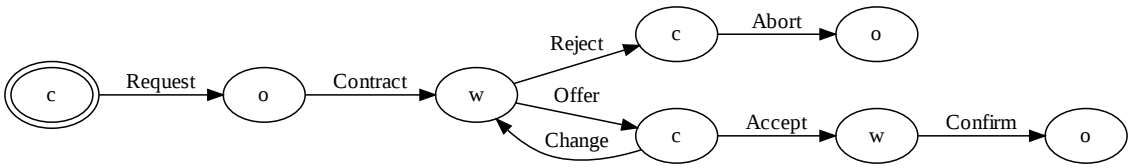


FIG. 3.16 – Shopping (figure 3.7)

proposée utilise le fait que les signatures des messages `Request` et `Contract` sont déjà connues par `o` : seules seraient transmises les signatures de `Reject` et `Abort`.

Plaçons-nous maintenant dans le cas d'une exécution partielle utilisant la boucle `Request-Contract-Offer-Change-Offer-Change-Offer-Change`. L'optimisation permet à `c` de n'envoyer que sa propre signature du message `Change` puisque `w` connaît déjà les signatures précédentes.

Notons enfin que cette optimisation n'élimine pas la possibilité d'avoir un nombre illimité de signatures à transmettre. Prenons l'exemple de la trace `Request-Contract-(Offer-Change)`^{n}`-Offer-Accept-Confirm` où le nombre d'itérations de la boucle est laissé en paramètre. Dans ce cas, le rôle `w` doit transmettre à `o` dans son message `Confirm` les $2n + 2$ signatures manquantes.

Le gain en taille sur chaque message proposé par cette optimisation ne permet cependant pas de s'affranchir de la vérification de la correction de signatures correspondant à un chemin de taille potentiellement arbitraire.

3.5.6 Visibilité

Rappelons avant toute chose que l'objectif ici est de ne demander que le minimum de vérifications lors de l'exécution permettant à un principal de s'assurer de la bonne exécution de la session jusque-là. Examinons quelques cas de figures sur un exemple.

Exemple 3.27 (Signatures uniques) Continuons à étudier la session `Shopping` de l'exemple 3.7, dont le graphe a été rappelé dans l'exemple 3.26 ci-dessus.

Étudions le chemin parcourant n tours de boucle et finissant par la branche supérieure `Request-Contract-(Offer-Change)`^{n}`-Reject-Abort`. Le contenu du message `Abort` que reçoit le rôle `o` doit le convaincre que la session s'est déroulée conformément à la spécification depuis le message `Contract` (qu'il a lui-même envoyé).

Supposons tout d'abord que les principaux jouant `c` et `w` soient malhonnêtes. Dans ce cas, le nombre de tours de boucles prétendument effectués par `c` et `w` n'a pas d'importance pour `o`. Dans le cas où au moins l'un des deux principaux est honnête, le nombre de signatures reflète le nombre de tours de boucles : les signatures de `(Offer-Change)`^{n}`-Reject-Abort` sont transmises. Cependant la présence des signatures de `Reject` et de `Abort` garantit qu'au moins un principal honnête s'est assuré de la correction de la session jusque-là. Les signatures de la boucle `(Offer-Change)`^{n} sont donc inutiles pour `o` dans tous les cas.

Généralisons le raisonnement présenté dans cet exemple. Pour un rôle recevant un message, recevoir l'ensemble des signatures produites par un autre rôle n'apporte pas plus de garanties que de ne recevoir que sa signature la plus récente. La raison est que, si un

rôle est joué par un principal malhonnête, aucune de ses signatures ne fait foi, tandis que si le principal est honnête, sa dernière signature suffit pour affirmer que de son point de vue la session s'est déroulée de façon satisfaisante jusque-là.

De cette façon, une seule signature de la part de chacun des rôles est suffisante pour vérifier l'intégrité de la session. La complexité du protocole s'en trouve ainsi considérablement réduite : le nombre de signatures à vérifier à chaque étape est borné par le nombre de rôles. La liste des signatures à vérifier à un état donné du protocole, exprimée sous la forme d'une liste d'étiquettes de messages, est appelée *séquence visible*.

Definition 3.3 (Messages visibles) Soit $\widetilde{g}$ un chemin initial entre le nœud initial m_0 et un nœud m dont le rôle est r. Soit $\widetilde{f}$ la séquence d'étiquettes obtenue à partir de $\widetilde{g}$ en effaçant chacune des étiquettes g :

1. dont le rôle émetteur est r, ou
2. qui est suivi dans $\widetilde{g}$ par une étiquette dont le rôle émetteur est r ou le rôle émetteur de g.

De cette manière, $\widetilde{f}$ ne contient que les étiquettes des derniers messages envoyés par chacun des rôles autres que r. On qualifie $\widetilde{f}$ de séquence *visible* depuis le nœud m.

Exemple 3.28 (Messages visibles - Shopping) Par exemple, dans la session `Shopping` (exemple 3.7, rappelé à l'exemple 3.26 ci-dessus), le nœud occupé par `o` après le chemin `Request-Contract-Offer-Accept-Confirm` a pour seule séquence visible `Accept-Confirm`. Le nœud occupé par `w` après un `Contract` ou un `Change` a deux séquences visibles suivant le chemin initial menant à ce nœud : `Request-Contract` et `Change`.

Pour des raisons d'efficacité, il est nécessaire de minimiser le nombre d'opérations effectuées lors de l'exécution du protocole. La stratégie retenue est donc de précalculer à la compilation l'ensemble des séquences visibles que les différents rôles sont en mesure d'attendre à chaque instant. Comme le nombre total des séquences visibles est fini, tous les cas peuvent être précalculés : les différentes vérifications à effectuer lors de la réception d'un message peuvent alors être statiquement déterminées à la compilation, idem pour la composition des messages à envoyer. L'exécution ne requiert alors plus le moindre calcul sur le graphe de session.

3.5.7 Protocole final

Pour résumer, le protocole qui est généré spécifiquement pour chaque session par notre compilateur repose sur un certain nombre d'éléments standards :

- Identifiant de session
- Horloge logique
- Cache
- Signatures

L'architecture que nous proposons fait reposer tous les calculs dépendant du graphe sur le compilateur : l'exécution du protocole repose sur des états pré-calculés, ce qui permet d'obtenir des performances proches de l'optimalité.

Les signatures et vérifications associées demandent cependant un temps de calcul important. Notre implémentation utilise des MACs à leur place. Une comparaison des temps d'exécution entre ces deux solution est effectuée dans la section 3.9.2. Les MACs requièrent l'établissement de clés partagées lors de l'échange des premiers messages. Ce protocole

d'échange de clés est discuté dans la section 3.7.3. Par simplicité, notre modèle formel ne considère cependant que l'utilisation de signatures.

Dans les sections suivantes, nous présentons les bibliothèques que notre implémentation utilise et dont l'implémentation symbolique fait partie de l'énoncé du théorème (section 3.6). Nous détaillons dans la section d'après (section 3.7) les mécanismes utilisés par notre compilateur pour générer l'implémentation sécurisée de la session.

3.6 Bibliothèques pour la cryptographie et les principaux

Dans cette section, nous décrivons les interfaces et l'implémentation des bibliothèques ML que nous utilisons pour les principaux et la cryptographie. Notre approche et nos bibliothèques sont très similaires à celles de Bhargavan et coll. [9] : sous une même interface, nous avons à la fois une implémentation symbolique et une implémentation concrète (en fait deux, reposant sur les bibliothèques OpenSSL ou .Net). L'implémentation symbolique est réalisée à l'aide de types algébriques dans le langage F et joue un rôle crucial dans notre modèle de sécurité.

La version symbolique des bibliothèques est utile pour le débogage (l'annexe C.4 donne la trace d'exécution d'un exemple). Mais principalement l'implémentation symbolique correspond au modèle formel de cryptographie que nous utilisons pour établir nos propriétés de sécurité. Plus précisément, nous nous plaçons dans un modèle d'adversaire de type Dolev-Yao : celui-ci peut contrôler des principaux corrompus (qui peuvent jouer n'importe quels rôles dans une session), intercepter, modifier et envoyer des message sur les canaux publics, et effectuer des opérations cryptographiques sur le matériel à sa disposition. L'adversaire ne peut cependant pas casser le chiffrement ou forger des signatures.

3.6.1 Types de données (module Data)

La première des bibliothèques est appelée `Data` et définit des types abstraits `str` et `bytes` pour représenter respectivement les chaînes de caractères et les suites d'octets. La figure 3.17 donne l'interface du module `Data`.

```
type str
type bytes

val base64 : bytes  → str
val ibase64 : str  → bytes
val utf8 : str  → bytes
val iutf8 : bytes  → str
val concat : bytes  → bytes  → bytes
val iconcat : bytes  → bytes * bytes
val cS : string  → str
val iS : str  → string
```

FIG. 3.17 – Interface de la bibliothèque Data

Ce module permet de manipuler les types de données `str` et `bytes`. Les fonctions présentes permettent la conversion en Base64 ([]), en Utf8 ([]) et entre le type `str` et le type natif `string` de ML. La fonction `concat` permet de concaténer deux suites d'octets. La fonction `iconcat` peut séparer deux suites d'octets concaténés avec `concat`.

Les relations algébriques que cette bibliothèque implémente sont les suivantes :

– lorsque `s` est de type `bytes`, `ibase64 (base64 s)` se réduit vers `s` ;
– lorsque `s` est de type `str`, `iutf8 (utf8 s)` se réduit vers `s` ;
– lorsque `s` et `s'` sont de type `bytes`, `iconcat (concat s s')` se réduit vers `(s,s')` ;
– lorsque `s` est de type `string`, `iS (cS s)` se réduit vers `s` ;

L'implémentation symbolique de cette bibliothèque est donnée dans la figure 3.18. Les types `str` et `bytes` sont implémentés à l'aide de types algébriques et d'un type auxiliaire `blob` qui représente les valeurs cryptographiques (voir section suivante). Les conversions entre les types `string`, `str` et `bytes` se font à l'aide des fonctions `cS`, `utf8`, `base64` et leurs inverses `iS`, `iutf8`, `ibase64` qui toutes reposent sur l'application ou la déconstruction des constructeurs algébriques `Literal`, `Base64` et `Utf8`. La concaténation et son inverse sont implémentées de la même manière.

```
type str =
  | Literal of string
  | Base64 of bytes
and bytes =
  | Concat of bytes * bytes
  | Utf8 of str
  | Bin of blob
and blob =
  | Hash of bytes
  | Nonce of bytes
  | AsymSign of bytes * bytes
  | AsymEncrypt of bytes * bytes
  | SymEncrypt of bytes * bytes
  | Mac of bytes * bytes

let cS s = Literal s
let iS = function
  | Literal s  → s
  | _  → failwith "iS failed"

let base64 b = Base64(b)
let ibase64 = function
  | Base64 s  → s
  | _  → failwith "ibase64 failed"

let utf8 x = Utf8 x
let iutf8 = function
  | Utf8 s  → s
  | _  → failwith "iutf8 failed"

let concat x y = Concat(x,y)
let iconcat = function
  | Concat(x,y)  → (x,y)
  | _  → failwith "iconcat failed"
```

FIG. 3.18 – Implémentation symbolique de la bibliothèque Data

3.6.2 Cryptographie

La partie cryptographie des bibliothèques est définie dans le module `Crypto`. L'interface de ce module est reproduite dans la figure 3.19.

```
open Data
val mkNonce : unit  → bytes
val sha1 : bytes  → bytes
val sha1_verify : bytes  → bytes  → unit
val rsa_encrypt : bytes  → bytes  → bytes
val rsa_decrypt : bytes  → bytes  → bytes
val rsa_sign : bytes  → bytes  → bytes
val rsa_verify : bytes  → bytes  → bytes  → unit
val mac : bytes  → bytes  → bytes
val mac_verify : bytes  → bytes  → bytes  → unit
```

FIG. 3.19 – Bibliothèque Crypto

Plus précisément, la fonction `mkNonce` crée un nonce qui va pouvoir être utilisé lors de la création d'un identifiant de session ou d'une clé partagée. La prise d'empreinte effectuée par la fonction `sha1` (fonction de hachage) peut être vérifiée par la fonction `sha1_verify` qui prend en argument à la fois l'objet à vérifier et son empreinte présumée. Les fonctions `rsa_encrypt` et `rsa_decrypt` permettent d'effectuer respectivement le chiffrement et le déchiffrement d'un message par un algorithme de type RSA à clés asymétriques (le premier argument de ces fonctions est la clé, le second le message). L'application de la fonction `rsa_sign` à une clé privée et un message produit une signature cryptographique. La fonction `rsa_verify` à laquelle sont données la clé publique, le message et la signature présumée, permet de vérifier celle-ci. Les fonctions `mac` et `mac_verify` réalisent des fonctions similaires à l'aides de MACs (Message Authentication Code) et de clés symétriques.

Ces fonctions cryptographiques fonctionnent selon les règles algébriques suivantes :

- lorsque `s` est de type `bytes`, `sha1_verify s (sha1 s)` ne lance pas d'exception ;
- lorsque `s`, `k` et `k'` sont de type `bytes` avec `k` la clé publique et `k'` la clé privée d'un même principal, `rsa_decrypt k'(rsa_encrypt k s)` se réduit vers `s` ;
- lorsque `s`, `k` et `k'` sont de type `bytes` avec `k` la clé publique et `k'` la clé privée d'un même principal, `rsa_verify k' s (rsa_sign k s)` ne lance pas d'exception ;
- lorsque `s` et `k` sont de type `bytes`, `mac_verify k s (mac k s)` ne lance pas d'exception ;

La gestion des clés est effectuée par la bibliothèque `Prins` décrite dans la section plus bas.

L'implémentation concrète de cette bibliothèque repose sur de la cryptographie standard. Par exemple, le type de donnée `bytes` est implémenté par un tableau d'octet et la fonction `rsa_sign` l'est par une fonction de signature à clef privée sur l'empreinte du message (RSA-SHA1).

```
let sha1 s = Bin (Hash s)
let sha1_verify s h =
  let ss = match s with Bin (Hash ss)  → ss
                      | _  → assert false in
  if s = ss then () else failwith "Verification of hash failed"

let sym_encrypt k t = Bin (SymEncrypt (k,s))
let sym_decrypt k s =
  let k',m = match s with Bin(SymEncrypt (k,t))  → (k,t)
                        | _  → assert false in
  if k = k' then m else failwith "Decryption failed"
```

FIG. 3.20 – Quelques fonctions de l'implémentation symbolique de la bibliothèque Crypto

La figure 3.20 donne l'implémentation symbolique de quelques unes des fonctions de la bibliothèque `Crypto`. Les fonctions de hachage, signature, MAC et chiffrement correspondent à de simples applications des constructeurs du type `blob`. Les fonctions de déchiffrement ou de vérification ne consistent qu'en un filtrage permettant de déconstruire les termes.

3.6.3 Principaux

La troisième bibliothèque utilisée par notre implémentation permet la gestion des principaux et des données qui leurs sont associées. Notre implémentation en a besoin pour effectuer les deux opérations associées aux principaux utilisées dans les sessions, c'est-à-dire la signature de valeurs et la réception de messages. L'interface de cette bibliothèques est donnée par la figure 3.21.

```
open Data
type pr = { id:string; cert:string; ip:string; port:int;}

val register : pr  → unit

val get_privkey : str  → bytes
val get_pubkey : str  → bytes
val get_mackey : str  → str  → bytes

val gen_keys : str  → str  → bytes
val reg_keys : str  → str  → bytes  → unit

val bind : string  → unit
val close : string  → unit

val psend : string  → str  → unit
val precv : string  → str

val check_cache : string  → string  → bytes  → unit
```

FIG. 3.21 – Bibliothèque Prins

Les principaux sont des enregistrements à quatre champs. L'identifiant `id` est une simple chaîne de caractères. À celui-ci sont associés un certificat `cert`, une adresse IP et un numéro de port. Le certificat est donné sous la forme d'un nom de fichier qui contient soit une clé publique (pour un principal distant) soit un couple clé publique/clé privée (pour un principal local). L'implémentation concrète utilise des certificats X.509 ou des fichiers `.key`/`.pub`. La fonction `register` permet d'enregistrer un nouveau principal dans la base de donnée. Une fois enregistré, un principal n'est désigné que par son champ `id`.

Les fonctions `get_privkey`, `get_pubkey` et `get_mackey` permettent d'accéder aux clés des principaux enregistrés. Une erreur est évidemment lancée lorsque la clé n'est pas disponible. Les clés symétriques (auxquelles on accède grâce à `get_mackey`) ne sont obtenues qu'au cours de l'exécution de la session : la fonction `gen_keys` les génère localement (à l'aide de la fonction `mkNonce` de la bibliothèque `Crypto`) tandis que la fonction `reg_keys` permet d'enregistrer les clés établies par d'autres principaux dans la base de donnée locale.

Les fonctions `bind` et `close` permettent respectivement d'initier et de fermer la connexion d'un principal au réseau. Les fonctions `psend` et `precv` permettent à un principal (le premier argument) d'envoyer ou de recevoir une valeur. Le second argument de la

fonction `psend` est le destinataire. Plus précisément, '`psend a v`' envoie de manière asynchrone le message `v` au principal `a`, tandis que '`precv a`' permet de recevoir un message envoyé au principal `a`.

Enfin, la fonction `check_cache` permet de vérifier et mettre à jour le cache que chaque principal possède pour éviter certaines attaques. Si le message est un message d'invite pour un rôle avec un identifiant déjà présent dans le cache associé au même rôle, une erreur est lancée. Dans le cas contraire, la fonction termine normalement et une nouvelle entrée est ajoutée au cache. Dans le cadre d'une implémentation concrète, il est possible de limiter la taille d'un tel cache tout en conservant sa correction par l'utilisation des techniques standards de cachets temporels (*time-stamps*).

3.6.4 Interface de l'adversaire

Dans notre modèle, nous supposons qu'il n'existe qu'un nombre fini et déterminé de principaux, ainsi qu'un prédicat arbitrairement fixé `safe` caractérisant le sous-ensemble des principaux honnêtes. Ce prédicat n'est utilisé que pour exprimer les propriétés de sécurité que satisfont les principaux honnêtes. Il est en effet impossible à notre implémentation de garantir la moindre propriété formelle à un principal dont les clefs privées sont accessibles. Notre sémantique distingue ainsi la façon dont les principaux honnêtes et malhonnêtes participent à une session. Rappelons que notre implémentation concrète ne fait bien sûr pas la différence.

L'adversaire n'a ainsi pas d'accès direct aux fonctions `psend`, `precv` et aux fonctions `get_privkey` et `get_mackey`. À la place, il dispose d'un canal $\texttt{psend}^\bullet$ pour envoyer des messages aux principaux honnêtes, d'une liste $\texttt{chans}^\bullet$ de canaux pour recevoir les messages envoyés aux principaux malhonnêtes, et d'une liste $\texttt{skeys}^\bullet$ des clefs de signature des principaux malhonnêtes (rappelons que notre modèle formel ne tient pas compte de l'utilisation de MACs à la place des signatures). En utilisant ces éléments, l'adversaire peut agir à la place des principaux malhonnêtes, et notamment signer pour eux n'importe quelle valeur. Notre sémantique suppose donc, au début de l'exécution d'un programme, la connaissance par l'adversaire des valeurs $\texttt{psend}^\bullet$, $\texttt{chans}^\bullet$ et $\texttt{skeys}^\bullet$, ainsi que de toutes les fonctions des bibliothèques mentionnées ci-dessus, à l'exception des fonctions `psend`, `precv` et `get_privkey`.

Le code de `psend`, `precv` et `antireplay` dans leur version symbolique est détaillé dans l'annexe C.1.

3.7 Génération du protocole et compilation

Dans cette partie, nous décrivons le processus de compilation qui, à partir des définitions de sessions décrites dans la section 3.2, génère pour chacun des rôles une implémentation reposant sur les bibliothèques décrites dans la section 3.6. La présentation de cette section correspond à un point intermédiaire entre [7] et [8]. Plus précisément, le code généré reprend le code de [7] en y remplaçant l'usage des signatures par des MACs plus efficaces et en y ajoutant un protocole d'échange de clés symétriques (pour les MACs). Les preuves de l'annexe D se rapportent à la version originelle de [7] avec signatures.

3.7.1 Génération de l'interface

La génération de l'interface a été décrite en détail dans la section 3.3.2. Nous récapitulons dans les figures 3.22 et 3.23 les divers éléments qui la composent.

Type `prins`

type prins = {
  Pour chaque rôle $r \in \mathcal{R}$, [prins_r : principal ;]
}

Types de renvoi

Pour chaque rôle $r \in \mathcal{R}$, *tel que* $(r{:}\tau = p) \in \Sigma$
[type result_r = τ]

Déclaration des rôles

val r_0 : prins → msg0 → result_r_0
Pour chaque rôle $r \neq r_0 \in \mathcal{R}$
[val r : principal → msgn_r → result_r]

FIG. 3.22 – Génération de l'interface : fonctions des rôles et types de retour

Syntaxe des processus nommés

$\chi ::= 0, 1, 2, ...$		noms de sous-processus
$\tau ::=$ unit $\mid$ int $\mid$ string		types de base
$p' ::= 0$		fin de processus
$\mid \chi : \text{send}(f_i : \tau_i \,;\, p'_i)_{i<k}$		envoi
$\mid \chi : \text{recv}[f_i : \tau_i \to p'_i]_{i<k}$		réception
$\mid \chi$		continuation vers un sous-processus
$\Sigma ::= (r_i : \tau_i = p'_i)_{i<n}$		session (processus initiaux pour chaque rôle)

Fonction msg$_r$

$\text{msg}_r(\chi : \text{send}(\_)) = \text{msg}\chi$
$\text{msg}_r(\chi : \text{recv}[\_]) = \text{msg}\chi$
$\text{msg}_r(\chi) = \text{msg}\chi$
$\text{msg}_r(0) =$ return_r

Génération des types msg$n$

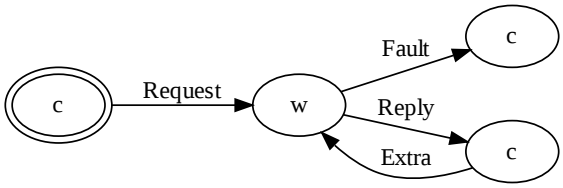
Pour chaque rôle $r \in \mathcal{R}$, *tel que* $(r{:}\tau = p') \in \Sigma$
$\langle \chi : \text{send}(f_i : \tau_i \,;\, p_i)_{i<k} \rangle^r = \{\text{type msg}\chi = \left[\mid f_i \text{ of } \tau_i * \text{msg}_r(p_i)\right]_{i<k}\} \cup \langle p_i \rangle^r_{i<k}$
$\langle \chi : \text{recv}[f_i : \tau_i \to p_i]_{i<k} \rangle^r = \{\text{type msg}\chi =$
$\qquad \{\text{h}f_i = \text{prins} * \tau_i \to \text{msg}_r(p_i)\,;\}_{i<k}\} \cup \langle p_i \rangle^r_{i<k}$
$\langle \chi \rangle^r = \varnothing$
$\langle 0 \rangle^r = \varnothing$

FIG. 3.23 – Génération de l'interface : types de pilotage

3.7.2 Stratégie d'implémentation

Notre implémentation a un objectif d'efficacité. Cela signifie notamment que tout calcul sur le graphe doit être évité lors de l'exécution : les séquences visibles qu'un rôle s'attend de voir à un instant donné doivent donc être pré-calculées. C'est dans cette optique que nous commençons par étudier quels sont les différents états dans lesquels l'implémentation d'un rôle peut se trouver.

Exemple 3.29 (Etats) Reprenons le graphe de la session `Wsn` (exemple 3.6).



Dans cette session, le rôle `w` n'annote qu'un seul nœud. Les séquences visibles qu'il attend correspondent cependant à deux états distincts : lorsque la session commence, `w` attend uniquement un message `Request` de la part de `c` (le premier message que `w` reçoit ne peut en aucun cas être `Extra`) ; lorsque `w` a répondu avec `Reply`, il n'attend qu'un message `Extra` (le message `Request` n'est plus possible). L'implémentation doit tenir compte de cette situation : le nœud courant d'un rôle dans le graphe de la session ne donne pas suffisamment d'information pour en déduire les messages et séquences visibles qu'il est en droit d'attendre.

L'état d'un rôle doit permettre de décider quelles séquences visibles celui-ci est en droit d'attendre (lorsqu'il reçoit un message), et quelles signatures il doit produire et envoyer (lorsqu'il envoie un message). Ces deux données ne sont accessibles que lorsque la position de chacun des rôles dans le graphe est connue. Or tout changement de position d'un rôle se retrouve dans la séquence visible acceptée par les rôles suivants, puisque la séquence visible enregistre les derniers messages (et donc les derniers états) des autres rôles (nous utilisons de fait la propriété 2 qui associe de façon unique une étiquette de message à chaque arête). Les états de l'exécution de chacun des rôles peuvent ainsi être représentés à l'aide de séquences d'étiquettes visibles.

Un *état* $\widetilde{g}$ d'une session est défini par la donnée de la dernière séquence d'étiquettes visibles acceptée par un rôle. L'état initial est la séquence vide (notée concrètement `start`). En pratique, un état $\widetilde{g}$ désigne deux états locaux : il désigne l'état atteint par l'émetteur du message après l'envoi et l'état atteint par le destinataire après réception. Un *état d'envoi* d'un rôle est donc donné par la dernière séquence visible qu'il a accepté. Un *état de réception* d'un rôle est donné par la séquence visible acceptée par le récepteur du dernier message que le rôle a envoyé.

Un graphe formé par les états locaux d'un rôle (graphe d'état local) peut alors être construit grâce à des arêtes liant en alternance des états d'envoi et des états de réception. De la même façon qu'un état d'une session correspond à deux états locaux, tout message de la session correspond à deux transitions locales : le passage d'un état d'envoi à un état de réception pour l'émetteur, et le passage d'un état de réception à un état d'envoi pour le destinataire. Le graphe formé de ces doubles transitions est appelé le *graphe d'état complet*.

Exemple 3.30 (Etats (suite)) Dans l'exemple `Wsn` rappelé dans la figure de

l'exemple 3.29, seuls deux rôles sont présents : les séquences visibles ne comportent donc qu'un seul message. La figure 3.24 donne les graphes locaux de chacun des rôles, les nœuds étant étiquetés par le rôle et l'état associé de la session.

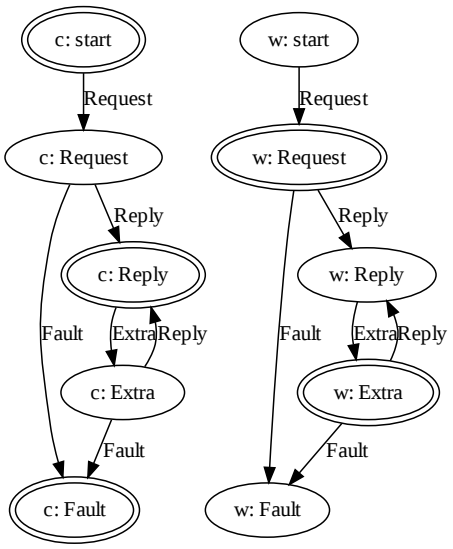


FIG. 3.24 – Wsn - Etats locaux

La figure 3.24 contient à droite le graphe du rôle `w` et à gauche celui du rôle `c`. On distingue deux types de nœuds : les nœuds d'envoi (états d'envoi, où le rôle a la main dans la session) qui ont un double encerclement tandis que les nœuds (états) de réception n'en ont qu'un simple.

Examinons le graphe du rôle `w`. L'état initial est `start`, moment où le rôle est en attente du premier message. Celui-ci ne peut être que `Request`. Une fois reçu, l'état devient donc `Request` (il s'agit de la séquence visible qu'il a vérifié) et le rôle `w` prend la main dans la session et a le choix d'envoyer un message `Fault` ou un message `Reply`. L'état devient alors respectivement `Fault` ou `Reply` (séquences visibles que le rôle `c` a vérifié lors de la réception). Une fois dans l'état `Reply`, seul le message `Extra` peut lui être envoyé. Les deux états d'exécution correspondant à un seul nœud dans le graphe d'origine sont donc bien séparés.

En ne représentant que les nœuds d'envoi, on peut obtenir une vision partielle du graphe d'état complet. La figure 3.25 donne ce graphe complet pour la session `Wsn`.

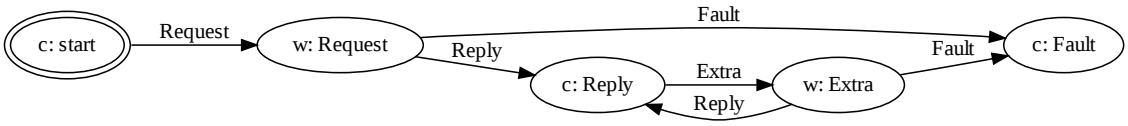


FIG. 3.25 – Wsn - Graphe d'état complet

Les états d'exécution du protocole sont ainsi indexés par les dernières séquences visibles acceptées par chacun des rôles. Le nombre de séquences visibles étant limité, la taille des graphes locaux et complets reste finie. La définition des séquences visibles donne au graphe complet une taille correspondante au dépliage d'un seul tour de chaque boucle. Ce dépliage est dans le pire des cas exponentiel (lorsque les boucles sont imbriquées), mais en pratique reste raisonnable (doublement de la taille du graphe).

Examinons un exemple avec un nombre plus important de rôles.

Exemple 3.31 (Etats de Shopping) Reprenons le graphe de la session `Shopping` (exemple 3.7).

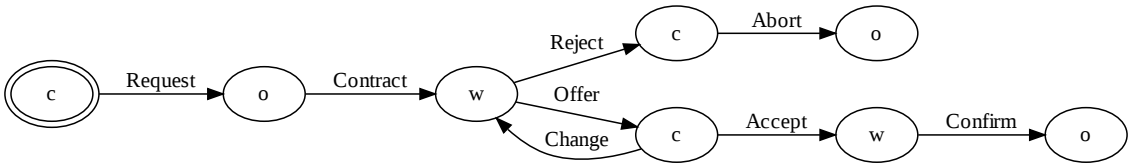


FIG. 3.26 – Shopping (figure 3.7)

Cette session comporte trois rôles, `c`, `o` et `w`. La figure 3.27 donne les états locaux des rôles `c` et `o`. Examinons le cas du rôle `c`. À partir de l'état `start`, le client `c` peut envoyer un message `Request`. L'état atteint est `Request`. Deux messages peuvent alors être reçus de la part du serveur web `w` : `Reject` mène vers l'état `ContractReject` tandis que `Offer` mène vers `ContractOffer`. Si `c` envoie un message `Change` et reçoit à nouveau `Offer`, l'état de `c` devient `Offer` : l'exécution du protocole nécessite de dupliquer le nœud de `c` participant à la boucle. La raison est que lors de la première itération de la boucle, le rôle `c` doit vérifier une signature de `Contract` de la part de `o` ; cette vérification n'est plus nécessaire lors des interactions `Offer-Change` suivantes.

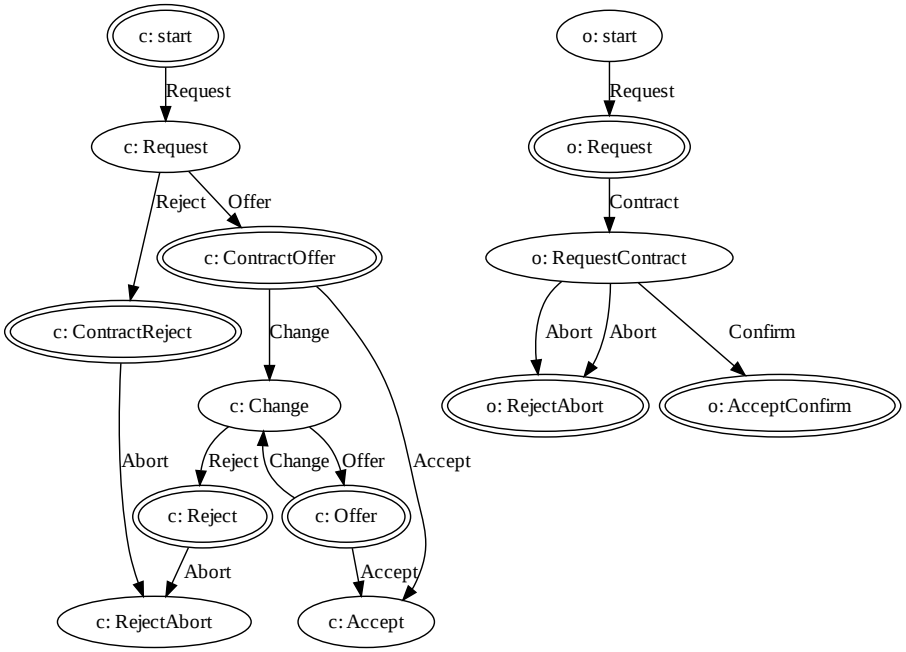


FIG. 3.27 – Shopping - Etats locaux de c et o

La figure 3.27 donne aussi le graphe des états du rôle `o`. Nous remarquons que l'arête `Abort` y est présente en double. Cette particularité est due au fait que le rôle `c` qui envoie ce message peut être dans deux états distincts lors de l'envoi. Pour l'implémentation du protocole du point de vue de `o`, il n'y a pas de différence entre le traitement de ces deux messages.

Les graphes d'état locaux vont donner le canevas de l'implémentation de chacun des rôles. Les doubles transitions du graphe d'état complet vont, elles, permettre de calculer les détails exacts du protocole : le contenu des messages à envoyer et les vérifications à faire sur les messages reçus dépendent de la situation globale et non seulement locale de la session.

3.7.3 Compilation

Nous décrivons maintenant les opérations réalisées par le compilateur avant la génération du code en tant que tel.

Parsing

La première étape de la compilation est la génération, par le compilateur, d'un arbre de syntaxe abstraite à partir de la description syntaxique de la session. Cette étape permet de vérifier certaines contraintes syntaxiques liées à l'utilisation d'Ocaml : les noms des étiquettes devenant des noms de constructeurs, ils doivent par exemple commencer par une majuscule.

La syntaxe abstraite d'une session consiste alors en une liste de rôles, ceux-ci étant représentés par leur processus en syntaxe abstraite. Cette version syntaxique des processus locaux est utilisée directement pour la génération de l'interface utilisateur.

Génération du graphe

Une fois l'arbre syntaxique de chacun des rôles construit, le compilateur en déduit le graphe global de la session. L'algorithme part du processus du rôle initial et suit le flot de la session. Les arêtes du graphe correspondent aux messages envoyés et reçus (le nœud d'origine est décoré par le rôle de l'émetteur, le nœud de destination par celui du récepteur) et, une fois une arête rencontrée, le processus du récepteur donne les arêtes suivantes.

Par cette phase, le compilateur s'assure de certaines propriétés de cohérence : notamment que tous les messages envoyés sont reçus par un rôle distinct et que les étiquettes sont uniques. Le graphe obtenu est ensuite converti en format `dot` [28] pour pouvoir être visualisé par le programmeur (tous les graphes de sessions présentés dans ce chapitre ont été générés automatiquement de cette manière).

Vérification de la dernière des conditions d'implémentation

Le compilateur utilise le graphe global pour vérifier la dernière des conditions d'implémentation. La propriété 3 de la section 3.5.3 est en effet nécessaire pour éviter les attaques qui consistent à exécuter plusieurs branches en parallèle. Les autres propriétés (1 et 2) ont été vérifiées lors de la construction du graphe.

Génération du graphe d'états complet

Nous avons vu précédemment que l'implémentation du protocole reposait sur des états indexés par des séquences visibles. Le compilateur, partant du graphe global, parcourt tous les chemins pour générer le graphe d'état complet : chaque chemin initial (où les boucles ne sont pas parcourues plus d'une fois) est abstrait en une séquence d'état. Dans le même temps, les graphes d'état locaux sont produits. Ceux-ci, ainsi que le graphe d'état complet, sont converti en format `dot`.

Génération des séquences visibles

À partir des états de réception de chacun des rôles, le compilateur calcule la relation de visibilité `visib`. Il s'agit d'avoir la liste des séquences visibles qu'il est possible d'accepter dans chaque état de réception. Le compilateur pourra ainsi, lorsqu'il génère le code associé à la réception dans un état donné, préciser les vérifications exactes qui sont à réaliser avant

d'accepter le message. Cette relation de visibilité se calcule très facilement à partir du graphe d'état complet, puisque les séquences visibles acceptables à partir d'un état sont les états d'envoi successeurs.

Génération de la relation `futur`

Les séquences visibles caractérisent les vérifications à effectuer lors de chaque réception. La relation `futur` donne, elle, la liste des MACs à produire lors d'un envoi à partir d'un état donné. Pour chaque état d'envoi et chaque message qu'il est possible d'envoyer, il est en effet nécessaire de savoir quels sont les destinataires qui demanderont une preuve d'envoi ou, autrement dit, dans les séquences visibles de quels rôles ce message apparaît.

Génération du protocole de routage des MACs

La relation `futur` donne la source des MACs, les séquences visibles donnent leur destination. Reste à organiser l'implémentation pour que les MACs partent de leur source et arrivent à destination. Le routage des MACs est organisé par une relation `fwd_mac` qui associe à chaque message du graphe d'état complet la liste des MACs à transmettre. Notons que cette liste contient au minimum les messages de la séquence visible du destinataire. Le protocole de routage utilise une heuristique pour limiter au nombre de rôles le nombre de messages nécessaire à la transmission d'un MAC de sa source à sa destination. Lorsqu'un MAC a un chemin à parcourir, cette heuristique consiste à transmettre le MAC à un rôle avec le premier message, puis à reprendre la transmission au moment de la dernière implication de ce rôle dans le chemin : l'implémentation du rôle conserve en mémoire le MAC en attendant le dernier moment. L'implémentation de chaque rôle comporte ainsi une mémoire pour stocker les MACs en attente de transmission. Dans un chemin, le protocole choisi transmet ainsi toujours un MAC d'un rôle à un autre, ce qui permet de limiter la taille des messages et de ne jamais transmettre inutilement une information.

Génération du protocole d'échange de clés

Dans le but d'améliorer le temps d'exécution, des MACs sont utilisés plutôt que des signatures. Ces MACs repose sur une infrastructure de clés symétriques qui ne sont pas présentes au début de la session. Les premiers messages de chaque session comportent ainsi une phase d'établissement de clés partagées à partir de l'infrastructure de clés publiques et privées déjà présente.

Comme nous ne souhaitons pas que l'implémentation utilise plus de messages que la session n'en spécifie, l'échange des clés symétriques est réalisé lors des premiers messages de la session. Le principe est le suivant : lorsqu'un rôle crée un MAC pour un autre rôle pour la première fois, il génére une clé symétrique qu'il chiffre avec la clé publique du principal jouant le rôle destinataire, puis signe avec sa propre clé privée. Lorsque le rôle destinataire reçoit cette clé, il est capable de l'authentifier puis de la déchiffrer avant de l'utiliser dorénavant pour vérifier les MACs provenant de ce même rôle. Pour éviter toute réutilisation par l'adversaire, les noms des deux principaux sont chiffrés avec la clé.

D'une façon identique au protocole de routage des MACs, une relation `fwd_key` est calculée pour permettre l'acheminement des clés vers leurs destinataires d'une façon relativement minimale.

Avant de passer à la génération du code, examinons les valeurs de ces différentes relations sur un exemple.

Exemple 3.32 (Auth - compilation) L'exemple `Auth` est un exemple simple (pas de choix, ni de boucle) qui comporte 4 rôles : un client `c` cherche une réponse d'un serveur web `w` ; pour cela il doit passer par un agent `a` qui va vérifier si son identité convient auprès

d'un annuaire `d`. Il s'agit d'une version un peu plus élaborée de l'exemple `Forward` 3.4. La figure 3.28 donne le graphe de la session `Auth`.

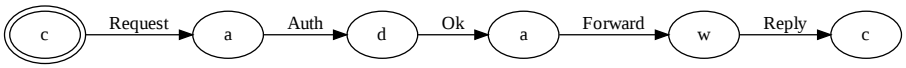


FIG. 3.28 – Auth

Le graphe des états locaux est donné dans la figure 3.29.

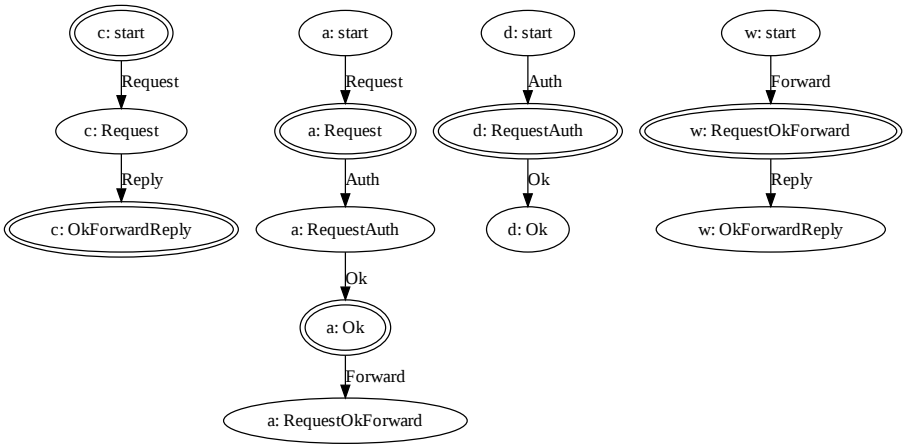


FIG. 3.29 – Auth - États locaux

Le graphe d'état complet n'est pas donné, vu qu'il est isomorphe au graphe d'origine (il n'y a pas de boucle). Nous listons les informations données par le compilateur :

```
[graph] [Visib] Visibility calculation starting.
[graph] [Visib] Nb of states = 5
[graph] [Visib] Visibility calculation done.
[graph] Visib:
  At state a: start, the visible sequences are [c_Request__]
  At state d: start, the visible sequences are [a_Auth_c_Request_]
  At state a: RequestAuth, the visible sequences are [d_Ok_]
  At state w: start, the visible sequences are [a_Forward_d_Ok_c_Request_]
  At state c: Request, the visible sequences are [w_Reply_a_Forward_d_Ok_]
[graph] Future relation:
  At point c: start, the msg Request needs to be MACed to rôles a d w
  At point a: Request, the msg Auth needs to be MACed to rôles d
  At point d: RequestAuth, the msg Ok needs to be MACed to rôles a w c
  At point a: Ok, the msg Forward needs to be MACed to rôles w c
  At point w: RequestOkForward, the msg Reply needs to be MACed to rôles c
[graph] MACs to fwd:
  At point c: start, the msg Request carries the macs caRequest,cdRequest,cwRequest
  At point a: Request, the msg Auth carries the macs cdRequest,adAuth
  At point d: RequestAuth, the msg Ok carries the macs daOk,dwOk,dcOk
  At point a: Ok, the msg Forward carries the macs cwRequest,dwOk,dcOk,awForward,
                                                                    acForward
  At point w: RequestOkForward, the msg Reply carries the macs dcOk,acForward,wcReply
[graph] Key fwding:
  At point a: Request, the msg Auth carries the keys (ad,cd)
  At point c: start, the msg Request carries the keys (ca,cd,cw)
  At point w: RequestOkForward, the msg Reply carries the keys (wc,ac,dc)
  At point a: Ok, the msg Forward carries the keys (aw,dw,cw,ac,dc)
  At point d: RequestAuth, the msg Ok carries the keys (da,dw,dc)
```

La première indication que le compilateur donne est le nombre d'états globaux : ici il y en a 5 sans compter l'état initial. Nous rappelons que chaque séquence d'état concerne en pratique deux rôles, l'un dans un état d'envoi, l'autre dans un état de réception. Cela nous donne 10 états locaux plus un état initial pour chaque rôle : il y a donc 14 états locaux au total, ce qui est confirmé par la figure 3.29.

Le compilateur donne ensuite pour chaque rôle dans un état de réception la liste des séquences d'étiquettes visibles qui correspondent aux vérifications à effectuer. Comme la session `Auth` est relativement simple, dans chacun des états de réception, il n'y a qu'une seule séquence visible. Par exemple, le rôle `a` dans l'état `RequestAuth` a une séquence visible réduite à un élément `d_Ok_` (ce qui signifie qu'il faut vérifier le MAC du message `Ok` de la part du rôle `d`). Dans le cas du rôle `w` dans son état initial `start`, la séquence visible est `a_Forward_d_Ok_c_Request_` : trois vérifications sont ainsi nécessaires (de la part de `a`, `d` et `c`). Notons que, comme le veut la définition de visibilité, le message `Auth` n'a pas à être revérifié.

Les deux dernières listes données par le compilateur concernent le routage des MACs et des clés. Examinons par exemple le cas du MAC du message `Request` de la part de `c` pour `w`, noté `cwRequest`. Ce MAC est transmis par `c` avec le message `Request`, puis par `a` avec le message `Forward`. Le rôle `d` n'est pas impliqué dans ce transport. La clé générée par `c` pour `w`, notée `cw`, suit le même chemin.

3.7.4 Génération de l'implémentation

La génération de l'interface a été décrite à la section 3.3.2. Nous décrivons ici l'organisation et les détails du code généré.

Les différents éléments du code généré sont les suivants :

- la mémoire locale `store` qui stocke les valeurs associées à l'exécution de la session ;
- les fonctions `content_`r`_`l créant les contenus des différents MACs ;
- les fonctions `mac_verify_`r`_`r'`_`l de vérification de chaque MAC ;
- les types `wired_`r`_`$\widetilde{g}$ permettant de transmettre les valeurs reçues sur le réseau ;
- les fonctions `receiveWired_`r`_`$\widetilde{g}$ de réception pour chaque état de réception ;
- les fonctions `sendWired_`l`_`r`_`$\widetilde{g}$ d'envoi pour chaque message à chaque état d'envoi ;
- les fonctions r`_`$\widetilde{g}$ (pour chaque état) et r permettant l'interaction entre l'utilisateur et la session.

Nous allons décrire chacun de ces éléments en utilisant les relations évoquées plus haut à la section 3.7, les fonctions des bibliothèques décrites à la section 3.6 et les types générés pour l'interface (section 3.3.2).

Mémoire locale : `store`

La mémoire locale doit, pour chaque rôle, contenir les valeurs pertinentes à l'exécution de la session. Ces valeurs sont les suivantes :

- les noms des principaux jouant chaque rôle ;
- les MACs à retenir le temps de les retransmettre ;
- les clés partagés, pour la même raison ;
- l'identifiant `sid` de la session ;
- la valeur de l'horloge logique `ts`.

La figure 3.30 décrit la génération de ces types enregistrement. Le type `prins` est en fait le même que celui présenté pour l'interface dans la section 3.3.2.

Le compilateur génère de façon triviale pour chacun des rôles des fonctions `empty_store_`r qui renvoient une mémoire vide à l'exception des champs des principaux

```
type prins = {
  Pour chaque rôle r ∈ R, [ prins_r : principal ; ]
}
type evidences = {
Pour chaque MAC pour un rôle r d'un message d'étiquette l, [ mac_rl : bytes ; ]
}
type sharedkeys = {
Pour tous les rôles r, r' ∈ R, [ key_rr' : bytes ; ]
}
type runparam = {
  ts : int ;
  sid : bytes }
type localstore =
 { peers : prins ;
   macs : evidences ;
   keys : sharedkeys ;
   header : runparam }
```

FIG. 3.30 – Génération de l'implémentation : `store`

connus et de l'horloge logique `ts` mise à 0.

Contenu des MACs : `content_`r`_`l

Chaque MAC ne contient que l'étiquette du message courant, la valeur de l'identifiant de session et de l'horloge logique. Les fonctions `content_`r`_`l ont pour tâche de construire le contenu de chaque MAC dans le but, soit de l'envoyer, soit de le confronter à un MAC reçu. La figure 3.31 donne le schéma de génération de ces fonctions.

```
Pour chaque rôle r ∈ R envoyant un message l,
[ let content_r_l = fun (ts:int) (store:localstore)  →
  let  nextstate = utf8 (cS "r_l") in
  let  ts_mar = utf8 (cS (string_of_int ts)) in
  let  header = concat store.header.sid ts_mar in
  let  content = concat header nextstate in
  content]
```

FIG. 3.31 – Génération de l'implémentaion : `content_`r`_`l

Vérification des MACs : `mac_verify_`r`_`r`_`l

Chaque MAC est vérifié par une fonction de vérification. Nous donnons le schéma de génération dans la figure 3.32.

Cette fonction prend en argument la valeur de l'horloge logique attendue `ts`, la mémoire `store`, la clé partagée `k` et la valeur du MAC à vérifier `m`. La fonction `mac_verify` provient de la bibliothèque `Crypto`. La fonction `test_eq` réalise un simple test d'égalité et, s'il n'est pas satisfait, renvoie une erreur.

Messages abstraits : `wired_`r`_`$\widetilde{g}$

Les types de donnée désignés `wired_`r`_`$\widetilde{g}$ permettent aux fonctions vérifiant la correction des messages reçus de retourner l'étiquette et la valeur du message afin qu'ils soient trans-

Pour chaque rôle $r \in \mathcal{R}$ et chaque MAC venant de r' d'un message d'étiquette l,

```
[ let mac_verify_r_r'_l = fun ts store k m  →
  let  content = content_r'_l ts store in
  let  cc = mac_verify k m content in
  let  () = test_eq cc content "MAC incorrect" in
  cc]
```

FIG. 3.32 – Génération de l'implémentation : `mac_verify_`r`_`r`_`l

mis à l'utilisateur. La figure 3.33 donne le schéma de génération de ces types-somme, qui sont les types de retour des fonctions `receive_Wired`.

Pour chaque rôle $r \in \mathcal{R}$ dans un état de réception $\widetilde{g}$,

```
[ type wired_r_g~ =
  Pour chaque message l de valeur de type t venant du rôle r' dans l'état g~',
  [ | Wired_in_r_g~_r'_g~' of t * store ]
]
```

FIG. 3.33 – Génération de l'implémentation : `wired_`r`_`$\widetilde{g}$

Les constructeurs `Wired_in_`r`_`$\widetilde{g}$`_`r'`_`$\widetilde{g}'$ sont annotés par le rôle destinataire r et son état de réception $\widetilde{g}$, et par le rôle r' et son état (de réception) après envoi $\widetilde{g}'$. La séquence $\widetilde{g}'$ correspond aussi à l'état de r après réception et à la séquence visible que r doit vérifier lors de cette réception.

Fonctions de réception et vérification : `receiveWired_`r`_`$\widetilde{g}$

La réception d'un message donne lieu à un ensemble de vérifications pour s'assurer de sa correction. La figure 3.34 donne le schéma de génération de cette série de fonctions.

Lorsqu'un rôle r dans un état de réception $\widetilde{g}$ reçoit un message grâce à `precv store.peers.prins_`r, l'implémentation commence par décomposer les différentes parties du messages, notamment la valeur de l'identifiant de session `sid` et de l'horloge logique `ts`.

Si le message permet au rôle r de rejoindre la session, les valeurs des principaux jouant les autres rôles sont lues et mises-à-jour dans la mémoire locale `store`. Les passages où `store` est modifié ont été omis de la figure 3.34. Ils sont par exemple de la forme `store.peers.prins_`$r \leftarrow v$. Une fois la mémoire à jour avec les principaux participant à la session, l'identifiant de la session est confronté au cache grâce à l'appel `check_cache store.peers.prins_`r `"`r`" sid`. Une fois le test passé, l'identifiant de la session est enregistré dans le `store`.

Si le message n'est pas un message initiateur, l'implémentation effectue deux tests pour éliminer la possibilité d'une attaque où l'adversaire rejoue un message ou réutilise un message d'une session différente. La fonction `test_inf` est similaire à la fonction `test_eq` mais utilise une comparaison plutôt qu'un test d'égalité.

Lorsque l'identifiant et l'horloge logique ont été acceptés, l'implémentation regarde quelle est l'étiquette `tag` du message. Cette étiquette donne pour confirmation le nom du rôle émetteur ainsi que son état $\widetilde{g}'$ après envoi, c'est-à-dire la séquence visible que le destinataire est censé vérifier. Le graphe d'état global (et plus directement la relation `visib`) donne la liste des séquences visibles qu'il est possible de recevoir à partir de l'état

```
Pour chaque rôle r ∈ R dans un état de réception g̃,
[ let receive_Wired_r_g̃ : store  → wired_r_g̃ = fun store  →
  let  mesg = precv store.peers.prins_r in
  let  start,content = iconcat (ibase64 mesg) in
  let  header,peers = iconcat start in
  let  sid,ts_mar = iconcat header in
  let  ts = int_of_string (iS (iutf8 ts_mar)) in
  let  oldts = store.header.ts in
 Si l'état est start
  (* ... Mise à jour de la mémoire avec les principaux participants ... *)
  let  () = check_cache store.peers.prins_r "r" sid in
  (* ... Mise à jour de la mémoire avec l'identifiant de la session ... *)
 Sinon
  let  _ = test_inf oldts ts "Replay attack!" in
  let  _ = test_eq store.header.sid sid "Session confusion attack!" in
 FinSi
  (* ... Mise à jour de l'horloge logique ... *)
  let  tag,payload = iconcat content in
  match iS (iutf8 tag) with
 Pour chaque message l venant du rôle r' et que l'on peut recevoir à l'état g̃,
  [ | "r'_g̃'"  →
  let  value,protocol = iconcat payload in
  let  macs,keys = iconcat protocol in
  let  x = [ () ou iS (iutf8 value) ou string_to_int (iS (iutf8 value))] in
  Pour chaque clé mentionnée par la relation fwd_key,
  [ let key_r0r1,keys = iconcat keys in
   let  () = reg_keys (cS store.peers.prins_r0)
         (cS store.peers.prins_r1) key_r0r1 in]
  Pour chaque MAC mentionné par la relation fwd_mac,
  [ let mac_r0l0,macs = iconcat macs in]
  (* ... Mise à jour de la mémoire avec les MACs reçues ... *)
  Pour chaque MAC mentionné par la relation visib,
  [ let macheader,maccontent = iconcat store.macs.mac_r0l0 in
   let  macsid,ts_mar = iconcat macheader in
   let  ts = int_of_string (iS (iutf8 ts_mar)) in
   let  _ = test_inf oldts ts "MAC verification" in
   let  oldts = ts in
   let  _ = test_eq macsid sid "MAC verification" in
   let  mackeyr0r = get_mackey store.peers.prins_r0 store.peers.prins_r in
   let  _ = mac_verify_r0_r_l0 ts store mackeyr0r maccontent in]
  let  _ = test_eq oldts store.header.ts "Time-stamp verification" in
  Wired_in_r_g̃_r'_g̃'(x,store)
  ]
]
```

FIG. 3.34 – Génération de l'implémentation : `receiveWired_r_g̃`

$\widetilde{g}$: ce sont les seules qui sont examinées.

Une fois l'étiquette connue, la connaissance du type (`unit`, `string` ou `int`) de la valeur transportée (on utilise ici la variable `x`) permet de la désérialiser. La relation `fwd_key` est ensuite utilisée pour générer le code permettant de désérialiser et, le cas éventuel, déchiffrer les clés symétriques. De même, la relation `fwd_mac` permet de savoir statiquement quels MACs sont transmis dans ce message. Les clés et MACs sont stockés dans la mémoire en attendant leur utilisation.

En dernier lieu, la relation `visib` donne la séquence d'étiquettes visibles associée au rôle r dans l'état $\widetilde{g}$ recevant un message d'étiquette l. Pour chaque MAC, pris dans l'ordre, à vérifier, l'implémentation le déconstruit pour en vérifier l'horloge logique et l'identifiant de session, puis pour en vérifier la validité grâce à la fonction `mac_verify_`r_0`_r_`l_0 dont la génération a été décrite précédemment. Le code est généré de manière à s'assurer de la progression des valeurs des horloges logiques. Un dernier test permet enfin de s'assurer que la valeur de l'horloge logique du MAC le plus récent correspond à la valeur courante (du message que l'on vient de recevoir).

Le constructeur `Wired_in_...` est enfin utilisé pour renvoyer la valeur `x` portée par le message ainsi que la valeur du `store` mis à jour.

Fonctions de construction de message et d'envoi : `sendWired_`l`_r_`$\widetilde{g}$

L'envoi d'un message consiste en sa construction (notamment la production des éléments cryptographiques), puis en l'envoi proprement dit. La figure 3.35 donne le schéma de génération de cette série de fonctions.

Lorsqu'un rôle r dans un état d'envoi $\widetilde{g}$ veut envoyer un message d'étiquette l à un rôle r', la fonction `sendWired_`r`_`$\widetilde{g}$ commence par incrémenter l'horloge logique `ts` et à construire l'en-tête `header` qui sera utilisé par le message et les MACs. Si le message envoyé est le premier message reçu par le destinataire, il doit contenir la liste de tous les principaux.

L'étape suivante consiste à s'occuper des clés : la relation `fwd_key` donne la liste des clés symétriques à transmettre. Parmi celle-ci, celles impliquant le rôle r correspondent aux clés qu'il faut générer à l'aide de la fonction `gen_key`. Les autres sont présentes dans la mémoire locale et ont juste besoin d'être sérialisées.

La relation `futur` donne ensuite la liste des MACs du message l à produire. Le contenu des MAC est obtenu grâce à la fonction `content_r_`l (de manière similaire aux fonctions `receiveWired_...`).

La relation `fwd_macs` donne enfin la liste des MACs à sérialiser dans le message en partance. Y est ajouté la valeur `x` que l'utilisateur a choisie pour être transmise dans ce message. La sérialisation de cette valeur est guidée par le type qui a été associé à ce message lors de la définition de la session.

La construction du message s'achève par l'addition de l'étiquette du message qui déclare d'une part le rôle émetteur et la séquence visible dont ce message apporte les MACs. L'envoi est effectué par `psend store.peers.prins_`r' `msg`. L'état de la mémoire après l'envoi est renvoyé pour son utilisation lors de la réception qui suivra.

Fonctions des rôles : r`_`$\widetilde{g}$ **et** r

Les fonctions `receiveWired_`r`_`$\widetilde{g}$ et `sendWired_`r`_`$\widetilde{g}$ s'occupent de tout ce qui concerne l'implémentation du protocole. Les fonctions r`_`$\widetilde{g}$ de chaque rôle permettent, elles, de s'assurer du respect du flot de la session et des interactions avec le code utilisateur. Ce sont les fonctions r qui sont exportées par l'interface. La figure 3.36 donne le schéma de génération de ces fonctions.

Pour chaque rôle r, les fonctions r`_`$\widetilde{g}$ s'appellent les unes les autres en suivant le graphe

```
Pour chaque rôle r ∈ R envoyant l vers r' dans un état d'envoi g̃,
[ let send_Wired_l_r_g̃ = fun x store  →
  let  nil = utf8 (cS "") in
  let  ts = store.header.ts + 1 in
  (* ... Mise à jour de l'horloge logique ... *)
  let  ts_mar = utf8 (cS (string_of_int store.header.ts)) in
  let  header = concat store.header.sid ts_mar in
  let  peers = nil in
 Si l'état du destinataire est start
  (* ... Sérialisation des principaux ... *)
 FinSi
  let  start = concat header prins in
  let  keys = nil in
  Pour chaque clé mentionnée par la relation fwd_key,
  [ let key_r0r1,keys = iconcat keys in
  Si r est l'emetteur de la clé
   let  key_rr1 = gen_keys (cS store.peers.prins_r)
        (cS store.peers.prins_r1) in
   let  keys = concat key_rr1 keys in
  Sinon
   let  keys = concat store.keys.key_r0r1 keys in
  FinSi
   Pour chaque MAC mentionné par la relation futur,
  [ let content = content_r_l store.header.ts store in
   let  mackeyrr0 = get_mackey store.peers.prins_r store.peers.prins_r0 in
   let  macmsg = mac mackeyrr0 content in
   let  mac_r0l = concat header macmsg in]
  (* ... Mise à jour de la mémoire avec les MACs générées ... *)
  let  macs = nil in
   Pour chaque MAC mentionné par la relation fwd_mac,
  [ let macs = concat store.macs.mac_r0l0 macs in]
  let  value = [ nil ou utf8 (cS x) ou utf8 (cS (int_to_string x))] in
  let  protocol = concat macs keys in
  let  payload = concat value protocol in
  let  tag = utf8 (cS "r_g̃'") in
  let  content = concat tag payload in
  let  mesg = base64 (concat start content) in
  let  () = psend store.peers.prins_r' mesg in
  store
]
```

FIG. 3.35 – Génération de l'implémentation : `sendWired_r_g̃`

```
Pour chaque rôle r ∈ R dans un état d'envoi g̃,
[ let rec r_g̃ = fun store  → function
Pour chaque message l que l'on peut envoyer à l'état g̃,
  [ | l(payload,next)  →
    let  newSt = sendWired_l_r_g̃ payload store in
  Si l'état atteint est final
    next
  Sinon
    r_g̃' newSt next ]
]
Pour chaque rôle r ∈ R dans un état de réception g̃,
[ let rec r_g̃ = fun store handlers  →
  let  r = receiveWired_r_g̃ store in
Pour chaque séquence visible g̃' venant de r' attendue à l'état g̃,
  [ | Wired_in_r_g̃_r'_g̃'(payload,store)  →
    let  next = handlers.hl (store.peers,payload) in
  Si l'état atteint est final
    next
  Sinon
    r_g̃' newSt next ]
]

Pour chaque rôle r ∈ R \ r0,
[ let r (host:principal) = fun user_input  →
  let  empty_store = empty_store_a host in
  let  () = bind host in
  let  result = r_start empty_store user_input in
  result
]

let  r0 (peers:prins) = fun user_input  →
  let  empty_store = empty_store_r0 peers in
  let  () = bind peers.prins_r0 in
  let  result = r0_start empty_store user_input in
  result
```

FIG. 3.36 – Génération de l'implémentation : $r\_\widetilde{g}$ et r

d'état local et sont ainsi mutuellement récursives lorsqu'une boucle est présente (de la même manière que les types de pilotage). Lorsque le rôle r est dans un état d'envoi $\widetilde{g}$, la fonction $r\_\widetilde{g}$ envoie par un appel à `sendWired_`l`_`$r\_\widetilde{g}$ le message l que l'utilisateur a spécifié. Lorsqu'il s'agit d'un état de réception, la fonction $r\_\widetilde{g}$ commence par un appel à `receiveWired_`$r\_\widetilde{g}$. Le résultat, du type `wired_`$r\_\widetilde{g}$, est déconstruit, et la continuation de `handlers` que l'utilisateur a fait correspondre au message l reçu est appelée. Ces fonctions vérifient les types `msg`n décrits à la section 3.3.2 puisqu'ils suivent le graphe d'état local, qui n'est qu'une version partiellement dépliée du processus local décrivant le rôle.

Les fonctions initiales r des rôles ont pour tâche d'allouer une mémoire locale à l'aide des fonctions `empty_store_`r, de lancer la session et d'en renvoyer le résultat. Le rôle initial r_0 a droit à un traitement particulier puisqu'il lui revient de préciser l'ensemble des principaux participants à la session.

3.8 Théorèmes d'intégrité

Dans le but de prouver le théorème 3.2, nous définissons ici une sémantique plus précise permettant de prendre en compte les actions de l'adversaire et une définition formelle de la relation entre programmes de haut-niveau et leur implémentation de bas-niveau. Nous aboutissons alors à un théorème 3.5 permettant de prouver le théorème 3.2.

3.8.1 Sémantique étiquetée pour l'adversaire

L'énoncé du théorème 3.2 repose sur une quantification sur l'ensemble des programmes O que l'adversaire peut écrire. Pour faire la preuve, il est plus simple d'utiliser une formulation différente qui caractérise l'adversaire par les actions qu'il peut réaliser et non plus par quels moyens. Nous donnons dans cette section la définition d'une sémantique étiquetée pour l'adversaire, maintenant représenté par un environnement abstrait.

Environnement/Adversaire

Dans les transitions décrites dans la section 3.4.2, il n'était pas tenu compte d'une façon particulière des sessions auxquelles l'adversaire participait. Les transitions que nous décrivons maintenant possèdent un environnement représentant l'adversaire et mémorisant les valeurs et sessions lui étant accessibles. De la même manière qu'à la section 3.4.2, le système de transition est donné en deux variantes, pour F et pour F+S.

Nous utilisons la lettre K pour représenter les connaissances et capacités de l'environnement/adversaire. Initialement, K contient ce que les interfaces laissent voir des bibliothèques à l'adversaire, et notamment les clés de vérification de tous les principaux, les clés privés des principaux corrompus, un certain nombre de nonces, ainsi que toute valeur exportée par U. Dans sa version haut-niveau, K contient initialement les éléments supplémentaires suivants : les clés privées des principaux honnêtes et un ensemble $\mathcal{N}$ de nonces additionnels. Ces deux éléments ne donnent aucun pouvoir supplémentaire à l'attaquant de haut-niveau puisque les participants n'en font aucun usage, mais facilitent les preuves et simplifient l'énoncé du théorème 3.5 en y évitant les questions de renommage de clés et de noms frais.

L'ensemble K croît à mesure que l'adversaire obtient de nouvelles valeurs en espionnant les messages passant sur le réseau. $\mathsf{Val}(K, \rho)$ représente l'ensemble des valeurs qui peuvent être construites en appliquant de façon répétée les constructeurs de type de ρ aux éléments de K et aux constantes des types de base.

Dans F+S, pour chaque session en cours d'exécution présente dans ρ, K contient l'état $s.p$ de chacun des rôles joués par les principaux malhonnêtes. Nous utilisons les notations suivantes pour manipuler les états de ces rôles : $K = K'[\sigma]$ signifie soit que K contient l'état $\sigma = s.p$ de l'exécution d'une session $s \in \rho$, soit que $K = K'$ et $\sigma = S.r_i^\bullet\, \widetilde{a}$ avec $\texttt{safe}\ (a_i) = \text{false}$.

Règles de réduction

La figure 3.37 définit les transitions des configurations de F et F+S avec K. La règle (KApply) permet à l'environnement d'appliquer des fonctions, pourvu que celles-ci, ainsi que leurs arguments, soient connues et que la fonction soit pure (ρ reste inchangé). Le reste des fonctions peut être appliqué indirectement via des communications.

Les règles (KSend) et (KRecv) correspondent aux communications ordinaires ayant lieu sur des canaux. La règle (KStep) permet à un processus P d'avancer dans son exécution silencieusement.

Les règles (KSendS) et (KRecvS) représentent les envois et réceptions au sein des sessions. Celles-ci utilisent les transitions $K, \rho \xrightarrow{\alpha}_{\mathsf{o}} K', \rho'$ pour modéliser les parties des

$$(\textsc{KApply})\ \frac{l_i, v_0, \ldots, v_k \in \mathsf{Val}(K,\rho) \quad (l_i\ x_0 \ldots x_k = e) \in \rho \quad \rho, l_i\ v_0 \ldots v_k \longrightarrow_{\mathsf{e}}{}^{*} \rho, w}{K, \rho, P \rightarrow_{\mathsf{K}} K \cup \{w\}, \rho, P}$$

$$(\textsc{KSend})\ \frac{\rho, P \xrightarrow{\overline{c}\ v}_{\mathsf{P}} \rho, P' \qquad c \in K}{K, \rho, P \xrightarrow{\overline{c}\ v}_{\mathsf{K}} K \cup \{v\}, \rho, P'} \qquad (\textsc{KRecv})\ \frac{\rho, P \xrightarrow{c\ v}_{\mathsf{P}} \rho, P' \quad c, v \in \mathsf{Val}(K,\rho)}{K, \rho, P \xrightarrow{c\ v}_{\mathsf{K}} K, \rho, P'}$$

$$(\textsc{KStep})\ \frac{\rho, P \rightarrow_{\mathsf{P}} \rho', P'}{K, \rho, P \rightarrow_{\mathsf{K}} K, \rho', P'}$$

$$(\textsc{KSendS})\ \frac{\rho, P \xrightarrow{s\overline{g}\ v}_{\mathsf{P}} \rho', P' \quad K, \rho' \xrightarrow{sg}_{\mathsf{o}} K', \rho''}{K, \rho, P \xrightarrow{s\overline{g}\ v}_{\mathsf{K}} K' \cup \{v\}, \rho'', P'}$$

$$(\textsc{KRecvS})\ \frac{K, \rho \xrightarrow{s\overline{g}}_{\mathsf{o}} K', \rho' \quad \rho', P \xrightarrow{sg\ v}_{\mathsf{P}} \rho'', P' \quad v \in \mathsf{Val}(K,\rho)}{K, \rho, P \xrightarrow{sg\ v}_{\mathsf{K}} K', \rho'', P'}$$

$$(\textsc{OStep})\ \frac{\rho, \sigma \xrightarrow{\eta}_{\mathsf{s}} \rho', s.p}{K[\sigma], \rho \xrightarrow{s\eta}_{\mathsf{o}} K[s.p], \rho'} \qquad (\textsc{OComm})\ \frac{K, \rho \xrightarrow{s\overline{g}}_{\mathsf{o}} \xrightarrow{sg}_{\mathsf{o}} \xrightarrow{s\overline{f}}_{\mathsf{o}} K', \rho'}{K, \rho \xrightarrow{s\overline{f}}_{\mathsf{o}} K', \rho'}$$

FIG. 3.37 – Transitions des configurations de F et F+S

sessions contrôlées par l'environnement. La règle (OSTEP) permet en particulier à l'environnement d'effectuer les opérations usuelles des sessions (début, envois, réceptions). La règle (OCOMM) sert dans le cas où des messages de session sont censés être échangés entre rôles contrôlés par l'environnement sans l'intervention de participants honnêtes.

Le lemme ci-dessous établit ensuite un lien entre la sémantique par réduction et la sémantique étiquetée de F+S. Un lemme similaire est vrai dans F. Ces lemmes nous permettent de prouver le théorème 3.2 par induction sur les traces. Nous notons $\overset{\varphi}{\Rightarrow}_{\mathsf{K}}$ les séries de transitions qui consistent en une série de transitions observables (dont les étiquettes sont φ) entrelacée par un nombre arbitraire de transitions silencieuses. La preuve de ce lemme est donnée en annexe D.6.

Lemma 3.4 (Correspondance)
Nous avons les transitions $K \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ pour un ensemble de noms frais $\widetilde{n}$ où $\omega \notin \psi$ si et seulement si $\rho, P \,|\, O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$ pour un processus O ne contenant pas ω, ne déconstruisant pas les valeurs de ρ, n'appelant que les fonctions pures de ρ et dont les valeurs qui sont définies dans ρ font toutes partie de $\mathsf{Val}(K,\rho)$.

Relation à l'exécution entre sessions et code généré

L'état de l'implémentation d'un rôle dans F n'est pas entièrement déterminé par la configuration de haut-niveau $H = K, \rho, P$: en plus de la définition de la session S et de l'état courant s présents dans ρ, en plus des sessions actives $s.p(e)$ dans P, l'implémentation a besoin de tenir compte de l'horloge globale, de l'identifiant de la session et de la séquence d'état $\widetilde{g}$. Ces informations sont sauvées dans la variable T, dont les différents éléments sont définis par :

- Pour chaque principal a :
  - $T.cache(a)$ correspond au contenu du cache anti-replay du principal a : c'est un ensemble de couples d'identifiants de session et de rôles $\{\widetilde{sid}, r\}$.
- Pour chaque session en cours d'exécution $s\,(a_i)_{i<n}\,\{\widetilde{r}\} : S$ de ρ :

  - $T.nonce(s)$ est un terme N_s de K ;
  - $T.path(s)$ est un chemin initial $\widetilde{f}$ de S, décoré par des entiers strictement croissants $\widetilde{j}$ et finissant par une étiquette envoyée ou reçue par un principal honnête ;
  - $T.stuck(s)$ est l'ensemble des rôles d'une session qui ont reçu un mauvais message. Dans ce cas, ces rôles sont bloqués.
- T contient un ensemble infini et co-infini de noms frais supplémentaires, $\mathcal{N}$, que nous utilisons pour fournir à l'environnement de haut-niveau un nonce frais à chaque fois que l'environnement de bas-niveau reçoit un nouvel identifiant de session.

Une signature est dite exportée par T lorsqu'il s'agit d'une signature d'une session s de H, dont l'étiquette et l'heure sont celles de $T.path(s)$, signée par un principal honnête, et telle que l'étiquette est visible par un rôle contrôlé par l'adversaire. De façon informelle, ces signatures sont les signatures que l'adversaire a reçu. Celles-ci ne sont pas connues par le code utilisateur tant que l'adversaire n'a pas explicitement décidé de les transmettre.

Nous définissons maintenant la traduction $[\![K, \rho, P]\!]_T$ des configurations de F+S vers les configurations de F.

Traduction de K Les connaissances de l'adversaire de bas-niveau sont les suivantes :
- Les sessions $s.p$ sont remplacées par les nonces $T.nonce(s)$;
- Les signatures exportées par T sont ajoutées ;
- Les clés privées des principaux honnêtes et les nonces supplémentaires $\mathcal{N}$ sont enlevées.

Traduction de ρ L'environnement de bas-niveau est le suivant :
- Les définitions de session $\widetilde{S}$ sont remplacées par les types et fonctions définies dans $M_{\widetilde{S}}$;
- Les sessions en cours d'exécution sont enlevées, ainsi que les nonces de $\mathcal{N}$ n'étant pas dans l'image de $T.nonce$.

Traduction de P Les processus de bas-niveau suivent l'implémentation sécurisée que nous avons définie dans ce chapitre.
- Le processus `F = forward()` est ajouté ;
- Pour chaque principal a, le processus $P_a = \text{send}\, cache_a\, T.cache(a)$ est ajouté, où $cache_a$ est le canal de $\rho_{L\widetilde{S}}$ contenant l'étant du cache anti-replay de a. (Le fonctionnement du cache est défini dans l'annexe C.1.)
- Les sessions en cours d'exécution présentes dans P sont traduites de la manière suivante (r est le rôle de p) :

$$
\begin{array}{rcll}
[\![s.p(w)]\!]_T & = & \mathbf{0} & \text{Si le rôle de } p \text{ est dans } T.stuck(s) \\
[\![s.p(w)]\!]_T & = & r\_\widetilde{g}\ st\ [\![w]\!]_T & \text{ou si } p \text{ est une réception} \\
[\![s.p(e)]\!]_T & = & \text{let } x_s = [\![e]\!]_T \text{ in } r\_\widetilde{g}\ st\ x_s & \text{ou si } p \text{ est un envoi} \\
[\![s.p(e)]\!]_T & = & [\![e]\!]_T & \text{ou si } p \text{ est } \mathbf{0}
\end{array}
$$

où $\widetilde{g}$ (la séquence d'état) et st (l'état de la mémoire locale) sont calculés à partir de $T.nonce(s)$ et $T.path(s)$. Remarquons que les expressions de la forme $S.r\_$ ne sont pas traduites, mais sont maintenant interprétées comme des appels de fonctions usuelles plutôt que des primitives de lancement de session.

Le processus `forward()` mentionné dans la définition de la traduction ci-dessus a pour objet de réaliser l'envoi asynchrone des messages pour le compte de l'adversaire. Il écoute sur le canal de $\texttt{psend}^\bullet$.

Correction de l'implémentation pour la sémantique étiquetée

Nous définissons $\rho_{L\widetilde{S}}$ comme le résultat de l'évaluation déterministe $\varnothing, L\, \widetilde{S}[\_] \rightarrow_{\mathsf{P}}^* \rho_{L\widetilde{S}}, [\_]$,

c'est-à-dire contenant toutes les bibliothèques et implémentations des protocoles après leur initialisation.

Une configuration $H = K, \rho, P$ de F+S est dite *valide* par rapport à T lorsque :

1. ρ contient $\rho_{L\widetilde{S}}$;
2. Chaque valeur de K définie dans $\rho_{L\widetilde{S}}$ est construite à partir des interfaces des bibliothèques de la section 3.6 ;
3. Pour chaque session $s\,(a_i)_{i<n}\{\widetilde{r}\} : S$ de ρ, il y a une session en cours d'exécution $s.p$ dans P pour chaque participant honnête et une session $s.p$ dans K pour chaque participant malhonnête, telles que chacun de ces rôles soit impliqué soit dans un envoi soit dans une réception de messages mentionnés dans $T.path(s)$;
4. Les clés privées des principaux honnêtes n'apparaissent dans P qu'à travers les signatures exportées par T ;
5. P ne contient aucun appel à la bibliothèque `Prins` ;
6. K contient toutes les clés privées ;
7. K et ρ contiennent l'ensemble de nonces supplémentaires $\mathcal{N}$.

Une configuration W de F est une *implémentation correcte* d'une configuration H de F+S lorsque H est valide et $W = [\![H]\!]_T$ pour un état de bas-niveau T. H est dite *avoir des réceptions correctes* lorsque T n'a pas enregistré de messages incorrects pour aucune session.

Une trace de bas-niveau dont les étiquettes sont φ est une *traduction directe* d'une trace de haut-niveau ψ lorsque φ est égale à ψ dans laquelle tous les envois et réceptions de session $s\,\eta$ ont été respectivement remplacés par des réceptions sur le canal $\text{psend}^\bullet$ et des envois sur les canaux de $\text{chans}^\bullet$.

Il est nécessaire de prendre en compte les traces de bas-niveau dans lesquelles certains messages sont ignorés (comme les messages rejoués) ou n'ont pas fini d'être traités par l'implémentation (les signatures n'ont pas encore été vérifiées). Une trace de bas-niveau est une *traduction* d'un trace de haut-niveau lorsqu'elle en est une traduction directe avec des réceptions supplémentaires sur $\text{psend}^\bullet$ entrelacées.

Un attaquant de bas-niveau peut copier les signatures reçues des principaux honnêtes pendant les exécution de session et les envoyer sur d'autres canaux (non-réservés aux sessions). Pour pouvoir refléter ce comportement à haut-niveau, l'adversaire du haut-niveau doit pouvoir recréer ces signatures vu que les communications par les sessions ne les créent plus. C'est la raison pour laquelle l'environnement de haut-niveau K contient les clés privées de tous les principaux. Notons que cette connaissance par l'environnement de haut-niveau de l'ensemble des clés ne rend pas l'attaquant plus puissant puisque, à haut-niveau, les sessions n'utilisent aucune cryptographie. La traduction vers le bas-niveau retire bien entendu de K les clés des principaux honnêtes et s'occupe d'α-renommer vers les nonces supplémentaires $\mathcal{N}$ les valeurs les employant.

Notre second théorème d'intégrité établit que tous les évènements se produisant à bas-niveau sur le réseau sont explicables par la sémantique de haut-niveau. Les attaquants de bas-niveau ne peuvent ainsi rien faire de plus en utilisant l'implémentation des sessions qu'en utilisant leur sémantique abstraite. La preuve se trouve dans l'annexe D.5.

Theorem 3.5 (Correction) Soit W une implémentation valide de H. Pour toute transition $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ dans F, il existe une implémentation valide W° de H° et une traduction φ de ψ telles que

$$H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ \qquad W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{K}} W'' \qquad W' \rightarrow^*_{\mathsf{K}} W''$$

Dans l'énoncé du théorème, à cause des vérifications silencieuses effectuées par l'implémentation, les transitions de haut-niveau $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ$ utilisées pour simuler $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ n'aboutiront pas exactement à des configurations correspondantes. C'est pourquoi il est nécessaire d'introduire les transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{K}} W''$ et $W' \rightarrow^*_{\mathsf{K}} W''$ qui, silencieusement, permettent aux configurations de haut et bas-niveau de correspondre.

Le théorème 3.5 permet, grâce au lemme 3.4, de démontrer le théorème 3.2.

Notre dernier théorème exprime de façon symétrique la complétude (*completeness*) de notre implémentation, sous l'hypothèse de l'absence d'erreur de réception. Ce théorème assure en particulier la correction de notre implémentation en l'absence d'adversaire. La démonstration est située dans l'annexe D.1.

Theorem 3.6 (Completeness) Soit W une implémentation valide d'une configuration H dont les réceptions sont correctes. Pour chaque transition $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H'$ dans F+S telle que ψ ne contient aucune clés privée des principaux honnêtes ni aucun élément de $\mathcal{N}$, il existe une implémentation valide W' de H' et une traduction directe φ de ψ telles que $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ dans F.

Ce théorème ne tient pas compte des transitions impliquant les clés privés des principaux honnêtes ou des nonces provenant de $\mathcal{N}$. Ces clés et nonces ne sont pas connus par l'adversaire de bas-niveau et sont pour cette raison α-renommés au moment de la traduction vers le bas-niveau (comme expliqué plus haut).

3.9 Résultats expérimentaux

Pour illustrer la facilité d'utilisation de nos sessions, nous présentons deux exemples supplémentaires et discutons des performances expérimentales de notre système.

3.9.1 Exemples supplémentaires

Les exemples suivants sont d'une complexité plus grande que les exemples utilisés jusqu'ici. Ils permettent de mettre en évidence la capacité de notre compilateur à traiter les sessions les plus complexes.

Un système de gestion de conférence

Exemple 3.33 (Conf) La figure 3.38 donne le graphe d'une session représentant le comportement simplifié d'un système de gestion de conférence. Cette session comporte trois rôles : le *program committee* `pc`, l'auteur `author` et le *conference manager* `confman`.

Après l'envoi par le `pc` d'un appel à contribution `Cfp`, l'auteur soumet un papier par le message `Upload` au `confman`. Celui-ci décide si le format est correct (message `Ok`) ou non (message `BadFormat`) dans quel cas l'auteur est invité à fournir une version corrigée. Une fois le format correct, l'auteur peut soumettre (message `Submit`) son papier au `confman`

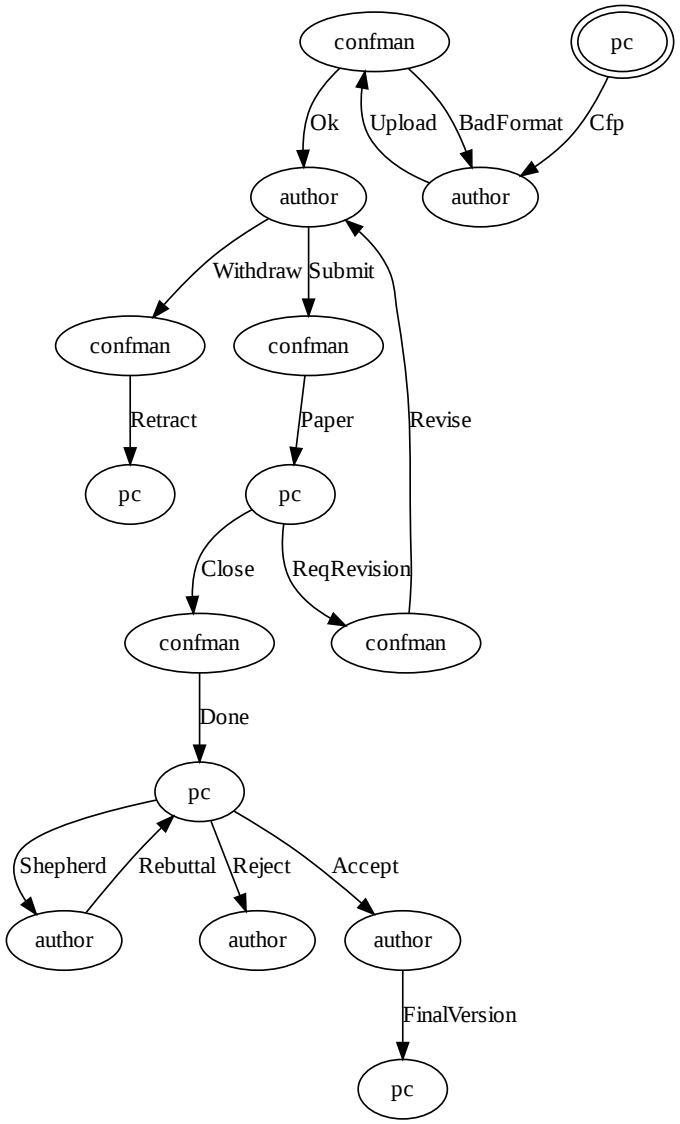


FIG. 3.38 – Un système de gestion de conférence simplifié

qui le transmet (message `Paper`) au `pc` pour appréciation. Le `pc` peut alors demander des révisions (messages `ReqRevision` puis `Revise`) ou s'arrêter là en fermant le site de soumission (messages `Close` et `Done`). L'auteur a aussi la possibilité de se retirer (messages `Withdraw` et `Retract`) au lieu de soumettre une nouvelle version. Une fois le `confman` arrêté, le `pc` donne sa décision à l'auteur : le papier peut être rejeté (`Reject`), mis en discussion (boucle `Shepherd-Rebuttal`) ou accepté (messages `Accept` puis `FinalVersion`).

Le code source de cette session est donné en annexe C.2.

Le processus législatif (simplifié)

Exemple 3.34 (Loi) La figure 3.39 donne le graphe d'une session représentant une version simplifiée d'une partie du processus législatif français. Cette session comporte six rôles : le gouvernement, un ministre, le conseil d'Etat, une commission, l'assemblée et un député.

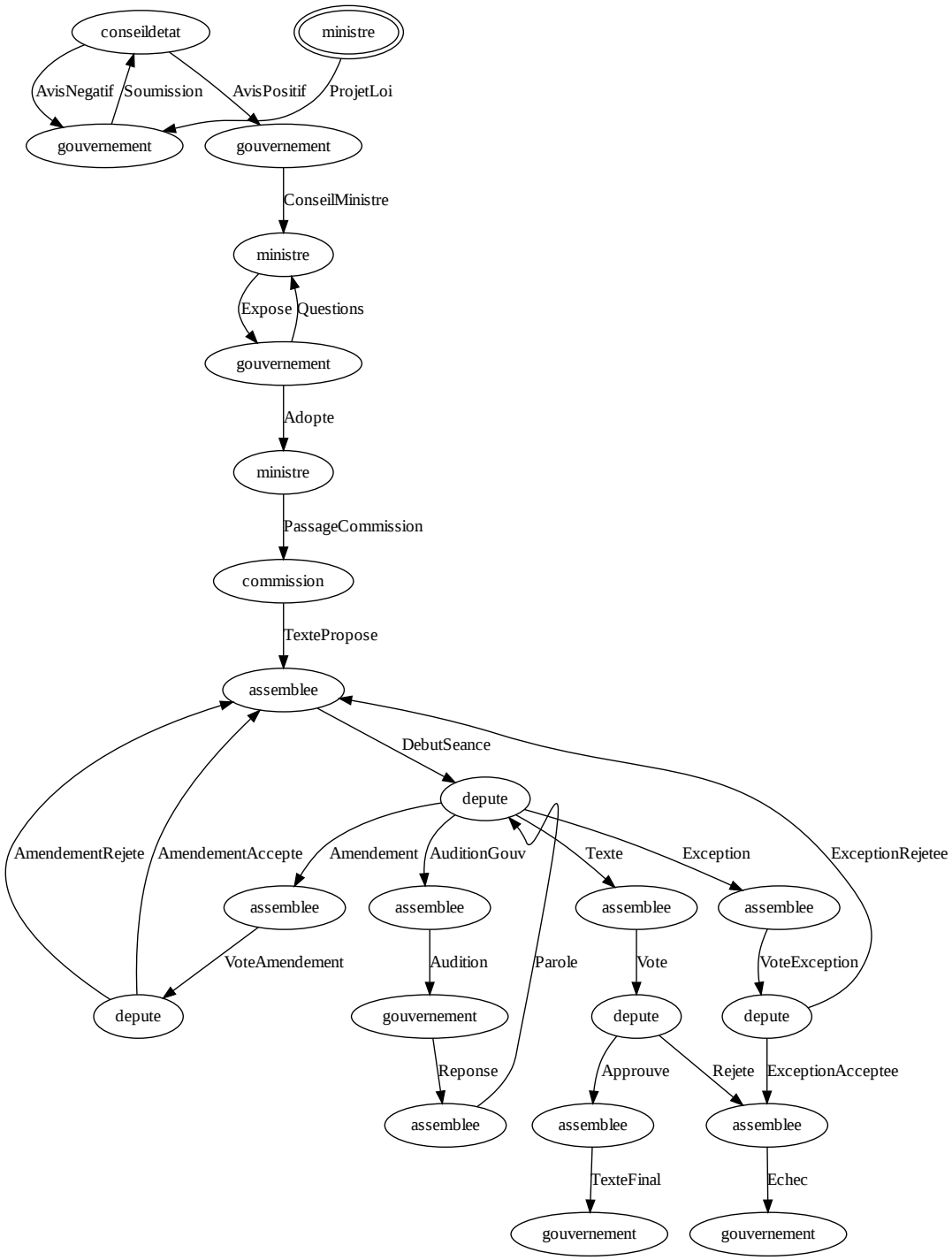


FIG. 3.39 – Session `Loi` : une partie simplifiée du système législatif

La session démarre par la soumission par un ministre d'un projet de loi (message `ProjetLoi`) au gouvernement. Celui-ci le transmet au conseil d'Etat (message `Soumission`) pour avis. Celui-ci donne son avis (messages `AvisPositif` et `AvisNegatif`), le gouvernement pouvant resoumettre tant que l'avis est négatif. Le projet de loi est ensuite discuté en conseil des ministres (message `ConseilMinistre`) durant lequel le ministre le présente (message `Expose`) et répond aux questions. Une fois accepté (message

`Adopte`), le gouvernement, via le ministre, transmet le projet à une commission de l'Assemblée (message `PassageCommission`) qui modifie le texte avant sa discussion (message `TextePropose`). Une fois en séance (message `DebutSeance`), le texte peut être amendé (messages `Amendement-VoteAmendement`, `AmendementAccepte` ou `AmendementRejete`), le gouvernement peut être interrogé (messages `AuditionGouv-Audition-Reponse-Parole`), une exception peut être soulevée et votée (messages `Exception-Vote`, `ExceptionAccepte` ou `ExceptionRejetee-Echec`), ou un vote solennel de l'ensemble du texte peut être demandé (`Texte-Vote`, puis `Approuve-TexteFinal` ou `Rejete-Echec`).

Le code source de cette session est donné en annexe C.3.

3.9.2 Evaluation

Afin d'évaluer la facilité d'utilisation de notre système, nous donnons dans la figure ci-dessous quelques chiffres se rapportant aux différents exemples utilisés : `Single` (exemple 3.1), `Rpc` (exemple 3.3), `Forward` (exemple 3.4), `Ws` (exemple 3.5), `Wsn` (exemple 3.6), `Wsne` (figure 3.6), `Shopping` (exemple 3.7), `Conf` (exemple 3.33) et `Loi` (exemple 3.34).

Programmation

Les données récoltées sont le nombre de rôles (entre 2 et 6), la taille du fichier décrivant la session, la taille d'une application test permettant de jouer tous les rôles de la session, une mesure de la taille du graphe de la session et des graphes d'état locaux utilisés par l'implémentation, la taille de l'interface générée, la taille du code généré, le temps total de la génération du code et de la compilation. Les tailles des fichiers sont données en nombre de lignes de code (loc). Les tailles des graphes sont exprimées comme la taille de leur représentation générée en format `dot` [28]. Cela correspond approximativement à la somme du nombre de nœuds et du nombre d'arêtes. Le temps de génération et de compilation est obtenu sur une machine Core 2 U7600 à 1.20GHz avec 2Go RAM, avec Linux 2.6.29 et Ocaml 3.11.0.

Session `S`	Rôles	Fichier `.session` (loc)	Application (loc)	Graphe (loc)	Graphes Locaux (loc)	`S.mli` (loc)	`S.ml` (loc)	Compilation (s)
`Single`	2	5	21	8	12	19	247	1.26
`Rpc`	2	7	25	10	18	23	377	1.35
`Forward`	3	10	33	12	25	34	632	1.66
`Auth`	4	15	45	16	38	49	1070	1.86
`Ws`	2	7	33	12	24	25	481	1.36
`Wsn`	2	15	44	13	42	29	782	1.50
`Wsne`	2	19	45	15	48	31	881	1.90
`Shopping`	3	29	70	21	85	49	1780	2.43
`Conf`	3	48	86	37	181	78	3451	3.32
`Loi`	6	101	189	57	310	141	7267	6.29

Ces chiffres nous permettent d'apprécier la qualité de notre système du point de vue du programmeur. Même pour les sessions les plus complexes, la description d'une session ne prend pas plus d'une centaine de lignes. Une fois écrite, la session n'est pas non plus très compliquée à utiliser : l'interface à suivre a une taille de l'ordre du double de la taille de la session. L'écriture d'une application test jouant tous les rôles de la session est donc simple, comme le montre la taille très réduite des applications utilisant ces quelques exemples.

Le bénéfice de l'utilisation des sessions ne s'apprécie enfin qu'en regardant la taille du code généré pour leur implémentation sécurisée. Cette implémentation, qui est réalisée aujourd'hui à la main et sans garantie formelle dans la majorité des grands systèmes distribués, est ici automatiquement générée et représente dans nos exemples jusqu'à 7000 lignes de code relativement complexe. Cette taille est à mettre en regard avec la modeste taille du compilateur : le code source de `s2ml` possède 4493 lignes de codes tandis que chaque version des bibliothèques fait de l'ordre de 700 lignes.

Notre dernier exemple montre enfin le bon comportement de la version courante de notre implémentation : la taille de la session `Loi` demande un temps de calcul plus important, mais qui reste raisonnable.

Performances

Nous avons fait tourner l'exemple `Conf` en utilisant la version OpenSSL des bibliothèques. Nous avons paramétré le nombre de tours de boucles de façon à ce que l'exécution de la session corresponde à l'échange de 10000 messages. Le tableau ci-dessous donne les temps d'exécution en secondes obtenus sur la machine mentionnée précédemment. Les communications sont locales de manière à ne pas mesurer la latence du réseau. La session a été exécutée en faisant varier plusieurs paramètres de notre protocole. Tout d'abord nous avons comparé la version utilisant des MACs avec une version utilisant à la place des signatures. Nous avons aussi mesuré le temps passé à la vérification que les MACs (ou les signatures) sont corrects. Nous avons ensuite mesuré le temps d'exécution lorsque les MACs et signatures sont absents (pas de production ni de vérification), et enfin lorsque même les vérifications du cache et l'établissement des clés partagées ont été oubliés (les messages ne portent plus que les valeurs données par l'utilisateur. Il va de soit que seules les versions utilisant toute la cryptographie (avec signatures ou MACs) jouissent des garanties de notre théorème.

Authentification avec	signatures	MACs
Temps total	93.92	1.77
Sans vérification	90.80	1.66
Sans cryptographie	1.43	
Communications seules	1.31	

Les résultats ci-dessus montrent deux choses. Premièrement, les choix d'utiliser les MACs pour authentifier les messages visibles est pertinent du point de vue des performances. Le coût de l'établissement des clés partagées est incomparablement plus faible que d'utiliser des signatures. Deuxièmement, lorsque les MACs sont utilisées, le coût d'utilisation du protocole cryptographique est raisonnable : de l'ordre de 35 %. L'implémentation statique (aucun calcul sur le graphe n'est effectué à l'exécution) et l'utilisation de la notion de visibilité en sont sans doute responsables.

Tous les exemples présentés dans ce chapitre, ainsi que le code source complet du compilateur `s2ml` et des bibliothèques sont téléchargeables en ligne [23].

Chapitre 4

Conclusion

4.1 Principaux résultats obtenus

Nous avons présenté dans cette thèse deux approches différentes des problèmes soulevés par la programmation distribuée.

Dans le chapitre 2, nous nous sommes intéressés à la préservation dans l'environnement réparti de l'abstraction locale que sont les types abstraits. Nous avons créé, à partir d'un langage existant, un calcul comportant modules, types abstraits et sous-typage qui satisfait les propriétés de préservation du typage et de progrès. Le sous-typage, que nous autorisons entre types abstraits et concrets, permet une plus grande flexibilité et donne notamment la possibilité à un système distribué d'accepter certaines mises à jour de modules et de programmes. Nous avons aussi discuté des conditions d'implémentation d'un tel langage.

Dans le chapitre 3, nous avons détaillé la conception et la génération d'une implémentation distribuée sécurisée des sessions. Nous avons tout d'abord défini un langage pour décrire cette spécification globale des interactions d'un système réparti. Nous avons ensuite proposé une famille de protocoles cryptographiques permettant à un participant d'une session d'avoir la garantie d'une exécution satisfaisante de cette dernière, en dépit de la corruption d'autres participants. Nous avons enfin présenté notre compilateur accompagné de données expérimentales validant notre approche.

4.2 Travaux futurs

Les travaux présentés dans cette thèse ne sont qu'une étape dans le chemin qui mène à une plus grande facilité de programmation des systèmes répartis et à une plus grande confiance en leur performance et fiabilité. Nous donnons ici quelques idées qui, si elles sont explorées, complètent ou dépassent les contributions mentionnées précédemment.

Techniques de preuve

Lorsque les démonstrations des théorèmes dépassent une certaine taille critique, l'utilisation de techniques automatiques de vérification ou d'un assistant de preuve deviennent préférables, voire nécessaires, pour s'assurer de la correction des arguments employés. Les preuves présentées dans cette thèse tombent dans cette catégorie et bénéficieraient d'une telle formalisation.

Deux approches sont possibles. D'un côté, nous pouvons transposer les preuves existantes dans un assistant de preuve comme Coq [27], Twelf [58] ou HOL/Isabelle [55]. Les travaux effectués dans le cadre du concours Poplmark [1] ou dans le cadre de la preuve de la correction du langage ML [45] concernent des preuves similaires et donnent une idée

des méthodes envisageables.

D'un autre côté, lorsque du code est généré, comme dans le cas de notre compilateur de session `s2ml`, des annotations peuvent y être ajoutées et faciliter une vérification automatique du code produit, par typage par exemple. C'est la solution que nous avons suivie dans nos travaux les plus récents sur la génération d'implémentations sécurisées de sessions [8].

Prototypes

Notre calcul explorant la préservation des abstractions dans un langage distribué en présence de sous-typage gagnerait à être intégré à un langage de programmation usuel comme Ocaml. Le travail effectué sur HashCaml [10] est une base de travail remarquable. Comme Ocaml ne comporte pas de sous-typage entre les enregistrements, il est logique dans un premier temps d'adapter notre travail au cas des objets (mais le typage est rendu plus complexe du fait du polymorphisme). Une intégration de la solution des crochets colorés avec un très expressif langage de module a déjà été étudiée dans la thèse de Gilles Peskine [57].

Concernant notre compilateur de sessions, de nombreuses améliorations sont possibles. Une intégration de la syntaxe des sessions à Ocaml via, par exemple, un plugin Camlp4 [48] simplifierait les étapes de compilation pour le programmeur. Le processus de compilation peut aussi être accéléré par des algorithmes plus efficaces que ceux utilisés sur les graphes de session pour la vérification des propriétés ou la génération du code. Une autre amélioration concerne enfin les bibliothèques : nous utilisons par simplicité nos propres liens vers OpenSSL ou .Net qui ne contiennent que le strict nécessaire. Utiliser des liens standard permettrait de partager la maintenance et le développement.

Valeurs dans les sessions

Les sessions présentées dans cette thèse se préoccupent des étiquettes des messages et de leurs séquences. Les valeurs transmises grâce aux messages par les utilisateurs d'un rôle à l'autre sont d'un égal intérêt. D'une part, nous pouvons étendre les types des valeurs transmises par les sessions au-delà des types de base (`unit`, `int`, `string`) : utiliser le travail effectué dans le chapitre 2 peut permettre la transmission de valeurs de type plus complexes, comme les types abstraits. L'utilisation de HashCaml pourrait être une première étape. Symétriquement, la sécurisation de HATS ou HashCaml est définitivement un enjeu intéressant.

Une autre idée est d'intégrer les valeurs transmises dans l'abstraction que forment les sessions : chaque session abstrait alors l'implémentation distribuée d'un espace de stockage global où les rôles peuvent écrire ou lire suivant les permissions données par la spécification de la session. Comme les rôles sont eux-mêmes des variables globales instantiées par des principaux, le choix dynamique des participants à une session peut se faire dans ce cadre. Cette solution a été présentée dans [8] et a été implémentée.

Des sessions plus expressives

Les sessions décrites dans le chapitre 3 ont une sémantique linéaire (elles s'exécutent comme des automates à états finis). Il est ainsi intéressant de considérer l'enrichissement de notre langage de sessions pour qu'il supporte, à la manière du calcul CCS [49], la possibilité de suivre plusieurs branches de manière concurrente (constructeur parallèle |). Les traces acceptables peuvent alors comporter un certain entrelacement des messages entre branches concurrentes. Cette extension a été réalisée [61], l'implémentation reste à effectuer.

Le multicast est de même un mécanisme usuel des protocoles : une situation fréquente est l'envoi simultané d'une requête à un ensemble de serveurs dont seule la réponse la plus rapide est prise en compte. Certains calculs de session utilisent cette sémantique [5].

Délégation

Notre langage de session suppose le fait que l'ensemble des participants à la session est déterminé lors de l'envoi du premier message. Un choix retardé de certains participants (présent dans [8]), voire la possibilité de réaffecter un rôle à un principal différent au cours de l'exécution permettrait d'améliorer l'expressivité de notre langage. Il serait intéressant notamment d'examiner les questions de sécurité soulevées par une telle capacité.

Plus de sécurité

D'un côté, la question de la sécurité de HATS ou de HashCaml est définitivement un enjeu intéressant. D'un autre, il existe des propriétés de sécurité complémentaires de celle que nous avons démontrée pour notre implémentation des sessions. Il peut être ainsi intéressant de formaliser et prouver des propriétés de confidentialité (par exemple en utilisant les technique de [2]) ou de progrès (en utilisant par exemple des méthodes standard contre les attaques par déni de service).

Sessions dynamiques et compositionnalité

Les sessions sont vérifiées statiquement et contrôlées dynamiquement, mais dans certains cas il pourrait être intéressant d'élaborer une session dynamiquement, avant de l'exécuter. Un exemple simple serait une session paramétrée par un entier spécifiant le nombre de tours de boucle [71]. Cet entier serait calculé avant l'exécution de la session et serait ensuite intégré au protocole généré. Une autre idée serait de fonctoriser (en Ocaml) le module généré en donnant la possibilité de paramétrer les types des valeurs envoyées dans les messages.

Dans cette optique, les sessions bénéficieraient d'une certaine compositionnalité. Il serait avantageux de pouvoir mettre bout à bout deux sessions ou de répéter une session en obtenant une propriété globale. De même, si un rôle déclare qu'il accepte les message `Request` et répond à chaque fois par un message `Reply`, pourquoi ne pourrait-il pas participer à toutes les sessions qui réclament ce comportement ? Les questions cruciales se rapportent évidemment à la préservation de la propriété d'intégrité et à l'implémentation qui, toutes deux, dépendent de la spécification globale.

Optimisations grâce aux relations de confiance

Le modèle du protocole cryptographique généré pour les sessions considère que tous les rôles peuvent être compromis. Dans bon nombre de cas, cette supposition n'est pas nécessaire car une certaine relation de confiance entre les participants est connue. Par exemple, lors d'une transaction commerciale, les participants peuvent décider de faire confiance à la banque. Lorsque de telles relations de confiance existent, il est possible de relâcher les conditions d'implémentation (voir section 3.5) et d'accepter ainsi plus de sessions sans rien changer à l'implémentation.

Les relations de confiance peuvent de plus permettre de ne pas effectuer toutes les vérifications que l'implémentation demande : lorsqu'on lui fait confiance, il n'est plus nécessaire de demander à la banque si la session s'est déroulée sans accroc jusqu'à son implication. La notion de confiance peut enfin être utile pour traiter la délégation dans les sessions.

Annexes

Plan

L'annexe A donne la définition complète de la syntaxe, du système de typage et de la sémantique du langage HATS obtenu à la fin du chapitre 2.

L'annexe B donne les preuves de l'ensemble des théorèmes mentionnés dans le chapitre 2.

L'annexe C présente les définitions des bibliothèques décrites à la section 3.6 du chapitre 3. Un exemple y est aussi détaillé en donnant notamment des extraits du code généré et des exemples de messages échangés.

L'annexe D, elle, démontre les théorèmes énoncés au chapitre 3.

L'annexe E donne un index des exemples, définitions, lemmes et théorèmes de cette thèse.

Note

Les annexes de cette thèse sont rédigées en anglais.

Annexe A

HATS : syntax, type system, semantics

In this chapter, we give the complete description of the HATS language.

A.1 Syntax

$e ::=$	expression
$()$	unit
$(e_1, ..., e_j)$	tuple $(2 \leqslant j)$
$\mathbf{proj}_i\, e$	projection
$\{l_1 = e_1, ..., l_j = e_j\}$	record $(1 \leqslant j)$
$e.l_i$	field
x	variable
$\lambda x : T.e$	function (x binds in e)
$e\, e$	application
$\mathbf{mar}\,(e : T)$	dynamic
$\mathbf{marshalled}_{\,c,H}(e : T)$	closed, colour-independent dynamic
$\mathbf{unmar}\, e : T$	undynamic
$!\, e$	send
$?$	receive
$U.\texttt{term}$	term-part of a module
$_{(T_1 <: T_2)}e$	explicit subtyping
$[e]_c^T$	type colouring
$\mathbf{Unmarfailure}^T$	undyn failure

$v^{c_0} ::=$	c_0-value, i.e. value in the colour c_0
$\widehat{v}^{c_0}$	values with subtyping
$[\widehat{v}^{c_1 \cup c_0}]_{c_1}^{h_1.\texttt{type}}$	type colouring where $h_1 \notin c_0$

$\widehat{v}^{c_0} ::=$	base values of the colour c_0
$\quad \widehat{\widehat{v}}^{c_0}$	values with subtyping
$\quad (TV_1^{c_0} \texttt{<:} TV_2^{c_0})\widehat{\widehat{v}}^{c_0}$	explicit subtyping where $TV_1^{c_0} = h_0.\texttt{type}$ or $TV_2^{c_0} = h_2.\texttt{type}$
$\widehat{\widehat{v}}^{c_0} ::=$	base values of the colour c_0
$\quad ()$	unit
$\quad (v_1^{c_0}, ..., v_j^{c_0})$	tuple $(2 \leqslant j)$
$\quad \{l_1 = v_1^{c_0}, ..., l_j = v_j^{c_0}\}$	record $(1 \leqslant j)$
$\quad \lambda x : T.e$	function (x binds in e)
$\quad \textbf{marshalled}_{\,c,H}(e : T)$	closed dynamic value
$C_{c_2}^{c_1} ::=$	single-level evaluation context
$\quad (v_1^{c_1}, .., v_{i-1}^{c_1}, \_, e_{i+1}, .., e_j)$	tuple ($2 \leqslant j$ and $1 \leqslant i \leqslant j$), if $c_1 = c_2$
$\quad \textbf{proj}_i \_$	projection, if $c_1 = c_2$
$\quad \{l_1 = v_1^{c_1}, ..., l_{i-1} = v_{i-1}^{c_1}, l_i = \_,$ $\quad\quad l_{i+1} = e_{i+1}, ..., l_j = e_j\}$	record ($1 \leqslant j$ and $1 \leqslant i \leqslant j$), if $c_1 = c_2$
$\quad \_.l_i$	field projection, if $c_1 = c_2$
$\quad \_\, e$	application left, if $c_1 = c_2$
$\quad v^{c_1}\ \_$	application right, if $c_1 = c_2$
$\quad (T_1 \texttt{<:} T_2)\_$	explicit subtyping
$\quad \textbf{mar}\,(\_ : T)$	dynamic, if $c_1 = c_2$
$\quad \textbf{unmar}\ \_ : T$	undynamic, if $c_1 = c_2$
$\quad !\,\_$	send, if $c_1 = c_2$
$\quad [\_]_c^T$	coloured bracket if $c \cup c_1 = c_2$
$CC_{c_2}^{c_1} ::=$	coloured evaluation context
$\quad CC_{c'}^{c_1}.C_{c_2}^{c'}$	extra level
$\quad \_$	identity, if $c_1 = c_2$
$T ::=$	type
$\quad \text{unit}$	unit
$\quad T_1 * ... * T_j$	tuple $(2 \leqslant j)$
$\quad \{l_1 : T_1; ...; l_j : T_j\}$	record $(1 \leqslant j)$
$\quad X$	variable
$\quad T \to T$	function
$\quad \text{bytes}$	dynamic
$\quad U.\texttt{type}$	type-part of a module
$\quad h.\texttt{type}$	hash
$\quad \top$	Top

$TV^c ::=$		types in head normal form in color c
	unit	unit
	$T_1 * ... * T_j$	tuple $(2 \leqslant j)$
	$\{l_1 : T_1; ...; l_j : T_j\}$	record $(1 \leqslant j)$
	$T \rightarrow T$	function
	bytes	dynamic
	h.type	hash if $h \notin c$
	$\top$	Top
$TC ::=$		first-level multi-hole type context
	unit	unit
	bytes	dynamic
	$_{-1} \rightarrow _{-2}$	function
	$_{-1} * ... * _{-j}$	tuple $(2 \leqslant j)$
	$\{l_1 : _{-1}; ...; l_j : _{-j}\}$	record $(1 \leqslant j)$
$c ::=$		colour
	$\bullet$	empty
	h	some hash
	$c \cup c$	union of hash sets
$h ::=$	$\mathbf{hash}(N, M : [X : \mathbf{Le}(T), T'])$	hash
$H ::=$		subhash relationship
	$\varnothing$	empty relationship
	$H \cup \kappa <: \kappa'$	addition of a statement
N		external name
$K ::=$		kind
	$\mathbf{Le}(T)$	(partially) opaque
	$\mathbf{Eq}(T)$	singleton
$M ::=$	$[T, v^\bullet : T']$	module structure (type part, term part)
$S ::=$	$[X : K, T]$	module signature (X binds in T)
$m ::=$		machine
	e	expression
	module $N\,U$ extends $(\kappa_1, ..., \kappa_i)$	module declaration
	restricts $(\kappa'_1, ..., \kappa'_j)$	(U binds in m),
	$= M : S$ in m	$0 \leqslant i$, $0 \leqslant j$.

$n ::=$	network
$\mathbf{0}$	null
$n \mid n$	parallel composition
H, e	expression and subhash on one machine
$\kappa ::=$	elements of the subhash relationship
U	module name
h	hash
$\zeta ::=$	variable
x	expression variable
X	type variable
U	module variable
$\chi ::=$	substitutable entity
X	type variable
$U.\texttt{type}$	type-part of a module
x	expression variable
$U.\texttt{term}$	term-part of a module
U	module variable
$\dddot{X} ::=$	type substitutable entity
X	type variable
$U.\texttt{type}$	type-part of a module
$\ddot{x} ::=$	expression substitutable entity
x	expression variable
$U.\texttt{term}$	term-part of a module
$E ::=$	environment
$E, x : T$	expression variable binding
$E, X : K$	type variable binding
$E, U(T) : S$	module variable binding
$\mathbf{nil}$	empty

$J ::=$	colourable statement
$\quad \mathrm{ok}$	environment correctness
$\quad K\,\mathrm{ok}$	kind correctness
$\quad K == K'$	kind equivalence
$\quad K <: K'$	subkinding
$\quad T : K$	kind of a type
$\quad T == T'$	type equivalence
$\quad T <: T'$	subtyping
$\quad S\,\mathrm{ok}$	signature correctness
$\quad S == S'$	signature equivalence
$\quad S <: S'$	subsignaturing
$\quad e : T$	type of an expression
$\quad M : S$	signature of a module expression
$\quad U : S$	signature of a module variable
$\quad m : T$	type of a machine
$\mathrm{CJ} ::= E \vdash^H_c J$	coloured judgement
$\mathrm{MJ} ::=$	monochrome judgement
$\quad \vdash c\,\mathrm{ok}$	hash set and subhash correctness
$\quad \vdash h\,\mathrm{ok}$	hash correctness
$\quad \vdash h \in c$	hash set membership
$\quad \vdash n\,\mathrm{ok}$	network correctness
$\mathrm{AJ} ::=$	judgement
$\quad \zeta \notin \mathrm{dom}\, E$	non-clash judgement
$\quad \mathrm{CJ}$	coloured judgement
$\quad \mathrm{MJ}$	monochrome judgement

Additionally, we use the following notations :

$\zeta : \tau ::= x : T \mid X : K \mid U(T) : S$	variable has sort
$\eta : \tau ::= e : T \mid T : K \mid M : S$	term has sort
$\tau\,\mathrm{ok} ::= T : \mathbf{Le}(\top) \mid K\,\mathrm{ok} \mid S\,\mathrm{ok}$	sort is correct
$=$	syntactic equality
$\in$	syntactic membership
σ	substitutions
Π	derivation (i.e. proof tree)
$\{\chi \leftarrow \eta\}\aleph$	substitution : replace χ by η in $\aleph$
fv	free variables (U, X, x)
fse	free substitutable entities (U, $U.\mathtt{type}$, $U.\mathtt{term}$, X, x) (if $U.\mathtt{type} \in \mathrm{fse}\,\aleph$ or $U.\mathtt{term} \in \mathrm{fse}\,\aleph$ then $U \in \mathrm{fse}\,\aleph$)

Finally, if $h = \mathbf{hash}(N, [T, v^\bullet : T'] : S)$, then we refer to T by the expression $\mathrm{impl}\,(h)$.

A.2 Typing rules

A.2.1 $\boxed{\zeta \notin \operatorname{dom} E}$ non-clash in environments

$$\frac{}{\zeta \notin \operatorname{dom} \mathbf{nil}}\ (\mathsf{clash.nil}) \qquad \frac{\zeta \notin \operatorname{dom} E \quad \zeta \neq \zeta'}{\zeta \notin \operatorname{dom} (E, \zeta' : \tau)}\ (\mathsf{clash.cons})$$

For any two distinct variables ζ and ζ', $\zeta \neq \zeta'$ is an axiom.

A.2.2 $\boxed{U \notin \operatorname{dom} H}$ non-clash in the subhash relationship

$$\frac{}{U \notin \operatorname{dom} \varnothing}\ (\mathsf{clash.hash.nil}) \qquad \frac{U \notin \operatorname{dom} H \quad U \neq \kappa \quad U \neq \kappa'}{U \notin \operatorname{dom} (H \cup \kappa <: \kappa)}\ (\mathsf{clash.hash.cons})$$

For any two distinct variables U and U', $U \neq U'$ is an axiom.

A.2.3 $\boxed{\vdash h\,\mathrm{ok}}$ hash correctness

$$\frac{\mathbf{nil} \vdash^{H}_{\bullet} M : [X : \mathbf{Le}(T), T']}{\vdash \mathbf{hash}(N, M : [X : \mathbf{Le}(T), T'])\,\mathrm{ok}}\ (\mathsf{hok.hash})$$

Note that :

1. N may be any external name.
2. we demand that a module have an opaque type (at least partially) in order to take its hash.
3. the existence of a correct H is assumed for every hash. However, in implementations, no hash correctness will be checked. If such a H is still required, we can annotate all hashes with the current H of the compilation. The operations on hashes such as equality $=$ or set operations would then need to ignore this annotation.

A.2.4 $\boxed{\vdash c\,\mathrm{ok}}$ hash sets correctness

$$\frac{}{\vdash \bullet\,\mathrm{ok}}\ (\mathsf{hmok.zero}) \qquad \frac{\vdash h\,\mathrm{ok}}{\vdash h\,\mathrm{ok}}\ (\mathsf{hmok.sing}) \qquad \frac{\vdash c_0\,\mathrm{ok} \quad \vdash c_1\,\mathrm{ok}}{\vdash c_0 \cup c_1\,\mathrm{ok}}\ (\mathsf{hmok.union})$$

A.2.5 $\boxed{E \vdash^{H}_{c}\ \mathrm{ok}}$ environment correctness

$$\frac{\vdash c\,\mathrm{ok}}{\mathbf{nil} \vdash^{\mathbf{nil}}_{c}\ \mathrm{ok}}\ (\mathsf{envok.nil}) \qquad \frac{\begin{array}{l} E \vdash^{H}_{c}\ \mathrm{ok} \\ E \vdash^{H}_{\bullet} T : \mathbf{Le}(\top) \\ x \notin \operatorname{dom} E \end{array}}{E, x : T \vdash^{H}_{c}\ \mathrm{ok}}\ (\mathsf{envok.x})$$

$$\frac{\begin{array}{l}\vdash \mathbf{hash}(N_0, [T_0, v_0^\bullet : T_0'''] : [X : \mathbf{Le}(T_0''), T_0']) \,\mathrm{ok} \\ \vdash \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1'''] : [X : \mathbf{Le}(T_1''), T_1']) \,\mathrm{ok} \\ \mathbf{nil} \vdash_c^H \,\mathrm{ok} \\ \mathbf{nil} \vdash_\bullet^H T_0 <: T_1 \\ \mathbf{nil} \vdash_\bullet^H \mathbf{Le}(T_0'') <: \mathbf{Le}(T_1'')\end{array}}{\mathbf{nil} \vdash_c^{H \cup \mathbf{hash}(N_0, [T_0, v_0^\bullet : T_0''']:[X:\mathbf{Le}(T_0''), T_0']) <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''']:[X:\mathbf{Le}(T_1''), T_1'])} \,\mathrm{ok}} \text{ (envok.hashhash)}$$

$$\frac{\begin{array}{l}\vdash \mathbf{hash}(N_0, [T_0, v_0^\bullet : T_0'''] : [X : \mathbf{Le}(T_0''), T_0']) \,\mathrm{ok} \\ E, U(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_c^H \,\mathrm{ok} \\ E, U(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_\bullet^H T_0 <: T_1 \\ E, U(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_\bullet^H \mathbf{Le}(T_0'') <: \mathbf{Le}(T_1'')\end{array}}{E, U(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_c^{H \cup \mathbf{hash}(N_0, [T_0, v_0^\bullet : T_0''']:[X:\mathbf{Le}(T_0''), T_0']) <: U} \,\mathrm{ok}} \text{ (envok.hashU)}$$

$$\frac{\begin{array}{l}\vdash \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1'''] : [X : \mathbf{Le}(T_1''), T_1']) \,\mathrm{ok} \\ E, U(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_c^H \,\mathrm{ok} \\ E, U(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_\bullet^H T_0 <: T_1 \\ E, U(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_\bullet^H \mathbf{Le}(T_0'') <: \mathbf{Le}(T_1'')\end{array}}{E, U(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_c^{H \cup <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''']:[X:\mathbf{Le}(T_1''), T_1'])} \,\mathrm{ok}} \text{ (envok.Uhash)}$$

$$\frac{\begin{array}{l}E_1, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'], E_2, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_c^H \,\mathrm{ok} \\ E_1, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'], E_2, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_\bullet^H T_0 <: T_1 \\ E_1, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'], E_2, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_\bullet^H \mathbf{Le}(T_0'') <: \mathbf{Le}(T_1'')\end{array}}{E_1, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'], E_2, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'] \vdash_c^{H \cup U_0 <: U_1} \,\mathrm{ok}} \text{ (envok.mod)}$$

$$\frac{\begin{array}{l}E_1, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'], E_2, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_c^H \,\mathrm{ok} \\ E_1, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'], E_2, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_\bullet^H T_0 <: T_1 \\ E_1, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'], E_2, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_\bullet^H \mathbf{Le}(T_0'') <: \mathbf{Le}(T_1'')\end{array}}{E_1, U_0(T_0) : [X : \mathbf{Le}(T_0''), T_0'], E_2, U_1(T_1) : [X : \mathbf{Le}(T_1''), T_1'] \vdash_c^{H \cup U_0 <: U_1} \,\mathrm{ok}} \text{ (envok.modopp)}$$

$$\frac{\begin{array}{l}E \vdash_c^H \,\mathrm{ok} \\ E \vdash_\bullet^H K \,\mathrm{ok} \\ X \notin \mathrm{dom}\, E\end{array}}{E, X : K \vdash_c^H \,\mathrm{ok}} \text{ (envok.X)} \qquad \frac{\begin{array}{l}E \vdash_c^H \,\mathrm{ok} \\ E \vdash_\bullet^H T_0 : K \\ E \vdash_\bullet^H [X : K, T] \,\mathrm{ok} \\ U \notin \mathrm{dom}\, E \\ U \notin \mathrm{dom}\, H\end{array}}{E, U(T_0) : [X : K, T] \vdash_c^H \,\mathrm{ok}} \text{ (envok.U)}$$

Note that the premise $U \notin \operatorname{dom} H$ is actually superfluous as it can be deduced from the other premises.

An alternate way of stating (envok.x) and (envok.X) could be to use a (envok. ?) rule :

$$\frac{E \vdash_c^H \tau \,\mathrm{ok} \quad \zeta \notin \operatorname{dom} E}{E, \zeta : \tau \vdash_c^H \mathrm{ok}} \text{(envok. ?)}$$

A.2.6 $\boxed{E \vdash_c^H K \,\mathrm{ok}}$ kind correctness

$$\frac{E \vdash_c^H T : \mathbf{Le}(\top)}{E \vdash_c^H \mathbf{Le}(T) \,\mathrm{ok}} \text{(Kok.Le)} \qquad \frac{E \vdash_c^H T : \mathbf{Le}(\top)}{E \vdash_c^H \mathbf{Eq}(T) \,\mathrm{ok}} \text{(Kok.Eq)}$$

A.2.7 $\boxed{E \vdash_c^H K == K'}$ kind equality

$$\frac{E \vdash_c^H T == T'}{E \vdash_c^H \mathbf{Le}(T) == \mathbf{Le}(T')} \text{(Keq.Le)} \qquad \frac{E \vdash_c^H T == T'}{E \vdash_c^H \mathbf{Eq}(T) == \mathbf{Eq}(T')} \text{(Keq.Eq)}$$

A.2.8 $\boxed{E \vdash_c^H K <: K'}$ subkinding

$$\frac{E \vdash_c^H T : \mathbf{Le}(\top)}{E \vdash_c^H \mathbf{Eq}(T) <: \mathbf{Le}(T)} \text{(Ksub.Eq)} \qquad \frac{E \vdash_c^H T <: T'}{E \vdash_c^H \mathbf{Le}(T) <: \mathbf{Le}(T')} \text{(Ksub.Le)}$$

$$\frac{E \vdash_c^H K == K'}{E \vdash_c^H K <: K'} \text{(Ksub.refl)} \qquad \frac{E \vdash_c^H K <: K' \quad E \vdash_c^H K' <: K''}{E \vdash_c^H K <: K''} \text{(Ksub.tran)}$$

Note that (Ksub.tran) is derivable.

A.2.9 $\boxed{E \vdash_c^H T : K}$ kind of a type

$$\frac{E \vdash_c^H T : K \quad E \vdash_c^H K <: K'}{E \vdash_c^H T : K'} \text{(TK.sub)} \qquad \frac{E \vdash_c^H T == T'}{E \vdash_c^H T : \mathbf{Eq}(T')} \text{(TK.Eq)}$$

$$\frac{E, X : K, E' \vdash_c^H \mathrm{ok}}{E, X : K, E' \vdash_c^H X : K} \text{(TK.var)} \qquad \frac{E \vdash_c^H T <: T'}{E \vdash_c^H T : \mathbf{Le}(T')} \text{(TK.Le)}$$

$$\frac{E \vdash_c^H U : [X : K, T]}{E \vdash_c^H U.\mathtt{type} : K} \text{(TK.mod)}$$

$$\frac{E \vdash_c^H \mathrm{ok} \qquad \vdash h \,\mathrm{ok}}{E \vdash_c^H h.\mathtt{type} : K} \text{(TK.hash)}$$

where $h = \mathbf{hash}(N, [T_0, v^\bullet : T_1] : [X : K, T_1'])$

A.2.10 $\boxed{E \vdash_c^H T == T'}$ type equivalence

$$\frac{E \vdash_c^H T : \mathbf{Eq}(T')}{E \vdash_c^H T == T'} \text{(Teq.Eq)}$$

$$\frac{E \vdash_c^H \text{ ok}}{E \vdash_c^H h.\texttt{type} == T_0} \text{(Teq.hash)}$$

where $h = \mathbf{hash}(N, [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T), T_1'])$ and $h \in c$

The (Teq.hash) and (Teq.Eq) are the more intersting rules as they introduce type equivalences. The others rules only propagate them.

$$\frac{E \vdash_c^H T : \mathbf{Le}(\top)}{E \vdash_c^H T == T} \text{(Teq.refl)} \qquad \frac{E \vdash_c^H T == T'}{E \vdash_c^H T' == T} \text{(Teq.sym)} \qquad \frac{\begin{array}{c}E \vdash_c^H T == T' \\ E \vdash_c^H T' == T''\end{array}}{E \vdash_c^H T == T''} \text{(Teq.tran)}$$

$$\frac{\begin{array}{c}E \vdash_c^H T_0 == T_0' \\ E \vdash_c^H T_1 == T_1'\end{array}}{E \vdash_c^H T_0 \to T_1 == T_0' \to T_1'} \text{(Teq.cong.fun)}$$

$$\frac{E \vdash_c^H T_i == T_i' \quad \forall i. 1 \leqslant i \leqslant j}{E \vdash_c^H T_1 * ... * T_j == T_1' * ... * T_j'} \text{(Teq.cong.tuple)}$$

$$\frac{E \vdash_c^H T_i == T_i' \quad \forall i. 1 \leqslant i \leqslant j}{E \vdash_c^H \{l_1 : T_1, ..., l_j : T_j\} == \{l_1 : T_1', ..., l_j : T_j'\}} \text{(Teq.cong.rec)}$$

Note that records are an unordered set of association between labels and values.

A.2.11 $\boxed{E \vdash_c^H T <: T'}$ subtyping

$$\frac{E \vdash_c^H T : \mathbf{Le}(T')}{E \vdash_c^H T <: T'} \text{(Tsub.Le)} \qquad \frac{E \vdash_c^H T == T'}{E \vdash_c^H T <: T'} \text{(Tsub.Equi)}$$

Transitivity follows from transitivity of subkinding.

$$\frac{E \vdash_c^H \text{ ok} \quad \kappa <: \kappa' \in H}{E \vdash_c^H \kappa.\texttt{type} <: \kappa'.\texttt{type}} \text{(Tsub.Subhash)}$$

For simplicity, we choose to ignore depth subtyping for records, though we have it for tuples.

$$\frac{E \vdash_c^H T_i : \mathbf{Le}(\top) \quad 1 \leqslant i \leqslant k}{E \vdash_c^H \{l_1 : T_1, ..., l_j : T_j, ..., l_k : T_k\} <: \{l_1 : T_1, ..., l_j : T_j\}} \text{(Tsub.cong.record.width)}$$

$$\frac{E \vdash_c^H T_0' <: T_0 \qquad E \vdash_c^H T_1 <: T_1'}{E \vdash_c^H T_0 \to T_1 <: T_0' \to T_1'} \text{(Tsub.cong.fun)}$$

$$\frac{E \vdash_c^H T_i <: T_i' \quad \forall i.1 \leqslant i \leqslant j}{E \vdash_c^H T_1 * ... * T_j <: T_1' * ... * T_j'} \text{(Tsub.cong.tuple)}$$

The following rules have to be present to ensure that T is well-formed iff T is a subtype of $\top$. In combination with (TK.Le), this allows us to prove that any well-formed T has kind $\mathbf{Le}(\top)$.

$$\frac{\vdash h \,\text{ok} \quad E \vdash_c^H \text{ok}}{E \vdash_c^H h.\texttt{type} <: \top} \text{(Tsub.hash)} \qquad \frac{E \vdash_c^H \text{ok}}{E \vdash_c^H \top <: \top} \text{(Tsub.top)}$$

$$\frac{E \vdash_c^H \text{ok}}{E \vdash_c^H \text{unit} <: \top} \text{(Tsub.unit)} \qquad \frac{E \vdash_c^H \text{ok}}{E \vdash_c^H \text{bytes} <: \top} \text{(Tsub.dyn)}$$

$$\frac{E \vdash_c^H T <: \top \quad E \vdash_c^H T' <: \top}{E \vdash_c^H T \to T' <: \top} \text{(Tsub.fun)} \qquad \frac{E \vdash_c^H T_i <: \top \quad \forall i.1 \leqslant i \leqslant j}{E \vdash_c^H T_1 * ... * T_j <: \top} \text{(Tsub.tuple)}$$

$$\frac{E \vdash_c^H T_i <: \top \quad \forall i.1 \leqslant i \leqslant j}{E \vdash_c^H \{l_1 : T_1, ..., l_j : T_j\} <: \top} \text{(Tsub.rec)}$$

A.2.12 $\boxed{E \vdash_c^H S \,\text{ok}}$ signature correctness

$$\frac{E, X : K \vdash_c^H T : \mathbf{Le}(\top)}{E \vdash_c^H [X : K, T] \,\text{ok}} \text{(Sok)}$$

A.2.13 $\boxed{E \vdash_c^H S == S'}$ signature equivalence

$$\frac{E \vdash_c^H K == K' \quad E, X : K \vdash_c^H T == T'}{E \vdash_c^H [X : K, T] == [X : K', T']} \text{(Seq.struct)}$$

Signature equivalence is an equivalence relation since kind equivalence and type equivalence are equivalence relations.

A.2.14 $\boxed{E \vdash_c^H S <: S'}$ subsignaturing

Note that we never use subsignaturing in typing judgements.

$$\frac{\begin{array}{c}E \vdash_c^H K <: K' \\ E, X : K \vdash_c^H T == T'\end{array}}{E \vdash_c^H [X : K, T] <: [X : K', T']} \text{(Ssub.struct)}$$

$$\frac{E \vdash_c^H S \,\text{ok}}{E \vdash_c^H S <: S} \text{(Ssub.refl)} \qquad \frac{\begin{array}{c}E \vdash_c^H S <: S' \\ E \vdash_c^H S' <: S''\end{array}}{E \vdash_c^H S <: S''} \text{(Ssub.tran)}$$

(Ssub.refl) and (Ssub.tran) are derivable.

A.2.15 $\boxed{E \vdash_c^H e : T}$ type of an expression

$$\frac{\begin{array}{c}E \vdash_c^H e : T \\ E \vdash_c^H T <: T'\end{array}}{E \vdash_c^H {}_{(T<:T')}e : T'} \text{(eT.sub)} \qquad \frac{\begin{array}{c}E \vdash_c^H e : T \\ E \vdash_c^H T == T'\end{array}}{E \vdash_c^H e : T'} \text{(eT.eq)}$$

$$\frac{E, x : T, E' \vdash_c^H \text{ok}}{E, x : T, E' \vdash_c^H x : T} \text{(eT.var)} \qquad \frac{\begin{array}{c}E \vdash_c^H U : [X : K, T] \\ E \vdash_c^H T : \mathbf{Le}(\top)\end{array}}{E \vdash_c^H U.\texttt{term} : T} \text{(eT.mod)}$$

In (eT.mod), the condition $E \vdash_c T : \mathbf{Le}(\top)$ guarantees that $X \notin \text{fv}\, T$.

$$\frac{\begin{array}{c}E \vdash_c^H e' : T \to T' \\ E \vdash_c^H e : T\end{array}}{E \vdash_c^H e'\, e : T'} \text{(eT.ap)} \qquad \frac{E, x : T \vdash_c^H e : T'}{E \vdash_c^H \lambda x : T.e : T \to T'} \text{(eT.fun)}$$

$$\frac{E \vdash_c^H e_i : T_i \quad \forall i. 1 \leqslant i \leqslant j}{E \vdash_c^H (e_1, .., e_j) : T_1 * ... * T_j} \text{(eT.tuple)} \qquad \frac{E \vdash_c^H e : T_1 * ... * T_j \quad 1 \leqslant i \leqslant j}{E \vdash_c^H \mathbf{proj}_i\, e : T_i} \text{(eT.proj)}$$

$$\frac{E \vdash_c^H e_i : T_i \quad \forall i. 1 \leqslant i \leqslant j}{E \vdash_c^H \{l_1 = e_1, .., l_j = e_j\} : \{l_1 : T_1, ..., l_j : T_j\}} \text{(eT.record)}$$

$$\frac{E \vdash_c^H e : \{l_1 : T_1, ..., l_j : T_j\} \quad 1 \leqslant i \leqslant j}{E \vdash_c^H e.l_i : T_i} \text{(eT.field)}$$

$$\frac{E \vdash_c^H \text{ok}}{E \vdash_c^H () : \text{unit}} \text{(eT.unit)} \qquad \frac{E \vdash_c^H \text{ok}}{E \vdash_c^H ? : \text{bytes}} \text{(eT.recv)}$$

$$\frac{E \vdash_c^H e : \text{bytes}}{E \vdash_c^H \,!\, e : \text{unit}}\ (\text{eT.send}) \qquad \frac{E \vdash_c^H e : T}{E \vdash_c^H \mathbf{mar}\,(e : T) : \text{bytes}}\ (\text{eT.mar})$$

$$\frac{\begin{array}{l} E \vdash_c^{H_0} \text{ok} \\ \mathbf{nil} \vdash_{\bullet}^{H_1} T : \mathbf{Le}(\top) \\ \mathbf{nil} \vdash_{c'}^{H_1} e : T \end{array}}{E \vdash_c^{H_0} \mathbf{marshalled}_{\,c',H_1}(e : T) : \text{bytes}}\ (\text{eT.marred})$$

The second premise of (eT.marred) is actually redundant because of Lemma B.53 (things have to be ok) and Lemma B.38 (colour and subhash change preserves type okedness).

$$\frac{\begin{array}{l} E \vdash_c^H T : \mathbf{Le}(\top) \\ E \vdash_c^H e : \text{bytes} \end{array}}{E \vdash_c^H (\mathbf{unmar}\, e : T) : T}\ (\text{eT.unmar}) \qquad \frac{E \vdash_c^H T : \mathbf{Le}(\top)}{E \vdash_c^H \mathbf{Unmarfailure}^T : T}\ (\text{eT.Undynfailure})$$

For (eT.unmar), the condition $E \vdash_c^H T : \mathbf{Le}(\top)$ ensures T is well-formed. Of course, nothing forces T to be closed; as usual, reduction will transform it into something closed before (ered.unmar) happens.

$$\frac{E \vdash_c^H T : \mathbf{Le}(\top) \quad E \vdash_{c \cup c'}^H e : T}{E \vdash_c^H [e]_{c'}^T : T}\ (\text{eT.col})$$

Note that the colors are additive here and that, by Lemma B.39 (colour change preserves type okedness), the first premise can be omitted.

A.2.16 $\boxed{E \vdash_c^H M : S}$ signature of a module expression

$$\frac{\begin{array}{l} E \vdash_c^H T : K \\ E, X : K \vdash_c^H T_1' : \mathbf{Le}(\top) \\ E, X : \mathbf{Eq}(T) \vdash_c^H T_1 <: T_1' \\ E \vdash_c^H v^c : T_1 \end{array}}{E \vdash_c^H [T, v^c : T_1] : [X : K, T_1']}\ (\text{MS.struct})$$

A.2.17 $\boxed{E \vdash_c^H U : S}$ signature of a module variable

$$\frac{E, U(T) : S, E' \vdash_c^H \text{ok}}{E, U(T) : S, E' \vdash_c^H U : S}\ (\text{US.var}) \qquad \frac{\begin{array}{l} E \vdash_c^H U : S \\ E \vdash_c^H S == S' \end{array}}{E \vdash_c^H U : S'}\ (\text{US.eq})$$

$$\frac{E \vdash_c^H U : [X : K, T]}{E \vdash_c^H U : [X : \mathbf{Eq}(U.\mathtt{type}), T]}\ (\text{US.self})$$

A.2.18 $\boxed{E \vdash^H_\bullet m : T}$ type of a machine

$$\frac{E \vdash^H_\bullet e : T}{E \vdash^H_\bullet e : T}\ (\mathsf{mT.expr})$$

In (mT.expr), the premise is a "type of an expression" judgement, while the conclusion is a "type of a machine" judgement.

$$\frac{\begin{array}{l} E \vdash^H_\bullet T : \mathbf{Le}(\top) \\ E \vdash^H_\bullet [T_0, v^\bullet : T_1] : S \\ E, U(T_0) : S \vdash^{H \cup \kappa_1 <: U \cup \ldots \cup \kappa_i <: U \cup U <: \kappa'_1 \cup \ldots \cup U <: \kappa'_j}_\bullet m : T \end{array}}{E \vdash^H_\bullet (\text{ module } NU \text{ extends } \kappa_1, ..., \kappa_i \text{ restricts } \kappa'_1, ..., \kappa'_j = [T_0, v^\bullet : T_1] : S \text{ in } m) : T}\ (\mathsf{mT.letext})$$

Note : in (mT.letext), the first premise is saying that U is not free in T.

A.2.19 $\boxed{\vdash n\,\mathrm{ok}}$ network correctness

$$\frac{\vdash n_1\,\mathrm{ok} \quad \vdash n_2\,\mathrm{ok}}{\vdash n_1 \mid n_2\,\mathrm{ok}}\ (\mathsf{nok.par}) \qquad \frac{}{\vdash \mathbf{0}\,\mathrm{ok}}\ (\mathsf{nok.zero}) \qquad \frac{\mathbf{nil} \vdash^H_\bullet e : \mathrm{unit}}{\vdash H, e\,\mathrm{ok}}\ (\mathsf{nok.expr})$$

A.3 Semantics

A.3.1 $\boxed{H, m \to_m H', m'}$ compile-time reduction

$$H,\ \text{module } N_U = [T_0, v^\bullet : T_1] : [X : \mathbf{Eq}(T'_0), T'_1] \text{ in } m \\ \to_m H, \{U.\mathtt{type} \leftarrow T'_0, U.\mathtt{term} \leftarrow_{(T_1 <: \{X \leftarrow T'_0\} T'_1)} v^\bullet\} m \qquad (\mathsf{mred.Eq})$$

$$H,\ \text{module } N_U \text{ extends } (h_1, ..., h_i) \text{ restricts } (h'_1, ..., h'_j) = [T, v^\bullet : T_1] : [X : \mathbf{Le}(T'), T'_1] \text{ in } m \\ \to_m H \cup h_1 <: h \cup \ldots \cup h_i <: h \cup h <: h'_1 \cup \ldots \cup h <: h'_j, \\ \{U \leftarrow h, U.\mathtt{type} \leftarrow h.\mathtt{type}, U.\mathtt{term} \leftarrow [_{(T_1 <: \{X \leftarrow h.\mathtt{type}\} T'_1)} v^\bullet]^{\{X \leftarrow h.\mathtt{type}\} T'_1}_{\{h\}}\} m \qquad (\mathsf{mred.Le})$$

$$\text{where} \quad h = \mathbf{hash}(N, [T, v^\bullet : T_1] : [X : \mathbf{Le}(T'), T'_1])$$

A.3.2 $\boxed{H, e \to_c H', e'}$ expression reduction

Expression

$$H, \mathbf{proj}_i\,(v^c_1, ..., v^c_j) \to_c H, v^c_i \qquad \text{if } 1 \leqslant i \leqslant j \qquad (\mathsf{ered.proj})$$

$$H, \{l_1 = v^c_1, ..., l_j = v^c_j\}.l_i \to_c H, v^c_i \qquad \text{if } 1 \leqslant i \leqslant j \qquad (\mathsf{ered.field})$$

$$H, (\lambda x : T.e)\, v^c \to_c H, \{x \leftarrow v^c\} e \qquad (\mathsf{ered.ap})$$

Marshalling

$$H, \mathbf{mar}\,(v^c : T) \to_c H, \mathbf{marshalled}_{\,c,H}(v^c : T) \qquad \text{(ered.mar)}$$

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{\,c',H'}(e : T)) : T' \to_c \begin{cases} H \cup H',\ {}_{(T<:T')}[e]^T_{c'} & \text{if } \mathbf{nil} \vdash^{H\cup H'}_{\bullet} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases} \qquad \text{(ered.unmar)}$$

The unmarshalling test happens in the empty colour so that the erasure of colours does not change the outcome.

Subtyping

$$H,\ {}_{(T_0<:T_1)}\widehat{v}^c \to_c H,\ {}_{(TV^c_2<:T_1)}\widehat{\widehat{v}}^c \quad \text{where } \widehat{v}^c = {}_{(TV^c_2<:\,TV^c_3)}\widehat{\widehat{v}}^c \qquad \text{(ered.sub.sub)}$$

$$H,\ {}_{(T_0<:h_0.\mathtt{type})}\widehat{\widehat{v}}^c \to_c H,\ {}_{(T_0<:T_1)}\widehat{\widehat{v}}^c \quad \text{when } h_0 \in c \text{ and } \mathrm{impl}\,(h_0) = T_1 \qquad \text{(ered.sub.typeright)}$$

$$H,\ {}_{(h_0.\mathtt{type}<:\,TV^c)}\widehat{\widehat{v}}^c \to_c H,\ {}_{(T_0<:\,TV^c)}\widehat{\widehat{v}}^c \quad \text{when } h_0 \in c \text{ and } \mathrm{impl}\,(h_0) = T_0 \qquad \text{(ered.sub.typeleft)}$$

$$H,\ {}_{((T_1*\ldots*T_j)<:(T'_1*\ldots*T'_j))}(v^c_1, ..., v^c_j) \to_c H, (\,{}_{(T_1<:T'_1)}v^c_1, ...,\ {}_{(T_j<:T'_j)}v^c_j) \qquad \text{(ered.sub.tuple)}$$

$$H,\ {}_{(\{l_1:T'_1;\ldots;l_i:T'_i\}<:\{l_1:T_1;\ldots;l_j:T_j\})}\{l_1 = v^c_1; ...; l_j = v^c_j\} \to_c H, \{l_1 = v^c_1; ...; l_i = v^c_i\} \qquad \text{(ered.sub.record)}$$

We suppose here that $i \leqslant j$. Note that we don't have depth subtyping so that we do not have to carry the annotations on the right-hand side.

$$H,\ {}_{((T_1\to T_2)<:(T'_1\to T'_2))}(\lambda x : T_0.e) \to_c H, (\lambda x : T'_1.\ {}_{(T_2<:T'_2)}\{x \leftarrow {}_{(T'_1<:T_1)}x\}e) \qquad \text{(ered.sub.fun)}$$

$$H,\ {}_{(\mathrm{bytes}<:\mathrm{bytes})}\mathbf{marshalled}_{\,c',H'}(e : T) \to_c H, \mathbf{marshalled}_{\,c',H'}(e : T) \qquad \text{(ered.sub.marshalled)}$$

$$H,\ {}_{(\mathrm{unit}<:\mathrm{unit})}() \to_c H, () \qquad \text{(ered.sub.unit)}$$

$$H,\ {}_{(T'<:T'')}([\widehat{v}^{c'\cup c}]^{h.\mathtt{type}}_{c'}) \to_c H, [\,{}_{(T'<:T'')}\widehat{v}^{c'\cup c}]^{T''}_{c'} \quad \text{where } h \notin c \qquad \text{(ered.sub.col)}$$

Bracket pushing

$$H, [[\widehat{v}^{c \cup c_0 \cup c_1}]_{c_0}^{h_0.\texttt{type}}]_{c_1}^{h_1.\texttt{type}} \rightarrow_c H, [\widehat{v}^{c \cup c_0 \cup c_1}]_{c_0 \cup c_1}^{h_1.\texttt{type}} \quad \text{if } h_0 \notin c_1 \cup c \text{ and } h_1 \notin c \quad \text{(ered.col.col)}$$

$$H, [\widehat{v}^{c' \cup c}]_{c'}^{h.\texttt{type}} \rightarrow_c H, [\widehat{v}^{c' \cup c}]_{c'}^{T} \qquad \text{when } h \in c \text{ and } \text{impl}\,(h) = T \qquad \text{(ered.col.type)}$$

$$H, [(v_1^{c' \cup c}, ..., v_j^{c' \cup c})]_{c'}^{T_1 * ... * T_j} \rightarrow_c H, ([v_1^{c' \cup c}]_{c'}^{T_1}, ..., [v_j^{c' \cup c}]_{c'}^{T_j}) \qquad \text{(ered.col.tuple)}$$

$$H, [\{l_1 = v_1^{c' \cup c}, .., l_j = v_j^{c' \cup c}\}]_{c'}^{\{l_1:T_1, ..., l_j:T_j\}} \rightarrow_c H, \{l_1 = [v_1^{c' \cup c}]_{c'}^{T_1}, .., l_j = [v_j^{c' \cup c}]_{c'}^{T_j}\} \qquad \text{(ered.col.record)}$$

$$H, [\lambda x : T.e]_{c'}^{T' \rightarrow T''} \rightarrow_c H, (\lambda x : T'.[e]_{c'}^{T''}) \qquad \text{(ered.col.fun)}$$

$$H, [\textbf{marshalled}_{\,c_0, H'}(e : T)]_{c'}^{\text{bytes}} \rightarrow_c H, \textbf{marshalled}_{\,c_0, H'}(e : T) \qquad \text{(ered.col.marred)}$$

$$H, [()]_{c'}^{\text{unit}} \rightarrow_c H, () \qquad \text{(ered.col.unit)}$$

Congruence

$$\frac{H, e \rightarrow_c H', e'}{H, C_c^{c'}.e \rightarrow_{c'} H', C_c^{c'}.e'} \text{ (ered.cong)}$$

A.3.3 $\boxed{n \equiv n'}$ network structural congruence

$\mathbf{0} \mid n \equiv n$ (nsc.id) $\quad n_1 \mid n_2 \equiv n_2 \mid n_1$ (nsc.commut) $\quad n_1 \mid (n_2 \mid n_3) \equiv (n_1 \mid n_2) \mid n_3$ (nsc.assoc)

Plus reflexivity, symmetry and transitivity of $\equiv$.

A.3.4 $\boxed{n \rightarrow_n n'}$ network reduction

$$\frac{H, e \rightarrow_\bullet H', e'}{H, e \rightarrow_n H', e'} \text{ (nred.expr)} \qquad \frac{n \rightarrow_n n'}{n \mid n'' \rightarrow_n n' \mid n''} \text{ (nred.par)}$$

$$\frac{n \equiv n_0 \rightarrow_n n_0' \equiv n'}{n \rightarrow_n n'} \text{ (nred.strcong)}$$

$$H, CC_c^\bullet.!\, v^c \mid H', CC_{c'}^\bullet.? \rightarrow_n H, CC_c^\bullet.() \mid H', CC_{c'}^\bullet.v^c \quad \text{(nred.comm)}$$

Annexe B

HATS : theorems and proofs

B.1 Proofs and derivations

We use the words "proof" and "derivation" indifferently, to mean a natural deduction-style tree of inference steps leading to a judgement.

Definition B.1 (smaller proof) A proof Π is smaller than a proof Π' iff Π contains at most as many inference steps as Π', i.e. the number of nodes in the tree Π is smaller.

A subproof is a particular case of a smaller proof.

Note that any proof is a subproof of itself and is smaller than itself. We will use the wordings "proper subproof" and "strictly smaller proof" to exclude equality (respectively, equal size).

B.2 Correctness

Definition B.2 (domain of an environment) The domain of an environment E, written $\operatorname{dom} E$, is a set of variables defined by induction as follows :

- $\operatorname{dom} \mathbf{nil} = \varnothing$
- $\operatorname{dom} (E, \zeta : \tau) = \operatorname{dom} E \cup \zeta$

Lemma B.3 (non-membership in domain is interpreted trivially) The judgement $\zeta \notin \operatorname{dom} E$ is provable iff ζ is not a member of the domain of E.

Proof. Induction on the derivation of $\zeta \notin \operatorname{dom} E$. The rules (clash.nil) and (clash.cons) trivially maintain this property. □

Definition B.4 (domain of a subhash relation) The domain of a subhash relation H, written $\operatorname{dom} H$, is a set of variables defined by induction as follows :

- $\operatorname{dom} \varnothing = \varnothing$
- $\operatorname{dom} (H \cup U <: U') = \operatorname{dom} H \cup U \cup U'$
- $\operatorname{dom} (H \cup U <: h) = \operatorname{dom} H \cup U$
- $\operatorname{dom} (H \cup h <: U) = \operatorname{dom} H \cup U$
- $\operatorname{dom} (H \cup h <: h') = \operatorname{dom} H$

Lemma B.5 (non-membership in subhash domain is interpreted trivially) The judgement $U \notin \operatorname{dom} H$ is provable iff U is not a member of the domain of H.

Proof. Induction on the derivation of $U \notin \operatorname{dom} H$. The rules (clash.hash.nil) and (clash.hash.cons) trivially maintain this property. □

We will freely make use of these lemma in the remainder of this section.

Lemma B.6 (colours have to be ok) If $E \vdash_c^H J$ then $\vdash c\,\mathsf{ok}$ by a proper subproof.

Proof. Induction on the derivation of $E \vdash_c^H J$. All rules whose conclusion is a coloured judgement have at least one premise that is a similarly coloured judgement, so induction applies. This leaves only the rule (envok.nil), which has $\vdash c\,\mathsf{ok}$ as a premise. □

Lemma B.7 (hashes have to be ok) If $E \vdash_c^H J$ or $\vdash c\,\mathsf{ok}$ or $\vdash h'\,\mathsf{ok}$ is derivable by a proof Π and h is a subterm of $E \vdash_c^H J$ or $\vdash c\,\mathsf{ok}$ or $\vdash h'\,\mathsf{ok}$ then $\vdash h\,\mathsf{ok}$ by a subproof of Π.

Proof. Induction on the structure of Π. Most metavariables in the conclusion of rules whose conclusion is a coloured judgement also appear in at least one premise that is a coloured judgement with the same color. If h is in the instantiation of such a metavariable then we have the desired result by induction. We list the remaining cases (including "exposed" hashes).

Case (hok.hash) : The conclusion is $\vdash \mathbf{hash}(N, M : S)\,\mathsf{ok}$. If $h = \mathbf{hash}(N, M : S)$ we have the desired result. Otherwise induction gives the desired result.

Case (hmok.zero), (hmok.sing), (hmok.union) : Trivial or trivial by induction.

Case subterm of the hash comparison in (envok.hashhash) : Two of the premises are the desired result : $\vdash \mathbf{hash}(N_0, [T_0, v_0^c : T_0'''] : S_0)\,\mathsf{ok}$ and $\vdash \mathbf{hash}(N_1, [T_1, v_1^c : T_1'''] : S_1)\,\mathsf{ok}$.

Case subterm of $\mathbf{hash}(N_0, [T_0, v_0^c : T_0'''] : S_0) <: U$ in (envok.hashU) : One of the premises is $\vdash \mathbf{hash}(N_0, [T_0, v_0^c : T_0'''] : S_0)\,\mathsf{ok}$, giving the desired result. Otherwise, we use the induction hypothesis.

Case subterm of $U <: \mathbf{hash}(N_1, [T_1, v_1^c : T_1'''] : S_1)$ in (envok.Uhash) : One of the premises is $\vdash \mathbf{hash}(N_1, [T_1, v_1^c : T_1'''] : S_1)\,\mathsf{ok}$, giving the desired result. Otherwise, we use the induction hypothesis.

Case subterm of $\mathbf{hash}(N, M : S)$ in (TK.hash) : One premise is $\vdash \mathbf{hash}(N, M : S)\,\mathsf{ok}$. If $h = \mathbf{hash}(N, M : S)$ we have the desired result. Otherwise induction gives the desired result.

Case subterm of $E \vdash_c^H h.\mathtt{type} == T$ in (Teq.hash) : Since h in c, induction gives the desired result.

Case h in $E \vdash_c^H h.\mathtt{type} <: \top$ in (Tsub.hash) : One of the premises is $\vdash h\,\mathsf{ok}$, giving the desired result.

Case N_U in (mT.letext) : Trivial ($\mathrm{fv}\, N = \varnothing$).

□

Lemma B.8 (environments and subhashes have to be ok) If $E \vdash_c^H J$ then $E \vdash_c^H \mathsf{ok}$ by a subproof.

Proof. Simultaneously with the following lemma. □

Lemma B.9 (prefixes of ok environments are ok) If $E, \zeta : \tau \vdash_c^H \mathsf{ok}$ then there exists a subset H' of H such that $E \vdash_c^{H'} \mathsf{ok}$ by a subproof. In particular, if $\zeta \notin \mathrm{dom}\, H$, we can take $H' = H$.

Proof. Induction on the derivation of $E, \zeta : \tau \vdash_c^H$ ok. Note that if $\zeta \notin \operatorname{dom} H$, $E, \zeta : \tau \vdash_c^H$ ok can only be derived from a judgement of the form $E \vdash_c^H J$, hence the result of the second lemma follows by induction from the first lemma. If $\zeta \in \operatorname{dom} H$, $E, \zeta : \tau \vdash_c^H$ ok can only be derived by one of the (envok.hashU), (envok.Uhash), (envok.mod) or (envok.modopp) rule. Each of these rules have one of their premises of the form $E, \zeta : \tau \vdash_c^{H'}$ ok with $H' \subseteq H$. We conclude by induction.

We now turn to the first lemma. Most rules whose conclusion is a coloured judgement have at least one premise that is a coloured judgement with the same colour, same environment and same H, and so we apply the induction hypothesis to that premise. We list the remaining cases.

Case (envok.*) : Trivial since the desired result is exactly the hypothesis.

Case (Seq.struct) : There exist X, K, T such that $J = [X : K, T]$ ok. The premise is $E, X : K \vdash_c^H T : \mathbf{Le}(\top)$. By induction, we get $E, X : K \vdash_c^H$ ok by a subproof. We conclude by the second lemma with $E \vdash_c^H$ ok since X is a type variable and can't bind in H.

Case (eT.fun) : There exist x, T, T', e such that $J = \lambda x : T.e : T \to T'$. The premise is $E, x : T \vdash_c^H e : T'$. Since x binds an expression variable, by induction, $E, x : T \vdash_c^H$ ok by a subproof, hence $E \vdash_c^H$ ok by the second lemma, as desired.

□

Lemma B.10 (ok environments have no repetition in the domain)
If $E, E' \vdash_c^H$ ok then $\operatorname{dom} E \cap \operatorname{dom} E' = \varnothing$.

Proof. Induction on the length of E' and of H. If $E' = \mathbf{nil}$, then $\operatorname{dom} E' = \varnothing$, so $\operatorname{dom} E \cap \operatorname{dom} E' = \varnothing$. Otherwise write $E' = (E'', \zeta : \tau)$. Then there are two cases :

Case $\zeta \notin \operatorname{dom} H$: $E, E' \vdash_c^H$ ok must have been derived by the appropriate (envok.*) rule, with the premises $E, E'' \vdash_c^H \tau$ ok and $\zeta \notin \operatorname{dom}(E, E'')$. From the first premise, by Lemma B.8 (environments and subhashes have to be ok), we get $E, E'' \vdash_c^H$ ok by a subproof, whence $\operatorname{dom} E \cap \operatorname{dom} E'' = \varnothing$ by induction. Then $\operatorname{dom} E \cap \operatorname{dom} E' = (\operatorname{dom} E \cap \operatorname{dom} E'') \cup (\operatorname{dom} E \cap \zeta) = \varnothing \cup \varnothing = \varnothing$ as desired.

Case $\zeta \in \operatorname{dom} H$: $E, E' \vdash_c^H$ ok must have been derived from some judgements of the form $E, E' \vdash_c^{H'}$ ok with H' strictly smaller than H. By induction we have $\operatorname{dom} E \cap \operatorname{dom} E' = \varnothing$.

□

Lemma B.11 (free variables of a judgement come from the environment)
If $E \vdash_c^H J$ then $\operatorname{fv}(J) \cup \operatorname{fv}(H) \subseteq \operatorname{dom}(E)$. For completeness's sake : if $\vdash c$ ok then $\operatorname{fv}(c) \subseteq \varnothing$; if $\vdash h$ ok then $\operatorname{fv}(h) \subseteq \varnothing$; if $\vdash n$ ok then $\operatorname{fv}(n) \subseteq \varnothing$.

Proof. We freely use Lemma B.3 (non-membership in domain is interpreted trivially) and Lemma B.5 (non-membership in subhash domain is interpreted trivially).

Induction on the size of the derivation Π of the judgement.

Most rules whose conclusion is a coloured judgement have the following property : every metavariable (including H) in the right-hand side of the conclusion $E \vdash_c^H J$ is present in the right-hand side, not under a binder, of a premise that is a coloured judgement with the same environment as the conclusion. Then, by induction on the premise, every free variable in the subterm matched by that metavariable is present in the domain of the environment.

Most rules whose conclusion is the correctness of a network have the following property : every metavariable in the conclusion is also present in one of the premises which is a network correctness judgement. Then, by induction, there is no free variable in the subterm matched by that metavariable.

We list the remaining cases.

Case (hok.hash) : The conclusion is $\vdash \mathbf{hash}(N, M : S)$ ok and the premise is $\mathbf{nil} \vdash^H_\bullet M : S$. By induction we have $\mathrm{fv}\,(M : S) \cup \mathrm{fv}\,(H) = \mathrm{fv}\,(\mathbf{hash}(N, M : S)) = \mathrm{dom}\,\mathbf{nil} = \varnothing$ as desired.

Case (hmok.zero) : Trivial.

Case (hmok.sing) : By induction, we have $\mathrm{fv}\,(h) = \varnothing$, so $\mathrm{fv}\,(h) = \varnothing$ as desired.

Case H in (envok.x), (envok.X), (envok.U) : One of the premises gives us by induction that $\mathrm{fv}\,(H) \subseteq \mathrm{dom}\,E$. Since the environment of the conclusion contains E, we can deduce the desired property.

Case h in (TK.hash) : The conclusion is $E \vdash^H_c \mathbf{hash}(N, M : [X : K, T']) : K$. One of the premises is $\vdash \mathbf{hash}(N, M : [X : K, T'])$ ok. By induction $\mathrm{fv}\,(\mathbf{hash}(N, M : [X : K, T'])) = \varnothing$, and in particular $\mathrm{fv}\,(K) = \varnothing \subseteq \mathrm{dom}\,E$ as desired.

Case (Teq.hash) : The conclusion is $E \vdash^H_c h.\mathtt{type} == T_0$ where $h = \mathbf{hash}(N, [T_0, v^c : T_1] : S)$ and $h \in c$. The premise is $E \vdash^H_c$ ok. By Lemma B.6 (colours have to be ok), $\vdash c$ ok by a proper subproof. By induction, we have $\mathrm{fv}\,c = \varnothing$. As $h \in c$, we have $\mathrm{fv}\,h = \varnothing$ and in particular $\mathrm{fv}\,T_0 = \varnothing$, so $\mathrm{fv}\,(h.\mathtt{type} == T_0) = \varnothing \subseteq \mathrm{dom}\,E$.

Case (Tsub.Subhash) : The conclusion is $E \vdash^H_c \kappa.\mathtt{type} <: \kappa'.\mathtt{type}$ with $E \vdash^H_c$ ok as a premise and $\kappa <: \kappa' \in H$. By induction hypothesis, we know that $\mathrm{fv}\,H \subseteq \mathrm{dom}\,E$. As $(\kappa <: \kappa') \in H$, we have $\mathrm{fv}\,(\kappa.\mathtt{type} <: \kappa'.\mathtt{type}) \subseteq \mathrm{dom}\,E$.

Case h in (Tsub.hash) : One of the premises is $\vdash h$ ok. By induction we then know that $\mathrm{fv}\,(h) = \varnothing$.

Cases T in (Sok); T and T' (Seq.struct) and (Ssub.struct) : These rules have a metavariable $\aleph$ in the conclusion that is under a binder for some variable ζ. In each case, there is a premise of the form $E, \zeta : \tau \vdash^H_c J'$ with $\aleph$ appearing not under a binder in J'. By induction, we get that $\mathrm{fv}\,\aleph \subseteq \mathrm{dom}\,E \cup \zeta$. Since ζ is bound in the occurence of $\aleph$ in the conclusion, this is the desired result.

Cases T' and T'' in (MS.struct); e in (eT.fun); m in (mT.letext) : Same case as (Sok).

Cases K in (TK.var) and (Sok); T in (eT.fun) : In each case, there is a premise of the form $E, \zeta : \aleph \vdash^H_c J'$ where E and c are the environment and the colour of the conclusion and $\aleph$ is the metavariable under consideration. By Lemma B.8 (environments and subhashes have to be ok) and reversing the appropriate (envok.*) rule, we have, by a proper subproof, respectively, $E \vdash^H_c K$ ok, $E \vdash^H_c T : \mathbf{Le}(\top)$, $E \vdash^H_c M : S$. In each case, by induction, we get $\mathrm{fv}\,\aleph \subseteq \mathrm{dom}\,E$.

Case T' in (eT.fun) : By induction as in the case of e, we get that $\mathrm{fv}\,T' \subseteq \mathrm{dom}\,E \cup x$. By Lemma B.7 (hashes have to be ok), for any hash h that is a subterm of T', we have $\vdash h$ ok by a (proper) subproof of Π. Thus, by induction, $\mathrm{fv}\,h = \varnothing$ and in particular $x \notin \mathrm{fv}\,h$. Given the syntax of types, the only place where T' might have a free expression variable is inside a hash, so $x \notin \mathrm{fv}\,T'$. Hence $\mathrm{fv}\,T' \subseteq \mathrm{dom}\,E$ as desired.

Cases (TK.var), (eT.var), (US.var) : The variable (X, x or U respectively) that the similarly written metavariable instantiates to is obviously present in the environment.

Case (eT.col) : The conclusion is $E \vdash_c [e]^T_{c'} : T$. All that remains to be shown is that $\mathrm{fv}\, c' \subseteq \mathrm{dom}\, E$. One premise of the rule is $E \vdash^H_{c'} e : T$. By Lemma B.6 (colours have to be ok), we have $\vdash c'\,\mathrm{ok}$ by a proper subproof, so by induction $\mathrm{fv}\, c' = \varnothing$ whence the desired result.

Case (eT.marred) The conclusion is $E \vdash^{H_0}_c \mathbf{marshalled}_{\,c',H_1}(e : T) : \mathrm{bytes}$. The type T, the expression e and the subhash relations H_1 from the conclusion are present in one of the premises that has an empty environment. By induction we can then state that they do not have any free variable. H_0 is present in one premise that has E as environment. By induction we know that $\mathrm{fv}\, H_0 \subseteq \mathrm{dom}\, E$.

Case N (in (mT.letext)) : Trivial as $\mathrm{fv}\, N = \varnothing$.

Case (nok.expr) : The conclusion is $\vdash H, e\,\mathrm{ok}$ and the premise is $\mathbf{nil} \vdash^H_\bullet e : \mathrm{unit}$. By induction we have $\mathrm{fv}\, e \subseteq \mathrm{dom}\, \mathbf{nil} = \varnothing$ as desired.

□

Lemma B.12 (ok environments are ok in every colour) If $E \vdash^H_c \mathrm{ok}$ and $\vdash c'\,\mathrm{ok}$ then $E \vdash^H_{c'} \mathrm{ok}$.

In particular, $E \vdash^H_c \mathrm{ok}$ implies $E \vdash^H_\bullet \mathrm{ok}$

Proof. By induction on the derivation of $E \vdash^H_c \mathrm{ok}$.

Most of the (envok.*) rules have only one premise with colour c. This premise is always a environment correctness judgement, so that we can apply the induction hypothesis to get the same correctness judgement in the c' colour. The other premises do not mention c. Then we can apply the same rule to get $E \vdash^H_{c'} \mathrm{ok}$.

The only remaining rule is (envok.nil) which we can reuse with c' to get the desired $\mathbf{nil} \vdash^{\mathbf{nil}}_{c'} \mathrm{ok}$. □

Definition B.13 (correctness judgement) A correctness judgement is a coloured judgement whose right-hand side is of one of the following forms :

$$\mathrm{ok},\ K\,\mathrm{ok},\ T : \mathbf{Le}(\top),\ S\,\mathrm{ok}. \tag{B.1}$$

Note that a derivation of a correctness judgement may involve other sorts of judgements. For example, in order to derive $U(T) : S \vdash^H_c U.\mathtt{type} : \mathbf{Le}(\top)$, one has to use (TK.mod), with a premise of the form $U(T) : S \vdash^H_c U : S'$.

Definition B.14 (type world judgement) A type world judgement is a coloured judgement whose right-hand side is of one of the following forms :

$$\mathrm{ok},\ K\,\mathrm{ok},\ K <: K',\ K == K',\ T <: T',\ T == T',\ T : K,\ S\,\mathrm{ok},\ S == S',\ S <: S',\ U : S. \tag{B.2}$$

Note that any derivation of a type world judgement contains only type world judgements and non-clash judgement, except in the proof of correctness of hashes.

B.3 Colours, substitutions and variables

Definition B.15 (hashes in something) The hashes in a syntactic entity are the subterms that are $\mathbf{hash}(N, M : S)$ and that are not themselves subterms of a hash.

Definition B.16 (substitution) Some potentially interesting cases :

$$\begin{aligned}
\sigma(\mathbf{mar}\,(e_0 : T_0)) &= \mathbf{mar}\,(\sigma e_0 : \sigma T_0)\\
\sigma(\mathbf{marshalled}_{\,c,H}(e_0 : T_0)) &= \mathbf{marshalled}_{\,c,H}(\sigma e_0 : \sigma T_0)\\
\sigma(\mathbf{unmar}\,e_0 : T_0) &= \mathbf{unmar}\,\sigma e_0 : \sigma T_0\\
\sigma([e_0]^T_{c_0}) &= [\sigma e_0]^{\sigma T}_{c_0}\\
\{U.\mathtt{type}{\leftarrow}T\}U.\mathtt{type} &= T\\
\{U.\mathtt{term}{\leftarrow}e\}U.\mathtt{term} &= e\\
\sigma(\kappa <: \kappa') &= \sigma\kappa <: \sigma\kappa'\\
\{U{\leftarrow}h\}(U <: h') &= h <: h'
\end{aligned}$$

Note that substitution performs all necessary alpha-conversions. We generally leave alpha-conversion implicit.

Lemma B.17 (stability of values by substitution) Let σ be any substitution, c be a colour and v^c be any c-value with correct hashes. Then σv^c is an c-value.

Proof. Induction on the structure of v^c. As per the syntax of values, the only places where a value may contain free variables are inside hashes or under a λ or a **marshalled**. Since the hashes in v^c are assumed to be correct, they are closed by Lemma B.11 (free variables of a judgement come from the environment). As for λ's and **marshalled**, they allow an arbitrary expression, hence they are stable by substitution. □

Lemma B.18 (connection between fv and fse) Let $\aleph$ be anything in the syntax. Then $\mathrm{fv}\,\aleph \subseteq \mathrm{fse}\,\aleph$. If $x \in \mathrm{fse}\,\aleph$ then $x \in \mathrm{fv}\,\aleph$. If $X \in \mathrm{fse}\,\aleph$ then $X \in \mathrm{fv}\,\aleph$. If $U \in \mathrm{fse}\,\aleph$ or $U.\mathtt{term} \in \mathrm{fse}\,\aleph$ or $U.\mathtt{type} \in \mathrm{fse}\,\aleph$ then $U \in \mathrm{fv}\,\aleph$.

We may use this lemma implicitly.

Proof. Trivial from the definition of fv and fse. □

Lemma B.19 (types do not contain free expression variables)
If $E \vdash^H_c T : \mathbf{Le}(\top)$ then $\mathrm{fse}\,T$ does not contain any expression substitutable entity (i.e. $\ddot{x}$). Also, if $E \vdash^H_c K\,\mathrm{ok}$ (respectively $E \vdash^H_c S\,\mathrm{ok}$) then $\mathrm{fse}\,K$ (respectively $\mathrm{fse}\,S$) does not contain any expression substitutable entity.

Note that $\mathrm{fse}\,\aleph$ not containing any expression substitutable entity implies that $\mathrm{fv}\,\aleph$ does not contain any free expression variable.

Proof. Let us first prove this lemma for a type T. Induct on the structure of T. Most cases are either obvious (X, $U.\mathtt{type}$) or obvious by induction (constructed type). The only non-trivial case is a hash type $h.\mathtt{type}$. By Lemma B.7 (hashes have to be ok), $\vdash h\,\mathrm{ok}$. By Lemma B.11 (free variables of a judgement come from the environment), $\mathrm{fv}\,h = \varnothing$, whence by Lemma B.18 (connection between fv and fse) $\mathrm{fse}\,h = \varnothing$.

If $E \vdash^H_c K\,\mathrm{ok}$, then by reversing (Kok.Eq) or (Kok.Le) we get $E \vdash^H_c T : \mathbf{Le}(\top)$, whence by the first part of this lemma $\mathrm{fse}\,K = \mathrm{fse}\,T$ has the desired property.

If $E \vdash^H_c [X : K, T]\,\mathrm{ok}$, then by reversing (Sok) we get $E, X : K \vdash^H_c T : \mathbf{Le}(\top)$. By the previous two paragraphs, neither $\mathrm{fse}\,K$ nor $\mathrm{fse}\,T$ contains any expression substitutable entity, so the same holds for $\mathrm{fse}\,S = \mathrm{fse}\,K \cup (\mathrm{fse}\,T \setminus X)$. □

Lemma B.20 (environments do not contain free expression variables)
If $E_0, E_1 \vdash^H_c$ ok then $\mathrm{fse}\, E_1$ does not contain any expression substitutable entity.

Note that in particular $\mathrm{fv}\, E_1$ does not contain any expression variable.

Proof. Induct on the length of E_1. If $E_1 = \mathbf{nil}$ the conclusion is obvious. Otherwise there exist E'_1, ζ and τ such that $E_1 = E'_1, \zeta : \tau$. By Lemma B.9 (prefixes of ok environments are ok), we have $H' \subseteq H$ and $E_0, E'_1 \vdash^{H'}_c$ ok by a subproof. By induction, we have $\ddot{x} \notin \mathrm{fse}\, E'_1$. Also, by Lemma B.19 (types do not contain free expression variables), $\ddot{x} \notin \mathrm{fse}\, \tau$ (whether τ is a type, kind or signature). Hence $\ddot{x} \notin \mathrm{fse}\, E_1$. □

Lemma B.21 (expression substitution in environments) If $E_0, E_1 \vdash^H_c$ ok and $\ddot{x}$ is an expression substitutable entity (i.e. an expression variable or $U.\mathtt{term}$ for some U) then $\{\ddot{x} \leftarrow \eta\} E_1 = E_1$.

Proof. Trivial consequence of Lemma B.20 (environments do not contain free expression variables). □

B.4 Weakening

Lemma B.22 ("type of a machine" judgements are not used to prove other coloured judgements) .

Proof. No rule whose conclusion is a coloured judgement other than "type of a machine" has a premise that is a "type of a machine" judgement. □

Lemma B.23 (colour stripping judgements)
If $E \vdash^H_c J$ and $\vdash c'$ ok and $c \subseteq c'$ then $E \vdash^H_{c'} J$ for all coloured statements J other than "type of a machine".

Proof. Induct on the derivation of $E \vdash^H_c J$. In most rules where the conclusion is a coloured judgement, all the premises either :

- do not involve the colour of the conclusion ; or
- are a coloured judgement of the same colour, other than "type of a machine", so we can use induction to prove them. Note that by Lemma B.22 ("type of a machine" judgements are not used to prove other coloured judgements), a premise that is a coloured judgement is never a "type of a machine" judgement.

If every premise of the last rule used in the derivation enjoys one of these properties, and if furthermore the rule applies to arbitrary colours, (so that replacing c by c' does yield an instance of the rule again), we have $E \vdash^H_{c'} J$. We list the remaining cases.

Case (envok.nil) : Trivial ($\vdash c'$ ok is assumed in this lemma).

Case (Teq.hash) : $h \in c \subseteq c'$, so $h \in c'$.

Case (eT.marred) : Trivial by induction.

Case (eT.col) : There exist e, T and c_0 such that the conclusion is $E \vdash^H_c [e]^T_{c_0} : T$. Since we have $c \cup c_0 \subseteq c' \cup c_0$ we can apply the induction hypothesis to the premise $E \vdash^H_{c \cup c_0} e : T$.

□

Lemma B.24 (subhash weakening)
If $E \vdash^H_c J$ and $E \vdash^{H'}_c$ ok and $H \subseteq H'$ then $E \vdash^{H'}_c J$.

Proof. Induct on the derivation of $E \vdash_c^H J$. In most rules where the conclusion has a subhash annotation, all the premises either :

- do not involve the subhash of the conclusion ; or
- are a judgement with the same subhash annotation and the same environment, so we can use induction to prove them.

We list the remaining cases.

Case $J = \text{ok}$ **:** The second hypothesis gives us the result.

Case (eT.fun) **:** The premise has an additional expression variable binding. We know that $E \vdash_c^{H'} \text{ok}$ and that $E \vdash_c^H \lambda x : T.e : T \to T'$ and $E, x : T \vdash_c^H e : T'$. By Lemma B.8 (environments and subhashes have to be ok), we get $E, x : T \vdash_c^H \text{ok}$ by a subproof. By reversing (envok.x) (x cannot be in $\operatorname{dom} H$), we have $E \vdash_c^H T : \mathbf{Le}(\top)$. By induction, we get $E \vdash_c^{H'} T : \mathbf{Le}(\top)$ and then by (envok.x) $E, x : T \vdash_c^{H'} \text{ok}$. We conclude by induction and (eT.fun).

Cases (Sok), (Ssub.struct), (MS.struct) **:** Similar to (eT.fun).

Case (mT.letext) **:** By the first premise $E \vdash_c^H T : \top$, we know that $U \notin \operatorname{dom} H$. The last premise is $E, U(T_0) : S \vdash_\bullet^{H \cup \kappa_1 <: U \cup \ldots \cup \kappa_i <: U \cup U <: \kappa'_1 \cup \ldots \cup U <: \kappa'_j} m : T$. By Lemma B.8 (environments and subhashes have to be ok) we get the judgement $E, U(T_0) : S \vdash_\bullet^{H \cup \kappa_1 <: U \cup \ldots \cup \kappa_i <: U \cup U <: \kappa'_1 \cup \ldots \cup U <: \kappa'_j} \text{ok}$ by a subproof. By reversing $i + j$ times (envok.Uhash) and (envok.hashU) we get $E, U(T_0) : S \vdash_\bullet^H \text{ok}$. Then we reverse (envok.U), apply the induction hypothesis and reapply (envok.U) to get $E, U(T_0) : S \vdash_\bullet^{H'} \text{ok}$. By induction and (envok.Uhash) and (envok.hashU) we have $E, U(T_0) : S \vdash_\bullet^{H' \cup \kappa_1 <: U \cup \ldots \cup \kappa_i <: U \cup U <: \kappa'_1 \cup \ldots \cup U <: \kappa'_j} \text{ok}$ and we conclude by induction and (mT.letext).

□

Lemma B.25 (weakening)
If $E, E', E'' \vdash_c^H \text{ok}$ and $E, E'' \vdash_c^H J$ then $E, E', E'' \vdash_c^H J$.

Proof. We freely use Lemma B.3 (non-membership in domain is interpreted trivially) and Lemma B.5 (non-membership in subhash domain is interpreted trivially).

Consider the variables that appear in the derivation of $E, E'' \vdash_c^H J$ but not in the judgement $E, E'' \vdash_c^H J$ itself. We can alpha-convert them to variables that are not present in $\operatorname{dom}(E, E', E'', H)$.

Induct on the derivation of $E, E'' \vdash_c^H J$.

Note that the inductive rules that define derivable coloured judgements are all rewriting rules (in other words, there is no side condition). Most rules have the following properties :

1. There is a distinguished environment metavariable $\widehat{E}$ such that the conclusion is a judgement with this metavariable at the leftmost position and no other.
2. For each premise, one of the following conditions holds :
   (a) The premise is a coloured judgement whose environment is either the same as the conclusion's or the one in the conclusion followed by exactly one more binding, with the same subhash relationship.
   (b) The premise is $\widehat{\zeta} \notin \operatorname{dom} \widehat{E}$ for some $\widehat{\zeta}$ that is in the domain of the conclusion.
   (c) The premise does not mention $\widehat{E}$.

Suppose that $E, E'' \vdash_c^H J$ was derived by an instance α of such a rule. There are two cases, depending on whether the instantiation of $\widehat{E}$ (as per condition 1) includes the whole of E or not.

Case $\widehat{E}$ is instantiated by E, E''' : Then there exists E'''' such that $E'' = E''', E''''$.

Since we have a raw term rewriting system, we get an instance ω of the same rule by instantiating $\widehat{E}$ by E, E', E''' and other variables as in α.

Let us prove that all the premises of ω hold. Consider a premise in α, depending on which case of condition 2 holds :

Case 2a : The premise is of the form $E, E'', E_i \vdash^H_{c_i} J_i$, where E_i is of length at most one. To apply the induction hypothesis, we need to prove $E, E', E'', E_i \vdash^H_{c_i}$ ok. By Lemma B.8 (environments and subhashes have to be ok), $E, E'', E_i \vdash^H_{c_i}$ ok by a subproof.

If E_i is empty, then we have $E, E', E'', E_i \vdash^H_{c_i}$ ok by using Lemma B.12 (ok environments are ok in every colour) on $E, E', E'' \vdash^H_c$ ok. Otherwise there exist ζ' and τ such that $E_i = \zeta' : \tau$ and $\zeta' \notin \mathrm{dom}\,(E, E'', H)$. The judgement $E, E'', \zeta' : \tau \vdash^H_{c_i}$ ok must have been derived by the appropriate (envok.x) or (envok.X) rule from $E, E'' \vdash^H_{c_i}$ ok and $E, E'' \vdash^H_\bullet \tau$ ok and $\zeta' \notin \mathrm{dom}\,(E, E'', H)$. By induction, we have $E, E', E'' \vdash^H_{c_i}$ ok and $E, E', E'' \vdash^H_\bullet \tau$ ok , whence by the same last (envok.x) or (envok.X) rule as previously : $E, E', E'', \zeta' : \tau \vdash^H_{c_i}$ ok (recalling that we performed alpha-conversion on the proof so that $\zeta' \notin \mathrm{dom}\, E''$ whence $\zeta' \notin \mathrm{dom}\,(E, E', E'', H)$). We finally get $E, E', E'', \zeta' : \tau \vdash^H_{c_i}$ ok.

In any case, $E, E', E'', E_i \vdash^H_{c_i}$ ok, so we can apply induction, getting $E, E', E'', E_i \vdash^H_{c_i} J_i$ as desired.

Case 2b : The premise is $\zeta' \notin \mathrm{dom}\,(E, E''')$ with ζ' in $\mathrm{dom}\,(E, E''', E'''')$. Hence $\zeta' \in \mathrm{dom}\, E''''$. Since $E, E', E''', E'''' \vdash^H_c$ ok, by Lemma B.10 (ok environments have no repetition in the domain), we have $\zeta \notin \mathrm{dom}\,(E, E', E''')$ as desired.

Case 2c : The premise in α is exactly the premise in ω.

The missing rules are (envok.hashU),(envok.Uhash),(envok.mod),(envok.modopp) and the (mT.letext).

Case (envok.hashU),(envok.Uhash),(envok.mod),(envok.modopp) : In these cases, the last rule of $E, E'' \vdash^H_c J$ and of $E, E', E'' \vdash^H_c$ ok is the same, and its object is the same binding in H. The premises of these two judgments can then be used to apply the induction hypothesis.

Case (mT.letext) : Our difficulty lies in the premise $E, E'', U(T) : S \vdash^{H \cup H'}_c J'$. To apply the induction hypothesis, we need to show that $E, E', E'', U(T) : S \vdash^{H \cup H'}_c$ ok. We first use Lemma B.8 (environments and subhashes have to be ok) to get $E, E'', U(T) : S \vdash^{H \cup H'}_c$ ok with $U \notin \mathrm{dom}\,(H)$. Then by reversing several times one of the The (envok.hashU),(envok.Uhash),(envok.mod),(envok.modopp) rules to peel off H', we get $E, E'', U(T) : S \vdash^H_c$ ok. The last rule is then (envok.U) : we can apply the induction hypothesis to the premises to finally get $E, E', E'', U(T) : S \vdash^H_c$ ok. Using this result and the induction hypothesis, we can rebuild the steps of the (envok.hashU),(envok.Uhash),(envok.mod),(envok.modopp) rules to yield the desired $E, E', E'', U(T) : S \vdash^{H \cup H'}_c$ ok.

Case $\widehat{E}$ is instantiated by a proper prefix E''' of E : Only the following cases are concerned :

Case (envok.x,X,U) : Trivial (take $\Pi' = \Pi$).

Case (*.var) : Then there exists E'''' such that $E = E''', \zeta' : \tau, E''''$. By assumption, we have a proof Π of $E''', \zeta' : \tau, E'''', E', E'' \vdash^H_c$ ok. By (*.var) we get a proof Π' of $E''', \zeta' : \tau, E'''', E', E'' \vdash^H_c \zeta' : \tau$ as desired.

The only rules whose conclusion is a coloured judgement that do not match the conditions above are (envok.nil) and (envok.hashhash). If the last step of the derivation uses one of these rules, then its conclusion is $\mathbf{nil} \vdash_c^H$ ok, and we desire a proof of $E' \vdash_c^H$ ok, which holds by assumption : take $\Pi' = \Pi$. □

Lemma B.26 (merging environments)
If $E, E' \vdash_c^H$ ok and $E, E'' \vdash_c^H$ ok and $\operatorname{dom} E' \cap \operatorname{dom} E'' = \varnothing$ then $E, E', E'' \vdash_c^H$ ok.

Proof. First we can deduce from $\operatorname{dom} E' \cap \operatorname{dom} E'' = \varnothing$ that free variables in H are only bound by E and are not present in E' or E''. Thus by several instances of Lemma B.9 (prefixes of ok environments are ok) $E \vdash_c^H$ ok.

We freely use Lemma B.3 (non-membership in domain is interpreted trivially) and Lemma B.5 (non-membership in subhash domain is interpreted trivially).

We induct on the length of E''. If $E'' = \varnothing$, the results are trivial. Now let us assume the lemma holds for $E'', \zeta' : \tau$, and we have $E, E'', \zeta' : \tau \vdash_c^H$ ok and $\operatorname{dom} E' \cap \operatorname{dom}(E'', \zeta' : \tau) = \varnothing$. Of course we know that $\zeta' \notin \operatorname{dom} H$.

By reversing the appropriate (envok.x,X,U) rule, we get $E, E'' \vdash_\bullet^H \tau$ ok (or its two equivalents in the (envok.U) case) and $E, E'' \vdash_c^H$ ok. By induction, we get $E, E', E'' \vdash_c^H$ ok. By Lemma B.12 (ok environments are ok in every colour) we have then $E, E', E'' \vdash_\bullet^H$ ok. Then we can apply Lemma B.25 (weakening) to get $E, E', E'' \vdash_\bullet^H \tau$ ok. We then apply the appropriate (envok.x,X,U) rule to get $E, E', E'', \zeta' : \tau \vdash_c^H$ ok as desired.

□

Lemma B.27 (combined weakening)
If $E, E' \vdash_c^H$ ok and $E, E'' \vdash_c^H J$ and $\operatorname{dom} E' \cap \operatorname{dom} E'' = \varnothing$ then $E, E', E'' \vdash_c^H J$.

Proof. Trivial combination of Lemma B.26 (merging environments) and Lemma B.25 (weakening). □

Lemma B.28 (environment and subhash weakening) If $E \vdash_c^H J$ and $E, E' \vdash_c^{H'}$ ok with $H \subseteq H'$ then $E, E' \vdash_c^{H'} J$.

Proof. By induction on the derivation of $E, E' \vdash_c^{H'}$ ok.

Case (envok.nil) : Trivial.

Cases (envok.x), (envok.X), (envok.U) : The conclusion is $E, E', \zeta : \tau \vdash_c^{H'}$ ok, and one premise is $E, E' \vdash_c^{H'}$ ok. By induction we get $E, E' \vdash_c^{H'} J$ and we conclude by Lemma B.25 (weakening).

Cases remaining (envok.*) rules : The conclusion is $E, E' \vdash_c^{H' \cup \{\kappa <: \kappa'\}}$ ok, and one premise is $E, E' \vdash_c^{H'}$ ok. By induction we get $E, E' \vdash_c^{H'} J$ and we conclude by Lemma B.24 (subhash weakening).

□

B.5 Type system

Lemma B.29 (reflexivity of kind equivalence) If $E \vdash_c^H K$ ok then $E \vdash_c^H K == K$.

Proof. If there exists T such that $K = \mathbf{Le}(T)$ or $K = \mathbf{Eq}(T)$, by reversing (Kok.Le) or (Kok.Eq), we get $E \vdash_c^H T : \mathbf{Le}(\top)$. By using (Teq.refl) and (Keq.Le), we have the desired $E \vdash_c^H K == K$. □

Lemma B.30 (transitivity of kind equivalence) If $E \vdash_c^H K == K'$ and $E \vdash_c^H K' == K''$ then $E \vdash_c^H K == K''$.

Proof. Both hypotheses have to be derived by the same rule.

Case (Keq.Le) : Trivial by (Teq.tran) and (Keq.Le).

Case (Keq.Eq) : Trivial by (Teq.tran) and (Keq.Eq).

□

Lemma B.31 (discreteness of subkinding) If $E \vdash_c^H K <: \mathbf{Eq}(T)$ then $E \vdash_c^H K == \mathbf{Eq}(T)$ by a subproof.

Proof. Induct on the derivation of $E \vdash_c^H K <: \mathbf{Eq}(T)$. If the last rule in the proof is (Ksub.tran), then the result holds by induction and Lemma B.30 (transitivity of kind equivalence). Otherwise the last rule is (Ksub.refl) and the premise is the desired result. □

Lemma B.32 (kinds are smaller than top) If $E \vdash_c^H K$ ok then $E \vdash_c^H K <: \mathbf{Le}(\top)$.

Proof. If there exists T such that $K = \mathbf{Le}(T)$, then by reversing (Kok.Le) we get $E \vdash_c^H T : \mathbf{Le}(\top)$. To have $E \vdash_c^H \mathbf{Le}(T) <: \mathbf{Le}(\top)$, we apply (Tsub.Le) and (Ksub.Le).

Otherwise there exists T such that $K = \mathbf{Eq}(T)$, then by reversing (Kok.Eq) and applying (Ksub.Eq) we get $E \vdash_c^H \mathbf{Eq}(T) <: \mathbf{Le}(\top)$. □

Lemma B.33 (transitivity of subtyping) If $E \vdash_c^H T <: T'$ and $E \vdash_c^H T' <: T''$ then $E \vdash_c^H T <: T''$.

Proof. By (Ksub.Le) we have $E \vdash_c^H \mathbf{Le}(T') <: \mathbf{Le}(T'')$. On the other side, we know that $E \vdash_c^H T : \mathbf{Le}(T')$ by (TK.Le). We use (TK.sub) to get $E \vdash_c^H T : \mathbf{Le}(T'')$. We conclude by (Tsub.Le). □

Lemma B.34 (signature equivalence is transitive) If $E \vdash_c^H S == S'$ and $E \vdash_c^H S' == S''$ then $E \vdash_c^H S == S''$.

Proof. Trivial by transitivity of type and kind equivalence. □

Lemma B.35 (components of modules are ok) If $E \vdash_c^H [T_0, v^c : T_1] : [X : K, T_1']$ then $E \vdash_c^H T_0 : K$ and $E, X : K \vdash_c^H T_1' : \mathbf{Le}(\top)$ and $E, X : \mathbf{Eq}(T_0) \vdash_c^H T_1 <: T_1'$ and $E \vdash_c^H v^c : T_1$ and $E \vdash_c^H K$ ok.

Proof. Reverse (MS.struct) to get the first four judgements. As for the last one, since $E, X : K \vdash_c^H T_1' : \mathbf{Le}(\top)$, by Lemma B.8 (environments and subhashes have to be ok) and reversing (envok.X) (X cannot be in dom H), we get $E \vdash_c K$ ok. □

Lemma B.36 (bindings in an ok environment are ok) If $E, \zeta : \tau, E' \vdash_c^H$ ok then we have a proof of $E, \zeta : \tau, E' \vdash_\bullet^H \tau$ ok or $E, \zeta : \tau, E' \vdash_\bullet^H \tau : \mathbf{Le}(\top)$ if τ is a type.

Proof. Induct on the derivation of $E, \zeta : \tau, E' \vdash_c^H$ ok.

Cases (envok.nil),(envok.hashhash) : The environment is empty.

Cases (envok.hashU), (envok.Uhash), (envok.mod), (envok.modopp) : A premise has then the form $E, \zeta : \tau, E' \vdash_\bullet^{H'} J$ with $H' \subseteq H$. Then by Lemma B.8 (environments and subhashes have to be ok), we have $E, \zeta : \tau, E' \vdash_c^{H'}$ ok by a subproof. By induction we get $E, \zeta : \tau, E' \vdash_c^{H'} \tau$ ok. By Lemma B.24 (subhash weakening), we finally have $E, \zeta : \tau, E' \vdash_c^H \tau$ ok.

Cases (envok.x),(envok.X),(envok.U) : If $E' = \mathbf{nil}$, then a premise will be $E \vdash^H_\bullet \tau\,\mathrm{ok}$ or $E \vdash^H_\bullet \tau : \mathbf{Le}(\top)$ if τ is a type. We conclude by Lemma B.25 (weakening). If $E' \neq \mathbf{nil}$, then there exist a ζ', a τ' and a E'' such that $E' = E'', \zeta' : \tau'$. A premise will then be $E, \zeta : \tau, E'' \vdash^H_c \mathrm{ok}$. By induction $E, \zeta : \tau, E'' \vdash^H_\bullet \tau\,\mathrm{ok}$. We conclude by Lemma B.25 (weakening).

□

Lemma B.37 (types are ok provided their hashes are) $E \vdash^H_c T : \mathbf{Le}(\top)$ iff $\mathrm{fv}\,T \subseteq \mathrm{dom}\,E$ and $E \vdash^H_c \mathrm{ok}$ and all the hashes in T are ok.

Proof. Through (Tsub.Le) and (TK.Le), we have $E \vdash^H_c T : \mathbf{Le}(\top)$ iff $E \vdash^H_c T <: \top$. We will use freely this equivalence in the following.

Suppose that $\mathrm{fv}\,T \subseteq E$ and $E \vdash^H_c \mathrm{ok}$, and all the hashes in T are ok. We prove that $E \vdash^H_c T <: \top$ by induction on the syntax of T.

Case $T =$ unit or $T =$ bytes or $T = \top$: Trivial by (Tsub.unit) or (Tsub.dyn) or (Tsub.top).

Case there exist T_1, T_2 such that $T = T_1 \to T_2$: Note that $\mathrm{fv}\,T_i \subseteq \mathrm{fv}\,T \subseteq \mathrm{dom}\,E$. By induction, $E \vdash^H_c T_i$ for $1 \leqslant i \leqslant 2$. By (Tsub.fun), we have $E \vdash^H_c T_1 \to T_2 <: \top$.

Case there exist $T_1, ..., T_j$ such that $T = T_1 * ... * T_j$: Note that $\mathrm{fv}\,T_i \subseteq \mathrm{fv}\,T \subseteq \mathrm{dom}\,E$. By induction, $E \vdash^H_c T_i <: \top$ for $1 \leqslant i \leqslant j$. By (Tsub.tuple), we have $E \vdash^H_c T_1 * ... * T_j <: \top$.

Case there exist $T_1, ..., T_j$ such that $T = \{l_1 : T_1; ...; l_j : T_j\}$: Note that $\mathrm{fv}\,T_i \subseteq \mathrm{fv}\,T \subseteq \mathrm{dom}\,E$. By induction, $E \vdash^H_c T_i <: \top$ for $1 \leqslant i \leqslant j$. By (Tsub.rec), we have $E \vdash^H_c \{l_1 : T_1; ...; l_j : T_j\} <: \top$.

Case T is a hash h.type : The hash h is ok by assumption, so we get $E \vdash^H_c h.\mathtt{type} <: \top$ by (Tsub.hash).

Case T is a type variable X : Since $X = \mathrm{fv}\,T \subseteq \mathrm{dom}\,E$, there exist E_1, K and E_2 such that $E = E_1, X : K, E_2$. By (TK.var), we get $E \vdash^H_c X : K$. Then we apply Lemma B.36 (bindings in an ok environment are ok) to get $E \vdash^H_\bullet K\,\mathrm{ok}$. By Lemma B.23 (colour stripping judgements), we get $E \vdash^H_c K\,\mathrm{ok}$. By Lemma B.32 (kinds are smaller than top), we have $E \vdash^H_c K <: \mathbf{Le}(\top)$, whence by (TK.sub) $E \vdash^H_c X : \mathbf{Le}(\top)$ as desired.

Case there exists U such that $T = U.\mathtt{type}$: Since $U = \mathrm{fv}\,T \subseteq \mathrm{dom}\,E$, there exist T_0, E_1, K, T' and E_2 such that $E = E_1, U(T_0) : [X : K, T'], E_2$. By (US.var) and (TK.mod), we get $E \vdash^H_c U.\mathtt{type} : K$. Then we apply Lemma B.36 (bindings in an ok environment are ok) to get $E \vdash^H_\bullet [X : K, T']\,\mathrm{ok}$. By reversing (Sok), we have $E, X : K \vdash^H_\bullet T' : \mathbf{Le}(\top)$. By Lemma B.8 (environments and subhashes have to be ok), we have $E, X : K \vdash^H_\bullet \mathrm{ok}$. By reversing (envok.X), we get $E \vdash^H_\bullet K\,\mathrm{ok}$. By Lemma B.23 (colour stripping judgements), we get $E \vdash^H_c K\,\mathrm{ok}$. By Lemma B.32 (kinds are smaller than top), we have $E \vdash^H_c K <: \mathbf{Le}(\top)$, whence by (TK.sub) and (Tsub.Le) $E \vdash^H_c U.\mathtt{type} <: \top$ as desired.

Now suppose $E \vdash^H_c T : \mathbf{Le}(\top)$. Then $\mathrm{fv}\,T \subseteq \mathrm{dom}\,E$ by Lemma B.11 (free variables of a judgement come from the environment). Also all the hashes in T are ok by Lemma B.7 (hashes have to be ok). Finally, $E \vdash^H_c \mathrm{ok}$ by Lemma B.8 (environments and subhashes have to be ok). □

Lemma B.38 (colour and subhash change preserves type okedness) If $\mathbf{nil} \vdash^{H_0}_{c_0} T : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash^{H_1}_{c_1} \mathrm{ok}$ then $\mathbf{nil} \vdash^{H_1}_{c_1} T : \mathbf{Le}(\top)$.

Proof. Trivial application of Lemma B.37 (types are ok provided their hashes are). □

Lemma B.39 (colour change preserves type okedness)
If $E \vdash^H_{c_0} T : \mathbf{Le}(\top)$ and $\vdash c_1$ ok then $E \vdash^H_{c_1} T : \mathbf{Le}(\top)$.

Proof. By Lemma B.37 (types are ok provided their hashes are) we know that $\mathrm{fv}\, T \subseteq \mathrm{dom}\, E$ and $E \vdash^H_{c_0}$ ok and all the hashes in T are ok.

By Lemma B.12 (ok environments are ok in every colour), we have $E \vdash^H_{c_1}$ ok

We finally apply the Lemma B.37 (types are ok provided their hashes are) in the other direction. □

Lemma B.40 (colour change preserves kind okedness)
If $E \vdash^H_{c_0} K$ ok and $\vdash c_1$ ok then $E \vdash^H_{c_1} K$ ok.

Proof. We reverse (Kok.Le) or (Kok.Eq), then apply Lemma B.39 (colour change preserves type okedness) and finally apply (Kok.Le) or (Kok.Eq) to get the desired result. □

Lemma B.41 (relating type-is-kind and subkinding) If $E \vdash^H_c T : K$ then $E \vdash^H_c \mathbf{Eq}(T) <: K$.

Proof. If there exists T' such that $K = \mathbf{Le}(T')$, then by applying (Tsub.Le) we get $E \vdash^H_c T <: T'$. From Lemma B.7 (hashes have to be ok) and Lemma B.8 (environments and subhashes have to be ok) and Lemma B.37 (types are ok provided their hashes are), we know that $E \vdash^H_c T : \mathbf{Le}(\top)$ and $E \vdash^H_c T' : \mathbf{Le}(\top)$. Then we use (Ksub.Eq) and (Ksub.Le) and (Ksub.tran) to get the desired $E \vdash^H_c \mathbf{Eq}(T) <: \mathbf{Le}(T')$.

If there exists T' such that $K = \mathbf{Eq}(T')$. Then we have a proof of $E \vdash^H_c T == T'$ by (Teq.Eq). We conclude by (Keq.Eq) and (Ksub.refl). □

Lemma B.42 (type equivalence is a congruence)
If $E \vdash^H_c T' == T''$ and $E \vdash^H_c T' <: T'''$ and $E, X : \mathbf{Le}(T''') \vdash^H_c T : \mathbf{Le}(\top)$ then $E \vdash^H_c \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$.

In particular, by using Lemma B.7 (hashes have to be ok) and Lemma B.37 (types are ok provided their hashes are), if $E \vdash^H_c T' == T''$ and $E, X : \mathbf{Le}(\top) \vdash^H_c T : \mathbf{Le}(\top)$ then $E \vdash^H_c \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$.

Proof. Induct on the structure of T.

Case $T =$ unit or $T =$ bytes or $T = U.\mathtt{type}$ or $T = Y \neq X$ or $T = h.\mathtt{type}$: Then $X \notin \mathrm{fv}\, T$ (if $T = h.\mathtt{type}$, this is because $\mathrm{fv}\, T = \varnothing$ by Lemma B.7 (hashes have to be ok) and Lemma B.11 (free variables of a judgement come from the environment)). By Lemma B.37 (types are ok provided their hashes are), the hashes of T are ok and $\mathrm{fv}\, T \subseteq \mathrm{dom}\, E \cup X$. By Lemma B.37 (types are ok provided their hashes are) in the other direction, since $\mathrm{fv}\, T \subseteq E$, we have $E \vdash^H_c T : \mathbf{Le}(\top)$. By (Teq.refl), we get $E \vdash^H_c T == T$ which is the desired result.

Case $T = X$: We have $E \vdash^H_c T' == T''$ as desired.

Case $T = T_1 \to T_2$: By induction, we have $E \vdash^H_c \{X \leftarrow T'\}T_i == \{X \leftarrow T''\}T_i$ for $i = 1, 2$. By (Teq.cong.fun), we get $E \vdash^H_c \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$ as desired.

Case $T = T_1 * ... * T_j$: By induction, we have $E \vdash^H_c \{X \leftarrow T'\}T_i == \{X \leftarrow T''\}T_i$ for $i = 1, ..., j$. By (Teq.cong.tuple), we get $E \vdash^H_c \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$ as desired.

Case $T = l_1 : T_1, ..., l_j : T_j$: By induction, we have $E \vdash^H_c \{X \leftarrow T'\}T_i == \{X \leftarrow T''\}T_i$ for $i = 1, ..., j$. By (Teq.cong.rec), we get $E \vdash^H_c \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$ as desired.

□

Lemma B.43 (variable substitution and equivalence)
If $E \vdash_c^H T : \mathbf{Le}(\top)$, and if $E \vdash_c^H X : \mathbf{Eq}(T_0)$ (or $E \vdash_c^H U.\mathtt{type} : \mathbf{Eq}(T_0)$) then $E \vdash_c^H T == \{X \leftarrow T_0\}T$ (or $E \vdash_c^H T == \{U.\mathtt{type} \leftarrow T_0\}T$).

This result is trivially extended to kinds and signatures equivalence.

Proof. By induction on the structure of T

Case $T =$ unit, $T =$ bytes, $T = U'.\mathtt{type} \neq U.\mathtt{type}$, $T = Y \neq X$ or $T = h.\mathtt{type}$: Then $X \notin \mathrm{fv}\, T$ (if $T = h.\mathtt{type}$, this is because $\mathrm{fv}\, T = \varnothing$ by Lemma B.7 (hashes have to be ok) and Lemma B.11 (free variables of a judgement come from the environment)). So that $\{X \leftarrow T_0\}T = T$. We conclude by (Teq.refl).

Case $T = X$: We have $\{X \leftarrow T_0\}T = T_0$ and we know by (Teq.Eq) that $E \vdash_c^H X == T_0$ which is the desired result.

Case $T = U.\mathtt{type}$: We have $\{U.\mathtt{type} \leftarrow T_0\}T = T_0$ and we know by (Teq.Eq) that $E \vdash_c^H U.\mathtt{type} == T_0$ which is the desired result.

Case $T = T_1 \to T_2$: By induction, we have $E \vdash_c^H T_i == \{X \leftarrow T_0\}T_i$ for $i = 1, 2$. By (Teq.cong.fun), we get $E \vdash_c^H T == \{X \leftarrow T_0\}T$ as desired.

Case $T = T_1 * ... * T_j$: By induction, we have $E \vdash_c^H T_i == \{X \leftarrow T_0\}T_i$ for $i = 1, ..., j$. By (Teq.cong.tuple), we get $E \vdash_c^H T == \{X \leftarrow T_0\}T$ as desired.

Case $T = \{l_1 : T_1, ..., l_j : T_j\}$: By induction, we have $E \vdash_c^H T_i == \{X \leftarrow T_0\}T_i$ for $i = 1, ..., j$. By (Teq.cong.rec), we get $E \vdash_c^H T == \{X \leftarrow T_0\}T$ as desired.

□

Lemma B.44 (type substitution in equivalence)
If $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$ and $E \vdash_c^H X == T_0$, and if $E \vdash_c^H \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$ or $E \vdash_c^H \{U.\mathtt{type} \leftarrow T_0\}T == \{U.\mathtt{type} \leftarrow T_0\}T'$ then $E \vdash_c^H T == T'$.

Proof. By (TK.Eq), we have $E \vdash_c^H X : \mathbf{Eq}(T_0)$ or $E \vdash_c^H U.\mathtt{type} : \mathbf{Eq}(T_0)$.

Then we can apply Lemma B.43 (variable substitution and equivalence) to get proofs of $E \vdash_c^H T == \{X \leftarrow T_0\}T$ and of $E \vdash_c^H T' == \{X \leftarrow T_0\}T'$, or $E \vdash_c^H T == \{U.\mathtt{type} \leftarrow T_0\}T$ and of $E \vdash_c^H T' == \{U.\mathtt{type} \leftarrow T_0\}T'$

By (Teq.tran) applied twice, we have $E \vdash_c^H \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$.

□

B.6 Type preservation by substitution

Definition B.45 (unresolved free variables of an environment) The unresolved free variables of an environment, written $\mathrm{ufv}\, E$, are defined as follows :

$$\begin{aligned}
\mathrm{ufv}\, \mathbf{nil} &= \varnothing \\
\mathrm{ufv}\, (x : T, E) &= (\mathrm{ufv}\, (E) \setminus x) \cup \mathrm{fv}\, T \\
\mathrm{ufv}\, (X : K, E) &= (\mathrm{ufv}\, (E) \setminus X) \cup \mathrm{fv}\, K \\
\mathrm{ufv}\, (U(T) : S, E) &= (\mathrm{ufv}\, (E) \setminus U) \cup (\mathrm{fv}\, T) \cup (\mathrm{fv}\, S)
\end{aligned}$$

It is immediate that $\mathrm{ufv}\, E \subseteq \mathrm{ufv}\, (E, E')$.

Lemma B.46 (computing unresolved free variables) $\mathrm{ufv}\,(E, E') = \mathrm{ufv}\,E \cup (\mathrm{ufv}\,E' \setminus \mathrm{dom}\,E)$

Proof. Induct on the length of E. The result is trivial if E is empty. If $E = \zeta : \tau, E''$, then $\mathrm{ufv}\,(E, E') = (\mathrm{ufv}\,(E'', E') \setminus \zeta) \cup \mathrm{fv}\,\tau$. By induction, $\mathrm{ufv}\,(E'', E') = \mathrm{ufv}\,E'' \cup (\mathrm{ufv}\,E' \setminus \mathrm{dom}\,E'')$. So $\mathrm{ufv}\,(E, E') = (\mathrm{ufv}\,E'' \cup (\mathrm{ufv}\,E' \setminus \mathrm{dom}\,E'') \setminus \zeta) \cup \mathrm{fv}\,\tau = (\mathrm{ufv}\,E'' \setminus \zeta) \cup (\mathrm{ufv}\,E' \setminus (\mathrm{dom}\,E'' \cup \zeta)) \cup \mathrm{fv}\,\tau = \mathrm{ufv}\,E \cup (\mathrm{ufv}\,E' \setminus \mathrm{dom}\,E)$ as desired. □

Lemma B.47 (ok environments have no unresolved free variables) If $E \vdash_c^H \mathrm{ok}$ then $\mathrm{ufv}\,E = \varnothing$.

Proof. We prove by induction on the derivation size that $E \vdash_c^H J$ implies $\mathrm{ufv}\,E = \varnothing$. Most rules whose conclusion is a coloured judgement $E \vdash_c^H J$ have at least one premise that is a coloured judgement whose environment is E, E' for some E', whence the induction hypothesis gives the desired result. Also, rules that have a conclusion of the form $\mathbf{nil} \vdash_c^H J$ are trivial.

The remaining rules are (envok.x,X,U). If we write the conclusion as $E, \zeta : \tau \vdash_c^H \mathrm{ok}$, one premise is $E \vdash_\bullet^H \tau\,\mathrm{ok}$. We have $\mathrm{ufv}\,E = \varnothing$ by induction. By Lemma B.11 (free variables of a judgement come from the environment), we have $\mathrm{fv}\,\tau \subseteq \mathrm{dom}\,E$. By Lemma B.46 (computing unresolved free variables), we have $\mathrm{ufv}\,(E, \zeta : \tau) = \mathrm{ufv}\,E \cup (\mathrm{ufv}\,(\zeta : \tau) \setminus \mathrm{dom}\,E) = \varnothing \cup (\mathrm{fv}\,\tau \setminus \mathrm{dom}\,E)$ thus $\mathrm{ufv}\,(E, \zeta : \tau) = \varnothing$ as desired. □

Lemma B.48 (only abstract modules are in subhashes) If $E \vdash_c^H J$ is derivable by a proof Π then any module name mentionned in a subhash relation referes to an abstract module.

Proof. Induct on the structure of Π. Most metavariables in the conclusion of rules whose conclusion is a coloured judgement also appear in at least one premise that is a coloured judgement. If H_0 is in the instantiation of such a metavariable then we have the desired result by induction. We list here the remaining cases where a subhash is explicited.

Case (envok.hashhash) : No module name is mentionned.

Cases (envok.hashU) and (envok.Uhash) : The module variable is binding an abstract module.

Cases (envok.mod) and (envok.modopp) : The two module variables explicitely mentionned are binding abstract modules.

Case (mT.letext) : The `extends` and `restricts` declarations are only present when the signature S is abstract and only accept parameters that are hashes or modules names that bind abstract modules.

□

Lemma B.49 (type preservation by substitution) If $E_0, \zeta : \tau, E \vdash_c^H J$ and $E_0 \vdash_\bullet^{H_0} \eta : \tau$ with $H_0 \subseteq H$ then $E_0, \sigma E \vdash_c^H \sigma J$ where $\sigma = \{\zeta \leftarrow \eta\}$ and $\zeta : \tau$ is an expression or type binding.

Proof. Induct on the derivation Π of $E_0, \zeta : \tau, E \vdash_c^H J$.

Note that the inductive rules that define derivable coloured judgements are all rewriting rules (in other words, there is no side condition). Most rules have the following properties :

1. There is a distinguished environment metavariable $\widehat{E}$ such that the conclusion is a judgement with this metavariable at the leftmost position and no other.
2. For each premise, one of the following conditions holds :

(a) The premise is a judgement with $\widehat{E}$ in the leftmost position, and in no other place.

(b) The premise is $\widehat{\zeta} \notin \operatorname{dom} \widehat{E}$ for some $\widehat{\zeta}$.

(c) The premise is a judgement with an empty environment and does not mention $\widehat{E}$.

Suppose that $E_0, \zeta : \tau, E \vdash^H_c J$ was derived by an instance α of such a rule. Without loss of generality, ζ is not in a binding position (including the domain of an environment) anywhere in $E_0, \zeta : \tau, E \vdash^H_c J$ except where shown; furthermore ζ is not in a binding position in any premise either, except in instances of $\widehat{E}$ if and where this includes $E_0, \zeta : \tau$. By condition 1, there are two possibilities :

General case : The instance α was obtained by instantiating the metavariable $\widehat{E}$ with $E_0, \zeta : \tau, E''$, where E'' is an environment. Hence E is of the form E'', E', where E' is an environment.

Since we have a raw term rewriting system, we also get an instance β of the same rule by instantiating $\widehat{E}$ by E_0, E'' and other metavariables as in α. Note that ζ does not appear in any binding position in β.

Note that the only places in the syntax where an expression (respectively type) variable is required are :

– in binders, which doesn't matter for σ as it does not affect variables that are bound in β.

(†)

– in the left-hand side of a non-clash judgement. These only occur in (envok.*) rules, in the form $\widehat{\zeta} \notin \operatorname{dom} \widehat{E}$. Let us write ζ' for the instantiation of $\widehat{\zeta}$. Then $\zeta' \neq \zeta$ as $\zeta' \notin \operatorname{dom}(E_0, \zeta : \tau, E'')$ is derivable.

Note furthermore that well-typed values are stable by substitution, as per Lemma B.17 (stability of values by substitution).[1] Hence σ is a well-sorted raw term substitution on β or any subterm thereof, so applying σ to β yields another instance ω of the rule. Note that the conclusion of ω is $\sigma E_0, \sigma E'' \vdash^H_c \sigma J$, which is also $E_0, \sigma E'' \vdash^H_c J$ as $\zeta \notin \operatorname{fv} E_0$ by Lemma B.8 (environments and subhashes have to be ok), Lemma B.9 (prefixes of ok environments are ok) and Lemma B.47 (ok environments have no unresolved free variables).

Consider the premises in α, depending on which case of condition 2 holds :

Case 2a : The premise is of the form $E_0, \zeta : \tau, E'', E'_i \vdash^H_{c_i} J_i$. We can apply induction, getting $E_0, \sigma E'', \sigma E'_i \vdash^H_{c_i} \sigma J_i$, which is the corresponding premise in ω (recall that $\sigma E_0 = E_0$).

Case 2b : The premise is of the form $\zeta' \notin \operatorname{dom}(E_0, \zeta : \tau, E'')$, and ζ' is not ζ (see (†) above). We need to prove that $\zeta' \notin \operatorname{dom}(E_0, \sigma E'')$. This follows easily, given that $\operatorname{dom}(E_0, \sigma E'') = \operatorname{dom}(E_0, E'') \subseteq \operatorname{dom}(E_0, \zeta : \tau, E'')$.

Case 2c : The premise is a judgement AJ with an empty environment, so by Lemma B.11 (free variables of a judgement come from the environment) ζ is not free in AJ. Hence $\sigma \mathrm{AJ} = \mathrm{AJ}$. Furthermore the premise does not include any instantation of $\widehat{E}$, so it is in fact exactly the premise needed in ω.

As all the premises of ω are derivable, its conclusion holds. It reads : $E_0, \sigma E'', \sigma E' \vdash^H_c \sigma J$, which is what we set out to prove.

Special cases : The instance was obtained by instantiating the metavariable $\widehat{E}$ to a prefix E_1 of E_0 : so there is E_2 such that $E_0 = E_1, E_2$. Only the following cases of the following rules are concerned.

[1] This is needed for (MS.struct), which requires a value in one place.

Cases (envok.x,X) : Then $E = \mathbf{nil}$ and $E_2 = \mathbf{nil}$. The proof obligation is $E_1 \vdash_c^H \text{ok}$, i.e. $E_0 \vdash_c^{H_0} \text{ok}$, which holds by Lemma B.9 (prefixes of ok environments are ok).

Cases (eT.var), (TK.var) : α is of the form

$$\frac{E_0, \zeta : \tau, E' \vdash_c^H \text{ok}}{E_0, \zeta : \tau, E' \vdash_c^H \zeta : \tau}$$

and $\sigma = \{\zeta \leftarrow \eta\}$, and we have $E_0 \vdash_\bullet^{H_0} \eta : \tau$. By induction, we have $E_0, \sigma E' \vdash_c^H \text{ok}$.

Furthermore, since $\vdash c\,\text{ok}$ by Lemma B.6 (colours have to be ok), $E_0 \vdash_c^{H_0} \eta : \tau$ by Lemma B.23 (colour stripping judgements). By Lemma B.28 (environment and subhash weakening) we get $E_0, \sigma E' \vdash_c^H \eta : \tau$ as desired.

Every remaining rule is inapplicable because the environment in the conclusion must be empty. □

Lemma B.50 (strengthening) If $E_0, \zeta : \tau, E \vdash_c^H J$ and J is a type-world judgement and $\zeta \notin \text{fv}\, E \cup \text{fv}\, J$ and $\zeta : \tau$ is a type or expression variable binding then $E_0, E \vdash_c^H J$.

Proof. Since $\zeta : \tau$ is a type or expression variable binding, we know that $\zeta \notin \text{dom}\, H$. By Lemma B.8 (environments and subhashes have to be ok) and Lemma B.9 (prefixes of ok environments are ok), we have $E_0, \zeta : \tau \vdash_c^{H'} \text{ok}$ with $H' \subseteq H$. Let us then reverse the rules (envok.x,X) that were applied to obtain this latter judgement :

Case $\zeta : \tau$ is $X : K$: Then we have $E_0 \vdash_\bullet^{H'} K\,\text{ok}$. Since there exists a type η such that $K = \mathbf{Le}(\eta)$ or $K = \mathbf{Eq}(\eta)$, by reversing (Kok.Le) or (Kok.Eq), we get $E_0 \vdash_\bullet^{H'} \eta : \mathbf{Le}(\top)$. By (Teq.refl) and (TK.Eq) or (Tsub.Equi) and (TK.Le), we get $E_0 \vdash_\bullet^{H'} \eta : K$.

Case $\zeta : \tau$ is $x : T$: Then we have $E_0 \vdash_\bullet^{H'} T : \mathbf{Le}(\top)$. Let η be $\mathbf{Unmarfailure}^T$. By (eT.Undynfailure), we have $E_0 \vdash_\bullet^{H'} \eta : T$.

In any case, we have $E_0 \vdash_\bullet^{H'} \eta : \tau$. By Lemma B.49 (type preservation by substitution), we have $E_0, \sigma E \vdash_c^H \sigma J$ where $\sigma = \{\zeta \leftarrow \eta\}$. However, by assumption, $\zeta \notin \text{fv}\, E \cup \text{fv}\, J$. Hence $\sigma E = E$ and $\sigma J = J$, so we get $E_0, E \vdash_c^H J$ as desired. □

Lemma B.51 (type preservation by expression substitution) If $E_0, x : T, E \vdash_c^H J$ and $E_0 \vdash_c^{H_0} e : T$ with $H_0 \subseteq H$ then $E_0, E \vdash_c^H \sigma J$ where $\sigma = \{x \leftarrow e\}$.

Proof. Induct on the derivation Π of $E_0, x : T, E \vdash_c^H J$.

Note that the inductive rules that define derivable coloured judgements are all rewriting rules (in other words, there is no side condition). Most rules have the following properties :

1. There is a distinguished environment metavariable $\widehat{E}$ such that the conclusion is a judgement with this metavariable at the leftmost position and no other.
2. For each premise, one of the following conditions holds :
   (a) The premise is a judgement with $\widehat{E}$ in the leftmost position, and in no other place, with the same colour.
   (b) The premise is $\zeta' \notin \text{dom}\, \widehat{E}$ for some ζ'.
   (c) The premise is a judgement with an empty environment and does not mention $\widehat{E}$.

Suppose that $E_0, x : T, E \vdash_c^H J$ was derived by an instance α of such a rule. Without loss of generality, x is not in a binding position (including the domain of an environment) anywhere in $E_0, x : T, E \vdash_c^H J$ except where shown ; furthermore x is not in a binding position in any premise either, except in instances of $\widehat{E}$ if and where this includes $E_0, x : \tau$. By condition 1, there are two possibilities :

General case : The instance α was obtained by instantiating the metavariable $\widehat{E}$ with $E_0, x : T, E''$, where E'' is an environment. Hence E is of the form E'', E', where E' is an environment.

Since we have a raw term rewriting system, we also get an instance β of the same rule by instantiating $\widehat{E}$ by E_0, E'' and other metavariables as in α. Note that x does not appear in any binding position in β.

Note that the only places in the syntax where an expression (respectively type) variable is required are :

(†)
- in binders, which doesn't matter for σ as it does not affect variables that are bound in β.
- in the left-hand side of a non-clash judgement. These only occur in (envok.*) rules, in the form $\widehat{\zeta} \notin \operatorname{dom} \widehat{E}$. Let us write y for the instantiation of $\widehat{\zeta}$. Then $y \neq x$ as $y \notin \operatorname{dom}(E_0, \zeta : \tau, E'')$ is derivable.

Note furthermore that well-typed values are stable by substitution, as per Lemma B.17 (stability of values by substitution).[2] Hence σ is a well-sorted raw term substitution on β or any subterm thereof, so applying σ to β yields another instance ω of the rule. Note that the conclusion of ω is $\sigma E_0, \sigma E'' \vdash_c^H \sigma J$, which is also $E_0, E'' \vdash_c^H \sigma J$ as $\zeta \notin \operatorname{fv} E_0$ by Lemma B.8 (environments and subhashes have to be ok), Lemma B.9 (prefixes of ok environments are ok) and Lemma B.21 (expression substitution in environments).

Consider the premises in α, depending on which case of condition 2 holds :

Case 2a : The premise is of the form $E_0, x : T, E'', E_i' \vdash_c^H J_i$. We can apply induction, getting $E_0, \sigma E'', \sigma E_i' \vdash_c^H \sigma J_i$, which is the corresponding premise in ω (recall that $\sigma E_0 = E_0$).

Case 2b : The premise is of the form $\zeta' \notin \operatorname{dom}(E_0, \zeta : \tau, E'')$, and ζ' is not ζ (see (†) above). We need to prove that $\zeta' \notin \operatorname{dom}(E_0, \sigma E'')$. This follows easily, given that $\operatorname{dom}(E_0, \sigma E'') = \operatorname{dom}(E_0, E'') \subseteq \operatorname{dom}(E_0, \zeta : \tau, E'')$.

Case 2c : The premise is a judgement AJ with an empty environment, so by Lemma B.11 (free variables of a judgement come from the environment) ζ is not free in AJ. Hence $\sigma\mathrm{AJ} = \mathrm{AJ}$. Furthermore the premise does not include any instantation of $\widehat{E}$, so it is in fact exactly the premise needed in ω.

The other rules are :

Cases (envok.hashU,Uhash,mod,modopp) : The premises with the empty colour are type-world judgements which implies, by Lemma B.19 (types do not contain free expression variables) that σ leaves them unchanged. We can use Lemma B.50 (strengthening) to get the desired premises.

Case (eT.col) : One of the premises uses the colour $c \cup c'$. We just need to use Lemma B.23 (colour stripping judgements) to get $E_0 \vdash_{c \cup c'}^{H_0} e : T$ and apply the induction hypothesis.

As all the premises of ω are derivable, its conclusion holds. It reads : $E_0, \sigma E'', \sigma E' \vdash_c^H \sigma J$, which is what we set out to prove.

[2] This is needed for (MS.struct), which requires a value in one place.

Special cases : The instance was obtained by instantiating the metavariable $\widehat{E}$ to a prefix E_1 of E_0 : so there is E_2 such that $E_0 = E_1, E_2$. Only the following cases of the following rules are concerned.

Cases (envok.x) : Then $E = \mathbf{nil}$ and $E_2 = \mathbf{nil}$. The proof obligation is $E_1 \vdash_c^H$ ok, i.e. $E_0 \vdash_c^{H_0}$ ok, which is exactly one of the premises.

Cases (eT.var) : α is of the form

$$\frac{E_0, x : T, E' \vdash_c^H \text{ ok}}{E_0, x : T, E' \vdash_c^H x : T}$$

and $\sigma = \{x \leftarrow e\}$, and we have $E_0 \vdash_c^{H_0} e : T$. By induction, we have $E_0, E' \vdash_c^H$ ok. By Lemma B.28 (environment and subhash weakening), we get $E_0, E' \vdash_c^H e : T$ as desired.

Every remaining rule is inapplicable because the environment in the conclusion must be empty. □

Lemma B.52 (weakening kind to ok kind in the environment) If $E_0 \vdash_\bullet^H K <: K'$ and $E_0 \vdash_c^H K$ ok and $E_0, X : K', E_1 \vdash_c^{H'} J$ and $H \subseteq H'$ and J is a type world judgement right-hand side then $E_0, X : K, E_1 \vdash_c^{H'} J$.

Note that the hypothesis $E_0 \vdash_c^H K$ ok is in fact superfluous (see Lemma (weakening kind in the environment) below).

Proof. Since $E_0 \vdash_c^H K$ ok, by Lemma B.40 (colour change preserves kind okedness), we have $E_0 \vdash_\bullet^H K$ ok.

By Lemma B.8 (environments and subhashes have to be ok), we know that $E_0 \vdash_c^H$ ok. Then, by (envok.X), $E_0, Y : K \vdash_c^H$ ok where Y is fresh. By Lemma B.25 (weakening), from $E_0, Y : K \vdash_c^H$ ok and $E_0 \vdash_c^H K'$ ok, we get $E_0, Y : K \vdash_c^H K'$ ok. By Lemma B.40 (colour change preserves kind okedness), we have then $E_0, Y : K \vdash_\bullet^H K'$ ok. By (envok.X), $E_0, Y : K, X : K' \vdash_c^H$ ok.

By Lemma B.28 (environment and subhash weakening), $E_0, Y : K, X : K', E_1 \vdash_c^{H'} J$.

From $E_0, Y : K \vdash_c^H$ ok, by Lemma B.12 (ok environments are ok in every colour) and by (TK.var), we get $E_0, Y : K \vdash_\bullet^H Y : K$. By Lemma B.25 (weakening), we also get $E_0, Y : K \vdash_\bullet^H K <: K'$. By (TK.sub), we get $E_0, Y : K \vdash_\bullet^H Y : K'$. By Lemma B.49 (type preservation by substitution), $E_0, Y : K', \{X \leftarrow Y\}E_1 \vdash_c^{H'} \{X \leftarrow Y\}J$. By alpha-conversion, we have $E_0, X : K', E_1 \vdash_c^{H'} J$ as desired. □

Lemma B.53 (things have to be ok)
If $E \vdash_c^H T : K$ then $E \vdash_c^H K$ ok.
If $E \vdash_c^H T == T'$ or $E \vdash_c^H T <: T'$ then $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$.
If $E \vdash_c^H K == K'$ or $E \vdash_c^H K <: K'$ then $E \vdash_c^H K$ ok and $E \vdash_c^H K'$ ok.
If $E \vdash_c^H S == S'$ or $E \vdash_c^H S <: S'$ then $E \vdash_c^H S$ ok and $E \vdash_c^H S'$ ok.
If $E \vdash_c^H e : T$ or $E \vdash_c^H m : T$ then $E \vdash_c^H T : \mathbf{Le}(\top)$.
If $E \vdash_c^H M : S$ or $E \vdash_c^H U : S$ then $E \vdash_c^H S$ ok.

Proof. Induct on the size of the derivation of the hypothesis. Consider the last rule used in said derivation.

Case (Keq.Le) : The conclusion is $E \vdash_c^H \mathbf{Le}(T) == \mathbf{Le}(T')$. The premise is $E \vdash_c^H T == T'$. By induction we get $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$, whence by (Kok.Le), $E \vdash_c^H \mathbf{Le}(T)$ ok and $E \vdash_c^H \mathbf{Le}(T')$ ok.

Case (Keq.Eq) : The conclusion is $E \vdash_c^H \mathbf{Eq}(T) == \mathbf{Eq}(T')$. The premise is $E \vdash_c^H T == T'$. By induction we get $E \vdash_c^H T : \mathbf{Type}$ and $E \vdash_c^H T' : \mathbf{Type}$, whence by (Kok.Eq), $E \vdash_c^H \mathbf{Eq}(T)$ ok and $E \vdash_c^H \mathbf{Eq}(T')$ ok.

Case (Ksub.Eq) : The conclusion is $E \vdash_c^H \mathbf{Eq}(T) <: \mathbf{Le}(T)$. The premise is $E \vdash_c^H T : \mathbf{Le}(\top)$. From this, by (Kok.Eq), we get $E \vdash_c^H \mathbf{Eq}(T)$ ok and by (Kok.Le), we get $E \vdash_c^H \mathbf{Le}(T)$ ok.

Case (Ksub.Le) : The conclusion is $E \vdash_c^H \mathbf{Le}(T) <: \mathbf{Le}(T')$. The premise is $E \vdash_c^H T <: T'$. By induction we know that $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$, whence by (Kok.Le), $E \vdash_c^H \mathbf{Le}(T)$ ok and $E \vdash_c^H \mathbf{Le}(T')$ ok.

Cases (Ksub.refl), (Ksub.tran) : Trivial by induction.

Case (TK.sub) : The conclusion is $E \vdash_c^H T : K'$ and one of the premises is $E \vdash_c^H K <: K'$. By induction, we get $E \vdash_c^H K'$ ok.

Case (TK.Eq) : The conclusion is $E \vdash_c^H T : \mathbf{Eq}(T')$. The premise is $E \vdash_c^H T == T'$. By induction we have $E \vdash_c^H T' : \mathbf{Le}(\top)$, hence $E \vdash_c^H \mathbf{Eq}(T')$ ok by (Kok.Eq).

Case (TK.Le) : The conclusion is $E \vdash_c^H T : \mathbf{Le}(T')$. The premise is $E \vdash_c^H T <: T'$. By induction we have $E \vdash_c^H T' : \mathbf{Le}(\top)$, hence $E \vdash_c^H \mathbf{Le}(T')$ ok by (Kok.Le).

Case (TK.var) : The conclusion is of the form $E, X : K, E' \vdash_c^H X : K$. The premise is $E, X : K, E' \vdash_c^H$ ok. By Lemma B.36 (bindings in an ok environment are ok), we get $E, X : K, E' \vdash_c^H K$ ok as desired.

Case (TK.mod) : The conclusion is $E \vdash_c^H U.\mathtt{type} : K$. The premise is $E \vdash_c^H U : [X : K, T]$. By Lemma B.11 (free variables of a judgement come from the environment), we know that $\operatorname{dom} H \subseteq \operatorname{dom} E$. By induction we have $E \vdash_c^H [X : K, T]$ ok. This must have been obtained by applying (Sok), with the premise $E, X : K \vdash_c^H T : \mathbf{Le}(\top)$, and we know that $X \notin \operatorname{dom} H$. By Lemma B.8 (environments and subhashes have to be ok), we have $E, X : K \vdash_c^H$ ok. Hence, by reversing (envok.X), we get a proof of $E \vdash_\bullet^H K$ ok which we can weaken to $E \vdash_c^H K$ ok by Lemma B.40 (colour change preserves kind okedness).

Case (TK.hash) : The conclusion is $E \vdash_c^H h.\mathtt{type} : K$ with $h = \mathbf{hash}(N, [T_0, v^c : T_1] : [X : K, T_1'])$. One of the premises is $\vdash \mathbf{hash}(N, [T_0, v^c : T_1] : [X : K, T_1'])$ ok. By reversing (hok.hash) we have $\mathbf{nil} \vdash_\bullet^{H'} [T_0, v^c : T_1] : [X : K, T_1']$. Note that by Lemma B.11 (free variables of a judgement come from the environment), we know that $\operatorname{dom} H' = \varnothing$. The last rule is then (MS.struct), we thus know that $X : K \vdash_\bullet^{H'} T_1' : \mathbf{Le}(\top)$. By Lemma B.8 (environments and subhashes have to be ok), $X : K \vdash_\bullet^{H'}$ ok. In all cases, we reverse (envok.X) to get $\mathbf{nil} \vdash_\bullet^{H'} K$ ok. By reversing (Kok.Le), and by the Lemma B.38 (colour and subhash change preserves type okedness) followed by another instance of (Kok.Le), we get $\mathbf{nil} \vdash_c^H K$ ok. We conclude by Lemma B.25 (weakening).

Case (Teq.Eq) : The conclusion is $E \vdash_c^H T == T'$. The premise is $E \vdash_c^H T : \mathbf{Eq}(T')$. By induction we have $E \vdash_c^H \mathbf{Eq}(T')$ ok, which must have been derived by (Kok.Eq) from $E \vdash_c^H T' : \mathbf{Le}(\top)$. From this, (Tsub.Le) and (Ksub.Le), we get $E \vdash_c^H \mathbf{Le}(T') <: \mathbf{Le}(\top)$. On the other side we can use (Ksub.Eq) to get $E \vdash_c^H \mathbf{Eq}(T') <: \mathbf{Le}(T')$. By (Ksub.tran) and (TK.sub), we have $E \vdash_c^H T : \mathbf{Le}(\top)$.

Case (Teq.hash) : The conclusion is $E \vdash_c^H h.\mathtt{type} == T_0$, and $h = \mathbf{hash}(N, [T_0, v^{c_1} : T_1] : S) \in c$. The premise is $E \vdash_c^H$ ok. By Lemma B.7 (hashes have to be ok), we get $\vdash \mathbf{hash}(N, [T, v^{c_1} : T_1] : S)$ ok. By reversing (hok.hash) we have $\mathbf{nil} \vdash_\bullet^{H'} [T_0, v^{c_1} : T_1] : S$ for some H'. Note that by Lemma B.11 (free variables of a judgement come from the environment), we know that $\operatorname{dom} H' = \varnothing$. We can reverse (MS.struct) to get $\mathbf{nil} \vdash_\bullet^{H'} T_0 : \mathbf{Le}(\top)$. By the Lemma B.38 (colour and subhash change preserves type

okedness), we have $\mathbf{nil} \vdash_c^H T_0 : \mathbf{Le}(\top)$. We conclude by Lemma B.25 (weakening). We also have $E \vdash_c^H h.\mathtt{type} <: \top$ by (Tsub.hash) using Lemma B.8 (environments and subhashes have to be ok). Then we know that $E \vdash_c^H h.\mathtt{type} : \mathbf{Le}(\top)$ by (TK.Le).

Case (Teq.refl) : Trivial.

Cases (Teq.sym), (Teq.tran) : Trivial by induction.

Case (Teq.cong.fun) : The conclusion is $E \vdash_c^H T_0 \to T_1 == T_0' \to T_1'$. By induction on the premises, we get $E \vdash_c^H T_j : \mathbf{Le}(\top)$ and $E \vdash_c^H T_j' : \mathbf{Le}(\top)$ for $j = 0, 1$. By (Tsub.Le) and (Tsub.fun), we get $E \vdash_c^H T_0 \to T_1 <: \top$ and $E \vdash_c^H T_0' \to T_1' <: \top$. We conclude by (TK.Le).

Case (Teq.cong.tuple) and (Teq.cong.rec) : Similar to case (Teq.cong.fun).

Case (Tsub.Le) : The conclusion is $E \vdash_c^H T <: T'$. The premise is $E \vdash_c^H T : \mathbf{Le}(T')$. By induction, we have $E \vdash_c^H \mathbf{Le}(T')\,\mathrm{ok}$, which must have been derived by (Kok.Le) from $E \vdash_c^H T' : \mathbf{Le}(\top)$. From this, (Tsub.Le) and (Ksub.Le), we get $E \vdash_c^H \mathbf{Le}(T') <: \mathbf{Le}(\top)$. Then we use (TK.sub) to get $E \vdash_c^H T : \mathbf{Le}(\top)$.

Case (Tsub.Equi) : Trivial by induction.

Case (Tsub.Subhash) : The conclusion is $E \vdash_c^H \kappa.\mathtt{type} <: \kappa'.\mathtt{type}$ with $\kappa <: \kappa' \in H$. The premise is $E \vdash_c^H$ ok. If κ or κ' is a hash, we use the Lemma B.7 (hashes have to be ok) and (Tsub.hash) and (TK.Le). If κ or κ' is a module variable U, then by Lemma B.11 (free variables of a judgement come from the environment), we know that $E = E_0, U(T) : S, E_1$ with $S = [X : K, T']$. By (US.var) we have a proof of $E \vdash_c^H U : S$. By Lemma B.35 (components of modules are ok) we know that $E \vdash_c^H K\,\mathrm{ok}$, and also by (TK.mod) we get $E \vdash_c^H U.\mathtt{type} : K$. We conclude with Lemma B.32 (kinds are smaller than top) and (TK.sub).

Case (Tsub.cong.record.width) : Trivial.

Case (Tsub.cong.fun) : The conclusion is $E \vdash_c^H T_0 \to T_1 <: T_0' \to T_1'$. By induction on the premises, we get $E \vdash_c^H T_j : \mathbf{Le}(\top)$ and $E \vdash_c^H T_j' : \mathbf{Le}(\top)$ for $j = 0, 1$. By (Tsub.Le) and (Tsub.fun), we get $E \vdash_c^H T_0 \to T_1 <: \top$ and $E \vdash_c^H T_0' \to T_1' <: \top$. We conclude by (TK.Le).

Case (Tsub.cong.tuple) : Similar to case (Tsub.cong.fun)

Cases (Tsub.hash), (Tsub.top), (Tsub.unit), (Tsub.fun), (Tsub.dyn), (Tsub.tuple), (Tsub.rec) : The conclusion is of the form $E \vdash_c^H T <: \top$ for some T. By Lemma B.8 (environments and subhashes have to be ok), $E \vdash_c^H$ ok, and we apply (Tsub.top) and (TK.Le) to get $E \vdash_c^H \top : \mathbf{Le}(\top)$. On the other side we just have to apply (TK.Le) to have $E \vdash_c^H T : \mathbf{Le}(\top)$.

Case (Seq.struct) : The conclusion is $E \vdash_c^H [X : K, T] == [X : K', T']$. One premise is $E, X : K \vdash_c^H T == T'$. By induction, we get $E, X : K \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$, whence the desired results by (Sok).

Case (Ssub.struct) : Similar to case (Seq.struct).

Case (Ssub.refl) : Trivial.

Case (Ssub.tran) : Trivial by induction.

Case (eT.sub) : The conclusion is $E \vdash_c^H {}_{(T<:T')}e : T'$. One of the premises is $E \vdash_c^H T <: T'$. By induction we have $E \vdash_c^H T' : \mathbf{Le}(\top)$.

Case (eT.eq) : The conclusion is $E \vdash_c^H e : T'$. One premise is $E \vdash_c^H T == T'$. By induction we get $E \vdash_c^H T' : \mathbf{Le}(\top)$.

Case (eT.var) : Similar to case (TK.var).

Case (eT.mod) : The conclusion is $E \vdash^H_c U.\mathtt{term} : T$. One premise is $E \vdash^H_c T : \mathbf{Le}(\top)$.

Case (eT.ap) : The conclusion is $E \vdash^H_c e\, e' : T'$. One premise is $E \vdash^H_c e' : T{\to}T'$. By induction we get $E \vdash^H_c T{\to}T' : \mathbf{Le}(\top)$. By Lemma B.37 (types are ok provided their hashes are) applied to $E \vdash^H_c T{\to}T' : \mathbf{Le}(\top)$ we get the desired $E \vdash^H_c T' : \mathbf{Le}(\top)$.

Case (eT.fun) : The conclusion is $E \vdash^H_c \lambda x : T.e : T{\to}T'$. The premise is $E, x : T \vdash^H_c e : T'$. By induction, we get $E, x : T \vdash^H_c T' : \mathbf{Le}(\top)$ by a proof Π. By Lemma B.19 (types do not contain free expression variables), $x \notin \mathrm{fv}\, T'$. By Lemma B.8 (environments and subhashes have to be ok), we have $E \vdash^H_c$ ok. By Lemma B.37 (types are ok provided their hashes are) applied once in each direction, we get first that the hashes in T' are ok and $\mathrm{fv}\, T' \subseteq \mathrm{dom}\, E \cup x$ (hence $\mathrm{fv}\, T' \subseteq \mathrm{dom}\, E$), then that $E \vdash^H_c T'$ ok. Also, by Lemma B.8 (environments and subhashes have to be ok), we get $E, x : T \vdash^H_c$ ok, whence $E \vdash^H_\bullet T : \mathbf{Le}(\top)$ by reversing (envok.x). By Lemma B.39 (colour change preserves type okedness), (Tsub.Le), (Tsub.fun) and (TK.Le), we get $E \vdash^H_c T{\to}T' : \mathbf{Le}(\top)$.

Cases (eT.tuple), (eT.record) : The conclusion is $E \vdash^H_c (e_1, ..., e_j) : T_1 * ... * T_j$ and $E \vdash^H_c \{l_1 = e_1, .., l_j = e_j\} : \{l_1 : T_1, ..., l_j : T_j\}$. The premises are $E \vdash^H_c e_i : T_i$ for $1 \leqslant i \leqslant j$. By induction, we have $E \vdash^H_c T_i : \mathbf{Le}(\top)$ for all i, whence by (Tsub.Le), (Tsub.tuple) or (Tsub.rec), and (TK.Le) : $E \vdash^H_c T_1 * ... * T_j : \mathbf{Le}(\top)$ and $E \vdash^H_c \{l_1 : T_1, ..., l_j : T_j\} : \mathbf{Le}(\top)$.

Cases (eT.proj), (eT.field) : Similar to (eT.ap).

Cases (eT.send), (eT.recv), (eT.mar), (eT.marred), (eT.unit) : By Lemma B.8 (environments and subhashes have to be ok), we have $E \vdash^H_c$ ok. Then (Tsub.unit) or (Tsub.dyn) with (TK.Le) gives the desired result.

Cases (eT.unmar), (eT.Undynfailure), (eT.col) : Trivial.

Case (MS.struct) : The conclusion is $E \vdash^H_c M : [X : K, T']$. One premise is $E, X : K \vdash^H_c T' : \mathbf{Le}(\top)$, whence by (Sok) : $E \vdash^H_c [X : K, T']$ ok.

Case (US.var) : Similar to case (TK.var).

Case (US.self) : The conclusion is $E \vdash^H_c U : [X : \mathbf{Eq}(U.\mathtt{type}), T]$. The premise is $E \vdash^H_c U : [X : K, T]$. By (TK.mod), we have $E \vdash^H_c U.\mathtt{type} : K$. And by induction we get $E \vdash^H_c [X : K, T]$ ok, whence by reversing (Sok) : $E, X : K \vdash^H_c T : \mathbf{Le}(\top)$.

By Lemma B.8 (environments and subhashes have to be ok) and reversing (envok.X), we have $E \vdash^H_\bullet K$ ok. By Lemma B.32 (kinds are smaller than top), we get $E \vdash^H_\bullet K <: \mathbf{Le}(\top)$. By (TK.sub), given that $E \vdash^H_\bullet U.\mathtt{type} : K$, we have $E \vdash^H_\bullet U.\mathtt{type} : \mathbf{Le}(\top)$, whence by (Kok.Eq) : $E \vdash^H_\bullet \mathbf{Eq}(U.\mathtt{type})$ ok.

If there exists a type T such that $K = \mathbf{Le}(T)$, then we have $E \vdash^H_\bullet \mathbf{Le}(U.\mathtt{type}) <: \mathbf{Le}(T)$ by (Tsub.Le) and (Ksub.Le). With (Ksub.Eq), and (Ksub.tran), we get $E \vdash^H_\bullet \mathbf{Eq}(U.\mathtt{type}) <: \mathbf{Le}(T)$. Otherwise there exists T such that $K = \mathbf{Eq}(T)$. From $E \vdash^H_\bullet U.\mathtt{type} : \mathbf{Eq}(T)$, by (Teq.Eq), (Keq.Eq) and (Ksub.refl), we get $E \vdash^H_\bullet \mathbf{Eq}(U.\mathtt{type}) <: \mathbf{Eq}(T)$. In either case we have $E \vdash^H_\bullet \mathbf{Eq}(U.\mathtt{type}) <: K$.

By Lemma B.52 (weakening kind to ok kind in the environment), $E, X : \mathbf{Eq}(U.\mathtt{type}) \vdash^H_c T : \mathbf{Le}(\top)$. Hence by (Sok) we have $E \vdash^H_c [X : \mathbf{Eq}(U.\mathtt{type}), T]$ ok.

Cases (mT.expr), (mT.letext) : Trivial by induction.

□

Lemma B.54 (type preservation by guarded expression variable substitution)

If $E_0, x : T, E \vdash^H_c J$ and $E_0 \vdash^{H_0}_{c'} e : T$ and $E_0 \vdash^{H_0}_\bullet$ ok, with $H_0 \subseteq H$, then $E_0, \sigma E \vdash^H_c \sigma J$ where $\sigma = \{x{\leftarrow}[e]^T_{c'}\}$.

Proof. $E_0 \vdash^{H_0}_{c'} T : \mathbf{Le}(\top)$ by Lemma B.53 (things have to be ok). By Lemma B.37 (types are ok provided their hashes are) applied one in each direction, we get $E_0 \vdash^{H_0}_{\bullet} T : \mathbf{Le}(\top)$. Applying (eT.col) to this and $E_0 \vdash^{H_0}_{c'} e : T$ yields $E_0 \vdash^{H_0}_{\bullet} [e]^T_{c'} : T$. We can now apply Lemma B.49 (type preservation by substitution) to get the desired result. □

Lemma B.55 (kind rewriting in environments) If $E_0, X : K, E_1 \vdash^H_c J$ and $E_0 \vdash^{H'}_c K' <: K$ with $H' \subseteq H$, then $E_0, X : K', E_1 \vdash^H_c J$.

In particular, thanks to (Ksub.refl), If $E_0, X : K, E_1 \vdash^H_c J$ and $E_0 \vdash^{H'}_c K == K'$ with $H' \subseteq H$, then $E_0, X : K', E_1 \vdash^H_c J$.

Proof. We induct on the derivation Π of $E_0, X : K, E_1 \vdash^H_c J$.

Note that the inductive rules that define derivable coloured judgements are all rewriting rules (in other words, there is no side condition). Most rules have one of the following properties :

1. The conclusion can be written $\widehat{E} \vdash^{H_0}_{c_0} J_0$, with $\widehat{E}$ a distinguished metavariable, or a concatenation of several metavariables with some expression and module variable bindings. Then, for each premises, one of the following conditions holds :
   (a) The premise can be written $\widehat{E}, \widehat{E}' \vdash^{H_1}_{c_1} J_1$ with $\widehat{E}'$ an optional additional binding.
   (b) The premise is a judgement with an empty environment and does not mention $\widehat{E}$.
2. The conclusion can be written $\widehat{E}, \zeta : \tau, \widehat{E}' \vdash^{H_0}_{c_0} J_0$, with $\widehat{E}$ and $\widehat{E}'$ some distinguished metavariables ($\widehat{E}'$ may be absent). Then, for each premise, one of the following conditions holds :
   (a) The premise is a judgement with exactly the same environment as the conclusion.
   (b) The premise is written $\widehat{E} \vdash^{H_1}_{c_1} J_1$.
   (c) The premise is $\zeta \notin \operatorname{dom} \widehat{E}$ or $\zeta \notin \operatorname{dom} H$.
   (d) The premise is a judgement with an empty environment and does not mention $\widehat{E}$, τ or $\widehat{E}'$.

If $E_0, X : K, E_1 \vdash^H_c J$ is derived by in instance α of one of these rules, we distinguish the cases.

Case 1 : For each premise, we just distinguish the cases :

Case 1a : We can apply the induction hypothesis.

Case 1b : We keep this part of the proof intact to build our result.

We conclude by applying the same rule to the new premises.

Case 2 : In this case, we first suppose that $\zeta : \tau$ is not instantiated by $X : K$. Then, for each premise, we distinguish the cases :

Case 2a : Concerning this premise, we can apply the induction hypothesis.

Case 2b : We just apply the induction hypothesis.

Case 2c : The variable is not X, so we can keep this part of the proof.

Case 2d : We keep this part of the proof.

Finally, since the rule did not mention $X : K$, we can apply this rule to the new premises to yield the desired result.

Now, we consider the case where $\zeta : \tau$ is instantiated by $X : K$. This is the case in exactly two rules.

Case (envok.X) : By Lemma B.53 (things have to be ok), we know that $E \vdash_c^{H'} K'$ ok. By Lemma B.24 (subhash weakening) and Lemma B.40 (colour change preserves kind okedness), we have $E_0 \vdash_\bullet^H K'$ ok. Then we can combine it with the already known premise $X \notin \operatorname{dom} E_0$, with (envok.X), to get $E_0, X : K' \vdash_c^H$ ok.

Case (TK.var) : The premise is $E_0, X : K, E_1 \vdash_c^H$ ok. By induction, we have a proof of $E_0, X : K', E_1 \vdash_c^H$ ok.

We have a proof of $E_0 \vdash_c^{H_0} K' <: K$. Then, by Lemma B.28 (environment and subhash weakening), we have $E_0, X : K', E_1 \vdash_c^H K' <: K$. Since, by (TK.var), we have $E_0, X : K', E_1 \vdash_c^H X : K'$, we conclude by (TK.sub), to get the desired $E_0, X : K', E_1 \vdash_c^H X : K$.

□

Lemma B.56 (signature rewriting in environments) If $E_0, U(T) : S, E_1 \vdash_c^H J$ and $E_0 \vdash_c^{H'} T == T'$ and $E_0 \vdash_c^{H'} S == S'$ with $H' \subseteq H$,then $E_0, U(T') : S', E_1 \vdash_c^H J$.

Proof. We induct on the derivation Π of $E_0, U(T) : S, E_1 \vdash_c^H J$.

Note that the inductive rules that define derivable coloured judgements are all rewriting rules (in other words, there is no side condition). Most rules have one of the following properties :

1. The conclusion can be written $\widehat{E} \vdash_{c_0}^{H_0} J_0$, with $\widehat{E}$ a distinguished metavariable, or a concatenation of several metavariables with some expression and module variable bindings. Then, for each premises, one of the following conditions holds :
   (a) The premise can be written $\widehat{E}, \widehat{E}' \vdash_{c_1}^{H_1} J_1$ with $\widehat{E}'$ an optional additional binding.
   (b) The premise is a judgement with an empty environment and does not mention $\widehat{E}$.
2. The conclusion can be written $\widehat{E}, \zeta : \tau, \widehat{E}' \vdash_{c_0}^{H_0} J_0$, with $\widehat{E}$ and $\widehat{E}'$ some distinguished metavariables ($\widehat{E}'$ may be absent). Then, for each premise, one of the following conditions holds :
   (a) The premise is a judgement with exactly the same environment as the conclusion.
   (b) The premise is written $\widehat{E} \vdash_{c_1}^{H_1} J_1$.
   (c) The premise is $\zeta \notin \operatorname{dom} \widehat{E}$ or $\zeta \notin \operatorname{dom} H$.
   (d) The premise is a judgement with an empty environment and does not mention $\widehat{E}$, τ or $\widehat{E}'$.

If $E_0, U(T) : S, E_1 \vdash_c^H J$ is derived by in instance α of one of these rules, we distinguish the cases.

Case 1 : For each premise, we just distinguish the cases :

Case 1a : We can apply the induction hypothesis.

Case 1b : We keep this part of the proof intact to build our result.

We conclude by applying the same rule to the new premises.

Case 2 : In this case, we first suppose that $\zeta : \tau$ is not instantiated by $U(T) : S$. Then, for each premise, we distinguish the cases :

Case 2a : Concerning this premise, we can apply the induction hypothesis.

Case 2b : We just apply the induction hypothesis.

Case 2c : The variable is not U, so we can keep this part of the proof.

Case 2d : We keep this part of the proof.

Finally, since the rule did not mention $U(T) : S$, we can apply this rule to the new premises to yield the desired result.

Now, we consider the case where $\zeta : \tau$ is instantiated by $U(T) : S$. This is the case in the following rules.

Case (envok.U) : We write $S = [X : K, T'']$.

By Lemma B.24 (subhash weakening), we have $E_0 \vdash_c^H T == T'$ and $E_0 \vdash_c^H S == S'$.

One of the premises is $E_0 \vdash_c^H T : K$. If $K = \mathbf{Eq}(T_1)$, then by (Teq.Eq), (Teq.sym), (Teq.tran) and (TK.Eq), we have $E_0 \vdash_c^H T' : K$. If $K = \mathbf{Le}(T_1)$, then by (Teq.sym), (Tsub.Equi), (TK.Le), (Tsub.Le), (Ksub.Le) and (TK.sub), we get $E_0 \vdash_c^H T' : K$.

By Lemma B.53 (things have to be ok), we know that $E \vdash_c^{H'} S'$ ok. By Lemma B.24 (subhash weakening) if necessary, we have $E_0 \vdash_c^H S'$ ok. Then we can combine it with the already known premises $U \notin \mathrm{dom}\, E_0$ and $U \notin \mathrm{dom}\, H$, with (envok.U), to get $E_0, U(T') : S' \vdash_c^H$ ok.

Cases (envok.hashU) and (envok.Uhash) and (envok.mod) and (envok.modopp) : The situation is very similar to the preceding case.

Case (US.var) : The premise is $E_0, U(T) : S, E_1 \vdash_c^H$ ok. By induction, we have a proof of $E_0, U(T') : S', E_1 \vdash_c^H$ ok. So that by (US.var), we get $E_0, U(T') : S', E_1 \vdash_c^H U : S'$.

We have a proof of $E_0 \vdash_c^{H'} S == S'$. Then, by Lemma B.28 (environment and subhash weakening), we have $E_0, U(T') : S', E_1 \vdash_c^H S == S'$ and, by symmetry, $E_0, U(T') : S', E_1 \vdash_c^H S' == S$. Then, by (US.eq), we have the desired $E_0, U(T') : S', E_1 \vdash_c^H U : S$.

□

Lemma B.57 (reversing subsignaturing judgement)
If $E \vdash_c^H [X : K, T] <: [X : K', T']$ then $E \vdash_c^H K <: K'$ and $E, X : K \vdash_c^H T == T'$.

Proof. Induct on the derivation of $E \vdash_c^H [X : K, T] <: [X : K', T']$.

Case (Ssub.refl) : The premise is $E \vdash_c^H [X : K, T]$ ok, which must have been derived by (Sok) from $E, X : K \vdash_c^H T : \mathbf{Le}(\top)$. By (Teq.refl), we have $E, X : K \vdash_c^H T == T$. By Lemma B.8 (environments and subhashes have to be ok), we have $E, X : K \vdash_c^H$ ok. By reversing (envok.X), we get $E \vdash_\bullet^H K$ ok whence $E \vdash_c^H K <: K$ by Lemma B.40 (colour change preserves kind okedness), Lemma B.29 (reflexivity of kind equivalence) and (Ksub.refl).

Case (Ssub.tran) : The premises are $E \vdash_c^H [X : K, T] <: [X : K'', T'']$ and $E \vdash_c^H [X : K'', T''] <: [X : K', T']$. By induction twice, we have : $E \vdash_c^H K <: K''$, $E, X : K \vdash_c^H T == T''$, $E \vdash_c^H K'' <: K'$ and $E, X : K'' \vdash_c^H T'' == T'$. By (Ksub.tran), we have $E \vdash_c^H K <: K'$. By Lemma B.55 (kind rewriting in environments) and (Teq.tran), we get $E, X : K \vdash_c^H T == T'$ as desired.

Case (Ssub.struct) : The premises are the desired judgements.

□

Lemma B.58 (type preservation by module substitution in coloured judgements)
Suppose $U(T_0) : [X : K, T], E \vdash_c^H J$ and z and Z are fresh.
If the signature is abstract, let h be a correct hash of a module with $[X : K, T]$ as signature and T_0 as internal type. In this case, we choose σ to be $\{U{\leftarrow}h, U.\mathtt{type}{\leftarrow}Z, U.\mathtt{term}{\leftarrow}z\}$. Otherwise $\sigma = \{U.\mathtt{type}{\leftarrow}Z, U.\mathtt{term}{\leftarrow}z\}$.
If $J = U : S'$ for some S' then we have the following judgement $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash_c^{\sigma H} [X : \mathbf{Eq}(Z), T] <: \sigma S'$. Else we can prove $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash_c^{\sigma H} \sigma J$.

Proof. Write $S = [X : K, T]$. Without loss of generality, we assume that $X \notin \operatorname{dom} E$. We also know that $\operatorname{fv}(T_0) = \operatorname{fv}(K) = \varnothing$.

We induct on the derivation Π of $U(T_0) : S, E \vdash_c^H J$.

Note that the inductive rules that define derivable coloured judgements are all rewriting rules (in other words, there is no side condition). Most rules have the following properties :

1. There is a distinguished environment metavariable $\widehat{E}$ such that the conclusion is a judgement with this metavariable at the leftmost position and no other.
2. For each premise, one of the following conditions holds :
   (a) The premise is a judgement with $\widehat{E}$ in the leftmost position, and in no other place. Also, the right-hand side of the premise is not $U : S'$ for any S'.
   (b) The premise is $\widehat{\zeta} \notin \operatorname{dom} \widehat{E}$ or $\widehat{\zeta} \notin \operatorname{dom} H$ for some $\widehat{\zeta}$.
   (c) The premise is a judgement with an empty environment and does not mention $\widehat{E}$ nor $\widehat{H}$.

Suppose that $U(T_0) : S, E \vdash_c^H J$ was derived by an instance α of such a rule. U is not in a binding position (including the domain of an environment) anywhere in α except as the first binding in an instance of $\widehat{E}$. We distinguish two possibilities, using 1 :

General case : The instance α was obtained by instantiating the metavariable $\widehat{E}$ with $U : S, E''$, where E'' is an environment. Hence E is of the form E'', E', where E' is an environment. Also, in this part of the proof, we assume that J is not of the form $U : S'$, and that the rule is not one of (TK.mod) or (eT.mod) with U in the conclusion.
Since we have a raw term rewriting system, we also get an instance β of the same rule by instantiating $\widehat{E}$ by $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E''$ and applying the substitution σ to the other parts of the rule.
Consider the premises in α, depending on which case of condition 2 holds :

Case 2a : The premise is of the form $U : S, E'', E_i' \vdash_{c_i}^{H_i} J_i$. Furthermore J_i is not $U : S'$ for any S', because we have excluded the rules (TK.mod) and (eT.mod) in the problematic case. By induction, we get $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E'', \sigma E_i' \vdash_{c_i}^{\sigma H_i} \sigma J_i$, which is the corresponding premise in β.

Case 2b : The premise is of the form $\zeta' \notin \operatorname{dom}(U : S, E'')$ or $\zeta' \notin \operatorname{dom} H$ for some ζ'. Given that $\operatorname{dom} \sigma E'' = \operatorname{dom} E'' \subseteq \operatorname{dom}(U : S, E'')$ and that $\operatorname{dom} \sigma \widehat{H} \subseteq \operatorname{dom} \widehat{H}$, we have $\zeta' \notin \operatorname{dom} \sigma E''$ and $\zeta' \notin \operatorname{dom} \sigma \widehat{H}$. Since Z and z are fresh, we have $\zeta' \notin \operatorname{dom}(Z : K, z : \{X{\leftarrow}Z\}T, \sigma E'')$, which is the corresponding premise in β.

Case 2c : The premise is a judgement AJ with an empty environment and a sub-hash relation H_1, so by Lemma B.11 (free variables of a judgement come from the environment) U is not free in AJ and $\operatorname{dom} H_1 = \varnothing$. Hence $\sigma\mathrm{AJ} = \mathrm{AJ}$. Furthermore the premise does not include any instantation of $\widehat{E}$, so it is in fact exactly the premise needed in β.

As all the premises of β are derivable, its conclusion holds. It reads : $Z : K, z : \{X \leftarrow Z\}T, \sigma E'', \sigma E' \vdash_c^{\sigma H} \sigma J$, which is what we set out to prove.

Case $\widehat{E}$ is instantiated to nil and J is not $U : S'$ for any S' : Only the following cases of the following rules are concerned.

Case (envok.U) : We have $U : S \vdash_c^H$ ok, and one of the premises is $\vdash_\bullet^H S$ ok. We know that U does not bind in it and then that $\sigma H = H$. By reversing (Sok), we have $X : K \vdash_\bullet^H T : \mathbf{Le}(\top)$. By Lemma B.39 (colour change preserves type okedness) and alpha-conversion, we get $Z : K \vdash_c^H \{X \leftarrow Z\}T : \mathbf{Le}(\top)$.

Case (US.var) : Impossible because J would be $U : S$.

Cases (envok.hashU) and (envok.Uhash) : The two rules are similar, so that we will only consider the rule (envok.Uhash). If we write h' the hash $\mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''''] : [X : \mathbf{Le}(T_1''), T_1'])$, the conclusion is the judgement $U(T_0) : [X : K, T] \vdash_c^{H \cup U <: h'}$ ok and the premises are $\vdash h'$ ok and $U(T_0) : [X : K, T] \vdash_\bullet^H T_0 <: T_1$ and $U(T_0) : [X : K, T] \vdash_\bullet^H K <: \mathbf{Le}(T_1'')$ and $U(T_0) : [X : K, T] \vdash_c^H$ ok. We can apply the induction hypothesis to get $Z : K, z : \{X \leftarrow Z\}T \vdash_\bullet^{\sigma H} \sigma T_0 <: \sigma T_1$ and $Z : K, z : \{X \leftarrow Z\}T \vdash_\bullet^{\sigma H} \sigma K <: \sigma(\mathbf{Le}(T_1''))$ and $Z : K, z : \{X \leftarrow Z\}T \vdash_c^{\sigma H}$ ok . Since $\mathrm{fv}(T_0) = \mathrm{fv}(K) = \varnothing$ and $\vdash h'$ ok we know that $\sigma T_0 = T_0$ and $\sigma T_1 = T_1$ and $\sigma K = K$ and $\sigma T_1'' = T_1''$. Also, by Lemma B.50 (strengthening), we get $\vdash_\bullet^{\sigma H} T_0 <: T_1$ and $\vdash_\bullet^{\sigma H} \mathbf{Le}(T_0'') <: \mathbf{Le}(T_1'')$ and $\vdash_c^{\sigma H}$ ok. Since we know that h is correct, we can apply (envok.hashhash) to get $: \vdash_c^{\sigma H \cup h <: h'}$ ok.

From Lemma B.53 (things have to be ok) we know that $Z : K, z : \{X \leftarrow Z\}T \vdash_c^{\sigma H} K$ ok. By Lemma B.50 (strengthening), we get $\vdash_c^{\sigma H} K$ ok. Since we have $\vdash_c^{\sigma H \cup h <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''''] : [X : \mathbf{Le}(T_1''), T_1'])}$ ok, by Lemma B.24 (subhash weakening) and Lemma B.40 (colour change preserves kind okedness), we get $\vdash_\bullet^{\sigma H \cup h <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''''] : [X : \mathbf{Le}(T_1''), T_1'])} K$ ok. Then we can apply (envok.X) to get $Z : K \vdash_c^{\sigma H \cup h <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''''] : [X : \mathbf{Le}(T_1''), T_1'])}$ ok.

From Lemma B.36 (bindings in an ok environment are ok) we know that $Z : K, z : \{X \leftarrow Z\}T \vdash_c^{\sigma H} \{X \leftarrow Z\}T : \mathbf{Le}(\top)$, and from Lemma B.50 (strengthening) $Z : K \vdash_c^{\sigma H} \{X \leftarrow Z\}T : \mathbf{Le}(\top)$. It is then possible to apply Lemma B.24 (subhash weakening) to get $Z : K \vdash_c^{\sigma H \cup h <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''''] : [X : \mathbf{Le}(T_1''), T_1'])} \{X \leftarrow Z\}T : \mathbf{Le}(\top)$. Then, by Lemma B.39 (colour change preserves type okedness) and (envok.x), we get $Z : K, z : \{X \leftarrow Z\}T \vdash_c^{\sigma H \cup h <: \mathbf{hash}(N_1, [T_1, v_1^\bullet : T_1''''] : [X : \mathbf{Le}(T_1''), T_1'])}$ ok.

Case (envok.mod) : The conclusion is $U(T_0) : [X : K, T], E_2, U_1(T_0') : [X : K', T'] \vdash_c^{H \cup U <: U_1}$ ok. The premises are $U(T_0) : [X : K, T], E_2, U_1(T_0') : [X : K', T'] \vdash_\bullet^H T_0' <: T_0$ and $U(T_0) : [X : K, T], E_2, U_1(T_0') : [X : K', T'] \vdash_\bullet^H K' <: K$ and $U(T_0) : [X : K, T], E_2, U_1(T_0') : [X : K', T'] \vdash_c^H$ ok. We can apply the induction hypothesis to get $: Z : K, z : \{X \leftarrow Z\}T, \sigma E_2, U_1(\sigma T_0') : [X : \sigma K', \sigma T'] \vdash_\bullet^{\sigma H} \sigma T_0' <: \sigma T_0$ and $Z : K, z : \{X \leftarrow Z\}T, \sigma E_2, U_1(\sigma T_0') : [X : \sigma K', \sigma T'] \vdash_\bullet^{\sigma H} \sigma K' <: \sigma K$ and $Z : K, z : \{X \leftarrow Z\}T, \sigma E_2, U_1(\sigma T_0') : [X : \sigma K', \sigma T'] \vdash_c^{\sigma H}$ ok with $\sigma = \{U \leftarrow h, U.\mathtt{type} \leftarrow Z, U.\mathtt{term} \leftarrow z\}$. Since we know that $\vdash h$ ok, we can apply (envok.Uhash) to get the desired result.

Case (envok.modopp) : Same case as (envok.mod) with the use of (envok.hashU) instead of (envok.Uhash).

Case (TK.mod) where $J = U.\mathtt{type} : K'$ for some K' : Then there exists T' such that the premise is $U : S, E \vdash_c^H U : [X : K', T']$. By induction, we get $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} [X : \mathbf{Eq}(Z), T] <: \sigma[X : K', T']$. By Lemma B.57 (reversing

subsignaturing judgement), we get $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} \mathbf{Eq}(Z) <: \sigma K'$. By Lemma B.8 (environments and subhashes have to be ok) and (TK.var), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} Z : K$. By Lemma B.53 (things have to be ok) and Lemma B.32 (kinds are smaller than top) and (TK.sub), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} Z : \mathbf{Le}(\top)$. By (Teq.refl), and (TK.Eq), we get $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} Z : \mathbf{Eq}(Z)$. By (TK.sub), we get $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} Z : \sigma K'$.

Case (eT.mod) where $J = U.\mathtt{term} : T'$ for some T' : Then there exists K' such that the premises are $U : S, E \vdash_c^H U : [X : K', T']$ and $U : S, E \vdash_c^H T' : \mathbf{Le}(\top)$. By applying induction, we get that $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} [X : \mathbf{Eq}(Z), T] <: \sigma[X : K', T']$. By Lemma B.57 (reversing subsignaturing judgement), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E, X : \mathbf{Eq}(Z) \vdash_c^{\sigma H} T == \sigma T'$.

By Lemma B.8 (environments and subhashes have to be ok), Lemma B.9 (prefixes of ok environments are ok) and (TK.var), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} Z : K$. By Lemma B.53 (things have to be ok), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} K$ ok. By Lemma B.40 (colour change preserves kind okedness) and Lemma B.32 (kinds are smaller than top) and (TK.sub), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_\bullet^{\sigma H} Z : \mathbf{Le}(\top)$. By (Teq.refl) and (TK.Eq), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_\bullet^{\sigma H} Z : \mathbf{Eq}(Z)$.

By Lemma B.49 (type preservation by substitution), we get $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} \{X \leftarrow Z\}T == \{X \leftarrow Z\}\{U.\mathtt{type} \leftarrow Z, U.\mathtt{term} \leftarrow z\}T'$. Since $U : S, E \vdash_c^H T' : \mathbf{Le}(\top)$, by Lemma B.11 (free variables of a judgement come from the environment), we have $X \notin \mathrm{fv}\, T'$, so we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} \{X \leftarrow Z\}T == \sigma T'$.

By Lemma B.8 (environments and subhashes have to be ok) and (eT.var), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} z : \{X \leftarrow Z\}T$. By (eT.eq), we get $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_c^{\sigma H} z : \sigma T'$ as desired.

Case $J = U : S'$ for some S' : We have $U : S, E \vdash_c^H U : S'$.

Subcase (US.var) : We have $U : S, E \vdash_c^H U : S$ (thanks to Lemma B.10 (ok environments have no repetition in the domain)). The premise is $U : S, E \vdash_c^H$ ok.

By induction and Lemma B.12 (ok environments are ok in every colour), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_\bullet^{\sigma H}$ ok.

By (TK.var) $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_\bullet^{\sigma H} Z : K$. By Lemma B.41 (relating type-is-kind and subkinding), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_\bullet^{\sigma H} \mathbf{Eq}(Z) <: K$.

On the other hand, by Lemma B.53 (things have to be ok), and (TK.sub) we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E \vdash_\bullet^{\sigma H} Z : \mathbf{Le}(\top)$.

Thus, by (Kok.Eq) and (envok.X), we have $Z : K, z : \{X \leftarrow Z\}T, \sigma E, X : \mathbf{Eq}(Z) \vdash_\bullet^{\sigma H}$ ok.

By Lemma B.8 (environments and subhashes have to be ok), Lemma B.9 (prefixes of ok environments are ok), reversing (envok.U) and Lemma B.53 (things have to be ok), we have $\vdash_\bullet^{H'} S$ ok with H' a subset of σH. By reversing (Sok), we get $X : K \vdash_\bullet^{H'} T : \mathbf{Le}(\top)$. By Lemma B.8 (environments and subhashes have to be ok), we have $X : K \vdash_\bullet^{H'}$ ok. By reversing Lemma B.36 (bindings in an ok environment are ok), we have $X : K \vdash_\bullet^{H'} K$ ok, so by (envok.X), we get $Z : K, X : K \vdash_\bullet^{H'}$ ok. By Lemma B.25 (weakening), we have $Z : K, X : K \vdash_\bullet^{H'} T : \mathbf{Le}(\top)$.

By (TK.var) and Lemma B.41 (relating type-is-kind and subkinding) as before, we have $Z : K \vdash_\bullet^{H'} \mathbf{Eq}(Z) <: K$. By Lemma B.55 (kind rewriting in environments), we have $Z : K, X : \mathbf{Eq}(Z) \vdash_\bullet^{H'} T : \mathbf{Le}(\top)$.

By Lemma B.28 (environment and subhash weakening), we can prove the judgement $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E, X : \mathbf{Eq}(Z) \vdash^{\sigma H}_{\bullet} T : \mathbf{Le}(\top)$. By (Teq.refl), we get $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E, X : \mathbf{Eq}(Z) \vdash^{\sigma H}_{\bullet} T == T$.

By (Ssub.struct), we get $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{\bullet} [X : \mathbf{Eq}(Z), T] <: [X : K, T]$.

By Lemma B.23 (colour stripping judgements), we get a proof of $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} [X : \mathbf{Eq}(Z), T] <: [X : K, T]$.

By Lemma B.11 (free variables of a judgement come from the environment), $U \notin \mathrm{fv}\, S$. So we have $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} [X : \mathbf{Eq}(Z), T] <: \sigma S$ as desired.

Subcase (US.self) : We have $U : S, E \vdash^{H}_{c} U : [X : \mathbf{Eq}(U.\mathtt{type}), T']$. There exists K' such that the premise is $U : S, E \vdash^{H}_{c} U : [X : K', T']$. By induction, we get $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} [X : \mathbf{Eq}(Z), T] <: \sigma[X : K', T']$. By Lemma B.57 (reversing subsignaturing judgement), we get $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E, X : \mathbf{Eq}(Z) \vdash^{\sigma H}_{c} T == \sigma T'$.

By Lemma B.8 (environments and subhashes have to be ok) and (TK.var), we have $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} Z : K$. By Lemma B.53 (things have to be ok) and Lemma B.32 (kinds are smaller than top), we have $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} Z : \mathbf{Le}(\top)$. By (Teq.refl), (Keq.Eq) and (Ksub.refl), we have $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} \mathbf{Eq}(Z) <: \mathbf{Eq}(Z)$.

By (Ssub.struct), we get $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} [X : \mathbf{Eq}(Z), T] <: [X : \mathbf{Eq}(Z), \sigma T']$ as desired.

Every remaining rule is inapplicable because the environment in the conclusion must be empty. □

Lemma B.59 (type preservation by module substitution in coloured judgements for type world judgements) Suppose $U(T_0) : [X : K, T], E \vdash^{H}_{c} J$ is a derivable type world judgement and J is not of the form $U : S'$ for any S' and Z is fresh and h is a hash of a module with $[X : K, T]$ as signature and T_0 as internal type and $\sigma = \{U{\leftarrow}h, U.\mathtt{type}{\leftarrow}Z\}$. Then $Z : K, \sigma E \vdash^{\sigma H}_{c} \sigma J$.

Proof. By Lemma B.58 (type preservation by module substitution in coloured judgements), for Z and z fresh, we have $Z : K, z : \{X{\leftarrow}Z\}T, E' \vdash^{H'}_{c} J'$ where $E' = \sigma' E$ and $J' = \sigma' J$ and $H' = \sigma' H$, with $\sigma' = \{U{\leftarrow}h, U.\mathtt{type}{\leftarrow}Z, U.\mathtt{term}{\leftarrow}z\}$.

Given the syntax of environments and type world judgements, z may appear free in E' or J' or H' only inside a hash. Given Lemma B.7 (hashes have to be ok), any hash in E' or J' or H' is ok, and by Lemma B.11 (free variables of a judgement come from the environment), none of these hashes has a free occurence of z. Therefore $z \notin \mathrm{fv}\, E'$ and $z \notin \mathrm{fv}\, J'$ and $z \notin \mathrm{fv}\, H'$.

Thus, with $\sigma = \{U{\leftarrow}h, U.\mathtt{type}{\leftarrow}Z\}$, we can write the judgement in question as $Z : K, z : \{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{c} \sigma J$.

Since $z \notin \mathrm{fv}\, \sigma E \cup \mathrm{fv}\, \sigma J \cup \mathrm{fv}\, \sigma H$, by Lemma B.50 (strengthening), we get $Z : K, \sigma E \vdash^{\sigma H}_{c} \sigma J$. □

Lemma B.60 (simplified module and type equality substitution for type world judgements) Suppose $U(T_0) : [X : \mathbf{Eq}(T), T'], E \vdash^{H}_{c} J$ and $\chi = U.\mathtt{type}$, or $X : \mathbf{Eq}(T), E \vdash^{H}_{c} J$ and $\chi = X$, where both are type world judgements and J is not of the form $U' : S'$. If h is a hash of a module with $[X : K, T]$ as signature and T_0 as internal type, then $\{\chi{\leftarrow}T\}E \vdash^{H'}_{c} \{\chi{\leftarrow}T\}J$, with $H' = \{U{\leftarrow}h\}H$ if $\chi = U.\mathtt{type}$, and $H' = H$ otherwise.

Proof. We consider each case.

$U(T_0) : [X : \mathbf{Eq}(T), T'], E \vdash_c^H J$ **:** By Lemma B.59 (type preservation by module substitution in coloured judgements for type world judgements), this case reduces to the other case.

$X : \mathbf{Eq}(T), E \vdash_c^H J$ **by some proof** Π **:** By Lemma B.8 (environments and subhashes have to be ok), $X : \mathbf{Eq}(T), E \vdash_c^H$ ok. By Lemma B.9 (prefixes of ok environments are ok), $X : \mathbf{Eq}(T) \vdash_c^{H_0}$ ok with $H_0 \subseteq H$. By reversing (envok.X), we have $\vdash_\bullet^{H_0} \mathbf{Eq}(T)$ ok. We reverse (Kok.Eq) to get $\vdash_\bullet^{H_0} T : \mathbf{Le}(\top)$. Then we can apply (Teq.refl) and (TK.Eq) to get $\vdash_\bullet^{H_0} T : \mathbf{Eq}(T)$. By Lemma B.49 (type preservation by substitution), $\{X{\leftarrow}T\}E \vdash_c^H \{X{\leftarrow}T\}J$, as desired.

□

Lemma B.61 (type preservation by fully carried out module substitution) If $U(T_0) : [X : \mathbf{Le}(T_0'), T_1'], E \vdash_\bullet^H J$ and J is not $U : S$ for any S and $\mathbf{nil} \vdash_\bullet^H [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1']$, then $\sigma E \vdash_\bullet^{\sigma H} \sigma J$, where $\sigma = \{U{\leftarrow}h, U.\mathtt{type}{\leftarrow}h, U.\mathtt{term}{\leftarrow}[_{(T_1 <: \{X\leftarrow h.\mathtt{type}\}T_1')} v^\bullet]_h^{\{X\leftarrow h.\mathtt{type}\}T_1'}\}$, where $h = \mathbf{hash}(N, [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1'])$, for any N.

Proof. From $\mathbf{nil} \vdash_\bullet^H [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1']$, and Lemma B.35 (components of modules are ok), we know that $X : \mathbf{Eq}(T) \vdash_\bullet^H T_1 <: T_1'$ and $X : \mathbf{Le}(T_0') \vdash_\bullet^H T_1' : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_\bullet^H T_0 : \mathbf{Le}(T_0')$ and $\mathbf{nil} \vdash_\bullet^H v^\bullet : T_1$. Also, by (hok.hash), we get $\vdash h$ ok.

From $\mathbf{nil} \vdash_\bullet^H v^\bullet : T_1$, by Lemma B.23 (colour stripping judgements), we get $\mathbf{nil} \vdash_{\{h\}}^H v^\bullet : T_1$. By Lemma B.8 (environments and subhashes have to be ok), $X : \mathbf{Eq}(T_0) \vdash_\bullet^H$ ok. By Lemma B.23 (colour stripping judgements), we get $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H$ ok. From $\mathbf{nil} \vdash_{\{h\}}^H v^\bullet : T_1$, by Lemma B.25 (weakening), we get $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H v^\bullet : T_1$.

By (Teq.hash), we also have $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H h.\mathtt{type} == T_0$ which gives, by (TK.var) and (Teq.Eq) and (Teq.tran) and (TK.Eq) $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H X : \mathbf{Eq}(h.\mathtt{type})$.

On the other hand, from $X : \mathbf{Eq}(T_0) \vdash_\bullet^H T_1 <: T_1'$, by Lemma B.23 (colour stripping judgements), we get $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H T_1 <: T_1'$. Then Lemma B.53 (things have to be ok) gives $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H T_1' : \mathbf{Le}(\top)$.

We apply Lemma B.43 (variable substitution and equivalence) to get $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H T_1' == \{X{\leftarrow}h.\mathtt{type}\}T_1'$.

Then, by (Tsub.Equi) and Lemma B.33 (transitivity of subtyping), we have a proof of $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H T_1 <: \{X{\leftarrow}h.\mathtt{type}\}T_1'$. Thus we can use (eT.sub) to get $X : \mathbf{Eq}(T_0) \vdash_{\{h\}}^H {}_{(T_1 <: \{X\leftarrow h.\mathtt{type}\}T_1')} v^\bullet : \{X{\leftarrow}h.\mathtt{type}\}T_1'$. Since X is not present in T_1 nor in $v^\bullet$, we can apply Lemma B.50 (strengthening) to get $\mathbf{nil} \vdash_{\{h\}}^H {}_{(T_1 <: \{X\leftarrow h.\mathtt{type}\}T_1')} v^\bullet : \{X{\leftarrow}h.\mathtt{type}\}T_1'$.

Now, by applying Lemma B.58 (type preservation by module substitution in coloured judgements) to $U : [X : \mathbf{Le}(T_0'), T_1'], E \vdash_\bullet^H J$ and performing alpha-conversion, we have $X : \mathbf{Le}(T_0'), x : T_1', \sigma_U E \vdash_\bullet^{\sigma_U H} \sigma_U J$, where $\sigma_U = \{U{\leftarrow}h, U.\mathtt{type}{\leftarrow}X, U.\mathtt{term}{\leftarrow}x\}$.

From the latter judgement and $\mathbf{nil} \vdash_\bullet^H h.\mathtt{type} : \mathbf{Le}(T_0')$, apply Lemma B.49 (type preservation by substitution). We get $x : \{X{\leftarrow}h.\mathtt{type}\}T_1', \{X{\leftarrow}h.\mathtt{type}\}\sigma_U E \vdash_\bullet^{\sigma_U H} \{X{\leftarrow}h.\mathtt{type}\}\sigma_U J$.

Finally, we apply Lemma B.54 (type preservation by guarded expression variable substitution) to the latter judgement combined with $\mathbf{nil} \vdash_{\{h\}}^H {}_{(T_1 <: \{X\leftarrow h.\mathtt{type}\}T_1')} v^\bullet : \{X{\leftarrow}h.\mathtt{type}\}T_1'$. If we name $\sigma' = \{x{\leftarrow}[_{(T_1 <: \{X\leftarrow h.\mathtt{type}\}T_1')} v^\bullet]_h^{\{X\leftarrow h.\mathtt{type}\}T_1'}\}$, then we obtain the desired $\sigma'\{X{\leftarrow}h.\mathtt{type}\}\sigma_U E \vdash_\bullet^{\sigma_U H} \sigma'\{X{\leftarrow}h.\mathtt{type}\}\sigma_U J$, i.e. $\sigma E \vdash_\bullet^{\sigma H} \sigma J$. □

B.7 Type decomposition

Lemma B.62 (shortening typing proof)
If $E \vdash^H_c e : T$ then there exists T' such that $E \vdash^H_c e : T'$ by a subproof that does not have (eT.eq) as the last rule used and $E \vdash^H_c T' == T$.

Proof. Induct on the structure of the derivation Π of $E \vdash^H_c e : T$.

If the proof $E \vdash^H_c e : T$ does not have an instance of (eT.eq) as the last step, then we have the desired result, given that $E \vdash^H_c T : \mathbf{Le}(\top)$ by Lemma B.53 (things have to be ok), whereupon we can apply (Teq.refl).

Otherwise there is T' such that $E \vdash^H_c e : T$ is derived from $E \vdash^H_c e : T'$ and $E \vdash^H_c T' == T$. By applying induction to the (proper) subproof Π' leading to $E \vdash^H_c e : T'$, we get that there is T'' such that $E \vdash^H_c e : T''$ by a subproof of Π' that does not have (eT.eq) as the last step used, and $E \vdash^H_c T'' == T'$. By (Teq.tran), we have $E \vdash^H_c T'' == T$, which completes our proof obligation. □

Lemma B.63 (reversing typing proof through a context)
If $\mathbf{nil} \vdash^H_{c'} CC^{c'}_c.e : T'$ then there exists T such that $\mathbf{nil} \vdash^H_c e : T$. If furthermore $\mathbf{nil} \vdash^H_c e_1 : T$ then $\mathbf{nil} \vdash^H_{c'} CC^{c'}_c.e_1 : T'$.

Proof. Induct on the structure of $CC^{c'}_c$. By Lemma B.62 (shortening typing proof), there exists T_0 such that $\mathbf{nil} \vdash^H_{c'} T_0 == T'$, and $\mathbf{nil} \vdash^H_{c'} CC^{c'}_c.e : T_0$ by a proof Π that does not end with (eT.eq).

Case $CC^{c'}_c = \_$ **:** Trivial.

Case $CC^{c'}_c = C^{c'}_{c_1}.CC^{c_1}_c$ **:** Then Π ends with an application of (eT.tuple), (eT.record), (eT.field), (eT.proj), (eT.ap), (eT.mar), (eT.marred), (eT.unmar), (eT.send) or (eT.col) (depending on $C^{c'}_{c_1}$). In any case, one premise is $\mathbf{nil} \vdash^H_{c_1} CC^{c_1}_c.e : T_1$ for some T_1. By induction we get $\mathbf{nil} \vdash^H_c e : T$ for some T.

If furthermore $\mathbf{nil} \vdash^H_c e_1 : T$, then we have $\mathbf{nil} \vdash^H_{c_1} CC^{c_1}_c.e_1 : T_1$ by induction. Each of the (eT.*) rules considered above is linear with respect to the expression metavariable instantiated by $CC^{c_1}_c.e$, with exactly one occurence above the line and one below. Instantiating this metavariable by $CC^{c_1}_c.e_1$ yields another instance of the rule. By replacing in Π the derivation leading to $\mathbf{nil} \vdash^H_{c_1} CC^{c_1}_c.e : T_1$ by that leading to $\mathbf{nil} \vdash^H_{c_1} CC^{c_1}_c.e_1 : T_1$, we get a derivation of $\mathbf{nil} \vdash^H_{c'} CC^{c'}_c.e_1 : T_0$. By (eT.eq), since $\mathbf{nil} \vdash^H_{c'} T_0 == T'$, we have $\mathbf{nil} \vdash^H_{c'} CC^{c'}_c.e_1 : T'$ as desired.

□

Definition B.64 (bare bones environment) A bare bones environment is one that contains only bindings to abstract types, i.e. of the form $X : \mathbf{Le}(T)$.

Definition B.65 (purely abstract environment) A purely abstract environment is one that contains only bindings of the form $U(T) : [X : \mathbf{Le}(T_0), T_1]$ or $X : \mathbf{Le}(T)$.

Lemma B.66 (substitutions in purely abstract environments)
If $E_0, X : \mathbf{Eq}(T), \{X \leftarrow T\}E_1 \vdash^H_c J$ with E_1 a purely abstract environment, then we have a proof of $E_0, X : \mathbf{Eq}(T), E_1 \vdash^H_c J$.

Proof. First, by Lemma B.9 (prefixes of ok environments are ok), we have $E_0, X : \mathbf{Eq}(T) \vdash^{H_0}_c$ ok with $H_0 \subseteq H$. Then, by (TK.var), we get $E_0, X : \mathbf{Eq}(T) \vdash^{H_0}_c X : \mathbf{Eq}(T)$,

which gives us $E_0, X : \mathbf{Eq}(T) \vdash_c^{H_0} X == T$. Then we can apply Lemma B.55 (kind rewriting in environments) and Lemma B.56 (signature rewriting in environments) to get the desired result. □

Lemma B.67 (equality kinding in an uncontributing environment)
If $E \vdash_c^H T : \mathbf{Eq}(T')$ and E is a bare bones environment then $E \vdash_c^H T == T'$ by a strictly smaller proof.

Note that if E was allowed to contain module bindings, we might not be able to obtain a strictly smaller proof. For example, take $T = T' = U.\texttt{type}$, with a proof whose last steps are (US.var), (US.self) and lastly (TK.mod). To prove that $E \vdash_c U.\texttt{type} == U.\texttt{type}$, we can't do any better than starting at $E \vdash_c$ ok and using (US.var), (TK.mod) and (Teq.refl). This gives an equal size proof, not a strictly smaller proof.

Proof. Induct on the structure of the proof.

Case (TK.sub) : The premises are $E \vdash_c^H T : K$ and $E \vdash_c^H K <: \mathbf{Eq}(T')$. By Lemma B.31 (discreteness of subkinding), $E \vdash_c^H K == \mathbf{Eq}(T')$ by a subproof, whence by reversing (Keq.Eq) there exists T'' such that $K = \mathbf{Eq}(T'')$ and $E \vdash_c^H T'' == T'$, the latter being derived by a proper subproof of the original proof. By induction on $E \vdash_c^H T : \mathbf{Eq}(T'')$, we get $E \vdash_c^H T == T''$ by a smaller proof. By (Teq.tran), we get $E \vdash_c^H T == T'$, by a proof that is at least one step smaller than the original proof.

Case (TK.Eq) : Trivial.

Case (TK.var) : Impossible since E only contains type variable bindings with kinds $\mathbf{Le}(T_0)$.

Case (TK.mod) : Impossible by Lemma B.11 (free variables of a judgement come from the environment), since E contains no module variable binding.

Case (TK.hash) : Impossible because hashes only have abstract kinds.

□

Lemma B.68 (variable equivalence in an uncontributing environment) If $E \vdash_c^H T == X$ (or the converse) and E is a bare bones environment then $T = X$.

Proof. Induct on the structure of the proof.

Case (Teq.Eq) : Then we know that $E \vdash_c^H T : \mathbf{Eq}(X)$ or $E \vdash_c^H X : \mathbf{Eq}(T)$. But Lemma B.67 (equality kinding in an uncontributing environment) gives us a smaller proof of $E \vdash_c^H T == X$ (or the converse). We can then apply the induction hypothesis.

Case (Teq.hash) : Then we have $E \vdash_c^H X == h.\texttt{type}$ and $X = \mathrm{impl}\,(h)$ which is impossible since h is closed.

Case (Teq.refl) : Then we have $T = X$.

Case (Teq.sym) : We trivially use the induction hypothesis.

Case (Teq.tran) : We apply the induction on the premise that mentions X. It shows that the intermediate type has to be X. We conclude with an application of the induction hypothesis on the other premise.

Cases (Teq.cong.*) : Impossible.

□

Lemma B.69 (type decomposition) Let E be a bare bones environment.

1. If $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_i)$, then $TC = TC'$ (with a possible permutation for records) and $i = j$ and for each $1 \leqslant k \leqslant j$, one of the following cases holds :
   (a) We have a stricly smaller proof of $E \vdash_c^H T_k == T'_k$
   (b) $T_k = T'_k$.
2. If $E \vdash_c^H TC(T_1, ..., T_j) == h.\texttt{type}$ (or the converse) then one of the following cases holds :
   (a) There is a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}(h)$ (or the converse) and $h \in c$.
   (b) $TC(T_1, ..., T_j) = \text{impl}(h)$ and $h \in c$.
3. If $E \vdash_c^H h.\texttt{type} == h'.\texttt{type}$, then one of the following cases holds :
   (a) There is a smaller proof of $E \vdash_c^H h.\texttt{type} == \text{impl}(h')$ and $h' \in c$.
   (b) $h.\texttt{type} = \text{impl}(h')$ and $h' \in c$.
   (c) There is a smaller proof of $E \vdash_c^H \text{impl}(h) == h'.\texttt{type}$ and $h \in c$.
   (d) $\text{impl}(h) = h'.\texttt{type}$ and $h \in c$.
   (e) $h = h'$.

Proof. E is a bare bones environment. This implies that there are not any $U.\texttt{type}$ in any of the type expressions that we consider. Moreover, by Lemma B.68 (variable equivalence in an uncontributing environment), we know that whenever we have an equivalence of the form $E \vdash_c^H TC(T_1, ..., T_j) == T$ or $E \vdash_c^H h.\texttt{type} == T$ (or symetric equivalences), then $T \neq X$. We freely use this throughout the proof.

We induct on the derivation of the equivalence hypothesis and we will now consider the last step of this proof.

Case (Teq.Eq) : The conclusion can be written $E \vdash_c^H T == T'$ and the premise is $E \vdash_c^H T : \mathbf{Eq}(T')$. Since E is a bare bones environment, we can apply the Lemma B.67 (equality kinding in an uncontributing environment) on the premise. Then we have a smaller proof of $E \vdash_c^H T == T'$. We conclude by applying the induction hypothesis.

Case (Teq.hash) : Let us examine the different cases :

1 : Impossible.

2 : The conclusion is $E \vdash_c^H TC(T_1, ..., T_j) == h.\texttt{type}$. The side-condition of this rule gives us that $TC(T_1, ..., T_j) = \text{impl}(h)$: we are in case 2b.

3 : The conclusion is $E \vdash_c^H h.\texttt{type} == h'.\texttt{type}$ with $h.\texttt{type} = \text{impl}(h')$ and $h' \in c$ or $E \vdash_c^H h'.\texttt{type} == h.\texttt{type}$ with $\text{impl}(h) = h'.\texttt{type}$; and $h \in c$. We are in cases 3b or 3d.

Case (Teq.refl) : Let us examine the different cases :

1 : The conclusion is $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_i)$. Then we know that $TC = TC'$ and $i = j$ and for all $1 \leqslant k \leqslant j$, $T_k = T'_k$ (case 1b).

2 : Impossible.

3 : The conclusion is $E \vdash_c^H h.\texttt{type} == h'.\texttt{type}$. Then we know that $h = h'$. We are in case 3e.

Case (Teq.sym) : We just apply the induction hypothesis on the premise.

Case (Teq.tran) : Let us examine the different cases :

Case 1 : The conclusion is $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_i)$ and we know that there exists a type T'' such that the premises are $E \vdash_c^H TC(T_1, ..., T_j) == T''$ and $E \vdash_c^H T'' == TC'(T'_1, ..., T'_i)$. We distinguish the possible shapes of T'', omiting the cases that are symmetric to the ones that we detail :

Case $T'' = TC''(T''_1, ..., T''_l)$: Then we can apply the induction hypothesis (case 1) on the two premises. It gives us that $TC = TC'' = TC$ and $i = l = j$. For each $1 \leqslant k \leqslant j$, we list the different cases :

Case 1a and 1a : We have for each of the premise, a strictly smaller proof of $E \vdash_c^H T_k == T''_k$ and $E \vdash_c^H T''_k == T'_k$. The we can use the rule (Teq.tran) to produce $E \vdash_c^H T_k == T'_k$ whose proof is strictly smaller than the proof of $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_i)$.

Case 1a and 1b : We have an equality $T''_k = T'_k$. Then the proof of $E \vdash_c^H T_k == T''_k$ is a proof of $E \vdash_c^H T_k == T'_k$. It is strictly smaller the original proof.

Case 1b and 1b : We simply use the transitivity of equality.

Case $T'' = h''.\texttt{type}$: Then we can apply the induction hypothesis (case 2) on the two premises. We list the different cases :

Case 2a and 2a : We have for each of the premises, a strictly smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}\,(h'')$ and $E \vdash_c^H \text{impl}\,(h'') == TC'(T'_1, ..., T'_i)$. Then we can apply (Teq.tran) to get a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_j)$. We conclude by applying the induction hypothesis in case 1.

Case 2a and 2b : We have an equality $TC'(T'_1, ..., T'_i) = \text{impl}\,(h'')$ and a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}\,(h'')$. We can use this latter as a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_i)$.We conclude by applying the induction hypothesis in case 1.

Case 2b and 2b : We simply use the transitivity of equality to get the case 1b.

Case 2 : The conclusion is $E \vdash_c^H TC(T_1, ..., T_j) == h.\texttt{type}$ (the symmetrical case is very similar) and we know that there exists a type T'' such that the premises are $E \vdash_c^H TC(T_1, ..., T_j) == T''$ and $E \vdash_c^H T'' == h.\texttt{type}$. We list the different cases :

Case $T'' = TC''(T''_1, ..., T''_l)$: We apply the induction hypothesis in the case 2 on the premise $E \vdash_c^H TC''(T''_1, ..., T''_l) == h.\texttt{type}$. There are two possibilities :

Case 2a : We have a smaller proof of $E \vdash_c^H TC''(T''_1, ..., T''_l) == \text{impl}\,(h)$ and $h \in c$. Then we can apply (Teq.tran) to get a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}\,(h)$.

Case 2b : We have $TC''(T''_1, ..., T''_l) = \text{impl}\,(h)$ and $h \in c$, and we can directly use the proof of the premise $E \vdash_c^H TC(T_1, ..., T_j) == TC''(T''_1, ..., T''_l)$ as a proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}\,(h)$. It is strictly smaller than the original proof.

Case $T'' = h''.\texttt{type}$: We apply the induction hypothesis in the case 3 on the premise $E \vdash_c^H h''.\texttt{type} == h.\texttt{type}$ and we list the different cases :

Case 3a : We have a smaller proof of $E \vdash_c^H h''.\texttt{type} == \text{impl}\,(h)$ and $h \in c$. Then we can apply (Teq.tran) to get a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}\,(h)$ with $h \in c$. It is the case 2a.

Case 3b : We know that $h''.\texttt{type} = \text{impl}(h)$ and $h \in c$. Then we just need the other premise, that is $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}(h)$ with $h \in c$. It is the case 2a.

Case 3c : We have a proof of $E \vdash_c^H \text{impl}(h'') == h.\texttt{type}$ and $h'' \in c$. We apply the induction hypothesis in the case 2 on the premise $E \vdash_c^H TC(T_1, ..., T_j) == h''.\texttt{type}$. We list the cases :

Case 2a : We have a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}(h'')$ and $h'' \in c$. Then we can apply (Teq.tran) to get a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == h.\texttt{type}$. We conclude by applying the induction hypothesis in its case 2.

Case 2b : We have $TC(T_1, ..., T_j) = \text{impl}(h'')$ and $h'' \in c$, and we can directly use the proof of $E \vdash_c^H h.\texttt{type} == \text{impl}(h'')$ as a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == h.\texttt{type}$. We conclude by applying the induction hypothesis in its case 2.

Case 3d : We know that $h.\texttt{type} = \text{impl}(h'')$ and $h'' \in c$. We apply the induction hypothesis in the case 2 on the premise $E \vdash_c^H TC(T_1, ..., T_j) == h''.\texttt{type}$. We list the cases :

Case 2a : We have a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == \text{impl}(h'')$ and $h'' \in c$. Then we can use it as a smaller proof of $E \vdash_c^H TC(T_1, ..., T_j) == h.\texttt{type}$. We conclude by applying the induction hypothesis in its case 2.

Case 2b : We have $TC(T_1, ..., T_j) = \text{impl}(h'')$ and $h'' \in c$, and then we have $TC(T_1, ..., T_j) = \text{impl}(h)$. Then we get the case 2b.

Case 3e : We know that $h = h''$. We conclude by the application of the induction hypothesis in the case 2.

Case 3 : The conclusion is $E \vdash_c^H h.\texttt{type} == h'.\texttt{type}$ and we know that there exists a type T'' such that the premises are $E \vdash_c^H h.\texttt{type} == T''$ and $E \vdash_c^H T'' == h'.\texttt{type}$. We list the different cases :

Case $T'' = TC''(T_1'', ..., T_l'')$: We apply the induction hypothesis in its case 2 on $E \vdash_c^H h.\texttt{type} == TC''(T_1'', ..., T_l'')$. We have two cases :

Case 2a : We now have a smaller proof of $E \vdash_c^H \text{impl}(h) == TC''(T_1'', ..., T_l'')$ and $h \in c$. Then we can use (Teq.tran) to get a smaller proof of $E \vdash_c^H \text{impl}(h) == h'.\texttt{type}$ and $h \in c$. It is the case 3c.

Case 2b : We have $\text{impl}(h) = TC''(T_1'', ..., T_l'')$ and $h \in c$. Then we can use the other premise $E \vdash_c^H TC''(T_1'', ..., T_l'') == h'.\texttt{type}$ as a smaller proof of $E \vdash_c^H \text{impl}(h) == h'.\texttt{type}$ and $h \in c$. It is the case 3a.

Case $T'' = h''.\texttt{type}$: We apply the induction hypothesis in its case 3 on $E \vdash_c^H h.\texttt{type} == h''.\texttt{type}$. We have five cases :

Case 3a : We now have a smaller proof of $E \vdash_c^H h.\texttt{type} == \text{impl}(h'')$ and $h'' \in c$. We apply the induction hypothesis in its case 3 on $E \vdash_c^H h''.\texttt{type} == h'.\texttt{type}$. We have five cases :

Case 3a 3a : We now have a smaller proof of $E \vdash_c^H h''.\texttt{type} == \text{impl}(h')$ and $h' \in c$. Then we can use (Teq.tran) to get a smaller proof of $E \vdash_c^H \text{impl}(h') == h.\texttt{type}$ with $h' \in c$.

Case 3a 3b : We have $\text{impl}(h') = h''.\texttt{type}$ and $h' \in c$. Then we can use the proof of $E \vdash_c^H h.\texttt{type} == h''.\texttt{type}$ as a smaller proof of $E \vdash_c^H h.\texttt{type} == \text{impl}(h')$ with $h' \in c$.

Case 3a 3c : We now have a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h'') == h'.\mathtt{type}$ and $h'' \in c$. We apply (Teq.tran) to get a smaller proof of $E \vdash_c^H h.\mathtt{type} == h'.\mathtt{type}$. We conclude by applying the induction hypothesis in its case 3.

Case 3a 3d : We have $h'.\mathtt{type} = \mathrm{impl}\,(h'')$ and $h'' \in c$. Then we take the proof of $E \vdash_c^H h.\mathtt{type} == \mathrm{impl}\,(h'')$ as a smaller proof of $E \vdash_c^H h.\mathtt{type} == h'.\mathtt{type}$. We conclude by applying the induction hypothesis in its case 3.

Case 3a 3e : We have $h'.\mathtt{type} = h''.\mathtt{type}$. Then we can use the proof of $E \vdash_c^H h''.\mathtt{type} == h.\mathtt{type}$ as a smaller proof of $E \vdash_c^H h.\mathtt{type} == h'.\mathtt{type}$. We conclude by the induction hypothesis in its case 3.

Case 3b : We have $h.\mathtt{type} = \mathrm{impl}\,(h'')$ and $h'' \in c$. We apply the induction hypothesis in its case 3 on $E \vdash_c^H h''.\mathtt{type} == h'.\mathtt{type}$. We have five cases :

Case 3b 3a : We now have a smaller proof of $E \vdash_c^H h''.\mathtt{type} == \mathrm{impl}\,(h')$ and $h' \in c$. Then we can use (Teq.tran) to get a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h') == h.\mathtt{type}$ with $h' \in c$.

Case 3b 3b : We have $\mathrm{impl}\,(h') = h''.\mathtt{type}$ and $h' \in c$. Then we can use the proof of $E \vdash_c^H h.\mathtt{type} == h''.\mathtt{type}$ as a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h') == h.\mathtt{type}$ with $h' \in c$.

Case 3b 3c : We now have a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h'') == h'.\mathtt{type}$ that we can use as a smaller proof of $E \vdash_c^H h.\mathtt{type} == h'.\mathtt{type}$. We conclude by applying the induction hypothesis in its case 3.

Case 3b 3d : We have $h'.\mathtt{type} = \mathrm{impl}\,(h'')$. Then we have $h'.\mathtt{type} = h.\mathtt{type}$. We are in case 3e.

Case 3b 3e : We have $h'.\mathtt{type} = h''.\mathtt{type}$. Then we can use the proof of $E \vdash_c^H h''.\mathtt{type} == h.\mathtt{type}$ as a smaller proof of $E \vdash_c^H h.\mathtt{type} == h'.\mathtt{type}$. We conclude by the induction hypothesis in its case 3.

Case 3c : We now have a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h) == h''.\mathtt{type}$ and $h \in c$. Then we can use (Teq.tran) to get a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h) == h'.\mathtt{type}$ with $h \in c$.

Case 3d : We have $\mathrm{impl}\,(h) = h''.\mathtt{type}$ and $h \in c$. Then we can use the proof of $E \vdash_c^H h''.\mathtt{type} == h'.\mathtt{type}$ as a smaller proof of $E \vdash_c^H \mathrm{impl}\,(h) == h'.\mathtt{type}$ with $h \in c$.

Case 3e : We have $h.\mathtt{type} = h''.\mathtt{type}$. Then we can use the proof of $E \vdash_c^H h''.\mathtt{type} == h'.\mathtt{type}$ as a smaller proof of $E \vdash_c^H h.\mathtt{type} == h'.\mathtt{type}$. We conclude by the induction hypothesis in its case 3.

Case (Teq.cong.fun) : The conclusion is of the form $E \vdash_c^H TC(T_1, ..., T_j) == TC'(T'_1, ..., T'_i)$, where $TC = TC' = \_\!\rightarrow\!\_$ and $i = j = 2$. The premises are the desired smaller proofs of $E \vdash_c^H T_k == T'_k$. The cases 2 and 3 are impossible.

Cases (Teq.cong.*) : Same as (Teq.cong.fun).

□

Lemma B.70 (decomposition of type equivalence)
If $\mathbf{nil} \vdash_c^H TC(T_1, ..., T_j) == TC(T'_1, ..., T'_j)$ then $\mathbf{nil} \vdash_c^H T_i == T'_i$ for $1 \leqslant i \leqslant j$.

Proof. Trivial consequence of part 1 of Lemma B.69 (type decomposition). □

Lemma B.71 (structural dependence of values on their types)
Suppose $\mathbf{nil} \vdash_c^H v^c : T_0$ and $\mathbf{nil} \vdash_c^H T_0 == TC(T_1, ..., T_j)$. Then one of the following cases holds.

1. There exist $h \notin c$ and $\widehat{v}^c$ and TV^c such that $\mathbf{nil} \vdash_c^H TV^c == T_0$ and $v^c = {}_{(h.\texttt{type<:}\, TV^c)}\widehat{v}^c$.
2. Consider the possible forms of TC.
   (a) If $TC = {}_{-1} \to {}_{-2}$, i.e. we have $\mathbf{nil} \vdash_c^H T_0 == T_1 \to T_2$, then there exist e and T_1' such that $v^c = \lambda x : T_1'.e$ and $\mathbf{nil} \vdash_c^H T_1' == T_1$.
   (b) If $TC = {}_{-1} * ... * {}_{-j}$, i.e. we have $\mathbf{nil} \vdash_c^H T_0 == T_1 * ... * T_j$ then there exists v_1^c, ..., v_j^c such that $v^c = (v_1^c, ..., v_j^c)$.
   (c) If $TC = \{l_1 : {}_{-1}, ..., l_j : {}_{-j}\}$, i.e. we have $\mathbf{nil} \vdash_c^H T_0 == \{l_1 : T_1; ...; l_j : T_j\}$, then there exists v_1^c, ..., v_j^c such that $v^c = \{l_1 = v_1^c; ...; l_j = v_j^c\}$.
   (d) If $TC = \text{unit}$, i.e. we have $\mathbf{nil} \vdash_c^H T_0 == \text{unit}$ then $v^c = ()$.
   (e) If $TC = \text{bytes}$, i.e. we have $\mathbf{nil} \vdash_c^H T_0 == \text{bytes}$ then there exist e and T and H' and c' such that $\mathbf{nil} \vdash_\bullet^{H'} T : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_{c'}^{H'} e : T$ and $v^c = \mathbf{marshalled}_{\,c',H'}(e : T)$.

Proof. Induct on the size of the derivation of $\mathbf{nil} \vdash_c^H v^c : T_0$.

Consider the last step of the derivation of $\mathbf{nil} \vdash_c^H v^c : T_0$. The rules not mentionned here cannot have been used because v^c is a value.

Case (eT.sub) : By Lemma B.69 (type decomposition), we know that T_0 being a $h.\texttt{type}$ for some h with $h \notin c$ is absurd. Then there are $h \notin c$ and $\widehat{v}^c$ and TV^c such that $v^c = {}_{(h.\texttt{type<:}\, TV^c)}\widehat{v}^c$. We proved the case 1 and we have $\mathbf{nil} \vdash_c^H TV^c == TC(T_1, ..., T_j)$.

Case (eT.eq) : There is T_0' such that $\mathbf{nil} \vdash_c^H T_0' == T_0$, and $\mathbf{nil} \vdash_c^H v^c : T_0'$ by a smaller proof compared with $\mathbf{nil} \vdash_c^H v^c : T_0$. Using (Teq.tran), we get $\mathbf{nil} \vdash_c^H T_0' == TC(T_1, ..., T_j)$. By induction we get the desired result.

Case (eT.fun) : There exist e, T_1' and T_2' such that $v^c = \lambda x : T_1'.e$ and $T_0 = T_1' \to T_2'$. By Lemma B.69 (type decomposition), we have proved case 2a, and we have $\mathbf{nil} \vdash_c^H T_1 == T_1'$.

Case (eT.unit) : Then $v^c = ()$ and $T_0 = \text{unit}$. By Lemma B.69 (type decomposition), we have proved case 2d.

Case (eT.tuple) : There exist $T_1', ..., T_j'$, $v_1^c, ..., v_j^c$ such that $v^c = (v_1^c, ..., v_j^c)$ and $T_0 = T_1' * ... * T_j'$. By Lemma B.69 (type decomposition), we have proved case 2b.

Case (eT.record) : There exist $T_1', ..., T_j'$, $v_1^c, ..., v_j^c$ such that $v^c = \{l_1 = v_1^c; ...; l_j = v_j^c\}$ and $T_0 = \{l_1 : T_1'; ...; l_j : T_j'\}$. By Lemma B.69 (type decomposition), we have proved case 2c.

Case (eT.marred) : There exist e and T and H' and c' such that $v^c = \mathbf{marshalled}_{\,c',H'}(e : T)$ and $T_0 = \text{bytes}$. By Lemma B.69 (type decomposition), we have proved case 2e. Two premises of (eT.marred) are $\mathbf{nil} \vdash_\bullet^{H'} T : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_{c'}^{H'} e : T$.

Case (eT.col) : Given that v^c is a value that is a bracket expression, T_0 can be but a hash $h_0.\texttt{type}$. By Lemma B.69 (type decomposition) in its case 2, we must have $h_0 \in c$. However $T_0 = h.\texttt{type}$ is impossible as per the definition of values.

□

Lemma B.72 (signature unicity) If $E \vdash_c^H U : [X : K, T]$ and $E \vdash_c^H U : [X : K', T']$ then $E \vdash_c^H T == T'$.

Proof. We induct on the structure of $E \vdash_c^H U : [X : K, T]$ and $E \vdash_c^H U : [X : K', T']$.

Case (US.eq) : If one of the proofs ends by (US.eq), we inverse (Seq.struct) and conclude by induction and (Teq.tran).

Case (US.self) : If one of the proofs ends by (US.self), we conclude by induction.

Case (US.var) : If both proofs end by (US.var), then $T = T'$. We conclude by Lemma B.53 (things have to be ok) and (Teq.refl).

□

Lemma B.73 (type unicity) If $E \vdash_c^H e : T_0$ and $E \vdash_c^H e : T_1$ then $E \vdash_c^H T_0 == T_1$.

Proof. We induct on the structure of $E \vdash_c^H e : T_0$ and $E \vdash_c^H e : T_1$.

If one of the proofs ends by (eT.eq), we conclude by induction and (Teq.tran). In all other cases, the typing rules are syntax-directed, which means that $E \vdash_c^H e : T_0$ and $E \vdash_c^H e : T_1$ end by the same typing rule. We list the different cases.

Case (eT.var) : From Lemma B.8 (environments and subhashes have to be ok), we have $E \vdash_c^H$ ok. By Lemma B.10 (ok environments have no repetition in the domain) and (eT.var), we know that $T_0 = T_1$. We conclude by Lemma B.53 (things have to be ok) and (Teq.refl).

Case (eT.mod) : We use Lemma B.72 (signature unicity).

Cases (eT.proj), (eT.field), (eT.ap) : We apply the induction hypothesis on the premises and conclude by Lemma B.69 (type decomposition).

Cases (eT.fun), (eT.tuple), (eT.record) : We apply repeatedly the induction hypothesis on the premises and conclude by the appropriate (Teq.cong.*).

Other rules : In these cases, $T_0 = T_1$. We conclude by Lemma B.53 (things have to be ok) and (Teq.refl).

□

Lemma B.74 (triviality of type equivalence in a trivial environment) If $\mathbf{nil} \vdash_\bullet^H T == T'$ then $T = T'$.

Proof. Induct on the structure of T.

Case T is of the form $TC(T_1, ..., T_j)$: By Lemma B.69 (type decomposition) and the trivial colour of the hypothesis, there exist $T'_1, ..., T'_j$ such that $\mathbf{nil} \vdash_\bullet^H T_i == T'_i$ for $1 \leqslant i \leqslant j$ and $T' = TC(T'_1, ..., T'_j)$. (We're possibly making use of (Teq.sym) here.) By induction on $\mathbf{nil} \vdash_\bullet^H T_i == T'_i$ for $1 \leqslant i \leqslant j$ we have $T_i = T'_i$ hence $T = T'$ as desired.

Case T is of the form $h.\texttt{type}$: By Lemma B.69 (type decomposition) and the trivial colour of the hypothesis, $T = T'$.

Case T is of the form $U.\texttt{type}$ or X : Impossible by Lemma B.11 (free variables of a judgement come from the environment) since the environment is empty.

□

Definition B.75 (skeletal constructor) A type T is said to have a skeletal constructor TC in the following cases :

- There exist $T_1, ..., T_i$ such that $T = TC(T_1, ..., T_i)$.
- There exist h such that $T = h.\texttt{type}$ and $\text{impl}\,(h)$ has a skeletal constructor TC.
- There exist $T_1, ..., T_i$ such that $T = TC(T_1, ..., T_i)$.

Note that this definition is recursive in the second case : we may have to traverse a chain of hashes before reaching the skeletal constructor.

We also remark that a type variable is the only correct type expression not to have a skeletal constructor and, conversely, that the types that do not contain any free substitutable entities always have a skeletal constructor.

Definition B.76 (common skeletal constructor) T_1 and T_2 are said to *share the same skeletal constructor* if they have the same skeletal constructor and, in the case of records, the record fields of T_2 are included in those of T_1, up to permutation.

Note that this relation is transitive but not symmetric.

Lemma B.77 (skeletal constructors in type equivalence) Let E be a bare bones environment. If $E \vdash_c^H T_0 == T_1$ and T_0 and T_1 have a skeletal constructor, then T_0 and T_1 share the same skeletal constructor.

Proof. By induction on the structure of T_0 and T_1. In each case, we use the Lemma B.69 (type decomposition) to get directly that T_0 and T_1 share the same constructor, or we have an equivalence proof between some smaller types that share the same constructors as T_0 and T_1. In the latter case we conclude by induction. □

Definition B.78 (related by subtyping) Two types T_0 and T_1 are said to be *related by subtyping* (written $T_0 \lhd T_1$) if there exist a bare bones environment E, a colour c, a subhash relation H such that we have a proof of one of the following judgements :

- $E \vdash_c^H T_0 <: T_1$
- $E \vdash_c^H T_0 : \mathbf{Le}(T_1)$
- $E \vdash_c^H \mathbf{Le}(T_0) <: \mathbf{Le}(T_1)$
- $E \vdash_c^H \mathbf{Eq}(T_0) <: \mathbf{Le}(T_1)$

Note that, when the environment E, colour c or subhash relation H are known, the fact that T_0 and T_1 are related by subtyping can be written $T_0 \lhd_{E,c,H} T_1$.

Lemma B.79 (essence of subtyping) If $T_0 \lhd T_1$ in an empty environment, then T_0 and T_1 share the same skeletal constructor.

Proof. Suppose T_0 and T_1 are related by subtyping, then we have a proof of $\vdash_c^H T_0 <: T_1$ or $\vdash_c^H T_0 : \mathbf{Le}(T_1)$ or $\vdash_c^H \mathbf{Le}(T_0) <: \mathbf{Le}(T_1)$ or $\vdash_c^H \mathbf{Eq}(T_0) <: \mathbf{Le}(T_1)$ for some colour c and subhash relation H.

Since the environment is empty, we know by Lemma B.11 (free variables of a judgement come from the environment) that T_0 and T_1 do not contain any substitutable entities.

We induct on this derivation : we have different cases depending on the last rule used.

Case (Ksub.Eq) : Then $T_0 = T_1$ and therefore share the same skeletal constructor.

Case (Ksub.Le) : The premise is $\vdash_c^H T_0 <: T_1$. We apply the induction hypothesis. Then we know that T_0 and T_1 share the same skeletal constructor.

Case (Ksub.refl) : The premise is $\vdash_c^H T_0 == T_1$. We apply the Lemma B.77 (skeletal constructors in type equivalence).

Case (Ksub.tran) : We have a new type T_2 which is related by subtyping to T_0 and T_1. By induction, we know that T_0 and T_2 share the same skeletal constructor and T_2 and T_1 also share the same skeletal constructor. Then we can deduce that T_0 and T_1 share the same skeletal constructor.

Case (TK.sub) : The two premises are $\vdash_c^H T_0 : K$ and $\vdash_c^H K <: \mathbf{Le}(T_1)$. If we have a type T_2 such that $K = \mathbf{Le}(T_2)$, then T_2 is related by subtyping to T_0 and T_1. By induction, we know that T_0 and T_2 share the same skeletal constructor and T_2 and T_1 share the same skeletal constructor. Then we can deduce that T_0 and T_1 share the same skeletal constructor. If we have a type T_2 such that $K = \mathbf{Eq}(T_2)$, then by Lemma B.67 (equality kinding in an uncontributing environment), we have a subproof of $\vdash_c^H T_0 == T_2$, and we can apply the Lemma B.77 (skeletal constructors in type equivalence). We induct on the other premise, and we can conclude that T_0 and T_1 share the same skeletal constructor.

Case (TK.Le) : The premise is $\vdash_c^H T_0 <: T_1$. We apply the induction hypothesis to know that T_0 and T_1 share the same skeletal constructor.

Case (TK.hash) : One of the premises is $\vdash h\,\mathrm{ok}$ with $h = \mathbf{hash}(N, [T_2, v^\bullet : T'] : [X : \mathbf{Le}(T_1) : T''])$ and by reversing (hok.hash) and (MS.struct), we have a subproof of $\mathbf{nil} \vdash_\bullet^{H'} T_2 : \mathbf{Le}(T_1)$. By induction, we know that T_1 and T_2 share the same skeletal constructor. Since $T_2 = \mathrm{impl}\,(h)$ we know that $T_0 = h.\mathtt{type}$ and T_2 share the same skeletal constructor. We conclude that T_0 and T_1 share the same skeletal constructor.

Case (Tsub.Le) : The premise is $\vdash_c^H T_0 : \mathbf{Le}(T_1)$. We apply the induction hypothesis to conclude that T_0 and T_1 share the same skeletal constructor.

Case (Tsub.Equi) : The premise is $\vdash_c^H T_0 == T_1$. We apply the Lemma B.77 (skeletal constructors in type equivalence).

Case (Tsub.Subhash) : $T_0 = h_0.\mathtt{type}$ and $T_1 = h_1.\mathtt{type}$. The premise is $\vdash_c^H$ ok and we know that $h_0.\mathtt{type} <: h_1.\mathtt{type} \in H$. By Lemma B.9 (prefixes of ok environments are ok), we know that there is a subproof of $\mathbf{nil} \vdash_c^H$ ok. Then we can repeatedly reverse (envok.hashhash) and apply the Lemma B.8 (environments and subhashes have to be ok) until we reach the rule introducing $h_0.\mathtt{type} <: h_1.\mathtt{type}$. One of the premises is then $\mathbf{nil} \vdash_\bullet^{H'} \mathrm{impl}\,(h_0) <: \mathrm{impl}\,(h_1)$. By induction, we get that $\mathrm{impl}\,(h_0)$ and $\mathrm{impl}\,(h_1)$ share the same skeletal constructor. Then we deduce that T_0 and T_1 share the same skeletal constructor.

Case (Tsub.cong.record.width) : T_0 and T_1 share the same skeletal constructor.

Case (Tsub.cong.fun) : T_0 and T_1 share the same skeletal constructor.

Case (Tsub.cong.tuple) : T_0 and T_1 share the same skeletal constructor.

□

Lemma B.80 (variable supertype in an uncontributing environment) If $T \lhd X$ in a bare bones environment then $T = X$.

Proof. Induct on the structure of the proof.

Case (Ksub.Eq) : Then $T = X$.

Case (Ksub.Le) : The premise is $E \vdash_c^H T <: X$. We apply the induction hypothesis.

Case (Ksub.refl) : The premise is $E \vdash_c^H T == X$. We apply the Lemma B.68 (variable equivalence in an uncontributing environment).

Case (Ksub.tran) : We have a new type T' such that $E \vdash_c^H T' <: X$. By induction, we know that $T' = X$ and we can apply the induction hypothesis to the other premise.

Case (TK.sub) : Similar to the (Ksub.tran) case.

Case (TK.Le) : The premise is $E \vdash_c^H T <: X$. We apply the induction hypothesis.

Case (TK.hash) : From Lemma B.11 (free variables of a judgement come from the environment), the supertype of $h.\texttt{type}$ does not contain any free variable. This case is thus impossible.

Case (Tsub.Le) : The premise is $E \vdash_c^H T : \mathbf{Le}(X)$. We apply the induction hypothesis.

Case (Tsub.Equi) : The premise is $E \vdash_c^H T == X$. We apply the Lemma B.68 (variable equivalence in an uncontributing environment).

Case (Tsub.Subhash) : Impossible.

Case (Tsub.cong.record.width) : Impossible.

Case (Tsub.cong.fun) : Impossible.

Case (Tsub.cong.tuple) : Impossible.

□

Definition B.81 (healthy proof environment) A *healthy proof environment* restricts the proofs to the ones where, whenever a judgement $E \vdash_c^H J$ is derived from (envok.*) or (TK.hash), then the immediately above (hok.hash) only introduces an arbitrary subhash relation H' such that $H' \subseteq H$.

An healthy proof environment expresses the fact that the subhash relations appearing in the (hok.hash) rule are not completely arbitrary.

Definition B.82 (simple subtyping proof) A *simple proof* is a proof where no (Teq.hash) rule appear in a position that is not above a ok judgement or not above a (Tsub.cong.*) or (Teq.cong.*).

Lemma B.83 (simplicity preservation) The application of the Lemma B.23 (colour stripping judgements) and Lemma B.24 (subhash weakening) preserve simplicity.

Proof. The proof tree transformation corresponding to Lemma B.23 (colour stripping judgements) and Lemma B.24 (subhash weakening) only modify the structure of the proof above ok judgements. □

Definition B.84 (really simple subtyping proof) A *really simple proof* is a proof where no (Teq.hash) rule appear in a position that is not above a (Tsub.cong.*) or (Teq.cong.*) or a reflexivity rule and such that the sub-proof leading to the premises of the (Teq.hash) rules are also really simple proofs.

Definition B.85 (underlying implementation) Given a hash c, we write $\text{impl}^*(h)$ the type corresponding to the result of the repeated application of the $\text{impl}()$ operator.

Lemma B.86 (subtyping simplification) In an healthy proof environment, let T and T' be related by subtyping in an empty environment, the colour c and the subhash relation H. Depending on the structure of T and T' we have the following properties :

- If $T = TC(T_1, ..., T_i)$ and $T' = TC(T'_1, ..., T'_i)$, then we have a simple proof of $\vdash_c^H TC(T_1, ..., T_i) <: TC(T'_1, ..., T'_i)$.
- If $T = TC(T_1, ..., T_i)$ and $T' = h.\texttt{type}$, then we have a simple proof of $\vdash_c^H TC(T_1, ..., T_i) <: \text{impl}^*(h)$.
- If $T = h.\texttt{type}$ and $T' = TC(T_1, ..., T_i)$, then we have a simple proof of $\vdash_c^H \text{impl}^*(h) <: TC(T_1, ..., T_i)$.
- If $T = h.\texttt{type}$ and $T' = h'.\texttt{type}$, then we have a simple proof of $\vdash_c^H \text{impl}^*(h) <: \text{impl}^*(h')$.

Proof. Since T and T' are related by subtyping, we have a proof of $\vdash_c^H T <: T'$ (or any of its alternatives).

Since the environment is empty, we know by Lemma B.11 (free variables of a judgement come from the environment) that T and T' do not contain any substitutable entities.

The first case is when $T = TC(T_1, ..., T_i)$ and $T' = TC(T'_1, ..., T'_i)$. We have different cases depending on the last rule used.

Case (Ksub.Eq) : In this case $T = T'$. The proof is simple.

Case (Ksub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Ksub.refl) : The premise is $\vdash_c^H T == T'$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Ksub.tran) : We have a new type T'' such that T is related by subtyping to T'' and T'' is related to T'. By induction, we know that we have simple proofs of $\vdash_c^H T <: \text{impl}^*(T'')$ and $\vdash_c^H \text{impl}^*(T'') <: T$. We conclude using transitivity.

Case (TK.sub) : The two premises are $\vdash_c^H T : K$ and $E \vdash_c^H K <: \mathbf{Le}(T')$. If we have a type T'' such that $K = \mathbf{Le}(T'')$, then this case is similar to the previous one. If we have a type T'' such that $K = \mathbf{Eq}(T'')$, then by Lemma B.67 (equality kinding in an uncontributing environment), we have a subproof of $\vdash_c^H T == T''$, and we can apply repeatedly the Lemma B.70 (decomposition of type equivalence) to get a simple proof of equivalence and then of subtyping by (Tsub.Equi). We conclude by (TK.sub).

Case (TK.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (TK.hash) : Impossible.

Case (TK.var) : Impossible.

Case (Tsub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Tsub.Equi) : The premise is $\vdash_c^H T_0 == T_1$. We proceed as in the (Ksub.refl) case.

Case (Tsub.Subhash) : Impossible.

Case (Tsub.cong.*) : Trivial.

The second case is when $T = TC(T_1, ..., T_i)$ and $T' = h.\texttt{type}$. We have different cases depending on the last rule used.

Case (Ksub.Eq) : Impossible.

Case (Ksub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Ksub.refl) : The premise is $\vdash_c^H T == T'$. We apply repeatedly the Lemma B.69 (type decomposition) to get a proof of $\vdash_c^H T == \text{impl}^*(h)$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Ksub.tran) : We have a new type T'' such that T is related by subtyping to T'' and T'' is related to T'. By induction, we know that we have simple proofs of $\vdash_c^H T <: \text{impl}^*(T'')$ and $\vdash_c^H \text{impl}^*(T'') <: T$. We conclude using transitivity.

Case (TK.sub) : The two premises are $\vdash_c^H T : K$ and $E \vdash_c^H K <: \mathbf{Le}(T')$. If we have a type T'' such that $K = \mathbf{Le}(T'')$, then this case is similar to the previous one. If we have a type T'' such that $K = \mathbf{Eq}(T'')$, then by Lemma B.67 (equality kinding in an uncontributing environment), we have a subproof of $\vdash_c^H T == T''$. and we can apply repeatedly the Lemma B.70 (decomposition of type equivalence) to get a simple proof of equivalence and then of subtyping by (Tsub.Equi). We conclude by (TK.sub).

Case (TK.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (TK.hash) : Impossible.

Case (TK.var) : Impossible.

Case (Tsub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Tsub.Equi) : The premise is $\vdash_c^H T == T'$.We apply repeatedly the Lemma B.69 (type decomposition) to get a proof of $\vdash_c^H T == \text{impl}^*(h)$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Tsub.Subhash) : Impossible.

Case (Tsub.cong.*) : Impossible.

The third case is when $T = h.\texttt{type}$ and $T' = TC(T_1, ..., T_i)$. We have different cases depending on the last rule used.

Case (Ksub.Eq) : Impossible.

Case (Ksub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Ksub.refl) : The premise is $\vdash_c^H T == T'$. We apply repeatedly the Lemma B.69 (type decomposition) to get a proof of $\vdash_c^H T == \text{impl}^*(h)$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Ksub.tran) : We have a new type T'' such that T is related by subtyping to T'' and T'' is related to T'.By induction, we know that we have simple proofs of $\vdash_c^H T <: \text{impl}^*(T'')$ and $\vdash_c^H \text{impl}^*(T'') <: T$. We conclude using transitivity.

Case (TK.sub) : The two premises are $\vdash_c^H T : K$ and $E \vdash_c^H K <: \mathbf{Le}(T')$. If we have a type T'' such that $K = \mathbf{Le}(T'')$, then this case is similar to the previous one. If we have a type T'' such that $K = \mathbf{Eq}(T'')$, then by Lemma B.67 (equality kinding in an uncontributing environment), we have a subproof of $\vdash_c^H T == T''$. and we can apply repeatedly the Lemma B.70 (decomposition of type equivalence) to get a simple proof of equivalence and then of subtyping by (Tsub.Equi). We conclude by (TK.sub).

Case (TK.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (TK.hash) : One of the premises is $\vdash h\,\text{ok}$ with $h = \mathbf{hash}(N, [\text{impl}(h), v^\bullet : T''] : [X : \mathbf{Le}(T'), T_0''])$ and by reversing (hok.hash) and (MS.struct), we have a subproof of $\vdash_\bullet^{H'} \text{impl}(h) : \mathbf{Le}(T')$ with $H' \subseteq H$. By induction we get a simple proof of $\vdash_\bullet^{H'} \text{impl}^*(h) <: T'$. By Lemma B.24 (subhash weakening) and Lemma B.23 (colour stripping judgements) and Lemma B.83 (simplicity preservation), we get a simple proof of $\vdash_c^H \text{impl}^*(h) <: T'$.

Case (TK.var) : Impossible.

Case (Tsub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Tsub.Equi) : The premise is $\vdash_c^H T == T'$.We apply repeatedly the Lemma B.69 (type decomposition) to get a proof of $\vdash_c^H T == \text{impl}^*(h)$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Tsub.Subhash) : Impossible.

Case (Tsub.cong.*) : Impossible.

The forth case is when $T = h.\texttt{type}$ and $T' = h'.\texttt{type}$. We have different cases depending on the last rule used.

Case (Ksub.Eq) : Impossible.

Case (Ksub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Ksub.refl) : The premise is $\vdash_c^H T == T'$. We apply repeatedly the Lemma B.69 (type decomposition) to get a proof of $\vdash_c^H \text{impl}^*(h) == \text{impl}^*(h')$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Ksub.tran) : We have a new type T'' such that T is related by subtyping to T'' and T'' is related to T'. By induction, we know that we have simple proofs of $\vdash_c^H T <: \text{impl}^*(T'')$ and $\vdash_c^H \text{impl}^*(T'') <: T$. We conclude using transitivity.

Case (TK.sub) : The two premises are $\vdash_c^H T : K$ and $E \vdash_c^H K <: \mathbf{Le}(T')$. If we have a type T'' such that $K = \mathbf{Le}(T'')$, then this case is similar to the previous one. If we have a type T'' such that $K = \mathbf{Eq}(T'')$, then by Lemma B.67 (equality kinding in an uncontributing environment), we have a subproof of $\vdash_c^H T == T''$. and we can apply repeatedly the Lemma B.70 (decomposition of type equivalence) to get a simple proof of equivalence and then of subtyping by (Tsub.Equi). We conclude by (TK.sub).

Case (TK.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (TK.hash) : One of the premises is $\vdash h\,\text{ok}$ with $h = \mathbf{hash}(N, [\text{impl}(h), v^\bullet : T''] : [X : \mathbf{Le}(T'), T_0''])$ and by reversing (hok.hash) and (MS.struct), we have a subproof of $\vdash_\bullet^{H'} \text{impl}(h) : \mathbf{Le}(T')$ with $H' \subseteq H$. By induction we get a simple proof of $\vdash_\bullet^{H'} \text{impl}^*(h) <: \text{impl}^*(h')$. By Lemma B.24 (subhash weakening) and Lemma B.23 (colour stripping judgements) and Lemma B.83 (simplicity preservation), we get a simple proof of $\vdash_c^H \text{impl}^*(h) <: \text{impl}^*(h')$.

Case (TK.var) : Impossible.

Case (Tsub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Tsub.Equi) : The premise is $\vdash_c^H T == T'$.We apply repeatedly the Lemma B.69 (type decomposition) to get a proof of $\vdash_c^H \text{impl}^*(h) == \text{impl}^*(h')$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Tsub.Subhash) : $T = h.\texttt{type}$ and $T' = h'.\texttt{type}$. The premise is $\vdash_c^H$ ok and we know that $h <: h' \in H$. By Lemma B.9 (prefixes of ok environments are ok), we know that there is a subproof of $\vdash_c^H$ ok. Then we can repeatedly reverse (envok.hashhash) and apply the Lemma B.8 (environments and subhashes have to be ok) until we reach the rule introducing $h <: h'$. One of the premises is then $\vdash_\bullet^{H'} \text{impl}(h) <: \text{impl}(h')$ with $H' \subseteq H$. By induction, we get a simple proof of $\vdash_\bullet^{H'} \text{impl}^*(h) <: \text{impl}^*(h')$. By Lemma B.24 (subhash weakening) and Lemma B.23 (colour stripping judgements) and Lemma B.83 (simplicity preservation), we get a simple proof of $\vdash_c^H \text{impl}^*(h) <: \text{impl}^*(h')$.

Case (Tsub.cong.*) : Impossible.

□

Lemma B.87 (absurd simplicity) There exist no simple proof of a judgement of the form $\vdash_c^H TC(T_1, ..., T_i) == h.\texttt{type}$ or $\vdash_c^H TC(T_1, ..., T_i) <: h.\texttt{type}$.

Proof. Suppose we take the smallest simple proof of a judgement of the form $\vdash_c^H TC(T_1, ..., T_i) == h.\texttt{type}$ (or any of its alternatives). We will prove that this supposition yields a contradiction.

Case (Teq.Eq) : The premise contradicts the fact that the original proof was the smallest.

Case (TK.Eq) : The premise contradicts the fact that the original proof was the smallest.

Case (Teq.hash) : Contradiction since the proof is simple.

Case (Teq.refl) : Impossible.

Case (Teq.sym) : The premise contradicts the fact that the original proof was the smallest.

Case (Teq.tran) and (TK.sub) : One of the premises contradicts the fact that the original proof was the smallest.

Cases (Teq.cong.*) : Impossible.

We have now proved that there exist no simple proof of a judgement of the form $\vdash_c^H TC(T_1, ..., T_i) == h.\texttt{type}$.

Suppose now that we take the smallest simple proof of a judgement of the form $\vdash_c^H TC(T_1, ..., T_i) <: h.\texttt{type}$ (or any of its alternatives). We will prove that this supposition yields a contradiction.

Case (Ksub.Eq) : Impossible.

Case (Ksub.Le) : The premise is $\vdash_c^H TC(T_1, ..., T_i) <: h.\texttt{type}$ and contradicts the fact that the original proof was the smallest.

Case (Ksub.refl) : The premise is $\vdash_c^H TC(T_1, ..., T_i) == h.\texttt{type}$. We get a contradiction from the first part of the lemma.

Case (Ksub.tran) : One of the premises contradicts the fact the the original proof was the smallest.

Case (TK.sub) : One of the premises contradicts the first part of the lemma of the fact that the original proof was the smallest.

Case (TK.Le) : The premise is $\vdash_c^H TC(T_1, ..., T_i) <: h.\texttt{type}$ and contradicts the fact that the original proof was the smallest.

Case (TK.hash) : Impossible.

Case (Tsub.Le) : The premise is $\vdash_c^H TC(T_1, ..., T_i) : \mathbf{Le}(h.\texttt{type})$ and contradicts the fact that the original proof was the smallest.

Case (Tsub.Equi) : The premise is $\vdash_c^H TC(T_1, ..., T_i) == h.\texttt{type}$. We get a contradiction from the first part of the lemma.

Case (Tsub.Subhash) : Impossible.

Case (Tsub.cong.*) : Impossible.

□

Lemma B.88 (subtyping decomposition) In an healthy proof environment, if $T = TC(T_1, ..., T_i)$ and $T' = TC(T'_1, ..., T'_i)$ are related by subtyping in an empty environment, the colour c and the subhash relation H, using a simple proof, then there is a proof of $\vdash_c^H TC(T_1, ..., T_i) <: TC(T'_1, ..., T'_i)$ ending by a congruence rule.

Proof. By induction on the derivation of $\vdash_c^H T <: T'$ (or any of its alternatives). We have different cases depending on the last rule used.

Case (Ksub.Eq) : Trivial.

Case (Ksub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Ksub.refl) : The premise is $\vdash_c^H T == T'$. Then by Lemma B.70 (decomposition of type equivalence), we can reorganize the proof to end by a congruence rule. The result is then a simple proof.

Case (Ksub.tran) : We have a new type T'' such that T is related by subtyping to T'' and T'' is related to T'. By Lemma B.87 (absurd simplicity), $T'' = TC(T''_1, ..., T''_i)$. We can use the induction hypothesis on both sides and then permute the transitivity rule with the congruence rules to get the desired result.

Case (TK.sub) : Case similar to the two previous ones.

Case (TK.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (TK.hash) : Impossible.

Case (Tsub.Le) : The premise is $\vdash_c^H T <: T'$. We apply the induction hypothesis.

Case (Tsub.Equi) : The premise is $\vdash_c^H T == T$. We proceed as in the (Ksub.refl) case.

Case (Tsub.Subhash) : Impossible.

Case (Tsub.cong.*) : Trivial.

□

Lemma B.89 (decomposition of subtyping) In an healthy proof environment, if $T = TC(T_1, ..., T_i)$ and $T' = TC(T'_1, ..., T'_i)$ are related by subtyping in the colour c and the subhash relation H, then there is a proof of $\vdash_c^H TC(T_1, ..., T_i) <: TC(T'_1, ..., T'_i)$ ending by a congruence rule.

Proof. Application of Lemma B.86 (subtyping simplification) and Lemma B.88 (subtyping decomposition). □

B.8 Decidability of type checking

B.8.1 Type equivalence

Definition B.90 (revelation of the implementation of a hash) The type $\text{reveal}_c T$ is obtained by replacing any subterm of T that is equal to a hash type $h.\texttt{type}$ where $h \in c$ by (the revelation of) the implementation type $\text{impl}(h)$. Thus :

- $\text{reveal}_c \text{unit} = \text{unit}$
- $\text{reveal}_c \top = \top$
- $\text{reveal}_c(T_1 * ... * T_j) = (\text{reveal}_c T_1) * ... * (\text{reveal}_c T_j)$
- $\text{reveal}_c\{l_1 : T_1; ...; l_j : T_j\} = \{l_1 : \text{reveal}_c T_1; ...; l_j : \text{reveal}_c T_j\}$
- $\text{reveal}_c(T_1 \to T_2) = (\text{reveal}_c T_1) \to (\text{reveal}_c T_2)$
- $\text{reveal}_c X = X$
- $\text{reveal}_c \text{bytes} = \text{bytes}$
- $\text{reveal}_c U.\texttt{type} = U.\texttt{type}$
- $\text{reveal}_c h.\texttt{type} = \text{reveal}_c(\text{impl}(h))$ if $h \in c$
- $\text{reveal}_c h'.\texttt{type} = h'.\texttt{type}$ if $h' \notin c$

Note that this definition relies on the fact that $\text{impl}(h)$ is a subterm of h.

Definition B.91 (partial type substitution associated to an environment)

$$\begin{aligned}
\text{partenvsub}_{\mathbf{nil}} &= \text{id} \\
\text{partenvsub}_{E,x:T} &= \text{partenvsub}_E \\
\text{partenvsub}_{E,X:\mathbf{Le}(T)} &= \text{partenvsub}_E \\
\text{partenvsub}_{E,X:\mathbf{Eq}(T)} &= \text{partenvsub}_E\{X \leftarrow T\} \\
\text{partenvsub}_{E,U:[X:\mathbf{Le}(T),T']} &= \text{partenvsub}_E \\
\text{partenvsub}_{E,U:[X:\mathbf{Eq}(T),T']} &= \text{partenvsub}_E\{U.\texttt{type} \leftarrow T\}
\end{aligned}$$

We will sometimes use the abbreviation σ_c^E to refer to the operation $\text{reveal}_c\text{partenvsub}_E$ and σ_c to refer to reveal_c.

Recall Definition B.65 (purely abstract environment).

Lemma B.92 (a purely abstract suffix does not change the substitution) If E_1 is a purely abstract environment then $\text{partenvsub}_{E_0,E_1} = \text{partenvsub}_{E_0}$.

Proof. Trivial from Definition B.91 (partial type substitution associated to an environment). □

Lemma B.93 (stability of types through revelation) If $E \vdash_c^H T : \mathbf{Le}(\top)$ then $E \vdash_c^H \text{reveal}_c T : \mathbf{Le}(\top)$ and $E \vdash_c^H T == \text{reveal}_c T$ and $E \vdash_c^H \text{partenvsub}_E T : \mathbf{Le}(\top)$ and $E \vdash_c^H T == \text{partenvsub}_E T$.

A trivial consequence that we also use is that if $E \vdash_c^H T : \mathbf{Le}(\top)$ then $E \vdash_c^H \text{reveal}_c\text{partenvsub}_E T : \mathbf{Le}(\top)$ and $E \vdash_c^H T == \text{reveal}_c\text{partenvsub}_E T$. Also, we freely use (Teq.sym) on the conclusion of this lemma.

Proof. We will first prove that $E \vdash_c^H T == \text{reveal}_c T$ by induction on the structure of T.

Case T is unit, bytes, $\top$, variable X or there exists U such that $T = U.\texttt{type}$: Then $\text{reveal}_c T = T$. We conclude by (Teq.refl).

Case there exist $T_1, ..., T_j$ such that $T = T_1 * ... * T_j$: By reversing (Tsub.tuple), we have the correctness of each of the T_i. Then we can apply the induction hypothesis to get $E \vdash_c^H T_i == \text{reveal}_c T_i$ for $1 \leqslant i \leqslant j$. Then we conclude by (Teq.cong.tuple).

Case there exist $T_1, ..., T_j$ such that $T = l_1 : T_1; ...; l_j : T_j$: By reversing (Tsub.rec), we have the correctness of each of the T_i. Then we can apply the induction hypothesis to get $E \vdash_c^H T_i == \text{reveal}_c T_i$ for $1 \leqslant i \leqslant j$. Then we conclude by (Teq.cong.rec).

Case there exist T_1, T_2 such that $T = T_1 \to T_2$: By reversing (Tsub.fun), we have the correctness of each of the T_i. Then we can apply the induction hypothesis to get $E \vdash_c^H T_i == \text{reveal}_c T_i$ for $i = 1, 2$. Then we conclude by (Teq.cong.fun).

Case T is a hash $h.\texttt{type}$: If $h \in c$, then $\text{reveal}_c h.\texttt{type} = \text{reveal}_c(\text{impl}(h))$. Since $\text{impl}(h)$ is a subterm of $h.\texttt{type}$, we can apply the induction hypothesis to get $E \vdash_c^H \text{impl}(h) == \text{reveal}_c(\text{impl}(h))$. In parallel, we can apply (Teq.hash) to get $E \vdash_c^H h.\texttt{type} == \text{impl}(h)$. We conclude by (Teq.tran). If $h \notin c$, then $\text{reveal}_c(h'.\texttt{type}) = h'.\texttt{type}$ and we conclude by (Teq.refl).

We get $E \vdash_c^H \text{reveal}_c T : \mathbf{Le}(\top)$ by Lemma B.53 (things have to be ok).

We will now prove that $E \vdash_c^H T == \text{partenvsub}_E T$ by generalizing to $E \vdash_c^H T == \text{partenvsub}_{E'} T$ and inducting on E'. By Lemma B.8 (environments and subhashes have to be ok) we know that $E \vdash_c^H$ ok.

Case $E = \mathbf{nil}$: Then $\text{partenvsub}_E T = T$. We conclude by (Teq.refl).

Case $E = E_0, x : T$ or $E = E_0, X : \mathbf{Le}(T')$ or $E = E_0, U(T_0) : [X : \mathbf{Le}(T'), T'']$: Then we know that $\text{partenvsub}_E = \text{partenvsub}_{E_0}$. We conclude by induction.

Case $E = E_0, X : \mathbf{Eq}(T')$: By induction we know that $E \vdash_c^H T == \text{partenvsub}_{E_0} T$ and from Lemma B.53 (things have to be ok) that $E \vdash_c^H \text{partenvsub}_{E_0} T : \mathbf{Le}(\top)$. On the other hand we have a proof of $E \vdash_c^H X : \mathbf{Eq}(T')$. Then we can apply Lemma B.43 (variable substitution and equivalence) to get $E \vdash_c^H \text{partenvsub}_{E_0} T == \text{partenvsub}_E T$. We conclude by (Teq.tran).

Case $E = E_0, U(T_0) : [X : \mathbf{Eq}(T'), T'']$ **:** By induction we know that $E \vdash_c^H T == \text{partenvsub}_{E_0} T$ and from Lemma B.53 (things have to be ok) that $E \vdash_c^H \text{partenvsub}_{E_0} T : \mathbf{Le}(\top)$. On the other hand we have a proof of $E \vdash_c^H U.\texttt{type} : \mathbf{Eq}(T')$. Then we can apply Lemma B.43 (variable substitution and equivalence) to get $E \vdash_c^H \text{partenvsub}_{E_0} T == \text{partenvsub}_E T$. We conclude by (Teq.tran).

Case $E = E_0, X : \mathbf{Le}(T)$ **:** Then we know that $\text{partenvsub}_E = \text{partenvsub}_{E_0}$.

□

Lemma B.94 (interpretation of type equivalence)
Let E be an bare bones environment. Then we have the following equivalence.
$E \vdash_c^H T == T'$ iff $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$ and $\text{reveal}_c T = \text{reveal}_c T'$.

This gives us an algorithm to test the type equivalence of two types.

Proof. Assume $E \vdash_c^H T == T'$. By Lemma B.53 (things have to be ok), we have $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c T' : \mathbf{Le}(\top)$. We will now prove the last statement by induction on T and T'.

Case $T = X$ **:** By Lemma B.68 (variable equivalence in an uncontributing environment), we have $T' = X = T$. By this same lemma, we cover the case where $T' = X$. We thus assume in the following case analysis that $T \neq X$ and $T' \neq X$.

Case $T = U.\texttt{type}$ **:** Impossible by Lemma B.11 (free variables of a judgement come from the environment).

Case $T = TC(T_1, ..., T_j)$ **:** By Lemma B.77 (skeletal constructors in type equivalence), T' has the same constructor. Then we distinguish between the different cases and apply the Lemma B.69 (type decomposition) :

Case $T' = TC(T'_1, ..., T'_j)$ **:** Then we have $E \vdash_c^H T'_i == T_i$ for each i and, by induction, for each i, we have $\text{reveal}_c T'_i = \text{reveal}_c T_i$. Then we have $\text{reveal}_c T' = \text{reveal}_c T$.

Case $T' = h'.\texttt{type} = \mathbf{hash}(N, [TC(T'_1, ..., T'_j), v : T_0] : [X : K, T'']) \in c$ **:** Then $\text{reveal}_c T' = TC(\text{reveal}_c T'_1, ..., \text{reveal}_c T'_j) = \text{reveal}_c(TC(T'_1, ..., T'_j))$. Since we know that $E \vdash_c^H T'_i == T_i$, we can apply the induction hypothesis to get $\text{reveal}_c(TC(T'_1, ..., T'_j)) = \text{reveal}_c T$.

Case $T = h.\texttt{type}$ **:** By Lemma B.69 (type decomposition), one of the following cases holds :

Case $h = \mathbf{hash}(N, [T'', v : T_0] : S) \in c$ **:** Then we have $E \vdash_c^H T'' == T'$ and by induction $\text{reveal}_c T'' = \text{reveal}_c T'$. Since we know that $\text{reveal}_c T = \text{reveal}_c T''$, we have $\text{reveal}_c T = \text{reveal}_c T'$. In the case where $T'' = T'$, we have also $\text{reveal}_c T = \text{reveal}_c T'' = \text{reveal}_c T'$.

Case $T' = c = \mathbf{hash}(N, [h_1, v : T_0] : [X : K, T'']) \in c$ **:** Then, as in the previous case, by induction, we have $\text{reveal}_c T = \text{reveal}_c T'$.

Case $T' = h.\texttt{type}$ **:** Trivial.

We turn to the other half of the proof. Assume that $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$ and $\text{reveal}_c T = \text{reveal}_c T'$. We induct on the structure of T and T'.

Case $T = X$ **:** Then $\text{reveal}_c T = T = T' = \text{reveal}_c T'$. By (Teq.refl), we have $E \vdash_c^H T == T'$.

Case $T = U.\texttt{type}$ **:** Impossible by Lemma B.11 (free variables of a judgement come from the environment).

Case $T = h.\texttt{type}$ **where** $h \notin c$ **:** Then $\text{reveal}_c T = T$. By definition of reveal_c, one of the following cases holds :

Case $T' = h'.\texttt{type}$ **with** $h' = \textbf{hash}(N, [T'', v] : [X : \textbf{Le}(\top), T'']) \in c$ **:** We have $E \vdash_c^H$ ok by Lemma B.8 (environments and subhashes have to be ok). Then, by (Teq.hash), we get $E \vdash_c^H T' == T$, whence $E \vdash_c^H T == T'$ by (Teq.sym).

Case $T' = T$ **:** Then $E \vdash_c^H T == T'$ by (Teq.refl).

Case $T = TC(T_1, ..., T_j)$ **:** By definition of reveal_c, one of the following cases holds :

Case $T' =$ **unit or** $T' =$ **mar :** Then $T = T'$, so $E \vdash_c^H T == T'$ by (Teq.refl).

Case $T' = TC(T'_1, ..., T'_j)$ **with** $j \neq 0$ **:** Still by the definition of reveal_c, we have $\text{reveal}_c T_i = \text{reveal}_c T'_i$ for all i. Then by induction we have $E \vdash_c^H T_i == T'_i$. By the appropriate rule amongst (Teq.cong.fun) and (Teq.cong.tuple) and (Teq.cong.rec), we have $E \vdash_c^H T == T'$.

Case $T' = h.\texttt{type}$ **with** $h = \textbf{hash}(N, [T'', v] : S) \in c$ **:** By definition of reveal_c, we have $\text{reveal}_c T = \text{reveal}_c T' = \text{reveal}_c T''$. Then by induction we have $E \vdash_c^H T == T''$. We conclude by (Teq.tran) and (Teq.hash) to get $E \vdash_c^H T == T'$.

Case $T = h.\texttt{type}$ **with** $h \in c$ **:** By definition of reveal_c, one of the following cases holds :

Case $T' = h.\texttt{type}$ **:** Then $E \vdash_c^H T == T'$ by (Teq.refl).

Case $T' = h'.\texttt{type}$ **with** $h' = \textbf{hash}(N, [T'', v] : S)$ **isin hm :** By definition of reveal_c, we have $\text{reveal}_c T = \text{reveal}_c T' = \text{reveal}_c T''$. Then by induction we have $E \vdash_c^H T == T''$. We conclude by (Teq.tran) and (Teq.hash) to get $E \vdash_c^H T == T'$.

□

B.8.2 Subtyping test algorithm

Given two types T and T' and a subhash relation H, we want to be able to know if there exists a proof of $\vdash_\bullet^H T <: T'$.

Definition B.95 (subtyping algorithm) We distinguish the different forms of T_0 and T'_0. Since the environment is empty, we know that T_0 and T'_0 are constructed types (i.e. $TC(T_1, ..., T_j)$) or hash types (i.e. $h.\texttt{type}$).

Case $TC(T_1, ..., T_i)$**,**$TC'(T'_1, ..., T'_j)$ **:** We test whether the constructors are equal. If it is the case : we induct on the subterms according to their variance.

Case $TC(T_1, ..., T_j)$**,**$h'.\texttt{type}$ **:** We find all the $h''.\texttt{type}$ such that $h'' <: h' \in H$: we recurse on the arguments $TC(T_1, ..., T_j)$ and $\sigma_c^E h''.\texttt{type}$.

Case $h.\texttt{type}$**,**$TC(T'_1, ..., T'_j)$ **:** We find all the $h''.\texttt{type}$ such that $h <: h'' \in H$ and the type T''_0 such that $h.\texttt{type} = \textbf{hash}(N, M : [X : \textbf{Le}(T''_0), ...])$: we recurse on these arguments in their normalized form and $TC(T'_1, ..., T'_j)$.

Case $h.\texttt{type}$**,**$h'.\texttt{type}$ **:** We find all the $h''.\texttt{type}$ such that $h <: h'' \in H$ and the type T''_0 such that $h.\texttt{type} = \textbf{hash}(N, M : [X : \textbf{Le}(T''_0), ...])$: we recurse on these arguments in their normalized form and $h'.\texttt{type}$.

Conjecture B.96 (correction of the subtyping algorithm) If we have $E \vdash_c^H T : \textbf{Le}(\top)$ and $E \vdash_c^H T' : \textbf{Le}(\top)$, and the subtyping algorithm returns true, then there exists a proof of $E \vdash_c^H T <: T'$.

Conjecture B.97 (completeness of the subtyping algorithm) If we have $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$, and there exists a proof of $E \vdash_c^H T <: T'$, then the subtyping algorithm will return true.

Conjecture B.98 (temination of the subtyping algorithm) If we have $E \vdash_c^H T : \mathbf{Le}(\top)$ and $E \vdash_c^H T' : \mathbf{Le}(\top)$, then the subtyping algorithm will terminate.

B.9 Type preservation by reduction

The following theorems are valid only in healthy proof environments.

Theorem B.99 (type preservation for expression reduction)
If $\mathbf{nil} \vdash_c^H e : T$ and $H, e \rightarrow_c H', e'$ then $\mathbf{nil} \vdash_c^{H'} e' : T$.

Proof. Note that by Lemma B.62 (shortening typing proof), there exists T' such that $\mathbf{nil} \vdash_c^H T' == T$ and $\mathbf{nil} \vdash_c^H e : T'$ by a proof that does not end in (eT.eq).

In the discussion below, we will often make use of the fact that apart from (eT.eq), typing of expressions is syntax-directed. Also, note that by Lemma B.53 (things have to be ok), we have $\mathbf{nil} \vdash_c^H T' : \mathbf{Le}(\top)$, and by Lemma B.8 (environments and subhashes have to be ok), we have $\mathbf{nil} \vdash_c^H$ ok.

We induct on the derivation of the reduction. In most cases we prove $\mathbf{nil} \vdash_c^H e' : T'$ which we can complete by (eT.eq) to get the desired $\mathbf{nil} \vdash_c^{H'} e' : T$.

Case (ered.proj) : Let $e = \mathbf{proj}_i\,(v_1^c, ..., v_j^c)$ and $e' = v_i^c$. Then $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.proj), and the ith premise is $\mathbf{nil} \vdash_c^H v_i^c : T'$.

Case(ered.field) : Let $e = \{l_1 = v_1^c; ...; l_j = v_j^c\}.l_i$ and $e' = v_i^c$. Then $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.field), and the ith premise is $\mathbf{nil} \vdash_c^H v_i^c : T'$.

Case (ered.ap) : Let $e = (\lambda x : T_0.e_0)\, v^c$ and $e' = \{x \leftarrow v^c\}e_0$. $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.ap), with the premises $\mathbf{nil} \vdash_c^H v^c : T_0$ and $\mathbf{nil} \vdash_c^H (\lambda x : T_0.e_0) : T_0 \rightarrow T'$.

By Lemma B.62 (shortening typing proof), there exists T_0' such that $\mathbf{nil} \vdash_c^H (\lambda x : T_0.e) : T_0'$ by a proof that does not end in (eT.eq), and $\mathbf{nil} \vdash_c^H T_0' == T_0 \rightarrow T'$. By reversing (eT.fun), there exists T_1 such that $T_0' = T_0 \rightarrow T_1$ and $x : T_0 \vdash_c^H e : T_1$. By Lemma B.70 (decomposition of type equivalence), we have $\mathbf{nil} \vdash_c^H T_1 == T'$.

By Lemma B.51 (type preservation by expression substitution), we have $\mathbf{nil} \vdash_c^H \{x \leftarrow v^c\}e : T_1$. By (eT.eq), we get $\mathbf{nil} \vdash_c^H e' : T'$.

Case (ered.mar) : Let $e = \mathbf{mar}\,(v^c : T_0)$ and $e' = \mathbf{marshalled}_{c,H}(v^c : T_0)$. $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.mar), with the premise $\mathbf{nil} \vdash_c^H v^c : T'$; also we know that $T' = \text{bytes}$.

By Lemma B.53 (things have to be ok) we know that $\mathbf{nil} \vdash_c^H T_0 : \mathbf{Le}(\top)$. By Lemma B.39 (colour change preserves type okedness) we get $\mathbf{nil} \vdash_\bullet^H T_0 : \mathbf{Le}(\top)$. By (eT.marred), we get $\mathbf{nil} \vdash_c^H e' : \text{bytes}$. By (eT.eq), we get $\mathbf{nil} \vdash_c^H e' : T'$.

Case (ered.unmar) : Let $e = \mathbf{unmar}\,(\mathbf{marshalled}_{c_0,H_0}(e_0 : T_0)) : T_0'$. By reversing (eT.unmar), we get $\mathbf{nil} \vdash_c^H T_0' : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_c^H \mathbf{marshalled}_{c_0,H_0}(e_0 : T) : \text{bytes}$. We also learn that $T_0' = T'$. By reversing (eT.marred) we have $\mathbf{nil} \vdash_c^H$ ok and $\mathbf{nil} \vdash_{c_0}^{H_0} e_0 : T_0$ and $\mathbf{nil} \vdash_\bullet^{H_0} T_0 : \mathbf{Le}(\top)$. There are two possible outcomes.

Case e' is $\mathbf{Unmarfailure}^{T_0'}$: By Lemma B.53 (things have to be ok), $\mathbf{nil} \vdash_c^H T_0' : \mathbf{Le}(\top)$, hence $\mathbf{nil} \vdash_c^H \mathbf{Unmarfailure}^{T_0'} : T_0'$ by (eT.Undynfailure).

Case e' is the value $_{(T_0 <: T_0')}[e_0]_{c_0}^{T_0}$: From $\mathbf{nil} \vdash_\bullet^{H_0} T_0 : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_{c_0}^{H_0} e_0 : T_0$, we can use (eT.col) to get $\mathbf{nil} \vdash_\bullet^{H_0} [e_0]_{c_0}^{T_0} : T_0$. By Lemma B.23 (colour stripping judgements), this implies that $\mathbf{nil} \vdash_c^{H_0} [e_0]_{c_0}^{T_0} : T_0$. Since we know that $\mathbf{nil} \vdash_\bullet^{H \cup H_0} T_0 <: T_0'$, by Lemma B.23 (colour stripping judgements) and Lemma B.24 (subhash weakening), we can apply (eT.sub) to get $\mathbf{nil} \vdash_c^{H \cup H_0} {}_{(T_0 <: T_0')}[e_0]_{c_0}^{T_0} : T_0'$.

Case (ered.sub.sub) : Let $e = {}_{(T_0 <: T_1)}({}_{(TV_2^c <: TV_3^c)}\widehat{\widehat{v}}^c)$ and $e' = {}_{(TV_2^c <: T_1)}\widehat{\widehat{v}}^c$. $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.sub). Then the premises are $\vdash_c^H T_0 <: T_1$ and $\vdash_c^H {}_{(TV_2^c <: TV_3^c)}\widehat{v}^c : T_0$. We also learn that $T' = T_1$. By Lemma B.62 (shortening typing proof), and by reversing (eT.sub), we get $\vdash_c^H TV_3^c == T_0$ and $\vdash_c^H {}_{(TV_2^c <: TV_3^c)}\widehat{\widehat{v}}^c : TV_3^c$ and $\vdash_c^H TV_2^c <: TV_3^c$ and $\vdash_c^H \widehat{\widehat{v}}^c : TV_2^c$. By Lemma B.33 (transitivity of subtyping) and (Tsub.Equi), we have $\vdash_c^H TV_2^c <: T_1$. By (eT.sub), we get $\vdash_c^H {}_{(TV_2^c <: T_1)}\widehat{\widehat{v}}^c : T_1$.

Case (ered.sub.typeright) : Let $e = {}_{(T_0 <: h_0.\mathtt{type})}\widehat{\widehat{v}}^c$ and $e' = {}_{(T_0 <: T_1)}\widehat{\widehat{v}}^c$ with $h_0 \in c$ and $\mathrm{impl}(h_0) = T_1$.We then learn that $T' = h_0.\mathtt{type}$ and $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash_c^H T_0 <: h_0.\mathtt{type}$ and $\mathbf{nil} \vdash_c^H \widehat{\widehat{v}}^c : T_0$. From Lemma B.8 (environments and subhashes have to be ok) and since $h_0 \in c$ and $\mathrm{impl}(h_0) = T_1$, we know that $\mathbf{nil} \vdash_c^H h_0.\mathtt{type} == T_1$ by (Teq.hash). By (Tsub.Equi) and the Lemma B.33 (transitivity of subtyping), we get $\mathbf{nil} \vdash_c^H T_0 <: T_1$. By (eT.sub), we know that $\mathbf{nil} \vdash_c^H {}_{(T_0 <: T_1)}\widehat{\widehat{v}}^c : T_1$. We conclude by (eT.eq).

Case (ered.sub.typeleft) : Let $e = {}_{(h_0.\mathtt{type} <: TV^c)}\widehat{\widehat{v}}^c$ and $e' = {}_{(T_0 <: TV^c)}\widehat{\widehat{v}}^c$ with $h_0 \in c$ and $\mathrm{impl}(h_0) = T_0$. We then learn that $T' = TV^c$ and $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.sub). The premises are $\mathbf{nil} \vdash_c^H h_0.\mathtt{type} <: TV^c$ and $\mathbf{nil} \vdash_c^H \widehat{\widehat{v}}^c : h_0.\mathtt{type}$. From Lemma B.8 (environments and subhashes have to be ok) and since $h_0 \in c$ and $\mathrm{impl}(h_0) = T_0$, then we know that $\mathbf{nil} \vdash_c^H h_0.\mathtt{type} == T_0$ by (Teq.hash). By (Tsub.Equi) and the Lemma B.33 (transitivity of subtyping), we get $\mathbf{nil} \vdash_c^H T_0 <: TV^c$. By (eT.eq) we have a proof of $\mathbf{nil} \vdash_c^H \widehat{\widehat{v}}^c : T_0$ which leads to know that $\mathbf{nil} \vdash_c^H {}_{(T_0 <: TV^c)}\widehat{\widehat{v}}^c : TV^c$ by (eT.sub).

Case (ered.sub.tuple) : Let $e = {}_{((T_1 * ... * T_j) <: (T_1' * ... * T_j'))}(v_1^c, ..., v_j^c)$ and its reduction $e' = ({}_{(T_1 <: T_1')}v_1^c, ..., {}_{(T_j <: T_j')}v_j^c)$.We then learn that $T' = (T_1' * ... * T_j')$ and $\mathbf{nil} \vdash_c^H e : T'$ must have been derived by (eT.sub). The premises are $\mathbf{nil} \vdash_c^H (v_1^c, ..., v_j^c) : (T_1 * ... * T_j)$ and $\mathbf{nil} \vdash_c^H T_1 * ... * T_j <: T_1' * ... * T_j'$. By theLemma B.89 (decomposition of subtyping) we know that for $1 \leqslant i \leqslant j$, $\vdash_c^H T_i <: T_i'$. We are now able to use (eT.sub) to prove, for $1 \leqslant i \leqslant j$, $\vdash_c^H {}_{(T_i <: T_i')}v_i^c : T_i'$. We then use (eT.tuple) to prove the desired $\vdash_c^H ({}_{(T_1 <: T_1')}v_1^c, ..., {}_{(T_j <: T_j')}v_j^c) : (T_1' * ... * T_j')$.

Case (ered.sub.record) : $\vdash_c^H {}_{(\{l_1 : T_1'; ...; l_j : T_j'\} <: \{l_1 : T_1; ...; l_i : T_i\})}\{l_1 = v_1^c; ...; l_j = v_j^c\} : \{l_1 : T_1; ...; l_i : T_i\}$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash_c^H \{l_1 = v_1^c; ...; l_j = v_j^c\} : \{l_1 : T_1; ...; l_j : T_j\}$ and $\vdash_c^H \{l_1 : T_1'; ...; l_j : T_j'\} <: \{l_1 : T_1; ...; l_i : T_i\}$. As we have only width subtyping, we know that, for $1 \leqslant k \leqslant i \leqslant j$, $\vdash_c^H T_k' == T_k$ by Lemma B.89 (decomposition of subtyping). By Lemma B.62 (shortening typing proof) we have $\mathbf{nil} \vdash_c^H \{l_1 = v_1^c; ...; l_j = v_j^c\} : \{l_1 : T_1''; ...; l_j : T_j''\}$ whose last rule is (eT.record) and $\mathbf{nil} \vdash_c^H \{l_1 : T_1'; ...; l_j : T_j'\} == \{l_1 : T_1''; ...; l_j : T_j''\}$. The premises are then, for $1 \leqslant k \leqslant j$, $\vdash_c^H v_k^c : T_k''$. By Lemma B.70 (decomposition of type equivalence) we know that for $1 \leqslant k \leqslant j$, $\vdash_c^H T_k' == T_k''$. We can now prove

that, for $1 \leqslant k \leqslant j$, $\vdash_c^H v_k^c : T_k$ using twice (eT.eq). By (eT.record) we now have $\vdash_c^H \{l_1 = v_1^c; ...; l_i = v_i^c\} : \{l_1 : T_1; ...; l_i : T_i\}$.

Case (ered.sub.fun) : $\vdash_c^H {}_{((T_1 \to T_2) <: (T_1' \to T_2'))}(\lambda x : T_0.e) : T_1' \to T_2'$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash_c^H (\lambda x : T_0.e) : T_1 \to T_2$ and $\mathbf{nil} \vdash_c^H T_1 \to T_2 <: T_1' \to T_2'$. By Lemma B.89 (decomposition of subtyping) we know that $\mathbf{nil} \vdash_c^H T_2 <: T_2'$ and $\mathbf{nil} \vdash_c^H T_1' <: T_1$. By Lemma B.62 (shortening typing proof) we have $\mathbf{nil} \vdash_c^H (\lambda x : T_0.e) : T_0 \to T_2''$ whose last rule is (eT.fun) and $\mathbf{nil} \vdash_c^H T_0 \to T_2'' == T_1 \to T_2$ (on which we can use the Lemma B.69 (type decomposition)). The premise is then $x : T_0 \vdash_c^H e : T_2''$. By Lemma B.8 (environments and subhashes have to be ok) and Lemma B.53 (things have to be ok) and Lemma B.39 (colour change preserves type okedness), we know that $\mathbf{nil} \vdash_c^H$ ok and $\mathbf{nil} \vdash_\bullet^H T_1' : \mathbf{Le}(\top)$. Then by (envok.x), we have $y : T_1' \vdash_c^H$ ok. First, we need to get $y : T_1' \vdash_c^H {}_{(T_1' <: T_1)} y : T_1$ from (eT.var) and (eT.sub) using Lemma B.25 (weakening). We deduce $y : T_1' \vdash_c^H {}_{(T_1' <: T_1)} y : T_0$ by (eT.eq). Then, by Lemma B.27 (combined weakening) we get $y : T_1', x : T_0 \vdash_c^H e : T_2''$. Finally, we use Lemma B.51 (type preservation by expression substitution) with substitution $\sigma = \{x \leftarrow {}_{(T_1' <: T_1)} y\}$ to get $y : T_1' \vdash_c^H \{x \leftarrow {}_{(T_1' <: T_1)} y\} e : T_2''$. We conclude by (eT.eq) and (eT.fun).

Case (ered.sub.marshalled) : $\vdash_c^H {}_{(\mathrm{bytes} <: \mathrm{bytes})} \mathbf{marshalled}_{c', H'}(e : T) : \mathrm{bytes}$ must have been derived by (eT.sub). One of the premises is then the desired $\vdash_c^H \mathbf{marshalled}_{c', H'}(e : T) : \mathrm{bytes}$.

Case (ered.sub.unit) : $\vdash_c^H {}_{(\mathrm{unit} <: \mathrm{unit})}() : \mathrm{unit}$ must have been derived by (eT.sub). One of the premises is then the desired $\vdash_c^H () : \mathrm{unit}$.

Case (ered.sub.col) : $\vdash_c^H {}_{(T' <: T'')}([\widehat{v}^{c' \cup c}]_{c'}^{h}.\mathtt{type}) : T''$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash_c^H [\widehat{v}^{c' \cup c}]_{c'}^{h.\mathtt{type}} : T'$ and $\mathbf{nil} \vdash_c^H T' <: T''$. Then by Lemma B.62 (shortening typing proof) we know that $\mathbf{nil} \vdash_c^H h.\mathtt{type} == T'$ and $\mathbf{nil} \vdash_c^H [\widehat{v}^{c' \cup c}]_{c'}^{h.\mathtt{type}} : h.\mathtt{type}$ whose last rule is (eT.col). One of the premises is then $\mathbf{nil} \vdash_{c \cup c'}^H \widehat{v}^{c' \cup c} : h.\mathtt{type}$. By applying twice Lemma B.23 (colour stripping judgements), (eT.eq) and (eT.sub) we get $\mathbf{nil} \vdash_{c \cup c'}^H {}_{(T' <: T'')} \widehat{v}^{c' \cup c} : T''$. Then, (eT.col) gives us $\vdash_c^H [{}_{(T' <: T'')} \widehat{v}^{c' \cup c}]_{c'}^{T''} : T''$.

Case (ered.col.col) : We know that $\mathbf{nil} \vdash_c^H [[\widehat{v}^{c \cup c_0 \cup c_1}]_{c_0}^{h_0.\mathtt{type}}]_{c_1}^{h_1.\mathtt{type}} : h_1.\mathtt{type}$. It must have been derived by (eT.col) from $\mathbf{nil} \vdash_c^H h_1.\mathtt{type} : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_{c \cup c_1}^H [\widehat{v}^{c \cup c_0 \cup c_1}]_{c_0}^{h_0.\mathtt{type}} : h_1.\mathtt{type}$. Then by Lemma B.62 (shortening typing proof) we know that $\mathbf{nil} \vdash_{c \cup c_1}^H h_0.\mathtt{type} == h_1.\mathtt{type}$ and that $\mathbf{nil} \vdash_{c \cup c_1}^H [\widehat{v}^{c \cup c_0 \cup c_1}]_{c_0}^{h_0.\mathtt{type}} : h_0.\mathtt{type}$ whose last rule is (eT.col). Then one of the premises is $\mathbf{nil} \vdash_{c \cup c_1 \cup c_0}^H \widehat{v}^{c \cup c_0 \cup c_1} : h_0.\mathtt{type}$. From $\mathbf{nil} \vdash_{c \cup c_1}^H h_0.\mathtt{type} == h_1.\mathtt{type}$, by Lemma B.23 (colour stripping judgements), we get $\mathbf{nil} \vdash_{c \cup c_0 \cup c_1}^H h_0.\mathtt{type} == h_1.\mathtt{type}$. Then we can use (eT.eq) to have $\mathbf{nil} \vdash_{c \cup c_1 \cup c_0}^H \widehat{v}^{c \cup c_0 \cup c_1} : h_1.\mathtt{type}$. Then we can derive $\mathbf{nil} \vdash_c [\widehat{v}^{c \cup c_0 \cup c_1}]_{c_0 \cup c_1}^{h_1.\mathtt{type}} : h_1.\mathtt{type}$.

Case (ered.col.type) : We know that $\vdash_c^H [\widehat{v}^{c' \cup c}]_{c'}^{h.\mathtt{type}} : h.\mathtt{type}$. It must have been derived by (eT.col) from $\vdash_{c \cup c'}^H \widehat{v}^{c' \cup c} : h.\mathtt{type}$. Since $h \in c \subseteq c' \cup c$ and $\mathrm{impl}(h) = T$, we can use (Teq.hash) and (eT.eq) to prove $\vdash_{c \cup c'}^H \widehat{v}^{c' \cup c} : T$. Then we use (eT.col) to get $\vdash_c^H [\widehat{v}^{c' \cup c}]_{c'}^{T} : T$. We conclude by (eT.eq).

Case (ered.col.tuple) : $\vdash_c^H [(\widehat{v}_1^{c' \cup c}, ..., \widehat{v}_j^{c' \cup c})]_{c'}^{T_1 * ... * T_j} : T_1 * ... * T_j$ must have been derived by (eT.col). One of the premises is then $\vdash_{c' \cup c}^H (\widehat{v}_1^{c' \cup c}, ..., \widehat{v}_j^{c' \cup c}) : T_1 * ... * T_j$. By Lemma B.62 (shortening typing proof) we have a proof of $\vdash_{c' \cup c}^H (\widehat{v}_1^{c' \cup c}, ..., \widehat{v}_j^{c' \cup c}) : T_1' * ... * T_j'$ whose last rule is (eT.tuple) and a proof of $\vdash_{c' \cup c}^H T_1' * ... * T_j' == T_1 * ... * T_j$. By

Lemma B.70 (decomposition of type equivalence) we know that for $1 \leqslant k \leqslant j$, $\vdash^H_{c'\cup c} T'_k == T''_k$. The premises of (eT.tuple) are, for $1 \leqslant k \leqslant j$, $\vdash^H_{c'\cup c} \widehat{v}_k^{c'\cup c} : T'_k$. Then we can use (eT.eq), (eT.col) and (eT.tuple) to get $\vdash^H_c ([\widehat{v}_1^{c'\cup c}]^{T_1}_{c'}, ..., [\widehat{v}_j^{c'\cup c}]^{T_j}_{c'}) : T_1 * ... * T_j$.

Case (ered.col.record) : $\vdash^H_c [\{l_1 = \widehat{v}_1^{c'\cup c}, .., l_j = \widehat{v}_j^{c'\cup c}\}]^{\{}_{c'} l_1 : T_1, ..., l_j : T_j\} : \{l_1 : T_1, ..., l_j : T_j\}$ must have been derived by (eT.col). One of the premises is then $\vdash^H_{c'\cup c} \{l_1 = \widehat{v}_1^{c'\cup c}, .., l_j = \widehat{v}_j^{c'\cup c}\} : \{l_1 : T_1, ..., l_j : T_j\}$. By Lemma B.62 (shortening typing proof) we have a proof of $\vdash^H_{c'\cup c} \{l_1 = \widehat{v}_1^{c'\cup c}, .., l_j = \widehat{v}_j^{c'\cup c}\} : \{l_1 : T'_1, ..., l_j : T'_j\}$ whose last rule is (eT.record), and a proof of $\vdash^H_{c'\cup c} \{l_1 : T'_1, ..., l_j : T'_j\} == \{l_1 : T_1, ..., l_j : T_j\}$. By Lemma B.70 (decomposition of type equivalence) we know that for $1 \leqslant k \leqslant j$, $\vdash^H_{c'\cup c} T'_k == T''_k$. The premises of (eT.record) are, for $1 \leqslant k \leqslant j$, $\vdash^H_{c'\cup c} \widehat{v}_k^{c'\cup c} : T'_k$.Then we can use (eT.eq), (eT.col) and (eT.record) to get $\vdash^H_c \{l_1 = [\widehat{v}_1^{c'\cup c}]^{T_1}_{c'}, .., l_j = [\widehat{v}_j^{c'\cup c}]^{T_j}_{c'}\} : \{l_1 : T_1, ..., l_j : T_j\}$.

Case (ered.col.fun) : $\vdash^H_c [\lambda x : T.e]^{T'\to T''}_{c'} : T'\to T''$ must have been derived by (eT.col). The premises are then $\mathbf{nil} \vdash_c T'\to T'' : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash^H_{c'\cup c} (\lambda x : T.e) : T'\to T''$. We can then deduce by Lemma B.37 (types are ok provided their hashes are) that $\mathbf{nil} \vdash_c T'' : \mathbf{Le}(\top)$.

By Lemma B.62 (shortening typing proof) we have $\mathbf{nil} \vdash^H_{c'\cup c} (\lambda x : T.e) : T\to T_0$ whose last rule is (eT.fun) and $\vdash^H_{c'\cup c} T\to T_0 == T'\to T''$ (on which we can use the Lemma B.69 (type decomposition)). The premise is then $x : T \vdash^H_{c'\cup c} e : T_0$. By Lemma B.53 (things have to be ok) we know that $\mathbf{nil} \vdash^H_{c'\cup c} T' : \mathbf{Le}(\top)$. Then by (envok.x), we have $y : T' \vdash^H_{c'\cup c}$ ok. First, we need to get $y : T' \vdash^H_{c'\cup c} [y]^T_{c'} : T$ from (eT.eq), (eT.var) and (eT.col), using Lemma B.25 (weakening). Then, by Lemma B.27 (combined weakening) and Lemma B.23 (colour stripping judgements), we get $y : T', x : T \vdash^H_{c'\cup c} e : T''$. Finally, we use Lemma B.51 (type preservation by expression substitution) with substitution $\sigma = \{x\leftarrow[y]^T_{c'}\}$ to get $y : T \vdash^H_{c'\cup c} \{x\leftarrow[y]^T_{c'}\}e : T''$. Then by applying (eT.col) we have $y : T \vdash^H_c [\{x\leftarrow[y]^T_{c'}\}e]^{T''}_{c'} : T''$. We conclude by applying (eT.fun).

Case (ered.col.marred) : $\vdash^H_c [\mathbf{marshalled}_{c_0,H'}(e_0 : T)]^{\text{bytes}}_{c'} : \text{bytes}$ must have been derived by (eT.col). One of the premises gives us a proof of $\mathbf{nil} \vdash^H_{c'\cup c} \mathbf{marshalled}_{c_0,H'}(e_0 : T) : \text{bytes}$ whose last rule is (eT.marred). One of the premises is then $\mathbf{nil} \vdash^H_{c'\cup c}$ ok. By Lemma B.12 (ok environments are ok in every colour) we get $\mathbf{nil} \vdash^H_c$ ok. So we can derive $\mathbf{nil} \vdash^H_c \mathbf{marshalled}_{c_0,H'}(e_0 : T) : \text{bytes}$ with (eT.marred).

Case (ered.col.unit) : We have a proof of $\vdash^H_c [()]^{\text{unit}}_{c'} : \text{unit}$. Then by Lemma B.8 (environments and subhashes have to be ok) we know that $\vdash^H_c$ ok. Then we can use (eT.unit) to get the desired $\vdash^H_c () : \text{unit}$.

Case (ered.cong) : We know that $\vdash^H_{c'} C^{c'}_c.e : T$ must have been derived by a syntactic rule which has as a premise $\vdash^H_c e : T_0$. None of the other premises mentions e. We know that $H, e \to_c H', e'$ and, by induction, that $\vdash^H_c e' : T_0$. Then we can use the same syntactic rule to get the desired $\vdash^H_{c'} C^{c'}_c.e' : T$.

In all cases, we have $\mathbf{nil} \vdash_c e' : T'$, whence by (eT.eq), $\mathbf{nil} \vdash_c e' : T$. □

Lemma B.100 (type preservation for network structural congruence) If $\vdash n$ ok and $n \equiv n'$ then $\vdash n'$ ok.

Proof. Induct on the derivation of $n \equiv n'$.

Case (nsc.id) : We have $n = \mathbf{0} \mid n'$. By reversing (nok.par), we get $\vdash n'$ ok.

Case (nsc.commut) : There exist n_1 and n_2 such that $n = n_1 \mid n_2$ and $n' = n_2 \mid n_1$. By reversing (nok.par) and applying it with the premises swapped, from $\vdash n\,\mathrm{ok}$, we get $\vdash n'\,\mathrm{ok}$.

Case (nsc.assoc) : There exist n_1, n_2, n_3 such that $n = n_1 \mid (n_2 \mid n_3)$ and $n' = (n_1 \mid n_2) \mid n_3$. By reversing (nok.par) twice, from $\vdash n\,\mathrm{ok}$, we get $\vdash n_i\,\mathrm{ok}$ for $1 \leqslant i \leqslant 3$, whence by (nok.par) twice $\vdash n'\,\mathrm{ok}$.

Reflexivity, symmetry, transitivity : Trivial (the latter two, by induction).

□

Corollary B.101 (type preservation for network reduction) If $\vdash n\,\mathrm{ok}$ and $n \rightarrow_n n'$ then $\vdash n'\,\mathrm{ok}$.

Proof. Induct on the derivation of the reduction $n \rightarrow_n n'$.

Case (nred.expr) : Trivial by Theorem B.99 (type preservation for expression reduction).

Case (nred.par) : There exist n_0, n_1, n_2 such that $n = n_0 \mid n_2$ and $n' = n_1 \mid n_2$. The premise is $n_0 \rightarrow_n n_1$. By reversing (nok.par), we have $\vdash n_0\,\mathrm{ok}$ and $\vdash n_2\,\mathrm{ok}$. By induction we have $\vdash n_1\,\mathrm{ok}$. By (nok.par), we have $\vdash n'\,\mathrm{ok}$.

Case (nred.strcong) : There exist n_0 and n_1 such that $n \equiv n_0 \rightarrow_n n_1 \equiv n'$. By Lemma B.100 (type preservation for network structural congruence), we get $\vdash n_0\,\mathrm{ok}$. By induction we get $\vdash n_1\,\mathrm{ok}$. By Lemma B.100 (type preservation for network structural congruence), we get $\vdash n'\,\mathrm{ok}$.

Case (nred.comm) : We know that $\vdash H, CC^{\bullet}_{c}.!\,v^c \mid H', CC^{\bullet}_{c'}.?\,\mathrm{ok}$ and, by reversing (nok.par) and (nok.expr), we have $\vdash^{H}_{\bullet} CC^{\bullet}_{c}.!\,v^c : \mathrm{unit}$ and $\vdash^{H'}_{\bullet} CC^{\bullet}_{c'}.? : \mathrm{unit}$. By Lemma B.63 (reversing typing proof through a context), we know that $\vdash^{H}_{c} !\,v^c : \mathrm{unit}$ and that $\vdash^{H}_{c} CC^{\bullet}_{c}.() : \mathrm{unit}$. By Lemma B.62 (shortening typing proof), we have $\vdash^{H}_{c} !\,v^c : \mathrm{unit}$ whose last rule is (eT.send). The premise is then $\vdash^{H}_{c} v^c : \mathrm{bytes}$. Then we can apply the Lemma B.71 (structural dependence of values on their types) to have a e_0 and T and c_0 and H_0 such that $\mathbf{nil} \vdash^{H_0}_{\bullet} T : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash^{H_0}_{c_0} e_0 : T$ and $v^c = \mathbf{marshalled}_{c_0,H_0}(e_0 : T)$.

On the other hand, by Lemma B.63 (reversing typing proof through a context), we get $\vdash^{H'}_{c'} ? : \mathrm{bytes}$. From Lemma B.8 (environments and subhashes have to be ok) we know that $\vdash^{H'}_{c'} \mathrm{ok}$.

Then we can apply (eT.marred) to get $\mathbf{nil} \vdash^{H'}_{c'} \mathbf{marshalled}_{c_0,H_0}(e_0 : T) : \mathrm{bytes}$. We now have $\vdash^{H'}_{\bullet} CC^{\bullet}_{c'}.\mathbf{marshalled}_{c_0,H_0}(e_0 : T_0) : \mathrm{bytes}$ by the second part of Lemma B.63 (reversing typing proof through a context).

We conclude by (mT.expr) twice, then (nok.par).

□

Theorem B.102 (type preservation for machine reduction) If $\mathbf{nil} \vdash^{H}_{\bullet} m : T$ and $H, m \rightarrow_m H', m'$ then $\mathbf{nil} \vdash^{H'}_{\bullet} m' : T$.

Proof. Consider each rule.

Case (mred.Eq) : We have $m =$ module $NU = [T_0, v^{\bullet} : T_1] : [X : \mathbf{Eq}(T'_0), T'_1]$ in m_0 and $m' = \{U.\mathtt{type} \leftarrow T'', U.\mathtt{term} \leftarrow_{(T_1 <: \{X \leftarrow T'_0\} T'_1)} v^{\bullet}\} m_0$, and we have a proof of $\mathbf{nil} \vdash^{H}_{\bullet}$ module $NU = [T_0, v^{\bullet} : T_1] : [X : \mathbf{Eq}(T'_0), T'_1]$ in $m_0 : T$ whose last rule can only be (mT.letext). The premises are then $\mathbf{nil} \vdash^{H}_{\bullet} T : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash^{H}_{\bullet} [T_0, v^{\bullet} : T_1] : [X : \mathbf{Eq}(T'_0), T'_1]$ and $U(T_0) : [X : \mathbf{Eq}(T'_0), T'_1] \vdash^{H}_{\bullet} m_0 : T$ where U doesn't bind in H. Since we know by Lemma B.48 (only abstract modules are in subhashes), that

the substituable entity U is not present anywhere in m_0, we can get $Z : \mathbf{Eq}(T_0'), z : \{X \leftarrow Z\}T_1' \vdash_c^H \{U.\mathtt{type} \leftarrow Z, U.\mathtt{term} \leftarrow z\}m_0 : T$ by Lemma B.58 (type preservation by module substitution in coloured judgements).

From Lemma B.35 (components of modules are ok) and from $\mathbf{nil} \vdash_\bullet^H [T_0, v^\bullet : T_1] : [X : \mathbf{Eq}(T_0'), T_1']$, we know that $X : \mathbf{Eq}(T_0) \vdash_\bullet^H T_1 <: T_1'$ and $X : \mathbf{Eq}(T_0') \vdash_\bullet^H T_1' : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_\bullet^H T_0 : \mathbf{Eq}(T_0')$ and $\mathbf{nil} \vdash_\bullet^H v^\bullet : T_1$ and $\mathbf{nil} \vdash_\bullet^H \mathbf{Eq}(T_0')$ ok.

By reversing the last rule ((Kok.Eq)) of the latter judgement, we get $\mathbf{nil} \vdash_\bullet^H T_0' : \mathbf{Le}(\top)$. With (Teq.refl) and (TK.Eq), we have $\mathbf{nil} \vdash_\bullet^H T_0' : \mathbf{Eq}(T_0')$.

Then we can apply the Lemma B.49 (type preservation by substitution) to get $z : \{X \leftarrow T_0'\}T_1' \vdash_c^H \{U.\mathtt{type} \leftarrow T_0', U.\mathtt{term} \leftarrow z\}m_0 : T$.

By Lemma B.8 (environments and subhashes have to be ok), we know that $X : \mathbf{Eq}(T_0) \vdash_\bullet^H$ ok. Then we can apply Lemma B.25 (weakening) and (eT.sub) to get $X : \mathbf{Eq}(T_0) \vdash_\bullet^H {}_{(T_1 <: T_1')} v^\bullet : T_1'$.

From $\mathbf{nil} \vdash_\bullet^H T_0 : \mathbf{Eq}(T_0')$ and (Teq.Eq), (Teq.sym) and (TK.Eq) we get $\mathbf{nil} \vdash_\bullet^H T_0' : \mathbf{Eq}(T_0)$. By Lemma B.49 (type preservation by substitution) we get $\mathbf{nil} \vdash_\bullet^H {}_{(T_1 <: \{X \leftarrow T_0'\}T_1')} v^\bullet : T_1'$. Then again, by Lemma B.49 (type preservation by substitution) we have the desired $\mathbf{nil} \vdash_c^H \{U.\mathtt{type} \leftarrow T_0', U.\mathtt{term} \leftarrow {}_{(T_1 <: \{X \leftarrow T_0'\}T_1')} v^\bullet\}m_0 : T$.

Case (mred.Le) : We have $m = \text{module } NU \text{ extends } (h_1, ..., h_i) \text{ restricts } (h_1', ..., h_j') = [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1']$ in m_0 and, with $h = \mathbf{hash}(N, [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1'])$, $m' = \{U \leftarrow h, U.\mathtt{type} \leftarrow T'', U.\mathtt{term} \leftarrow [{}_{(T_1 <: \{X \leftarrow T_0'\}T_1')} v^\bullet]_{\{h\}}^{\{X \leftarrow h.\mathtt{type}\}T_1'}\}m_0$.

We have a proof of $\mathbf{nil} \vdash_\bullet^H \text{module } NU \text{ extends } (h_1, ..., h_i) \text{ restricts } (h_1', ..., h_j') = [T_0, v^\bullet : T_1] : [X : \mathbf{Le}(T_0'), T_1']$ in $m_0 : T$ whose last rule can only be (mT.letext). The premises are then $\mathbf{nil} \vdash_\bullet^H T : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_\bullet^H [T_0, v^\bullet : T_1] : [X : \mathbf{Eq}(T_0'), T_1']$ and $U(T_0) : [X : \mathbf{Eq}(T_0'), T_1'] \vdash_\bullet^{H \cup h_1 <: U \cup ... \cup h_i <: U \cup U <: h_1' \cup ... \cup U <: h_j'} m_0 : T$ where U doesn't bind in H.

Then we apply the Lemma B.61 (type preservation by fully carried out module substitution) to get $\mathbf{nil} \vdash_\bullet^{\sigma(H \cup h_1 <: U \cup ... \cup h_i <: U \cup U <: h_1' \cup ... \cup U <: h_j')} \sigma m_0 : T$ with $\sigma = \{U \leftarrow h, U.\mathtt{type} \leftarrow h.\mathtt{type}, U.\mathtt{term} \leftarrow [{}_{(T_1 <: \{X \leftarrow h.\mathtt{type}\}T_1')} v^\bullet]_{\{h\}}^{\{X \leftarrow h.\mathtt{type}\}T_1'}\}$.

□

B.10 Progress

Definition B.103 (waiting for communication) An expression e is *waiting for communication* iff one of the following cases holds :

- e is ready to output, i.e. there exists $CC_c^{c'}$ and c_0 and H_0 and T_0 and e_0 such that $e = CC_c^{c'}.!\,\mathbf{marshalled}_{c_0, H_0}(e_0 : T_0)$
- e is ready to input, i.e. there exists $CC_c^{c'}$ such that $e = CC_c^{c'}.?$

Definition B.104 (dormant) An expression e is *dormant* iff one of the following cases holds :

- e is waiting for communication
- e is dead, i.e. there exists $CC_c^{c'}$ and T such that $e = CC_c^{c'}.\mathbf{Unmarfailure}^T$.

Lemma B.105 (dormancy in context) If e is dormant and $CC_c^{c'}$ is a coloured evaluation context then $CC_c^{c'}.e$ is dormant.

Proof. Composing coloured evaluation contexts yields a coloured evaluation context. □

Lemma B.106 (reduction in context) If $e \rightarrow_c$ and $CC_c^{c'}$ is an evaluation context then $CC_c^{c'}.e \rightarrow_{c'}$.

Proof. Apply (ered.cong) as many times as the size of $CC_c^{c'}$ requires. □

Definition B.107 (legitimately stuck expressions) An expression e is *legitimately stuck* in c iff one of the following cases holds :

- e is a c-value
- e is dormant.

Theorem B.108 (progress of expressions) If $\mathbf{nil} \vdash_c^H e : T$ then one of the following cases holds :

- e is legitimately stuck in c.
- e can reduce, i.e. there exists e' and H' such that $H, e \rightarrow_c H', e'$.

Proof. Induct on the type derivation. Consider the rule used in the last step of the proof.

Cases (eT.var) and (eT.mod) : Impossible by Lemma B.11 (free variables of a judgement come from the environment) since the environment is empty.

Case (eT.sub) : There exist e_0, T_0 such that $e = {}_{(T_0 <: T)}e_0$ and $\mathbf{nil} \vdash_c^H e_0 : T_0$ and $\mathbf{nil} \vdash_c^H T_0 <: T$. Apply the induction hypothesis to e_0.

Case e_0 can reduce : there exists e_0' and H' such that $H, e_0 \rightarrow_c H', e_0'$. By (ered.cong), $H, e \rightarrow_c H', {}_{(T_0 <: T)}e_0'$.

Case e_0 is ready to output : there exist $CC_{c'}^c$ and $v^{c'}$ such that $e_0 = CC_{c'}^c.!\, v^{c'}$, thus $e = ({}_{(T_0 <: T)}\_).CC_{c'}^c.!\, v^{c'}$ is ready to output.

Case e_0 is ready to input : Similar to the output case.

Case e_0 is dead : Then e is dead.

Case e_0 is a c-value : Then we have to distinguish the different cases :

Case e_0 is a $\widehat{\widehat{v}}^c$: If $T = h.\mathtt{type}$ with $h \in c$ then we can apply (ered.sub.typeright). If $h \notin c$, then T is a type value TV^c. In this case, if $T_0 = h_0.\mathtt{type}$ with $h_0 \in c$ then we can apply (ered.sub.typeleft). If $h \notin c$, then T_0 is a type value TV_0^c, and then e is legitimately stuck as a value. We finished the cases where at least one of the types is abstract.

We thus know that T_0 is of the form $TC(T_1, ..., T_i)$. We then apply the Lemma B.71 (structural dependence of values on their types) to get the possible forms of e_0. Since we also have $\mathbf{nil} \vdash_c^H T_0 <: T$, by Lemma B.79 (essence of subtyping), we know that T is also of the form $TC(T_1', ..., T_j')$ with the same type constructor as T_0. We are in the cases where (ered.sub.unit) or (ered.sub.marshalled) or (ered.sub.fun) or (ered.sub.record) or (ered.sub.tuple) can be applied.

Case e_0 is a ${}_{(TV_1^{c_0} <: TV_2^{c_0})}\widehat{\widehat{v}}^{c_0}$: We can reduce e by (ered.sub.sub).

Case e_0 is a $[\widehat{v}^{c_1 \cup c_0}]_{c_1}^{h_1}.\mathtt{type}$: Since e_0 is a c-value, we know that $h_1 \notin c$. Then we can apply (ered.sub.col).

Case (eT.eq) : The inductive hypothesis is the desired result.

Case (eT.ap) : There exists e_0, e_1, T_1 such that $e = e_0\, e_1$ and $\mathbf{nil} \vdash_c^H e_0 : T_1 \rightarrow T$ and $\mathbf{nil} \vdash_c^H e_1 : T_1$. Apply the inductive hypothesis to e_0.

Case e_0 can reduce : there exists e_0' and H' such that $H, e_0 \rightarrow_c H', e_0'$. By (ered.cong), $H, e \rightarrow_c H', e_0'\, e_1$.

Case e_0 is ready to output : there exist $CC^c_{c'}$ and $v^{c'}$ such that $e_0 = CC^c_{c'}.!\,v^{c'}$, thus $e = (\_\, e_1).CC^c_{c'}.!\,v^{c'}$ is ready to output.

Case e_0 is ready to input : similar to the output case.

Case e_0 is dead : then e is dead.

Case e_0 is a c-value : Apply the inductive hypothesis to e_1. Note that $e_0\ \_$ is an evaluation context.

Case e_1 can reduce : there exists e_1' and H' such that $H, e_1 \to_c H', e_1'$. By (ered.cong), $H, e \to_c H', e_0\, e_1'$.

Case e_1 is ready to output : there exist $CC^c_{c'}$ and $v^{c'}$ such that $e_1 = CC^c_{c'}.!\,v^{c'}$, thus $e = (e_0\ \_).CC^c_{c'}.!\,v^{c'}$ is ready to output.

Case e_1 is ready to input : similar to the output case.

Case e_1 is dead : then e is dead.

Case e_1 is a c-value : By Lemma B.71 (structural dependence of values on their types), there exists e_2 such that $e_0 = (\lambda x : T_1.e_2)$. Then $e = (\lambda x : T_1.e_2)\, e_1$. By (ered.ap), $H, e \to_c H, \{x \leftarrow e_1\}e_2$.

Case (eT.fun) : e is a value.

Case (eT.send) : e is ready to output.

Case (eT.recv) : e is ready to input.

Case (eT.mar) : There exist an e_0 and a T_0 such that $e = \mathbf{mar}\,(e_0 : T_0)$, and $T = \text{bytes}$. If e_0 is dormant or reduces then the same holds for e by Lemma B.106 (reduction in context) and Lemma B.105 (dormancy in context). Otherwise, by the inductive hypothesis on e_0, e_0 is a c-value, so by (ered.mar), $H, e \to_c H, \mathbf{marshalled}_{c',H}(e_0 : T)$.

Case (eT.marred) : There exist an e_0 and a T_0 and a H_0 and a c_0 such that $e = \mathbf{marshalled}_{c_0,H_0}(e_0 : T_0)$, and $T = \text{bytes}$. Then e is a value.

Case (eT.unmar) : There is an e_0 such that $e = (\mathbf{unmar}\, e_0 : T)$. If e_0 is dormant or reduces then the same holds for e by Lemma B.106 (reduction in context) and Lemma B.105 (dormancy in context). Otherwise, by the inductive hypothesis on e_0, e_0 is a value. Its type is bytes, so by Lemma B.71 (structural dependence of values on their types), there is a e_1 and a T_0 and a H_0 and a c_0 such that $e_0 = \mathbf{marshalled}_{c_0,H_0}(e_1 : T_0)$. Then e reduces by (ered.unmar).

Case (eT.Undynfailure) : e is dormant.

Case (eT.unit) : e is a value.

Case (eT.tuple) : There are $e_1, ..., e_j$ such that $e = (e_1, ..., e_j)$. Let i be the smallest index k such that e_1 through e_{k-1} are values. If $i = j + 1$ then e is a value. Otherwise, apply the inductive hypothesis to e_i. Since e_i is not a value, it is dormant or reduces, and in either case, the same holds for e by Lemma B.105 (dormancy in context) and Lemma B.106 (reduction in context), as $(e_1, ..., e_{i-1}, \_, e_{i+1}, ..., e_j)$ is an evaluation context.

Case (eT.record) : Similar to (eT.tuple).

Case (eT.proj) : There is an e' such that $e = \mathbf{proj}_i\, e'$. If e' is dormant or reduces then the same holds for e by Lemma B.106 (reduction in context) and Lemma B.105 (dormancy in context). Otherwise, by the inductive hypothesis on e, e' is a value. We know that $\mathbf{nil} \vdash^H_c e' : T_1 * ... * T_j$. By Lemma B.71 (structural dependence of values on their types), there are $v^c_1, ..., v^c_j$ such that $e' = (v^c_1, ..., v^c_j)$. Then $H, e \to_c H, v^c_i$ by (ered.proj).

Case (eT.field) : Similar to (eT.proj).

Case (eT.col) : There is an e_0 and an c_0 such that $e = [e_0]^T_{c_0}$. Apply the inductive hypothesis to e_0 in the colour $c \cup c_0$; if e_0 is a value, the discussion depends on its form and that of $c \cup c_0$.

By Lemma B.62 (shortening typing proof), there is a type T' such that $\mathbf{nil} \vdash_{c \cup c_0} e_0 : T'$ by a smaller proof that does not use (eT.eq) as its last step and $\mathbf{nil} \vdash_{c \cup c_0} T' == T$.

Case e_0 is dormant : e is dormant.

Case e_0 reduces : There is an e'_0 such that $e_0 \rightarrow_{c \cup c_0} e'_0$. By (ered.cong), $e \rightarrow_c [e'_0]^T_{c_0}$.

Case e_0 is an $c \cup c_0$-value :

Case $e_0 = \widehat{\widehat{v}}^{c \cup c_0}$: Then, by reversing the appropriate rule amongst (eT.unit), (eT.tuple), (eT.fun) or (eT.marred), we have that T' is some constructed type $TC(T'_1, ..., T'_j)$. Since $\mathbf{nil} \vdash_{c \cup c_0} T' == T$, knowing that the environment is empty, by Lemma B.69 (type decomposition), one of the following cases holds :

Case T is a constructed type : There exist $T_1, ..., T_j$ such that for all i, $T'_i = T_i$ or $\mathbf{nil} \vdash_{c \cup c_0} T'_i == T_i$, and $T = TC(T_1, ..., T_j)$. Then e reduces by the appropriate rule amongst (ered.col.unit), (ered.col.tuple), (ered.col.record), (ered.col.fun) or (ered.col.marred).

Case $T = h_0.\texttt{type}$ and $h_0 \in c \cup c_0$: If $h_0 \in c$ then $e = [e_0]^{h_0}_{c_0}.\texttt{type}$ reduces by (ered.col.type). Else, it is a value.

Case $e_0 = {}_{(TV_1^{c \cup c_0} <: TV_2^{c \cup c_0})}\widehat{\widehat{v}}^{c \cup c_0}$: Then we know that $T' = TV_2^{c \cup c_0}$ with $\mathbf{nil} \vdash_{c \cup c_0} T' == T$.

If $T' = h.\texttt{type}$ with $h \notin c \cup c_0$, then by Lemma B.69 (type decomposition), we know that T can only be $h.\texttt{type}$ or an $h'.\texttt{type}$ with $h' \in c \cup c'$. In the first case, we have a value and in the second case we can apply (ered.col.type).
If T' is a constructed type, then we know that $TV_1^{c \cup c_0} = h.\texttt{type}$ with $h \notin c \cup c_0$. By reversing (eT.sub), we get a proof of $\mathbf{nil} \vdash_{c \cup c_0} \widehat{\widehat{v}}^{c \cup c_0} : h.\texttt{type}$. By Lemma B.62 (shortening typing proof) we have a proof of $\mathbf{nil} \vdash_{c \cup c_0} \widehat{\widehat{v}}^{c \cup c_0} : T''$ whose last rule is not (eT.eq), and a proof of $\mathbf{nil} \vdash_{c \cup c_0} h.\texttt{type} == T''$. However, we know that $\widehat{\widehat{v}}^{c \cup c_0}$ is syntactically a constructed value, and it implies (by reversing (eT.fun), (eT.tuple), (eT.record), (eT.unit), (eT.marred)) that T'' is a constructed type. Then the Lemma B.69 (type decomposition) tells us that h has to be in $c \cup c_0$ which is a contradiction. Then T' cannot be a constructed type.

Case $e_0 = [v^{c_0 \cup c_1}]^{h_1}_{c_1}.\texttt{type}$ with $h_1 \notin c \cup c_0$: By reversing (eT.col) and applying Lemma B.62 (shortening typing proof) we get a proof of $\mathbf{nil} \vdash_{c \cup c_0} h_1.\texttt{type} == T$. From Lemma B.69 (type decomposition) there are two possibilities. First, $T = h_1.\texttt{type}$ and then we can reduce using (ered.col.col). Or $T = h_0.\texttt{type}$ with $h_0 \in c$ and then we can reduce e using (ered.col.type).

□

Corollary B.109 (progress of networks) If $\vdash n\,\text{ok}$ then one of the following cases holds :

- n is stopped, i.e. there exists $n_{()}$ and n_{fail} such that $n \equiv n_{()} \mid n_{\text{fail}}$.
- n is waiting to input, i.e. there exists $n_{()}$ and n_{fail} and $n_?$ such that $n \equiv n_{()} \mid n_{\text{fail}} \mid n_?$

– n is waiting to output, i.e. there exists $n_{()}$ and n_{fail} and $n_!$ such that $n \equiv n_{()} \mid n_{\text{fail}} \mid n_!$
– n can reduce, i.e. there exists n' such that $n \rightarrow_n n'$

where

$$
\begin{array}{lll}
n_{()} ::= & \mathbf{0} & \text{null} \\
 & n_{()} \mid n_{()} & \text{parallel composition} \\
 & () & \text{unit} \\
 & & \\
n_{\text{fail}} ::= & \mathbf{0} & \text{null} \\
 & n_{\text{fail}} \mid n_{\text{fail}} & \text{parallel composition} \\
 & CC^{\bullet}_{c}.\mathbf{Unmarfailure}^{T} & \text{dead} \\
 & & \\
n_? ::= & n_? \mid n_? & \text{parallel composition} \\
 & CC^{\bullet}_{c}.? & \text{waiting to input} \\
 & & \\
n_! ::= & n_! \mid n_! & \text{parallel composition} \\
 & CC^{\bullet}_{c}.!\, v & \text{waiting to output}
\end{array}
$$

Proof. Induct on the derivation of $\vdash n\,\text{ok}$.

Case (nok.zero) : Trivial.

Case (nok.par) : There exist n_0 and n_1 such that $n = n_0 \mid n_1$. If either n_0 or n_1 reduces then n reduces. If n_0 is stopped then n has the same form as n_1, and vice versa. If n_0 and n_1 are both waiting to input (or both to output) then so is n. Otherwise $n_0 \equiv n_{()} \mid n_{\text{fail}} \mid n_?$ and $n_1 \equiv n_{()} \mid n_{\text{fail}} \mid n_!$ (or the converse) : then n reduces by (nred.comm).

Case (nok.expr) : In this case, n is an expression. Therefore, we can apply the Theorem B.108 (progress of expressions), and thus one of the following cases holds :

Case n is a $\bullet$-value : By (nok.expr), the value n has type unit. By Lemma B.71 (structural dependence of values on their types), n is an $n_{()}$.

Case n is dead : n is an n_{fail}.

Case n is waiting for input : n is an $n_?$.

Case n is waiting for output : n is an $n_!$.

Case n reduces : n reduces.

□

Theorem B.110 (progress of machines) If $\mathbf{nil} \vdash^{H}_{\bullet} m : T$ then either m is an expression or it reduces under $\rightarrow_m$.

Proof. Induct on the type derivation.

Case (mT.expr) : Trivial.

Case (mT.letext) : It is obvious that m reduces, by either (mred.Le) or (mred.Eq).

□

B.11 Determinism of reduction

Theorem B.111 (determinism of machine reduction) Reduction of machines is deterministic, i.e. if $m \to_m m_1$ and $m \to_m m_2$ then $m_1 = m_2$ and both reductions use the same rule on the same redex.

Proof. Induct on the structure of m.

Case m is an expression : Impossible (m does not reduce).

Case $m =$ module $NU = M : [X : \mathbf{Le}(T'), T]$ in m' : The only applicable rule is (mred.Le).

Case $m =$ module $NU = M : [X : \mathbf{Eq}(T'), T]$ in m' : The only applicable rule is (mred.Eq).

□

Lemma B.112 (values do not reduce) If $H, e \to_c H', e'$ then e is not an c-value.

Proof. We prove that if e is an c-value then e does not reduce in c. We induct on the structure of values.

Case $\widehat{\widehat{v}}^c = ()$: No reduction rule applies.

Case $\widehat{\widehat{v}}^c = (v_1^c, ..., v_j^c)$: The only reduction rule that is not obviously inapplicable is (ered.cong). If that rule applies, then it is with a context of the form $(v_1^c, ..., v_{i-1}^c, \_, v_{i+1}^c, ..., v_j^c)$. But then $v_i^c \to_c$, which is impossible by induction.

Case $\widehat{\widehat{v}}^c = \{l_1 = v_1^c; ...; l_j = v_j^c\}$: The only reduction rule that is not obviously inapplicable is (ered.cong). If that rule applies, then it is with a context of the form $\{l_1 = v_1^{c_1}, .., l_{i-1} = v_{i-1}^{c_1}, l_i = \_, l_{i+1} = e_{i+1}, .., l_j = e_j\}$. But then $v_i^c \to_c$, which is impossible by induction.

Case $\widehat{\widehat{v}}^c = (\lambda x : T.e)$: No reduction rule applies.

Case $\widehat{\widehat{v}}^c =$ marshalled $_{c',H'}(e : T)$: No reduction rule applies.

Case $\widehat{v}^c = {}_{(TV_1^c <: TV_2^c)}\widehat{\widehat{v}}^c$: Since $TV_1^c = h_0.\texttt{type}$ or $TV_2^c = h_0.\texttt{type}$ (ered.sub.{tuple,record.*, fun,marshalled,unit}) do not apply. The rule (ered.sub.sub) requires $\widehat{\widehat{v}}^c$ to start with some explicit subtyping annotation which is syntactically impossible. The rules (ered.sub.typeright) and (ered.sub.typeleft) require the hash types to have their hashes in c which is impossible by definition of TV^c. By induction on $\widehat{\widehat{v}}^c$, (ered.cong) cannot be applied.

Case $v^c = [\widehat{v}^{c_1 \cup c}]_{c_1}^{h_1.\texttt{type}}$ where $h_1 \notin c$: The rule (ered.col.type) requires $h_1 \in c$, a contradiction. The other rules (ered.col.*) do not apply as they require the type annotation on the bracket not to be a hash. If (ered.cong) applies, then it is with a context of the form $[\_]_{c_1}^{h_1.\texttt{type}}$. But then $\widehat{v}^{c_1 \cup c} \to_{c_1 \cup c}$, which is impossible by induction.

□

Theorem B.113 (determinism of expression reduction) Reduction of expressions and machines is deterministic, i.e. if $H, e \to_c H', e'$ and $H, e \to_c H'', e''$ then $e' = e''$ and $H' = H''$ an both reductions use the same rule on the same redex.

Proof. Induct on the structure of e.

Cases $e = x$, $e = U.\texttt{term}$, $e =$ Unmarfailure^T : No reduction is possible.

Cases $e = ()$, $e = (v_1^c, ..., v_j^c)$, $e = \lambda x : T.e_0$, $e = \mathbf{marshalled}_{c',H'}(e_0 : T)$ **:** No reduction is possible, by Lemma B.112 (values do not reduce).

Case $e = (e_1, ..., e_j)$ **:** Let i be the smallest k such that e_1 through e_{k-1} are c-values. The case $i = j + 1$ has already been treated. Given Lemma B.112 (values do not reduce), the only possibility of reduction is (ered.cong) with the context $(e_1, ..., e_{k-1}, \_, e_{k+1}, ..., e_j)$. By induction, only one reduction is possible.

Case $e = \{l_1 = e_1; ...; l_j = e_j\}$ **:** Similar to $(e_1, ..., e_j)$.

Case $e = \mathbf{proj}_i\, e_0$ **:** If e_0 is a c-value, given Lemma B.112 (values do not reduce), the only possibility of reduction is (ered.proj). Otherwise, by induction, only one reduction of e_0 is possible, and the only possibility for e to reduce is using (ered.cong) with the context $\mathbf{proj}_i\, \_$.

Case $e = e_0.l_i$ **:** Similar to $\mathbf{proj}_i\, e_0$.

Case $e = e_1\, e_2$ **:** If e_1 and e_2 are both c-values, given Lemma B.112 (values do not reduce), the only possibility of reduction is (ered.ap). If e_1 is an c-value and e_2 is not an c-value, then the only possibility for reduction is to use (ered.cong) with the context $e_1\, \_$; by induction, this yields at most one possible reduction. Similarly, if e_1 is not an c-value, then the only possibility of reduction is (ered.cong) with the context $\_\, e_2$.

Case $e = \mathbf{mar}\,(e_0 : T)$ **:** If e_0 is an c-value, then e_0 does not reduce by Lemma B.112 (values do not reduce), so (ered.mar) is the only possibility of reduction. Otherwise the only possibility of reduction is (ered.cong) with the context $\mathbf{mar}\,(\_ : T)$, so by induction, only one reduction is possible.

Case $e = \mathbf{unmar}\, e_0 : T$ **:** The only possibility of reduction is (ered.unmar), which has only one possible outcome for any given e_0 and T.

Case $e = !\, e_0$ **:** If e_0 is a c-value, given Lemma B.112 (values do not reduce), no reduction is possible (communication happens at the network level). Otherwise, by induction, only one reduction of e_0 is possible, and the only possibility for e to reduce is using (ered.cong) with the context $!\, \_$.

Case $e = ?$ **:** No reduction is possible (communication happens at the network level).

Case $e = [e_1]^T_{c_1}$ **:**

Case e_1 **is not a** $c \cup c_1$**-value :** Then the only possibility of reduction is (ered.cong), so by induction, only one reduction is possible. We then assume in the remaining cases that e_1 is a $c \cup c_1$-value.

Case $e_1 = \widehat{v}^{c \cup c_1}$ **:** Then (ered.cong) does not apply since $\widehat{v}^{c_1 \cup c}$ is a value and we have Lemma B.112 (values do not reduce). If e_1 is a constructed value, the only rules that may apply are the rules to push brackets in ((ered.col.unit), (ered.col.tuple), (ered.col.record), (ered.col.fun), (ered.col.marred)) which are mutually exclusive. If e_1 starts with a subtyping annotation, then only (ered.col.type) could be applied.

Case $e_1 = [\widehat{v}]^{h.\mathtt{type}}_{c_0}$ **:** Then (ered.cong) does not apply since e_1 is a value and we have Lemma B.112 (values do not reduce). Then only (ered.col.col) can be applied.

Case $e = {}_{(T_1 <: T_2)}e_1$ **:**

Case e_1 **is not a** c**-value :** Then the only possibility of reduction is (ered.cong), so by induction, only one reduction is possible. We then assume in the remaining cases that e_1 is a c-value.

Case $e_1 = \widehat{\widehat{v}}^{\,c}$ **:** Then (ered.cong) does not apply since $\widehat{\widehat{v}}^{\,c}$ is a value and we have Lemma B.112 (values do not reduce). If T_1 and T_2 are constructed types, the only rules that may apply are the rules to push the annotation in ((ered.sub.unit), (ered.sub.tuple), (ered.sub.record), (ered.sub.fun), (ered.sub.marshalled)) which are mutually exclusive. If $T_2 = h_2.\texttt{type}$ with $h_2 \in c$, then we can only reduce with (ered.sub.typeright). If $h_2 \notin c$, then we look at T_1. If $T_1 = h_1.\texttt{type}$ with $h_1 \in c$, then we can only reduce with (ered.sub.typeleft), else $(h_1 \notin c)$ e is a value and we have Lemma B.112 (values do not reduce).

Case $e_1 = {}_{(TV_1^c <: TV_2^c)}\widehat{\widehat{v}}^{\,c}$ **:** Then (ered.cong) does not apply since e_1 is a value and we have Lemma B.112 (values do not reduce). Then the only rule that can be applied is (ered.sub.sub).

Case $e_1 = [\widehat{v}]^h_{c_0}.\texttt{type}$ **:** Then (ered.cong) does not apply since e_1 is a value and we have Lemma B.112 (values do not reduce). Then the only rule that can be applied is (ered.sub.col).

□

B.12 Erasure

Definition B.114 (bracket and subtyping reduction subsystem) The bracket and subtyping reduction subsystem is a reduction relation $\rightarrow^{\text{bse}}_c$ on expressions consisting of (ered.cong) and the (ered.col.*) and (ered.sub.*) rules.

Lemma B.115 (determinism of bracket and subtyping reduction) The bracket and subtyping reduction is deterministic.

Proof. Trivial consequence of Theorem B.113 (determinism of expression reduction). □

Lemma B.116 (termination of bracket and subtyping reduction) Bracket and subtyping reduction is strongly normalising.

Proof. Define the bracket and subtyping reduction weight of an expression $w_{\text{bse}}(e)$ structurally on the expression. We use $\mid T \mid$ as the number of hashes in a type expression. It trivially satisfies $\mid h.\texttt{type} \mid > \mid \text{impl}\,(h) \mid$.

$$
\begin{aligned}
w_{\text{bse}}(()) &= 1 \\
w_{\text{bse}}((e_1, ..., e_j)) &= w_{\text{bse}}(e_1) + \cdots + w_{\text{bse}}(e_j) + 1 \\
w_{\text{bse}}(\{l_1 = e_1; ...; l_j = e_j\}) &= w_{\text{bse}}(e_1) + \cdots + w_{\text{bse}}(e_j) + 1 \\
w_{\text{bse}}(\mathbf{proj}_i\, e) &= w_{\text{bse}}(e) + 1 \\
w_{\text{bse}}(e.l_i) &= w_{\text{bse}}(e) + 1 \\
w_{\text{bse}}(x) &= 1 \\
w_{\text{bse}}(\lambda x : T.e) &= 1 \\
w_{\text{bse}}(e_1\, e_2) &= w_{\text{bse}}(e_1) + w_{\text{bse}}(e_2) + 1 \\
w_{\text{bse}}(\mathbf{mar}\,(e : T)) &= w_{\text{bse}}(e) + 1 \\
w_{\text{bse}}(\mathbf{marshalled}\,(e : T)) &= w_{\text{bse}}(e) + 1 \\
w_{\text{bse}}(\mathbf{unmar}\, e : T) &= w_{\text{bse}}(e) + 1 \\
w_{\text{bse}}(!\, e) &= w_{\text{bse}}(e) + 1 \\
w_{\text{bse}}(?) &= 1 \\
w_{\text{bse}}(U.\texttt{term}) &= 1 \\
w_{\text{bse}}([e]^T_c) &= (2 + \mid T \mid) w_{\text{bse}}(e) \\
w_{\text{bse}}({}_{(T <: T')}e) &= (w_{\text{bse}}(e) + 1)^{2 + |T| + |T'|} \\
w_{\text{bse}}(\mathbf{Unmarfailure}^T) &= 1
\end{aligned}
$$

It is obvious that $w_{\text{bse}}(e)$ is always a strictly positive integer.

We prove that if $e \rightarrow_c^{\text{bse}} e'$ then $w_{\text{bse}}(e) > w_{\text{bse}}(e')$. We induct on the derivation of the reduction.

Case (ered.col.unit) : Here $e = [()]_{c'}^{\text{unit}}$ and $e' = ()$. We have $w_{\text{bse}}(e) = 2 > 1 = w_{\text{bse}}(e')$.

Case (ered.col.marred) : Here $e = [\textbf{marshalled}\,(e_0 : T)]_{c'}^{\text{bytes}}$ and $e' = \textbf{marshalled}\,(e_0 : T)$. We have $w_{\text{bse}}(e) = 2(1 + w_{\text{bse}}(e_0)) > 1 + w_{\text{bse}}(e_0) = w_{\text{bse}}(e')$.

Case (ered.col.fun) : Here $e = [\lambda x : T.e_0]_{c'}^{T' \to T''}$ and $e' = \lambda x : T'.[\{x \leftarrow [x]_c^{T'}\}e_0]_{c'}^{T''}$. We have $w_{\text{bse}}(e) = 2 + |T'| + |T''| > 1 = w_{\text{bse}}(e')$

Case (ered.col.record) : Here $e = [\{l_1 = e_1; ...; l_j = e_j\}]_{c'}^{T_1 * ... * T_j}$ and $e' = \{l_1 = [e_1]_{c'}^{T_1}; ...; l_j = [e_j]_{c'}^{T_j}\}$. We have $w_{\text{bse}}(e) = (2+\sum_i |Ti|)(1+\sum_i w_{\text{bse}}(e_i)) > 1+\sum_i(2+|Ti|)w_{\text{bse}}(e_i) = w_{\text{bse}}(e')$.

Case (ered.col.tuple) : Here $e = [(e_1, ..., e_j)]_{c'}^{T_1 * ... * T_j}$ and $e' = ([e_1]_{c'}^{T_1}, ..., [e_j]_{c'}^{T_j})$. We have $w_{\text{bse}}(e) = (2 + \sum_i |Ti|)(1 + \sum_i w_{\text{bse}}(e_i)) > 1 + \sum_i(2 + |Ti|)w_{\text{bse}}(e_i) = w_{\text{bse}}(e')$.

Case (ered.col.type) : Here $e = [e_0]_{c'}^{h.\texttt{type}}$ and $e' = [e_0]_{c'}^{\text{impl}(h)}$. Since $\mid h.\texttt{type} \mid > \mid \text{impl}\,(h) \mid$, we have $w_{\text{bse}}(e) = (2 + |h.\texttt{type}|)w_{\text{bse}}(e_0) > (2 + |\text{impl}\,(h)\,|)w_{\text{bse}}(e_0) = w_{\text{bse}}(e')$.

Case (ered.col.col) : Here $e = [[e_0]_{c_0}^{h_0.\texttt{type}}]_{c_1}^{h_1.\texttt{type}}$ and $e' = [e_0]_{c_0 \cup c_1}^{h_1.\texttt{type}}$. We have $w_{\text{bse}}(e) = (2 + |h_1.\texttt{type}|)(2 + |h_0.\texttt{type}|)w_{\text{bse}}(e_0) > (2 + |h_1.\texttt{type}|)w_{\text{bse}}(e_0) = w_{\text{bse}}(e')$.

Case (ered.sub.col) : Here $e = {}_{(T' <: T'')}([e_0]_{c'}^{h.\texttt{type}})$ and $e' = [{}_{(T' <: T'')}e_0]_{c'}^{T''}$. We need to prove that $w_{\text{bse}}(e) = ((2 + |h.\texttt{type}|)w_{\text{bse}}(e_0) + 1)^{2+|T'|+|T''|} > (2 + |T''|)(w_{\text{bse}}(e_0) + 1)^{2+|T'|+|T''|} = w_{\text{bse}}(e')$. We rely on the fact that $\forall m > 1, \forall n > 0, m^n > n$. Then we have $(1.5)^{2+|T'|+|T''|} > 2 + \mid T'' \mid$. On the other hand we have $\left(\frac{(2+|h.\texttt{type}|)w_{\text{bse}}(e_0)+1}{w_{\text{bse}}(e_0)+1}\right)^{2+|T'|+|T''|} > (1.5)^{2+|T'|+|T''|}$. We can conclude easily.

Case (ered.sub.unit) : Here $e = {}_{(\text{unit} <: \text{unit})}()$ and $e' = ()$. We have $w_{\text{bse}}(e) = 4 > 1 = w_{\text{bse}}(e')$.

Case (ered.sub.marshalled) : Here $e = {}_{(\text{bytes} <: \text{bytes})}\textbf{marshalled}_{\,c',H'}(e_0 : T)$ and $e' = \textbf{marshalled}_{\,c',H'}(e_0 : T)$. We have $w_{\text{bse}}(e) = (w_{\text{bse}}(e') + 1)^2 > w_{\text{bse}}(e')$.

Case (ered.sub.fun) : Here our two expressions are $e = {}_{((T_1 \to T_2) <: (T_1' \to T_2'))}(\lambda x : T_0.e_0)$ and $e' = (\lambda x : T_1'.\, {}_{(T_2 <: T_2')}\{x \leftarrow {}_{(T_1' <: T_1)}x\}e)$. Then we get $w_{\text{bse}}(e) = 4 > 1 = w_{\text{bse}}(e')$.

Case (ered.sub.record) : Here $e = {}_{(\{l_1:T_1';...;l_i:T_i'\} <: \{l_1:T_1;...;l_j:T_j\})}\{l_1 = v_1^c; ...; l_j = v_j^c\}$ and $e' = \{l_1 = v_1^c; ...; l_i = v_i^c\}$. We have $w_{\text{bse}}(e) = (2 + \sum_{k \le j} w_{\text{bse}}(e_k))^{2+\sum_{k\le i}|Tk|+\sum_{k\le j}|Tk'|} > 1 + \sum_{k \le i} w_{\text{bse}}(e_k) = w_{\text{bse}}(e')$.

Case (ered.sub.tuple) : Here our two expressions are $e = {}_{((T_1 * ... * T_j) <: (T_1' * ... * T_j'))}(e_1, ..., e_j)$ and $e' = ({}_{(T_1 <: T_1')}e_1, ..., {}_{(T_j <: T_j')}e_j)$.

We have then $w_{\text{bse}}(e) = (2 + \sum_{k \le j} w_{\text{bse}}(e_k))^{2+\sum_{k\le j}|Tk|+\sum_{k\le j}|Tk'|} > 1 + \sum_{k \le j}(1 + w_{\text{bse}}(e_k))^{2+|Tk|+|Tk'|} = w_{\text{bse}}(e')$.

Case (ered.sub.typeleft) : Here $e = {}_{(h_0 <: .)}\texttt{type}\,TV^c e_0$ and $e' = {}_{(\text{impl}(h_0) <: TV^c)}e_0$. We have $w_{\text{bse}}(e) = (1 + w_{\text{bse}}(e_0))^{2+|h_0.\texttt{type}|+|TV^c|} > (1 + w_{\text{bse}}(e_0))^{2+|\text{impl}(h_0)|+|TV^c|} = w_{\text{bse}}(e')$.

Case (ered.sub.typeright) : Here $e = {}_{(T <: h_0)}.\texttt{type}e_0$ and $e' = {}_{(T <: \text{impl}(h_0))}e_0$. We have $w_{\text{bse}}(e) = (1 + w_{\text{bse}}(e_0))^{2+|T|+|h_0.\texttt{type}|} > (1 + w_{\text{bse}}(e_0))^{2+|T|+|\text{impl}(h_0)|} = w_{\text{bse}}(e')$.

Case (ered.sub.sub) : Here $e = {}_{(T_0 <: T_1)}({}_{(T_2 <: T_3)})e_0$ and $e' = {}_{(T_2 <: T_1)}e_0$. We have $w_{\text{bse}}(e) = (1 + (1 + w_{\text{bse}}(e_0))^{2+|T_0|+|T_1|})^{2+|T_2|+|T_3|} > (1 + w_{\text{bse}}(e_0))^{2+|T_2|+|T_1|} = w_{\text{bse}}(e')$.

Case (ered.cong) : Here $e = C^c_{c_0}.e_0$ and $e' = C^c_{c_0}.e'_0$ and $e_0 \rightarrow^{\text{bse}}_{c_0} e'_0$. By induction, $w_{\text{bse}}(e_0) > w_{\text{bse}}(e'_0)$.
If $C^c_{c_0} = [\_]^{T_0}_{c_0}$, then $w_{\text{bse}}(e) = (2 + |T_0|)w_{\text{bse}}(e_0) > (2 + |T_0|)w_{\text{bse}}(e'_0) = w_{\text{bse}}(e')$.
If $C^c_{c_0} = {}_{(T_0 <: T_1)}\_$, then $w_{\text{bse}}(e) = (1+w_{\text{bse}}(e_0))^{2+|T_0|+|T_1|} > (1+w_{\text{bse}}(e'_0))^{2+|T_0|+|T_1|} = w_{\text{bse}}(e')$. Otherwise, there exists an integer k such that $w_{\text{bse}}(e) = k + w_{\text{bse}}(e_0) > k + w_{\text{bse}}(e'_0) = w_{\text{bse}}(e')$.

Since the weight of an expression is a positive integer that decreases at each step of reduction, bracket reduction is strongly normalising (bracket reduction of e terminates in at most $w_{\text{bse}}(e)$ steps). □

Definition B.117 (stripped expressions) A stripped expression is one that does not have any coloured brackets nor subtyping annotations nor annotations on λ in it. We also forget about colour annotation on **marshalled** . Ditto for types (where annotations are removed from hashes), values, contexts, and networks.

$\underline{e}$ ::=	stripped expression
$()$	unit
$(\underline{e}_1, ..., \underline{e}_j)$	tuple $(2 \leqslant j)$
$\mathbf{proj}_i\ \underline{e}$	projection
$\{l_1 = \underline{e}_1, ..., l_j = \underline{e}_j\}$	record $(1 \leqslant j)$
$\underline{e}.l_i$	field
x	variable
$\lambda x.\underline{e}$	stripped function (x binds in e)
$\underline{e}\ \underline{e}$	application
$\mathbf{mar}\,(\underline{e} : \underline{T})$	dynamic
$\mathbf{marshalled}\,(\underline{e} : \underline{T})$	closed, colour-independent dynamic
$\mathbf{unmar}\ \underline{e} : \underline{T}$	undynamic
$!\,\underline{e}$	send
$?$	receive
$U.\texttt{term}$	term-part of a module
$\mathbf{Unmarfailure}^{\underline{T}}$	undyn failure

$\underline{T}$ ::=	stripped type
...	grammar similar to T
$\underline{h}.\texttt{type}$	stripped hash type
...	

$\underline{h}$ ::=$\mathbf{hash}(N, [\underline{T}, \underline{v} : \underline{T}] : [X : \mathbf{Le}(\underline{T}), \underline{T}])$	stripped hash

$\underline{v}$ ::=	stripped value
$()$	unit
$(\underline{v}_1, ..., \underline{v}_j)$	tuple $(2 \leqslant j)$
$\{l_1 = \underline{v}_1, ..., l_j = \underline{v}_j\}$	record $(1 \leqslant j)$
$\lambda x.\underline{e}$	function (x binds in e)
$\mathbf{marshalled}\,(\underline{e} : \underline{T})$	closed dynamic value

$C ::=$	single-level evaluation context
$(\underline{v}_1, .., \underline{v}_{i-1}, \_, \underline{e}_{i+1}, .., \underline{e}_j)$	tuple ($2 \leqslant j$ and $1 \leqslant i \leqslant j$)
$\mathbf{proj}_i \_$	projection
$\{l_1 = \underline{v}_1, ..., l_{i-1} = \underline{v}_{i-1}, l_i = \_,$ $l_{i+1} = \underline{e}_{i+1}, ..., l_j = \underline{e}_j\}$	tuple ($1 \leqslant j$ and $1 \leqslant i \leqslant j$)
$\_.l_i$	projection
$\_\ \underline{e}$	application left
$\underline{v}\ \_$	application right
$\mathbf{mar}\,(\_ : \underline{T})$	dynamic
$\mathbf{unmar}\ \_ : \underline{T}$	undynamic
$!\,\_$	send
$CC ::=$	stripped coloured evaluation context
$CC.C$	extra level
$\_$	identity
$\underline{\kappa} ::=$	elements of the subhash relationship
U	module name
$\underline{h}$	hash
$\underline{H} ::=$	subhash relationship
$\mathbf{nil}$	empty relationship
$\underline{H} \cup \underline{\kappa} <: \underline{\kappa}'$	addition of a statement
$\underline{n} ::=$	stripped network
$\mathbf{0}$	null
$\underline{n} \mid \underline{n}$	parallel composition
$\underline{H}, \underline{e}$	expression and subhash on one machine

Definition B.118 (stripped reductions) Define $\underline{H}, \underline{e} \rightarrow_{\text{nb}} \underline{H}', \underline{e}'$ on stripped expressions as given by the (ered.*) rules other than (ered.col.*) and (ered.sub.*), with (ered.unmar) modified not to introduce brackets and subtyping annotations : the right-hand side, in the case where $\mathbf{nil} \vdash_c^{H \cup H'} T <: T'$, becomes only $\underline{e}$. We also strip the type annotation present on λ and the colour annotation on **marshalled**.

Also define $\underline{n} \rightarrow_{\text{nb}} \underline{n}'$ for networks in the obvious way.

Definition B.119 (erasure relation) We have erase (φ, ψ) if φ is any syntactic entity and ψ is φ with all brackets outside hashes, explicit subtyping annotations, λ type annotations, and **marshalled** colour annotation erased, and records potentially extended by garbage fields linking to stripped values. In particular :

- erase $([e]_c^T, \underline{e})$ if erase $(e, \underline{e})$
- erase $(\ _{(T <: T')} e, \underline{e})$ if erase $(e, \underline{e})$
- erase $(\lambda x : T.e, \lambda x.\underline{e})$ if erase $(e, \underline{e})$
- erase $(\mathbf{marshalled}\ _{c,H}(e : T), \mathbf{marshalled}\,(\underline{e} : \underline{T}))$ if erase $(e, \underline{e})$ and erase $(T, \underline{T})$
- erase $(\{l_1 = e_1; ...; l_i = e_i\}, \{l_1 = \underline{e}_1; ...; l_j = \underline{e}_j\})$ if $i \leqslant j$ and for $1 \leqslant k \leqslant i$,

erase $(e_k, \underline{e}_k)$ and for $i+1 \leqslant k' \leqslant j$, $\underline{e}_{k'}$ is a stripped value.

In a similar way, we can define a function strip such that if φ is any syntactic entity, then strip (φ) is φ with all brackets outside hashes, explicit subtyping annotations, λ typing annotations and **marshalled** colour annotation erased. Then we have erase $(\varphi, \text{strip}\,(\varphi))$.

Lemma B.120 (erasure preservation by bracket and subtyping reduction) If e is a well-typed expression under c and H and $\underline{e}$ is such that erase $(e, \underline{e})$, and if e can be reduced by several (ered.col.*) or (ered.sub.*) rules to an expression e', then erase $(e', \underline{e})$.

Note also that, by Theorem B.99 (type preservation for expression reduction), e' is well-typed.

Proof. We induct on the length of the derivations, the case where no reduction happens being trivial.

Let e_0 be the result of the application of the first reduction rule. If the reduction rule is (ered.sub.record), then the record in the redex of e has more fields than the result in e_0. We use in this case the part of the definition of erase concerning the extension of records with extra fields.

If the reduction corresponds to a different rule, then the erase relation is the same for e and e_0.

In all cases, erase $(e_0, \underline{e})$. We conclude by the induction hypothesis. □

Lemma B.121 (progress and determinism of stripped expression reduction) If e is a well-typed expression under c and H and $\underline{e}$ is such that erase $(e, \underline{e})$, then exactly one of the following cases holds :

- $\underline{e}$ is a stripped value ;
- $\underline{e}$ is dormant, i.e. is $\mathbf{Unmarfailure}^T$ or a communication in a stripped evalution context ;
- $\underline{e}$ reduces by $\rightarrow_{\text{nb}}$ under H ; moreover there is exactly one rule with one redex applicable.

Note also that if e is a c-value, then $\underline{e}$ is a stripped value.

Proof. By induction on the number of brackets and subtyping reduction rules ((ered.col.*) or (ered.sub.*) rules) that can consecutively reduce e.

By Theorem B.108 (progress of expressions) and Theorem B.113 (determinism of expression reduction), we know that there are three cases for e.

If e can be reduced, it is only by one rule at one redex. If the rule is a bracket or subtyping reduction rule, then by Lemma B.120 (erasure preservation by bracket and subtyping reduction), we can apply the induction hypothesis. If the rule is a different rule, then $\underline{e}$ can be reduced by the corresponding $\rightarrow_{\text{nb}}$ rule : we verify in particular that a greater number of fields in records does not prevent any reduction.

If e is a value, then $\underline{e}$ is a stripped value (records only add fields with values).

If e is dormant, then $\underline{e}$ is also dormant. □

Theorem B.122 (erasure preserves expression reduction outcomes) If $\mathbf{nil} \vdash_c^H e : T$ and $H, e \rightarrow_c H', e'$ and if a stripped expression $\underline{e}$ and subhash relation $\underline{H}$ are such that erase $(e, \underline{e})$ and erase $(H, \underline{H})$, then there exists a $\underline{e}'$ such that $\underline{H}, \underline{e} \rightarrow_{\text{nb}}^? \underline{H}', \underline{e}'$ and erase $(e', \underline{e}')$ and erase $(H', \underline{H}')$.

Note that we write $\rightarrow_{\text{nb}}^?$ for at most one $\rightarrow_{\text{nb}}$ reduction.

Proof. We induct on the derivation of the reduction.

If $H, e \to_c H', e'$ by (ered.col.*) or (ered.sub.*) : Then we take $\underline{e'} = \underline{e}$ and $\underline{H'} = \underline{H}$ and do not use any reduction rule. We conclude by using Lemma B.120 (erasure preservation by bracket and subtyping reduction).

If $H, e \to_c H', e'$ by another reduction rule : Then the corresponding rule applies to $\underline{H}$ and $\underline{e}$ to produce $\underline{H'}$ and $\underline{e'}$ and no rule breaks the erasure preservation : erase $(e', \underline{e'})$ and erase $(H', \underline{H'})$.

If $H, e \to_c H', e'$ by (ered.cong) : By induction.

□

Theorem B.123 (erasure preserves reduction outcomes) If $\vdash n\,\mathrm{ok}$ and $n \to_n n'$ and there is a $\underline{n}$ such that erase $(n, \underline{n})$, then there exists a $\underline{n'}$ such that $\underline{n} \to_{\mathrm{nb}}^{?} \underline{n'}$ and erase $(n', \underline{n'})$.

Proof. Follows from Theorem B.122 (erasure preserves expression reduction outcomes). □

Theorem B.124 (erasure does not add expression reduction outcomes) If $\mathbf{nil} \vdash_c^H e : T$ and there is a $\underline{e}$ and a $\underline{H}$ such that erase $(e, \underline{e})$ and erase $(H, \underline{H})$ and if $\underline{H}, \underline{e} \to_{\mathrm{nb}} \underline{H'}, \underline{e'}$ then there exists e' and H' such that erase $(e', \underline{e'})$ and erase $(H', \underline{H'})$ and $H, e \to_c^+ H', e'$.

Note that we write $\to_c^+$ for at least one $\to_c$ reduction.

Proof. By Theorem B.108 (progress of expressions), we know that e can be in one of the three following cases :

Case e is an c-value : Then $\underline{e}$ is a stripped value, so by Lemma B.121 (progress and determinism of stripped expression reduction) does not reduce by $\to_{\mathrm{nb}}$, a contradiction.

Case e is dormant : Then $\underline{e}$ is a stripped dormant expression, so by Lemma B.121 (progress and determinism of stripped expression reduction) does not reduce by $\to_{\mathrm{nb}}$, a contradiction.

Case e reduces by a bracket or subtyping reduction rule : Then by Lemma B.120 (erasure preservation by bracket and subtyping reduction) and Theorem B.122 (erasure preserves expression reduction outcomes), we know that there exists an expression e_0 such that erase $(e_0, \underline{e})$ and that e_0 the next possible reduction is not a bracket or subtyping reduction rule (we apply the bracket and subtyping reduction rules as much as possible). By Theorem B.99 (type preservation for expression reduction), we also know that e_0 is well-typed under c and H. e_0 is then in one of the three other cases.

Case e reduces by another rule : Therefore there exists e' and H' such that $H, e \to_c H', e'$. By Lemma B.121 (progress and determinism of stripped expression reduction), we can apply Theorem B.122 (erasure preserves expression reduction outcomes) to know that the only possible corresponding reduction for $\underline{e}$ is $\underline{H}, \underline{e} \to_{\mathrm{nb}} \underline{H'}, \underline{e'}$ and we have erase $(e', \underline{e'})$ and erase $(H', \underline{H'})$.

□

Theorem B.125 (erasure does not add reduction outcomes) If $\vdash n\,\mathrm{ok}$ and there exists a $\underline{n}$ such that erase $(n, \underline{n})$ and $\underline{n} \to_{\mathrm{nb}} \underline{n'}$ then there exists n' such that erase $(n', \underline{n'})$ and $n \to_n^+ n'$.

This means that the erasure of all annotations does not prevent the existence of a reduction strategy that is very similar to the original, type preserving, one. $\rightarrow_{\text{nb}}$ represents a possible implementation strategy.

Proof. Follows from Theorem B.124 (erasure does not add expression reduction outcomes). □

Annexe C

Secure sessions : libraries and example

C.1 Symbolic code for the libraries

In this appendix we provide the symbolic implementations for the libraries described in Section 3.6. To bridge the gap between our formal syntax and the Ocaml syntax we used for our presentation, we rely on syntactic sugar : if ... then ... else is a shortcut for standard pattern-matching on the result of the test ; the semi-colon, which expresses sequentiality, can be written with our let construct ; function application where arguments are not values can be unfolded using let bindings ; and anonymous functions introduced with fun can be replaced by a freshly named let binding for the function body.

The code presented here is the one mentionned in the theorems statements. The only important difference with the libraries presented earlier regards the anti-replay cache : its check is done in the `psend` function instead of the (more intuitive) `precv` as it allows seamlessly all the principal to maintain a local cache.

Symbolic code for the Data and Crypto libraries We first list the `Crypto` library, which implements cryptographic types as algebraic datatypes :

```
type keybytes = SKey of name | VKey of keybytes
type bytes = Nonce of name
           | Hash of bytes
           | Concat of bytes * bytes
           | Sign of bytes * keybytes
           | Utf8 of string

let nonce (n: name) : bytes = Nonce n
let genskey (n: name) : keybytes = SKey n
let genvkey (n:keybytes) : keybytes =
  match n with
    | SKey _  → VKey n
let hash (b:bytes) : bytes = Hash b
let concat (m1 : bytes) (m2 : bytes) : bytes = Concat (m1, m2)
let sign (m : bytes) (k : keybytes) : bytes = Sign (m, k)
let verify (m : bytes) (s : bytes) (k : keybytes) : bool =
  match s with
    | Sign (mm, sk)  →
        if k = VKey sk && mm = m then true else false
    | _  → 0
```

```
let iconcat (m : bytes ) : (bytes * bytes) =
  match m with
    | Concat (m1, m2)  → (m1, m2)
let utf8 (s:string) = Utf8 s
let iutf8 (m: bytes) = match m with | Utf8 m1  → m1
```

Symbolic code for the Prins library In our model, the implementation of the `Prins` library is parameterized by a finite list of principals and a safety predicate on those principals : we do not model the registration process in our formalism. (In contrast, our concrete prototype implementation retrieves cryptographic materials from a partial database and does not serve the opponent !)

```
let prins = ... (* a fixed list of all principals *)
let safe (a:principal) = ... (* a fixed predicate on principals *)
let skeys = List.map (fun a  → (a, genskey (new()))) prins
let skey (a : principal) = List.assoc a skeys
let vkey (a : principal) = genvkey (skey a)
let chans = List.map
   (fun a  → let (n:name) = new() in (a, n)) prins

type cache_contents = (bytes * int) list
type cache_result = Stale | Fresh of cache_contents

let asend m a = fork (fun ()  → send a m)
let caches = List.map
   (fun a  → let (n:name) = new() in (a, n)) prins
let _ = map (asend []) caches (* caches init *)

let header s =
  let (msg, sigs) = iconcat (ibase64 s) in
  let (joinflag,header,payload) = iconcat3 (msg) in
  let (host2, dest2, sid) = iconcat3 header in
  let join = if (iS (iutf8 joinflag)) = "J" then true else false in
    ((int_of_string (iS (iutf8 dest2)),sid),join)

let antireplay old a msg =
  let ((sid, r) as k), joining = header message in
  if joining then
    if List.mem k old then Stale
    else Fresh(k::old)
  else Fresh(old)

let psend (a : principal) (m : bytes) =
  let ch = List.assoc a chans in
  let cache = List.assoc a caches in
  let oldcache = recv cache in
  let r = antireplay oldcache a m in
  match r with
  | Fresh(newcache)  → asend cache newcache; send ch m
  | Stale  → asend cache oldcache

let precv (a : principal) = recv (List.assoc a chans)

(* for modelling the opponent's knowledge only: *)
let psend• = new()
```

```
let rec forward () =
  let a,m = recv psend• in
  fork forward; psend a m in
fork forward
let chans• = List.filter (fun (a,n) → not (safe a)) chans
let skeys• = List.filter (fun (a,k) → not (safe a)) skeys
```

The `psend•` channel implements a small server that receives requests to call `psend`; this enables the opponent to send messages to safe principals without having a direct access to `psend`.

The opponent is given access to `prins`, `safe`, `vkey`, `psend•`, `chans•`, and `skeys•`. Our generated protocol implementations access `safe`, `skey`, `vkey`, `psend`, and `precv`. User code is given access only to principal constants.

C.2 Conference session

Session `Conf` is defined as follows :

```
session Conf =
  role pc =
    send Cfp:string ;
    start:
    recv [ Paper:string →
           send ( Close;
                  recv Done →
                  discuss:
                  send ( Accept:string ;
                         recv FinalVersion:string
                       + Reject
                       + Shepherd:string ;
                         recv Rebuttal:string →
                         discuss )
                + ReqRevision:string; start )
         | Retract]
  role author : string =
    recv Cfp:string →
    start:
    send Upload:string ;
    recv [ BadFormat:string → start
         | Ok → subm :
           send (Submit:string;
                 discuss :
                 recv [ Accept:string →
                        send FinalVersion:string
                      | Reject
                      | Shepherd:string →
                        send Rebuttal:string ; discuss
                      | Revise → subm ]
                  + Withdraw) ]
  role confman =
    start :
    recv Upload:string →
    send ( Ok;
           subm:
           recv [ Submit:string →
```

```
                send Paper:string;
                recv [
                | Close  → send Done
                | ReqRevision:string  →
                  send Revise ; subm
                ]
              | Withdraw  → send Retract
              ]
        + BadFormat:string;
          start)
```

C.3 Legislative session

Session `Loi` is defined as follows :

```
session Loi =
  role ministre =
    send ProjetLoi : string ;
    recv ConseilMinistre : string  →
    conseildesministres:
    send Expose : string ;
    recv
    [ Questions : string →  conseildesministres
    | Adopte  → send PassageCommission : string ]
  role conseildetat =
    avis:
    recv Soumission : string  →
    send
    ( AvisPositif
    + AvisNegatif :string ; avis
    )
  role gouvernement =
    recv ProjetLoi : string  →
    conseil:
    send Soumission : string ;
    recv
    [ AvisNegatif : string  → conseil
    | AvisPositif  →
      send ConseilMinistre : string ;
      conseildesministres:
      recv Expose : string  →
      send
      ( Questions : string ; conseildesministres
      + Adopte ;
        seance:
        recv
        [ Audition  → send Reponse : string ; seance
        | TexteFinal
        | Echec
        ]
      )
    ]
  role commission =
    recv PassageCommission : string  →
    send TextePropose : string
```

```
role assemblee =
  saisie:
  recv TextePropose : string →
  depart:
  send DebutSeance : string ;
  seance:
  recv
  [ AuditionGouv → send Audition ;
    recv Reponse : string →
    send Parole ; seance
  | Exception →
    send VoteException ;
    recv
    [ ExceptionRejetee → depart
    | ExceptionAcceptee → fin
    ]
  | Amendement : string →
    send VoteAmendement ;
    recv
    [ AmendementRejete → depart
    | AmendementAccepte : string → depart
    ]
  | Texte : string →
    send Vote ;
    recv
    [ Approuve → send TexteFinal
    | Rejete → fin: send Echec
    ]
  ]
role depute =
  depart:
  recv DebutSeance : string →
  seance:
  send
  ( AuditionGouv ; recv Parole → seance
  + Exception ;
    recv VoteException →
    send
    ( ExceptionRejetee ; depart
    + ExceptionAcceptee
    )
  + Amendement : string ;
    recv VoteAmendement →
    send
    ( AmendementRejete ; depart
    + AmendementAccepte : string ; depart
    )
  + Texte : string ;
    recv Vote →
    send
    ( Approuve ;
    + Rejete
    )
  )
```

C.4 Example code

We present in detail the code for the session `Shopping` from example 3.7.

We first recall the session definition and graphs, then we give a excerpt of the generated interface and the `client` user code that drives the session.

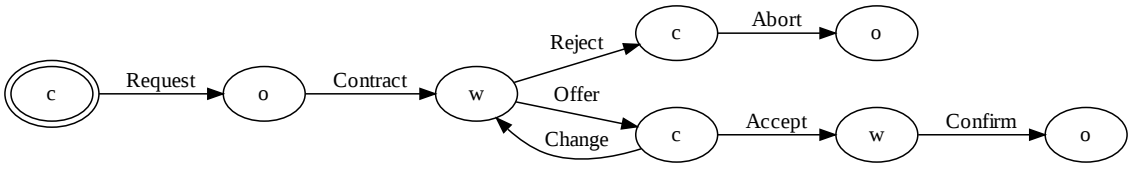
Syntactic session Session `Shopping` is defined as follows :

```
session Shopping =
  role c:unit =
   send Request:string;
   start:
   recv
   [ Offer:string  →
      send ( Change:string; start
           + Accept:unit )
   |  Reject:string  → send Abort:unit
   ]

  role o:unit =
    recv Request:string  →
    send Contract:string ;
    recv
    [ Confirm:unit
    | Abort:unit
    ]

  role w:string =
    recv Contract:string  →
    loop:
    send
    ( Offer:string ;
      recv
      [ Change:string  → loop
      |  Accept:unit  → send Confirm:unit
      ]
    +  Reject:string )
```

Graph



Generated interface and user code The user code drives the session for the roles. We give here the programming interface of the client `c`.

```
(* Function for role c *)

type result_c = unit

type msg0 =
   Request of (string * msg1)
and msg1 = {
```

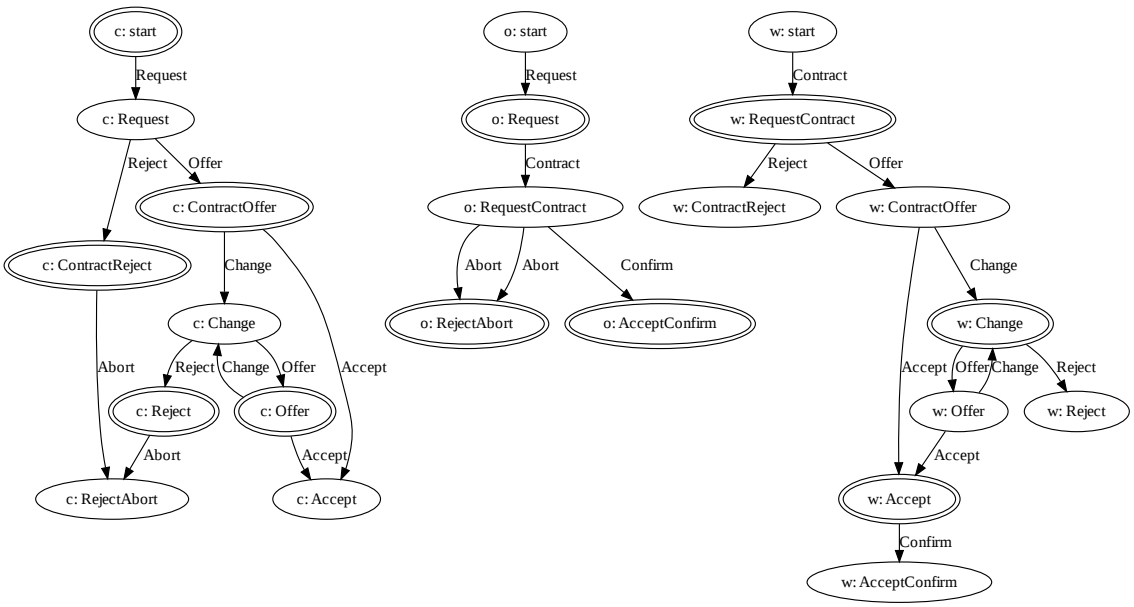
```
   hOffer : (prins * string  → msg2) ;
   hReject : (prins * string  → msg4)}
and msg2 =
   Change of (string * msg1)
 | Accept of (unit * result_c)
and msg4 =
   Abort of (unit * result_c)

val c : prins  → msg0  → result_c
```

Now we give the user code for the client role (negotiating a delivery).

```
type choiceOnOffer = UChange of string | UAccept

let start_client () =
  let offer_ui offer =
    if offer = "Redmond, 8am-9am"
    then UChange "Cambridge"
    else UAccept in
  let prins = {
    prins_c = "alice";
    prins_o = "charlie";
    prins_w = "bob"; } in
  let r = "12 March 2007" in
  let rec msg1 = {
      hReject = (fun (_,s)  → printf "%s\n" s;Abort((),()));
      hOffer = (fun (_,offer)  →
        match offer_ui offer with
          | UChange location  → Change(location,msg1)
          | UAccept  → Accept((),()))} in
  let msg0 = Request(r,msg1) in
  let worker () =
    Shopping.c prins msg0;
    printf "Client: session complete.\n\n" in
  worker()
```



Implementation graph

Generated code excerpts The generated code for session `Shopping` is as follows (we only show one example of a sending of `Request` and receiving of `Reject`) :

```
let sendWired_Request_c_start = fun x → fun store →
  let empty_str = cS "" in
  let nil = utf8 empty_str in
  let ts = store.header.ts + 1 in
  let store = {
      prins = store.prins;
      macs = store.macs;
      keys = store.keys;
      header = {store.header with ts = ts; }} in
  let ts_string = string_of_int store.header.ts in
  let ts_str = cS ts_string in
  let ts_mar = utf8 ts_str in
  let prins = nil in
  let string_w = cS store.prins.prins_w in
  let mar_w = utf8 string_w in
  let prins = concat mar_w prins in
  let string_o = cS store.prins.prins_o in
  let mar_o = utf8 string_o in
  let prins = concat mar_o prins in
  let string_c = cS store.prins.prins_c in
  let mar_c = utf8 string_c in
  let prins = concat mar_c prins in
  let header = concat store.header.sid ts_mar in
  let start = concat header prins in
  let keys = nil in
  let keycw = gen_keys (cS store.prins.prins_c) (cS store.prins.prins_w) in
  let keys = concat keycw keys in
  let keyco = gen_keys (cS store.prins.prins_c) (cS store.prins.prins_o) in
  let keys = concat keyco keys in
  (* Generation of a MAC from state c_start to role o *)
  let content = content_c_Request store.header.ts store in
  let mackeyco = get_mackey store.prins.prins_c store.prins.prins_o in
  let macmsg = mac mackeyco (pickle content) in
  let mac_oRequest = concat header macmsg in
  (* Generation of a MAC from state c_start to role w *)
  let content = content_c_Request store.header.ts store in
  let mackeycw = get_mackey store.prins.prins_c store.prins.prins_w in
  let macmsg = mac mackeycw (pickle content) in
  let mac_wRequest = concat header macmsg in
  let store = {
      prins = store.prins;
      macs = {store.macs with mac_oRequest = mac_oRequest;
                                             mac_wRequest = mac_wRequest; };
      keys = store.keys;
      header = store.header} in
  let macs = nil in
  let macs = concat store.macs.mac_wRequest macs in
  let macs = concat store.macs.mac_oRequest macs in
  let value = utf8 (cS x) in
  let protocol = concat macs keys in
  let payload = concat value protocol in
  let visib = cS "c_Request__" in
  let visib_string = utf8 visib in
```

```
  let  content = concat visib_string payload in
  let  msg = base64 (concat start content) in
  let  () = psend store.prins.prins_o msg in
  store
```

```
let  receiveWired_c_Change : store  → wired_c_Change = fun store  →
  let  msg = precv store.prins.prins_c in
  let  header,content = iconcat (ibase64 msg) in
  let  sid,ts_mar = iconcat header in
  let  ts_str = iutf8 ts_mar in
  let  ts_string = iS ts_str in
  let  ts = int_of_string ts_string in
  let  oldts = store.header.ts in
  let  _ = test_inf oldts ts "Replay attack!" in
  let  _ = test_eq store.header.sid sid "Session confusion attack!" in
  let  store = {
      prins = store.prins;
      macs = store.macs;
      keys = store.keys;
      header = {store.header with ts = ts; }} in
  let  tag,payload = iconcat content in
  match iS (iutf8 tag) with
  | "w_Reject__"  →
  let  value,protocol = iconcat payload in
  let  macs,keys = iconcat protocol in
  (* Unmarshalling value *)
  let  x = iS (iutf8 value) in
  (* Unmarshalling keys *)
  let  key_wo,_ = iconcat keys in
  (* Unmarshalling MACs *)
  let  mac_cReject,macs = iconcat macs in
  let  mac_oReject,_ = iconcat macs in
  let  store = {
      prins = store.prins;
      macs = {store.macs with mac_cReject = mac_cReject;
                                            mac_oReject = mac_oReject; };
      keys = {store.keys with key_wo = key_wo; };
      header = store.header} in
  (* Verification of a MAC from state w_Reject*)
  let  macheader,maccontent = iconcat store.macs.mac_cReject in
  let  macsid,ts_mar = iconcat macheader in
  let  ts_str = iutf8 ts_mar in
  let  ts_string = iS ts_str in
  let  ts = int_of_string ts_string in
  let  _ = test_inf oldts ts "MAC verification" in
  let  oldts = ts in
  let  _ = test_eq macsid store.header.sid "MAC verification" in
  let  mackeywc = get_mackey store.prins.prins_w store.prins.prins_c in
  let  _ = mac_verify_c_w_Reject ts store mackeywc maccontent in
  let  _ = test_eq oldts store.header.ts "Time-stamp verification" in
  (* Verification Ended *)
    let  wired = Wired_in_c_Change_of_w_Reject__(x,store) in
    wired
  | "w_Offer__"  → ...
```

Auxiliary functions are of the form :

```
let content_w_Reject = fun ts store →
  let nextstate = cS "w_Reject" in
  let nextstate_string = utf8 nextstate in
  let ts_string = string_of_int ts in
  let ts_str = cS ts_string in
  let ts_mar = utf8 ts_str in
  let header = concat store.header.sid ts_mar in
  let content = concat header nextstate_string in
  content
```

```
let mac_verify_o_w_Reject = fun ts store k m →
  let content = content_w_Reject ts store in
  let up = mac_verify k m content in
  let _ = test_eq up content "MAC incorrect" in
  up
```

Symbolic run Finally, we report on the execution of the above code for session Shopping :

```
[impl] Executing role c ...
[impl] Executing role w ...
[impl] Executing role o ...
[Prins] Generating shared keys for alice and bob: 'nonce1'
[Prins] Encrypted key: RSA-Enc{'bob.pub'}['nonce1']
                    | RSA-SHA1{'alice.key'}[RSA-Enc{'bob.pub'}['nonce1']]
[Prins] Generating shared keys for alice and charlie: 'nonce2'
[Prins] Encrypted key: RSA-Enc{'charlie.pub'}['nonce2']
                    | RSA-SHA1{'alice.key'}[RSA-Enc{'charlie.pub'}['nonce2']]
[Prins] Looking for shared mac key between alice and charlie
[Prins] Found 'nonce2'
[Prins] Looking for shared mac key between alice and bob
[Prins] Found 'nonce1'
[prins] sent to charlie: Initial message nb 1 'c_Request__':
  Principals = '"alice"','"charlie"','"bob"'
  Payload = 12 March 2007
  Keys = RSA-Enc{'charlie.pub'}['nonce2'],RSA-Enc{'bob.pub'}['nonce1']
  MACS =
   - Mac{nonce2} of message c_Request (sid: [SHA1('nonce0' |...], ts: 1),
   - Mac{nonce1} of message c_Request (sid: [SHA1('nonce0' |...], ts: 1)
[prins] charlie received the message!
[Prins] Shared keys for alice and charlie already registered.
[Prins] Looking for shared mac key between alice and charlie
[Prins] Found 'nonce2'
[crypto] Verifying MAC: Mac{nonce2} of message c_Request (ts: 1)
[crypto] against    : Mac{nonce2} of message c_Request (ts: 1)
[Prins] Generating shared keys for charlie and alice: 'nonce3'
[Prins] Encrypted key: RSA-Enc{'alice.pub'}['nonce3']
                    | RSA-SHA1{'charlie.key'}[RSA-Enc{'alice.pub'}['nonce3']]
[Prins] Generating shared keys for charlie and bob: 'nonce4'
[Prins] Encrypted key: RSA-Enc{'bob.pub'}['nonce4']
                    | RSA-SHA1{'charlie.key'}[RSA-Enc{'bob.pub'}['nonce4']]
[Prins] Looking for shared mac key between charlie and bob
[Prins] Found 'nonce4'
[Prins] Looking for shared mac key between charlie and alice
[Prins] Found 'nonce3'
[prins] sent to bob: Initial message nb 2 'o_Contract_Request__c_Request__':
```

```
  Principals = '"alice"','"charlie"','"bob"'
  Payload = 12 March 2007
  Keys = RSA-Enc{'bob.pub'}['nonce4'],
         RSA-Enc{'bob.pub'}['nonce1'],
         RSA-Enc{'alice.pub'}['nonce3']
  MACS =
   - Mac{nonce4} of message o_Contract_Request
                                    (sid: [SHA1('nonce0' |...], ts: 2),
   - Mac{nonce3} of message o_Contract_Request
                                    (sid: [SHA1('nonce0' |...], ts: 2),
   - Mac{nonce1} of message c_Request (sid: [SHA1('nonce0' |...], ts: 1)
[prins] bob received the message!
[Prins] Shared keys for charlie and bob already registered.
[Prins] Shared keys for alice and bob already registered.
[Prins] Looking for shared mac key between alice and bob
[Prins] Found 'nonce1'
[crypto] Verifying MAC: Mac{nonce1} of message c_Request (ts: 1)
[crypto] against    : Mac{nonce1} of message c_Request (ts: 1)
[Prins] Looking for shared mac key between charlie and bob
[Prins] Found 'nonce4'
[crypto] Verifying MAC: Mac{nonce4} of message o_Contract_Request (ts: 2)
[crypto] against    : Mac{nonce4} of message o_Contract_Request (ts: 2)
Server: session starting for 12 March 2007.
[Prins] Generating shared keys for bob and alice: 'nonce5'
[Prins] Encrypted key: RSA-Enc{'alice.pub'}['nonce5']
                    | RSA-SHA1{'bob.key'}[RSA-Enc{'alice.pub'}['nonce5']]
[Prins] Looking for shared mac key between bob and alice
[Prins] Found 'nonce5'
[prins] sent to alice: Message nb 3 'w_Offer_Contract__o_Contract_Request__':
  Payload = Paris, 8am-9am
  Keys = RSA-Enc{'alice.pub'}['nonce5'],RSA-Enc{'alice.pub'}['nonce3']
  MACS =
   - Mac{nonce3} of message o_Contract_Request
                                     (sid: [SHA1('nonce0' |...], ts: 2),
   - Mac{nonce5} of message w_Offer_Contract (sid: [SHA1('nonce0' |...], ts: 3)
[prins] alice received the message!
[Prins] Shared keys for bob and alice already registered.
[Prins] Shared keys for charlie and alice already registered.
[Prins] Looking for shared mac key between charlie and alice
[Prins] Found 'nonce3'
[crypto] Verifying MAC: Mac{nonce3} of message o_Contract_Request (ts: 2)
[crypto] against    : Mac{nonce3} of message o_Contract_Request (ts: 2)
[Prins] Looking for shared mac key between bob and alice
[Prins] Found 'nonce5'
[crypto] Verifying MAC: Mac{nonce5} of message w_Offer_Contract (ts: 3)
[crypto] against    : Mac{nonce5} of message w_Offer_Contract (ts: 3)
[Prins] Looking for shared mac key between alice and bob
[Prins] Found 'nonce1'
[Prins] Looking for shared mac key between alice and charlie
[Prins] Found 'nonce2'
[prins] sent to bob: Message nb 4 'c_Accept__':
  Payload =
  Keys =
  MACS =
   - Mac{nonce1} of message c_Accept (sid: [SHA1('nonce0' |...], ts: 4),
   - Mac{nonce2} of message c_Accept (sid: [SHA1('nonce0' |...], ts: 4)
```

```
Client: session complete.

[prins] bob received the message!
[Prins] Looking for shared mac key between alice and bob
[Prins] Found 'nonce1'
[crypto] Verifying MAC: Mac{nonce1} of message c_Accept (ts: 4)
[crypto] against    : Mac{nonce1} of message c_Accept (ts: 4)
[Prins] Generating shared keys for bob and charlie: 'nonce6'
[Prins] Encrypted key: RSA-Enc{'charlie.pub'}['nonce6']
                     | RSA-SHA1{'bob.key'}[RSA-Enc{'charlie.pub'}['nonce6']]
[Prins] Looking for shared mac key between bob and charlie
[Prins] Found 'nonce6'
[prins] sent to charlie: Message nb 5 'w_Confirm_Accept__c_Accept__':
  Payload =
  Keys = RSA-Enc{'charlie.pub'}['nonce6']
  MACS =
   - Mac{nonce2} of message c_Accept (sid: [SHA1('nonce0' |...], ts: 4),
   - Mac{nonce6} of message w_Confirm_Accept
                                (sid: [SHA1('nonce0' |...], ts: 5)
Store: Done! in Paris, 8am-9am.
[prins] charlie received the message!
[Prins] Shared keys for bob and charlie already registered.
[Prins] Looking for shared mac key between alice and charlie
[Prins] Found 'nonce2'
[crypto] Verifying MAC: Mac{nonce2} of message c_Accept (ts: 4)
[crypto] against    : Mac{nonce2} of message c_Accept (ts: 4)
[Prins] Looking for shared mac key between bob and charlie
[Prins] Found 'nonce6'
[crypto] Verifying MAC: Mac{nonce6} of message w_Confirm_Accept (ts: 5)
[crypto] against    : Mac{nonce6} of message w_Confirm_Accept (ts: 5)

Office: run confirmed.
```

Annexe D

Secure sessions : theorems and proofs

This annex shares its content with [7].

We first describe series of low-level transitions that implement each high-level transition, which essentially provides the proof of Theorem 3.6. We then use this description as the basis of the case analysis in the proof of Theorem 3.5. We finally prove Lemma 3.4 and obtain Theorem 3.2 as a corollary of Theorem 3.5.

In order to simplify the writing of the proofs, we adopt the following conventions :

- let `init` be the shorter name of `empty_store` ;
- let `gen_`$\widetilde{g}$`_`g be a meta-function building the message g that is sent from state $\widetilde{g}$;

  val `gen_`$\widetilde{g}$`_`f`: localstore * `$payload(f)$ `→ bytes`
- let `verify_`$\widetilde{g}f$ be a meta-function verifying the session id and time-stamp of the message f received in state $\widetilde{g}$;
- let `check_`$\widetilde{g}$`_`$\widetilde{g}'$ be a meta-function verifying the signature received in state $\widetilde{g}$ from a sender reaching state $\widetilde{g}'$ (the visibility sequence is $\widetilde{g}'$).

  val `check_`$\widetilde{g}$`_`$\widetilde{g}'$ `: localstore * bytes →  localstore *` $payload(\widetilde{g}')$

For any path in the graph, there is a single active role r, which can send a message to a role r' with label selected from a set $\mathcal{F}$ that collects the possible outgoing labels at this particular node. With those notations, the generated code is reduced to :

```
for all sending states g̃ with corresponding roles r, r', F:
[[let rec|and] send_g̃ st msg = match msg with
   for each f ∈ F: [ | f(v,w)  →
       let  a' = st.peers.r' in let m = gen_g̃_f st v in psend a' m ;
       if the next node is terminal :  w else :  recv_g̃f st w]]
for all reachable receiving states g̃ with r, r', F and for each f ∈ F:
[and recv_g̃ st w = let a = st.peers.r in
   let  m = precv a in verify_g̃f st m w
 and verify_g̃ st m w = let path = visible_g̃ m in
   match path with
    for each g̃' visible from g̃:
   [ | t_g̃'  → let st,payload = chk_g̃_g̃' st m in
   if the next node is terminal :  w.last(g̃') st.peers payload
   else :  let next = w.last(g̃') st.peers payload in
       send_g̃+g̃' st next ]]
```

D.1 Proof of the completeness theorem

Lemma D.1 (Correctness) Let W be a valid implementation of H. For every transition $H \xrightarrow{\alpha}_{\mathsf{K}} H'$ in F+S, where α does not contain any signing key of a safe principal nor any element of $\mathcal{N}$, there exists a valid implementation W' of H such that $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F.

Proof. Except for the transition steps that involve sessions, every high-level step carries over to a low-level step with the same label, as their redexes are preserved by the translation. We distinguish seven kinds of high-level session transitions, depending on the base rules they use : rules KSendS (using Init or Step), KRecvS (using Join or Step), and KStep (using CommL, CommR and EndS).

Unfolding the function definitions given in Section 3.7 and Appendix C.1, we exhibit a series of low-level transitions that implement these high-level session transitions, leading to a valid implementation of H' for some possibly updated state T' .

We first set up auxiliary notations. We write the elements of high-level configurations without subscripts : $H = K, \rho, P$ or $H' = K', \rho', P'$; we use subscripts for the elements of low-level configurations : $W_0 = K_0, \rho_0, P_0$ or $W_1 = K_1, \rho_1, P_1$. We also use italics for metavariables.

Cache transitions We begin with auxiliary low-level transitions for filtering messages through the anti-replay cache. We write $W_1 = K_1, \rho_1, P_1 \mid E_1$ for the low-level adversary knowledge, environment, processes and expression context. For a given safe principal a, we let $P_a =$ send $cache_a$ $content_a$ abbreviate the message that holds the state of the cache for principal a on channel $cache_a$: hence, $content_a$ is the list of session identifiers and roles already seen by a, recorded in $T.cache(a)$.

Starting from a message m sent to a in expression context E_1, under the premise that '`header` m' evaluates either to '`(`sid`,`$r`)`,true' with sid,$r \notin content_a$ or to '`_`,false ', by definition of `psend` we have transitions :

$$
\begin{aligned}
&K_1, \rho_1, P_1 \mid P_a \mid E_1[\text{psend } a\ m] \\
&\xrightarrow{\textsc{Apply}(psend)} \xrightarrow{\textsc{Apply,LetVal}}{}^{*} \\
&K_1, \rho_1, P_1 \mid P_a \mid E_1[\text{let oldcache} = \text{recv } cache_a \text{ in let isreplay} = [...] \text{ in } [...]] \\
&\xrightarrow{\textsc{CommL}} \\
&K_1, \rho_1, P_1 \mid () \mid E_1[\text{let oldcache} = content_a \text{ in let isreplay} = [...] \text{ in } [...]\ ] \\
&\xrightarrow{\textsc{LetVal}} \\
&K_1, \rho_1, P_1 \mid () \mid E_1[\text{let isreplay} = \text{antireplay } content_a\ a\ m \text{ in match } [...]] \\
&\xrightarrow{\textsc{Apply,LetVal}}{}^{*} \xrightarrow{\textsc{Match,Mismatch}}{}^{*} \\
&K_1, \rho_1, P_1 \mid () \mid E_1[\text{match } isreplay \text{ with } [...]\ ] \qquad \text{(D.1)} \\
&\xrightarrow{\textsc{Match}} \\
&K_1, \rho_1, P_1 \mid () \mid E_1[\text{asend } cache_a\ newcache\ ;\ \text{send } ch_a\ m\ ] \\
&\xrightarrow{\textsc{Apply}} \xrightarrow{\textsc{Fork}} \xrightarrow{\textsc{LetVal}} \\
&K_1, \rho_1, P_1 \mid () \mid \text{send } cache_a\ newcache \mid E_1[\text{send } ch_a\ m\ ]
\end{aligned}
$$

where $newcache = (sid,r)::content_a$ if '`header` m' evaluates to '`(`sid`,`$r`)`,true', and $newcache = content_a$ otherwise. We let $\xrightarrow{\text{Cache}}$ abbreviate this sequence of transitions, which represents successful anti-replay filtering.

Conversely, if '`header` m' evaluates to '`(`sid`,`r`)`,true' with sid,$r \in content_a$, then after (D.1) we have the transitions :

$$(D.1)\quad \xrightarrow{\textsc{Match},\textsc{Mismatch}}{}^{*}\ K_1, \rho_1, P_1 \mid () \mid E_1[\text{asend } cache_a\ content_a] \\ \xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\ K_1, \rho_1, P_1 \mid () \mid P_a \mid E_1[()]$$

(This case plays a role in the proof of Theorem 3.5, but not in the proof of Theorem 3.6.) In the rest of the proof, we may omit inert threads consisting of just the () expression, such as the thread left after P_a in the transitions above.

Session transitions We now translate the high-level session transitions.

EndS : The transition is trivially simulated : by definition, we have $[\![s\texttt{.0(}e\texttt{)}]\!]_T = [\![e]\!]_T$, so no low-level transition step is needed to simulate this step.

Init : The high-level transition is of the form

$$K, \rho, P \mid E[S.r_0^b\ (a_i)_{i<n}(g(\widetilde{v}), w)] \xrightarrow{s\overline{g}\ \widetilde{v}}_{\mathsf{K}} K \cup \{\widetilde{v}\}, \rho', P \mid E[s.p'\ (w)]$$

Let $P_0 \mid P_a \mid E_0[S.r\ \ peers\ (g(\widetilde{v}), w)]$ match $[\![P \mid E[S.r_0^b\ (a_i)_{i<n}(g(\widetilde{v}), w)]]\!]_T$. Let also $K_0 = [\![K]\!]_T$ and $\rho_0 = [\![\rho]\!]_T$. We have the following sequence of transitions :

$$\begin{array}{l} K_0, \rho_0, P_0 \mid P_a \mid E_0[S.r\ \ peers\ (g(\widetilde{v}), w)] \\ \xrightarrow{\textsc{Apply}} \\ K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{let}\ \ \text{st}\ = \text{init}\ \ peers\ \text{in send\_}\emptyset\ \text{st}\ \ (g(\widetilde{v}), w)] \\ \xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*} \\ K_0, \rho_1, P_0 \mid P_a \mid E_0[\text{send\_}\emptyset\ \ st\ (g(\widetilde{v}), w)] \\ \xrightarrow{\textsc{Apply}}\xrightarrow{\textsc{Mismatch}}{}^{*}\xrightarrow{\textsc{Match}}\xrightarrow{\textsc{LetVal}} \\ K_0, \rho_1, P_0 \mid P_a \mid E_0[\text{let}\ \ \text{m} = \text{gen\_}\emptyset\text{\_}g\ \ st\ \widetilde{v}\ \text{in psend}\ a\ \text{m}\ ;\ \text{recv\_}g\ \ st\ w] \\ \xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*} \\ K_0, \rho_1, P_0 \mid P_a \mid E_0[\text{psend}\ a\ m\ ;\ \text{recv\_}f\ \ st\ w|] \\ \xrightarrow{\textsc{Cache}}\xrightarrow{\textsc{LetVal}} \\ K_0, \rho_1, P_0 \mid \text{send}\ cache_a\ ((sid,r)::content_a) \mid E_0[\text{send}\ ch\ m\ ;\ \text{recv\_}g\ \ st\ w] \\ \xrightarrow{\overline{ch}\ m(\textsc{KSend})}_{\mathsf{K}}\xrightarrow{\textsc{LetVal}} \\ K_0 \cup \{m\}, \rho_1, P_0 \mid \text{send}\ cache_a\ ((sid,r)::content_a) \mid E_0[\text{recv\_}g\ \ st\ w] \end{array} \qquad (D.2)$$

where `(`sid`,`r`)` comes from the evaluation of '`header` m'. In (D.2), the `init` function has some side-effects, transforming ρ_0 into $\rho_1 = [\![\rho']\!]_T$.

Also, the message m added to the low-level K is not the exact translation of the updated high-level K, which contains the message payload v, new signatures, and the nonce. To obtain matching knowledge, we apply KApply steps at the low level to retrieve these values from m.

KSendS : The high-level transition is of the form

$$K, \rho, P \mid E[s.p\ (g(\widetilde{v}), w)] \xrightarrow{s\overline{g}\ \widetilde{v}}_{\mathsf{K}} K \cup \{\widetilde{v}\}, \rho, P \mid E[s.p'\ (w)]$$

Let $P_0 \mid P_a \mid E_0[\text{let } \mathrm{x}_s = (g(\widetilde{v}), w) \text{ in } \text{send\_}\widetilde{g}\ st\ \mathrm{x}_s]$ match $[\![P \mid E[s.p\ (g(\widetilde{v}), w)]]\!]_T$. Let also $K_0 = [\![K]\!]_T$ and $\rho_0 = [\![\rho]\!]$. We have the following sequence of transitions :

$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{let } \mathrm{x}_s = (g(\widetilde{v}), w) \text{ in } \text{send\_}\widetilde{g}\ st\ \mathrm{x}_s]$$
$$\xrightarrow{\textsc{LetVal}}$$
$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{send\_}\widetilde{g}\ st\ (g(\widetilde{v}), w)]$$
$$\xrightarrow{\textsc{Apply}}\xrightarrow{\textsc{Mismatch}}{}^{*}\xrightarrow{\textsc{Match}}\xrightarrow{\textsc{LetVal}}$$
$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{let } \mathrm{m} = \text{gen\_}\widetilde{g}\text{\_}g\ \mathrm{st}\ \widetilde{v} \text{ in } \text{psend}\ a\ \mathrm{m}\ ;\ \text{recv\_}\widetilde{g}g\ st\ w]$$
$$\xrightarrow{\textsc{Apply,LetVal}}{}^{*}$$
$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{psend}\ a\ m\ ;\ \text{recv\_}\widetilde{g}g\ st\ w]$$
$$\xrightarrow{\text{Cache}}\xrightarrow{\textsc{LetVal}}$$
$$K_0, \rho_0, P_0 \mid \text{send}\ cache_a\ newcache \mid E_0[\text{send}\ ch\ m\ ;\ \text{recv\_}\widetilde{g}g\ st\ w]$$
$$\xrightarrow{\overline{ch}\ m(\textsc{KSend})}_{\mathsf{K}}\xrightarrow{\textsc{LetVal}}$$
$$K_0 \cup \{m\}, \rho_0, P_0 \mid \text{send}\ cache_a\ newcache \mid E_0[\text{recv\_}\widetilde{g}g\ st\ w]$$

We apply KApply steps as above, to obtain matching knowledge.

Join : The high-level transition is of the form

$$K, \rho, P \mid E[S.r_0^b\ a_j(w)] \xrightarrow{sg\ \widetilde{v}}_{\mathsf{K}} K', \rho', P \mid E[s.p'\ (w.g\ \widetilde{a}\ \widetilde{v})]$$

We write $K_0 = [\![K]\!]_T$ and $\rho_0 = [\![\rho]\!]$. Let $P_0 \mid P_a \mid \mathrm{F} \mid E_0[S.r\ \ self\ w]$ match $[\![P \mid E[S.r_0^b\ a_j(w)]]\!]_T$. We also note `F = forward ()` the `forward` process and $P_1 = P_0 \mid \text{send}\ cache_a\ content_a$.

The translation of this transition is the following sequence of transitions :

$$K_0, \rho_0, P_0 \mid P_a \mid \mathrm{F} \mid E_0[S.r\ st\ w]$$
$$\xrightarrow{\textsc{Apply}}$$
$$K_0, \rho_0, P_0 \mid P_a \mid \text{let } \mathrm{a,m} = \text{recv psend}^{\bullet} \text{ in } [...] \mid E_0[S.r\ self\ w]$$
$$\xrightarrow{\text{psend}^{\bullet}\ (a,m0)(\textsc{KRecv})}_{\mathsf{K}}\xrightarrow{\textsc{LetVal}}\xrightarrow{\textsc{Fork}}\xrightarrow{\textsc{LetVal}}$$
$$K_0, \rho_0, P_0 \mid P_a \mid \mathrm{F} \mid \text{psend}\ a\ m \mid E_0[S.r\ self\ w]$$
$$\xrightarrow{\text{Cache}}\xrightarrow{\textsc{LetVal}} \tag{D.3}$$
$$K_0, \rho_0, P_1 \mid \mathrm{F} \mid \text{send}\ ch\ m0 \mid E_0[S.r\ self\ w]$$
$$\xrightarrow{\textsc{Apply}}$$
$$K_0, \rho_0, P_1 \mid \mathrm{F} \mid \text{send}\ ch\ m0 \mid E_0[\text{let } \mathrm{m0} = \text{precv}\ self \text{ in } [...]]$$
$$\xrightarrow{\textsc{Apply}}$$
$$K_0, \rho_0, P_1 \mid \mathrm{F} \mid \text{send}\ ch\ m0 \mid E_0[\text{let } \mathrm{m0} = \text{recv}\ ch \text{ in } [...]]$$
$$\xrightarrow{\textsc{CommL}}\xrightarrow{\textsc{LetVal}}$$
$$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{let } \mathrm{st,m} = \text{join}\ m0 \text{ in } [...]]$$
$$\xrightarrow{\textsc{Apply,LetVal}}{}^{*}\xrightarrow{\textsc{LetVal}}$$
$$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{if } \mathrm{st.peers}.r = self \text{ then verify\_}\emptyset\ st\ m\ w]$$
$$\xrightarrow{\textsc{Match}}$$

$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\mathrm{verify\_}\ \emptyset\ st\ m\ w]$
$\xrightarrow{\textsc{Apply}}$

$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\mathrm{let}\ \ \mathrm{path} = \mathrm{visible\_}\emptyset\ m\ \mathrm{in}\ \ \mathrm{match}\ \mathrm{path}\ \mathrm{with}\ [...]\ ]$
$\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\mathrm{match}\ path\ \mathrm{with}\ [...]]$
$\xrightarrow{\textsc{Mismatch}}{}^{*}\ \xrightarrow{\textsc{Match}}$

$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\mathrm{let}\ \ \mathrm{st}, \mathrm{payload} = \mathrm{check\_}\emptyset\_\widetilde{g'}\ st\ m\ \mathrm{in}\ \ \mathrm{let}\ \ \mathrm{next} = [...]\ \ ]$
$\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\mathrm{let}\ \ \mathrm{next} = w.last(\widetilde{g'})\ st.\mathrm{peers}\ payload\ \ \mathrm{in}\ \ [...]\ \ ]$

The cache transition (D.3) always succeeds, since there is no bad input.

KRecvS : The high-level transition is of the form

$$K, \rho, P \mid E[s.p\ (w)] \xrightarrow{sg\ \widetilde{v}}_{\mathsf{K}} K', \rho, P \mid E[s.p'\ (w.g\ \widetilde{a}\ \widetilde{v})]$$

We write $K_0 = [\![K]\!]_T$ and $\rho_0 = [\![\rho]\!]$. Let $P_0 \mid \mathrm{F} \mid E_0[\mathrm{recv\_}\widetilde{g}f\ \ \mathrm{st}\ w]$ match $[\![P \mid E[s.p\ (w)]]\!]_T$. We also note `F = forward ()` the `forward` process.
The translation of this transition is the following sequence of transitions :

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\mathrm{recv\_}\widetilde{g}f\ \ st\ w]$
$\xrightarrow{\textsc{Apply}}\ \xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}$

$K_0, \rho_0, P_0 \mid \mathrm{let}\ \ \mathrm{a}, \mathrm{m} = \mathrm{recv}\ \mathrm{psend}^{\bullet}\ \mathrm{in}\ [...]\ \ \mid E_0[\mathrm{let}\ \ \mathrm{m} = \mathrm{recv}\ ch\ \mathrm{in}\ [...]\ ]$
$\xrightarrow{\mathrm{psend}^{\bullet}\ (a,m)(\textsc{KRecv})}_{\mathsf{K}}\ \xrightarrow{\textsc{LetVal}}\ \xrightarrow{\textsc{Fork}}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid \mathrm{psend}\ a\ m \mid E_0[\mathrm{let}\ \ \mathrm{m} = \mathrm{recv}\ ch\ \mathrm{in}\ \mathrm{verify\_}\widetilde{g}f\ \ st\ \mathrm{m}\ w]$
$\xrightarrow{\textsc{Cache}}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid \mathrm{send}\ ch\ m \mid E_0[\mathrm{let}\ \ \mathrm{m} = \mathrm{recv}\ ch\ \mathrm{in}\ \mathrm{verify\_}\widetilde{g}f\ \ st\ \mathrm{m}\ w]$
$\xrightarrow{\textsc{CommL}}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\mathrm{verify\_}\ \widetilde{g}f\ \ st\ m\ w]$
$\xrightarrow{\textsc{Apply}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\mathrm{let}\ \ \mathrm{path} = \mathrm{visible\_}\widetilde{g}f\ \ m\ \mathrm{in}\ \mathrm{match}\ \mathrm{path}\ \mathrm{with}\ [...]]$
$\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\mathrm{match}\ path\ \mathrm{with}\ [...]]$
$\xrightarrow{\textsc{Mismatch}}{}^{*}\ \xrightarrow{\textsc{Match}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\mathrm{let}\ \ \mathrm{st}, \mathrm{payload} = \mathrm{check\_}\widetilde{g}f\_\widetilde{g'}\ \ st\ m\ \mathrm{in}\ \ \mathrm{let}\ \ \mathrm{next} = [...]\ \ ]$
$\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\ \xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\mathrm{let}\ \ \mathrm{next} = \mathrm{w}.last(\widetilde{g'})\ st.\mathrm{peers}\ payload\ \mathrm{in}\ [...]]$

KStep : High-level session communication steps are translated as a high-level send (without extending K) followed by a high-level receive.

□

We are now ready to prove completeness for our implementation :

Restatement of Theorem 3.6. *Let W be a valid implementation of H with no bad inputs. For all transitions $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H'$ in* F+S *such that ψ contains neither any signing key of a safe principal nor any elements of $\mathcal{N}$, there exist a valid implementation W' of H' with no bad inputs and a direct translation φ of ψ such that $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in* F.

Proof. By induction on the number of high-level transitions, applying Lemma D.1 for each transition. □

D.2 Notations for the soundness theorem

Administrative transitions Among the transitions that are part of the implementation of a high-level session transition (made explicit in the proof of lemma D.1), we designate as *administrative* the ones that do not concern communication (i.e. different from Send or Recv or CommL). We use $\rightarrow_{\mathsf{KA}}$ as an abbreviation for those transitions.

Transition names In order to easily reason about the reordering of the low-level transitions, we give names to some selected sequences of low-level transitions that are part of the session implementation detailed in the proof of Lemma D.1.

- The low-level implementation of Init and KSendS have only one non-administrative transition, labelled with $\overline{ch}\ m$, corresponding to the application of the (KSend) rule. We name this transition ϵ_0 and write the sequence of transitions as follows :

$$\xrightarrow{\mathrm{pre}_{\epsilon_0}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_0}_{\mathsf{K}} \xrightarrow{\mathrm{post}_{\epsilon_0}}{}^{*}_{\mathsf{KA}}$$

- The low-level implementation of KRecvS and Join have three non-administrative transitions, which we name as follows :
  - $\epsilon_1 = \mathtt{psend}^{\bullet}(a, m)(\mathrm{KRecv})$
  - ϵ_2 is the silent transition corresponding to a (CommL) on a $cache_a$ channel
  - ϵ_3 is the silent transition corresponding to a (CommL) on a ch_a channel

  The sequence of transitions is then written

$$\xrightarrow{\mathrm{pre}_{\epsilon_1}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_1}_{\mathsf{K}} \xrightarrow{\mathrm{pre}_{\epsilon_2}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_2}_{\mathsf{K}} \xrightarrow{\mathrm{pre}_{\epsilon_3}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_3}_{\mathsf{K}} \xrightarrow{\mathrm{post}_{\epsilon_3}}{}^{*}_{\mathsf{KA}}$$

  In the proof of theorem 3.5, we refer more generically to the pre and (possibly empty) post transitions of a given non-administrative transition ϵ_n.

Thread and commutativity We designate as a *thread* an expression placed in evaluation context. Each session role implementation uses only one thread, so administrative transitions (which are purely local computations) can be reordered with respect to other threads' transitions.

Visibility We write V for the visibility function : if $\widetilde{g}$ is the sequence of labels on a given path from the initial node m_0 to a node m with role r, $V(\widetilde{g})$ is the corresponding visible sequence, that is, the result of erasing from $\widetilde{g}$ every label g (1) whose sending role is r ; or (2) that is followed by a label whose sending role is either r or g's sending role.

D.3 An extended translation

The proof of theorem 3.5 relies on the translation relation and other invariants that track the link between low and high-level configurations. However, there exist low-level intermediate states of our session implementation that do not have a direct high-level reflection. We thus extend the translation function so that it coincides with the translation given in Section 3.8 on their joint domain, and so that it also keeps track of the session implementation intermediate steps.

We first define an extended version of the state T. The only added elements are the inner and outer functions ($T.inner$ and $T.outer$). For every session record $s\ (a_i)_{i<n}\ \{\widetilde{r}\} : S$ in ρ,

- $T.nonce(s)$ is a term N_s in K.
- $T.path(s)$ is an initial path $\widetilde{f}$ of S, decorated with strictly-increasing integers $\widetilde{j}$, ending by a label sent or received by a safe principal ;
- $T.stuck(s)$ is the set of roles that have received a bad input so far—in that case, the roles have silently terminated.

For every principal a,

- $T.cache(a)$ is the content of the cache of principal a : a set of pairs of session identifiers and roles $(\widetilde{sid}, r)$.
- $T.outer(a)$ is the multiset of messages $\widetilde{m}$ that have been received on $\mathtt{ch}_a$but have not been checked against the cache yet.
- $T.inner(a)$ is the multiset of messages $\widetilde{n}$ that have passed the cache test but have not received further treatment, with the following properties : (1) the multiset contains at most one joining message for each role of each running session, and (2) all joining message are also registered in the cache.

Finally, T provides an infinite (and co-infinite) set of fresh supplementary names, $\mathcal{N}$, which is formally used to supply a fresh nonce to the high-level environment whenever the low-level environment receives a fresh session nonce.

We define the translation $[\![K, \rho, P]\!]_T$ from F+S configurations to F configurations, as follows :

To translate K :

- we replace session records $s.p$ with the session nonces $T.nonce(s)$;
- we add the signatures exported by T ;
- we remove the signing keys of all safe principals and the supplementary nonces $\mathcal{N}$.

To translate the environment ρ :

- we replace session type definitions $\widetilde{S}$ with the types and function definitions of $M_{\widetilde{S}}$;
- we remove the session entries and the nonces of $\mathcal{N}$ not in the image of $T.nonce$.

To processes, we add the following ones :

- the forwarding process `F = forward ()` ;
- for each principal a, the process $P_a =$ send $cache_a\ T.cache(a)$, where $cache_a$ is the channel of $\rho_{L\widetilde{S}}$ that holds the state of the anti-replay cache for a. (The forwarding code and cache management are defined in Appendix C.1.)
- for each principal a and for each message n of $T.inner(a)$, a sending process send $ch_a\ n$ (ch_a denotes the principal's communication channel).
- for each principal a and for each message m of $T.outer(a)$, a sending process `psend` $a\ m$.

To translate the process P :

- we translate all running session roles within P as follows :

$$
\begin{array}{lcll}
[\![s.p(w)]\!]_T & = & \mathbf{0} & \text{if } p\text{'s role is in } T.stuck(s) \\
[\![s.p(w)]\!]_T & = & \texttt{S.recv\_}\widetilde{g}\,st\,[\![w]\!]_T & \text{else if } p \text{ is an input} \\
 & & & \text{or is an output of a message} \\
 & & & \text{in } T.inner \text{ or } T.outer \\
[\![s.p(e)]\!]_T & = & \text{let } x_s = [\![e]\!]_T \text{ in } \texttt{S.send\_}\widetilde{g}\,st\,(x_s) & \text{else if } p \text{ is an output} \\
[\![s.p(e)]\!]_T & = & [\![e]\!]_T & \text{else if } p \text{ is } \mathbf{0}
\end{array}
$$

where $\widetilde{g}$ and st are computed from $T.nonce(s)$ and $T.path(s)$. Note that the translation knows for each process $s.p$ which session S it implements, the reason being that the translation acts on configurations which include the necessary information in environments ρ.

– Expressions of the form $S.r\ldots$ are unchanged but now interpreted as function calls, rather than primitive session entries.

The refined translation coincides with the original translation when both the inner and outer functions ($T.inner$ and $T.outer$) are empty (as in the proof of theorem 3.6).

D.4 Auxiliary path properties

Lemma D.2 (Visibility) Consider a session Σ and a node n with role r. Let $\widetilde{f}$ be a visible sequence ending in n. Let $\widetilde{g}$ be the longest suffix such that r is not active, of a pre-image of $\widetilde{f}$ from the visibility function V. Then the set of active roles in $\widetilde{f}$ is the set of active roles in $\widetilde{g}$.

Proof. By definition of visibility. □

Lemma D.3 (No blind fork) Consider a session Σ and an initial path $\widetilde{g}$ leading to a node n_2 with role r. Let $\widetilde{f}$ be the longest suffix of $\widetilde{g}$ for which r is not active. Let n_1 be the first node of $\widetilde{f}$.

Then the set of active roles of all paths starting at n_1 where r is not active is included in the set of active roles of $\widetilde{f}$.

Proof. The proof relies on the absence of "blind forks" in session graphs, excluded by property 3 of section 3.2. □

D.5 Proof of the soundness theorem

Restatement of Theorem 3.5. *Let W be a valid implementation of H. For all transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F, there exist a valid implementation W° of H° and a translation φ of ψ such that*

$$H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ \qquad W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{K}} W'' \qquad W' \rightarrow^*_{\mathsf{K}} W''$$

Proof. By induction on the number of non-administrative transitions in $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$. We start from a low-level configuration W that is the translation in a state T of a high-level configuration $H = K, \rho, U$, thus $W = (K_0, \rho_0, U_0) = ([\![K]\!]_T, [\![\rho]\!]_T, [\![U]\!]_T)$. We first explain our induction principle and the step reordering it uses, then conduct the main case analysis.

Outline In this proof, our initial hypothesis is that there are transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ that lead a low-level configuration W to a configuration W' and that have an observable trace φ. The invariant of our proof relates a high-level configuration H, a state T and a low-level configuration W by the translation function described above : $W = [\![H]\!]_T$. Our goal is then to propagate this invariant over the transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ by finding intermediary configurations W_i where we have high-level configuration H_i and states T_i such that $W_i = [\![H_i]\!]_{T_i}$ and that the H_i are related by high-level transitions. However the intermediary low-level configurations W_i cannot always be found on the initial path from W to W' because of the interleaving of steps allowed by the session implementation. Thus, we first need to reorder (and complete) the original series of transitions before matching high-level and low-level transitions.

We present this reordering in an inductive manner. We start by examining the low-level transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ that we can decompose and complete in the following way :

$$W \xrightarrow{\text{extra + pre}_\epsilon}{}^*_{\mathsf{KA}} W_0 \xrightarrow{\epsilon}_{\mathsf{K}} W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$$

The sequence $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ starts with some administrative steps $W \xrightarrow{\text{extra+pre}_\epsilon}{}^*_{\mathsf{KA}} W_0$ followed by a session-related communication or a user-code transition $W_0 \xrightarrow{\epsilon}_{\mathsf{K}} W_1$. This transition may be silent or not : we use the metavariable ϵ to denote either the presence and content of a label or its absence. Among the administrative transitions that precede the ϵ transition, we distinguish the ones that causally precede ϵ : pre_ϵ ; and the others : 'extra'. We write φ' for the sequence of labels that are on the remaining transitions $W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$. We have thus $\varphi = \epsilon\ \varphi'$.

We then commute the 'extra' steps after the ϵ transition. We possibly need to add administrative post_ϵ steps to fully match in the low-level transitions the implementation of a high-level session transition. The result is the following series of transitions :

$$W \xrightarrow{\text{pre}_\epsilon}{}^*_{\mathsf{KA}} \xrightarrow{\epsilon}_{\mathsf{K}} \xrightarrow{\text{post}_\epsilon}{}^*_{\mathsf{KA}} W^\circ \xrightarrow{\text{extra}}{}^*_{\mathsf{KA}} W_1' \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W''$$

where the extra transitions are moved after the complete session step consisting in pre_ϵ, ϵ, post_ϵ. The transitions $W_1' \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W''$ consist of those in $W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$ after having removed any transitions that is also in post_ϵ. If all of the post_ϵ transitions are included in $W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$, then we have $W' = W''$. Otherwise the missing ones can be applied to W' to reach W''.

We iterate the reordering from configuration W°. The remaining transitions

$$W^\circ \xrightarrow{\text{extra}}{}^*_{\mathsf{KA}} W_1' \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W''$$

contain fewer non-administrative steps than the original series of transitions.

The next part of the proof consists in a case analysis on the transitions of the form $W \xrightarrow{\text{pre}_\epsilon}{}^*_{\mathsf{KA}} \xrightarrow{\epsilon}_{\mathsf{K}} \xrightarrow{\text{post}_\epsilon}{}^*_{\mathsf{KA}} W^\circ$ that correspond to specific high-level transitions.

Case analysis We now proceed with the inductive step of the proof : for each transitions $W \xrightarrow{\text{pre}_\epsilon}{}^*_{\mathsf{KA}} \xrightarrow{\epsilon}_{\mathsf{K}} \xrightarrow{\text{post}_\epsilon}{}^*_{\mathsf{KA}} W^\circ$, we exhibit a corresponding high-level transition that preserves the valid implementation relation.

Apply, Match, Mismatch, LetVal, LetFun, Type and Fresh (KStep) :

The different KStep transitions are reflexions of P-transitions and e-transitions. By hypothesis those silent transitions are not administrative steps, so the redex they reduce is also present in the original high-level U and is invariant under translation. The low level transitions, which are available at high level, may therefore be mirrored identically.

KApply : Since we have excluded administrative steps, the applied function appears in ρ before and after the translation. The arguments applied to the function used in the low-level KApply transition may consist of values built from cryptographic material (signatures, nonces) received by the low-level attacker as a byproduct of session communication. The matching between high and low-level adversarial knowledge provided by our invariant ensures that a high-level application of KApply directly simulate the low-level one.

KSend on a principal channel in chans$^\bullet$: In this case, the channel belongs to an unsafe principal b and the message is thus sent by the implementation of a running session role, for some principal a, at node n. As a consequence, the message is among the pending messages in $T.inner(b)$. The low-level process that initiates this message send is of the form send ch_b m ; [...], itself derived from let $x_s = [\![e]\!]_T$ in S.send_$\widetilde{g}$ st (x_s).

We know that $T.path(s)$ is an initial path. If the message m is the first of the session, then $T.path(s)$ is empty and we set $T.path(s)$ to the label and time-stamp, thus maintaining the relation between $T.path(s)$ and the role processes. If s is already started, the implementation for the principal a imposes that it previously received a signed label that, by definition of a valid implementation, is in $T.path(s)$, and that there is only one safe participant a that is in a sending state. Thus $T.path(s)$ ends in the current node n. Appending the last label and time-stamp of m to $T.path(s)$ therefore preserves the invariant.

As part of m, the attacker receives a sequence of signatures that are added to K. These signatures correspond to the updated translation under the new $T.path(s)$. However, to have matching high and low-level adversarial knowledge as required by the valid implementation invariant, the high-level configuration need to do some KApply transitions to build the corresponding signatures from the available keys.

When the attacker joins the session for the first time (i.e. has not already joined as a different role), the low-level message contains a nonce that we assume, without loss of generality, is already included in the supplementary infinite set of nonces $\mathcal{N}$ in the high-level K set. We therefore add N_s to $T.nonce(b)$ to maintain our invariant.

In all cases, the high-level transition is a KSendS.

KRecv on the psend$^\bullet$ channel : This a communication from the adversary to a safe principal a using the psend$^\bullet$ channel and therefore handled by the forward process, defined as let `a,m = recv psend•` in `[...]`. The reception of the message yields the process

$$\text{let } \mathrm{a,m} = a, m \text{ in fork forward ; psend a m}$$

which performs administrative transitions then yields process `psend` a m. To conclude, we update T by appending the message m to the multiset of pending messages in $T.outer(a)$, and leave the high-level configuration unchanged.

CommL or CommR on a safe principal channel from chans : By definition of the translation, this transition is enabled only for a receiver in the implementation of a running role r instantiated by a safe principal a. (Reception for unsafe principals is handled in case KRecv). Then we know that the received message m has passed the cache test and is among the pending messages in $T.inner(a)$. The sending process is then of the form : send ch m while the receiving one is :

$$\text{let } \mathrm{m} = \text{recv } ch \text{ in verify\_}\widetilde{g}g \text{ st } \mathrm{m}\ [\![w]\!]_T$$

Recall that $\widetilde{g}$ ends at the node from which r previously sent g; let n_0 be the node receiving g. Let f be the label corresponding to the m message and n_2 be the node in which r receives f.

The message m contains a session id, and a list of signed labels and time-stamps. Checking that the session id corresponds to a running session (or a new one) gives us the correct high-level s. The verify_ $\widetilde{g}g$ function also checks that the signatures are correct, the time-stamps are consecutive and the list of labels corresponds to a visible sequence $\widetilde{f}f$.

By our invariant, the labels in $\widetilde{f}$ that are signed by safe principals are in $T.path(s)$. Since we know that $T.path(s)$ is an initial path, we can deduce that $T.path(s)$ contains in particular the last label that is signed by a safe principal in $\widetilde{f}$. We call n_1 the receiving node of this label.

We next prove that $T.path(s)$ ends in n_1. Suppose for contradiction that $T.path(s)$ goes further. Then there is a path starting in n_1 which does not meet r and which leads a node n_3 where role r', different from r, is active and is instantiated by a safe principal. By Lemma D.3, this role r' is also active in all paths from n_1 to n_2. By Lemma D.2, this role has signed a label among the visible sequence $\widetilde{f}$ received by r in n_2 and thus $T.path(s)$ contains this label between n_1 and n_2, a contradiction. Therefore $T.path(s)$ ends in n_1 and there is no label signed by a safe principal between n_1 and n_2 in $\widetilde{f}$.

If a compliant principal is the sender of the message m, then we can simulate the f message exchange at high-level by a CommL or CommR. If the message m is sent by an unsafe principal, then there is (possibly) a multi-step gap between n_1 and n_2. The signed visible labels $\widetilde{f}$ received by r in n_2 give then a skeleton of internal adversary communication. By definition of visibility, we know that we can complete it to a contiguous path ending by f, going from n_1 to n_2. We therefore append this path to $T.path(s)$ to maintain the invariant. These internal communication steps are simulated at high-level through the use of (multiple) OComm steps inside the global KRecvS transition.

Finally, the message m is removed from $T.inner(a)$.

CommL or CommR on a cache channel : In this case, the receiving process is produced by a call `psend` a m that either is itself called by the implementation code

$$\text{let} \quad x_s \;=\; [\![e]\!]_T \;\text{ in S.send\_}\widetilde{g}\;\; st\;\; (x_s)$$

or that comes from the presence of m in $T.outer(a)$. The object of this transition is the communication of the current version of the cache of a and then administrative transitions decide if m can be forwarded to a. If m is a joining message and the session id s and the role r played by a in s are already recorded in the cache, then a new cache process is forked and r is added to $T.stuck(s)$ because m is a replay attack. Otherwise if m passes the cache test, we add it to $T.inner(a)$ and remove it from $T.outer(a)$. If m is a joining message we add the corresponding role and session id to $T.cache(a)$. We make no transition at high-level. The new low-level configuration is the image of the high-level configuration under the updated state T, modulo the identification of stuck processes with $\mathbf{0}$.

KSend, KRecv, CommL or CommR on any other channel : Since in this case the channels are not part of the session, these transitions correspond to redexes that are identically present in the original high-level U. We therefore reflect these transitions identically at high-level. Note that, in the KRecv case, a value v from

the low-level $\mathsf{Val}(K,\rho)$ is sent from the attacker : our invariant that matches high and low-level adversarial knowledge ensures that such a value can be built and sent at high-level.

□

D.6 Proof of the correspondance lemma

Restatement of Lemma 3.4. *We have transitions $K \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ for some fresh names $\widetilde{n}$ where $\omega \notin fn(\psi)$ if and only if $\rho, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$ for some process O that does not contain ω, does not match on constructors in ρ, that calls only pure functions of ρ, and whose values defined in ρ are all included in $\mathsf{Val}(K,\rho)$.*

Proof. ($\Rightarrow$) First, for every K, ρ, P, and ψ where $\overline{\omega}() \notin fn(\psi)$ we exhibit a process $O = e_O \mid \prod_{\sigma\in K} e_\sigma$, such that
(1) each e_σ represents an active session role controlled by the attacker and is of the form $(\texttt{let}\ \ \mathtt{x}_i\ \texttt{=}\ e_i\ \ \texttt{in})_{i\in[1..n]}\ \ \sigma\ \ v$, where $n \geq 0$;
(2) the processes in O do not contain ω ;
(3) the processes in O do not match on constructors and call only pure functions of ρ ;
(4) the values in O are all contained in $\mathsf{Val}(K \uplus \{c\}, \rho \uplus \{c\})[\omega := \omega']$;
(5) if $K, \rho, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ then $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$.
Here, $c \notin \rho$ is a distinguished channel used for communicating values between the different subprocesses of O ; $\omega' \notin \rho$ is also a distinguished name used wherever K uses ω instead.
We proceed by induction on the length of $\overset{\psi}{\Rightarrow}_{\mathsf{K}}$.
Base Case : $\overset{\psi}{\Rightarrow}_{\mathsf{K}}$ is empty.
Hence, P sends on ω (KSend). Let O be an arbitrary process satisfying the five conditions above ; say $O = \mathbf{0} \mid \prod_{\sigma\in K} \sigma()$. Then $\rho \uplus \{c, \omega'\}, P \mid O \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$ (Send,ParL).
Inductive Case : $K, \rho, P \xrightarrow{\alpha}_{\mathsf{K}} K', \rho', P' \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$, where $\alpha \neq \overline{\omega}()$. By the inductive hypothesis, there exists a process $O' = e_{O'} \mid \prod_{\sigma\in K'} e'_\sigma$ such that the five conditions above hold. By case analysis on the first transition, we construct a process $O = e_O \mid \prod_{\sigma\in K} e_\sigma$ that also satisfies the conditions :

KStep The transition is wholly in P. Let $O = O'$; P performs the same transition in $\rho \uplus \{c, \omega'\}, P \mid O$.

KApply The transition has the form $K, \rho, P \rightarrow_{\mathsf{K}} K \cup \{w\}, \rho, P$, where $\rho, l_i\ v_0 \ldots v_k \longrightarrow_{\mathsf{e}}{}^{*} \rho, w$ and $l_i, v_0, \ldots, v_k \in \mathsf{Val}(K,\rho)$ We spell out this case in detail, the construction of O in the other cases is similar. Here, $K' = K \cup \{w\}$, $\rho' = \rho$, and $P' = P$. First, we rename any occurrences of ω in $v_0, \ldots, v_k$ to ω', obtaining $v'_0, \ldots, v'_k$ By the inductive hypothesis, O' may contain $w[\omega = \omega']$ and so must be of the form : $e_{O'}[x := w][\omega = \omega'] \mid \prod_{\sigma\in K} e'_\sigma[x := w][\omega = \omega']$, where x is some fresh variable. Let e_O be $\texttt{let}\ \ \mathtt{x}\ \texttt{=}\ l_i\ \ v'_0 \ldots v'_k\ \ \texttt{in}\ \ (\texttt{let}\ \ \_\ \texttt{=}\ \texttt{send}\ \ c\ \ \mathtt{x}\ \ \texttt{in})_{\sigma\in K}\ \ e_{O'}$ and for each $\sigma \in K$, let $e_\sigma = \texttt{let}\ \ \mathtt{x}\ \texttt{=}\ \texttt{recv}\ \ c\ \ \texttt{in}\ \ e'_\sigma$. Here, e_O computes the function result, broadcasts it to all the active session processes in O, and continues with $e_{O'}$, while each e_σ receives this result and continues with e'_σ. (This is a common pattern that we use to distribute any new values that are generated in K among the subprocesses of O.) Then $O = e_O \mid \prod_{\sigma\in K} e_\sigma$ satisfies the induction hypothesis ; it extends O' with the application of a pure function l_i, with some new values $v'_0, \ldots, v'_k \in \mathsf{Val}(K,\rho)[\omega := \omega']$; it introduces some message sends and receives on the channel c but none on the channel ω ; moreover, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \rho \uplus \{c, \omega'\}, P \mid O'$ (Apply,CommR) ; hence $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KSend $K, \rho, P \xrightarrow{\overline{ch}\ w}_{\mathsf{K}} K \cup \{w\}, \rho, P'$, where P sends a message w on a channel ch in K. Here, $\omega \notin fn(ch, w)$; hence, $ch, w \in \mathsf{Val}(K \uplus \{w\}, \rho)[\omega := \omega']$, and $O' = e_{O'}[x := w] \mid \prod_{\sigma \in K} e'_\sigma[x := w]$.

Let $e_O = \text{let}\ \ \mathtt{x} = \text{recv}\ ch\ \text{in}\ (\text{let}\ \_ = \text{send}\ c\ \mathtt{x}\ \text{in})_{\sigma \in K}\ e_{O'}$, and for each $\sigma \in K$, let e_σ be $\text{let}\ \ \mathtt{x} = \text{recv}\ c\ \text{in}\ e'_\sigma$. Then $O = e_O \mid \prod_{\sigma \in K} e_\sigma$ satisfies the induction hypothesis ; in particular, CommRgives us $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \rho \uplus \{c, \omega'\}, P' \mid O' \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KRecv $K, \rho, P \xrightarrow{ch\ w}_{\mathsf{K}} K, \rho, P'$, where $ch, w \in \mathsf{Val}(K, \rho)$ and $\omega \notin fn(ch, w)$; hence $ch, w \in \mathsf{Val}(K, \rho)[\omega = \omega']$, and $O' = e_{O'} \mid \prod_{\sigma \in K} e'_\sigma$.

Let e_O be $\text{let}\ \ \_ = \text{send}\ ch\ w\ \text{in}\ e_{O'}$. Then $O = e_O \mid \prod_{\sigma \in K} e'_\sigma$ satisfies the induction hypothesis ; it extends O' with a message w sent on ch, where $w \in \mathsf{Val}(K, \rho)$; it does not introduce a message send on ω, since by the inductive hypothesis $ch \neq \omega$; using CommL, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KSendS The transition has the form $K[\sigma_r], \rho, P \xrightarrow{s\overline{g}\ w}_{\mathsf{K}} K[s.p] \cup \{w\}, \rho'', P'$, where $\omega \notin fn(w)$, and $\rho, P \xrightarrow{s\overline{g}\ w}_{\mathsf{P}} \rho', P'$, and $\rho', \sigma_r \xrightarrow{sg}_{\mathsf{s}} \rho'', s.p$. Hence $O' = e_{O'}[x := w] \mid \prod_{\sigma \in K[s.p]} e'_\sigma[x := w]$; in particular $e_{s.p}$ corresponds to the session process $s.p$.

Let $e_{\sigma_r} = \sigma_r(v)$ where $v.g\ \ \widetilde{a}\ x = (\text{let}\ \_ = \text{send}\ c\ x\ \text{in}\ )_{\sigma \in K[s.p]} e_{s.p}$; and for each $\sigma \in K[\sigma_r]$, if $\sigma \neq \sigma_r$ then let $e_\sigma = \text{let}\ \ \mathtt{x} = \text{recv}\ c\ \text{in}\ e'_\sigma$. Let e_O be $\text{let}\ \ x = \text{recv}\ c\ \text{in}\ e_{O'}$. Then $O = e_O \mid e_\sigma \prod_{\sigma \in K[\sigma]} e'_\sigma$ satisfies the induction hypothesis ; using RecvS, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KRecvS The transition has the form $K[\widetilde{\sigma}], \rho, P \xrightarrow{sg\ w}_{\mathsf{K}} K[\widetilde{s.p}, s'.p'], \rho''', P'$, where $\omega \notin fn(w)$, $K[\widetilde{\sigma}], \rho \xrightarrow{s\overline{g_1}}_{\mathsf{o}} \xrightarrow{sg_1}_{\mathsf{o}} \cdots \xrightarrow{s\overline{g_m}}_{\mathsf{o}} \xrightarrow{sg_m}_{\mathsf{o}} K[\widetilde{s.p}, \sigma_s], \rho'$, and $\rho', \sigma_s \xrightarrow{s\overline{g}}_{\mathsf{s}} s'.p', \rho''$, and $\rho'', P \xrightarrow{sg\ w}_{\mathsf{P}} \rho''', P'$. That is, the session role processes $\widetilde{\sigma}$ in K perform session communications with each other, resulting in the processes $\widetilde{s.p}$ and a process σ_s that then performs a session send and interacts with a session receive in P.

We construct a process O that simulates this sequence of steps. First we construct the process O_m corresponding to $K[\widetilde{s.p}, \sigma_s]$,$\rho'$, just before the last session send. By the inductive hypothesis, $O' = e_{O'} \mid \prod_{\sigma \in K[\widetilde{s.p}, s'.p']} e'_\sigma$, where $e_{s'.p'}$ must be of the form $(\text{let}\ \ \mathtt{x}_i = e_i\ \text{in})_{i \in [1..n]}\ \ s'.p'(v)$, where v is a record of handlers $h_1, \ldots, h_k$. We let $e_{\sigma_s} = \sigma_s(g(w), v')$, where v' has the same message handlers $h_1, \ldots, h_k$ as v, and for each h_i, we let $v'.h_i\ \widetilde{a}\ x = \text{let}\ \ (\mathtt{x}_1, \ldots, \mathtt{x}_n) = \text{recv}\ c\ \text{in}\ e_{s.p}$; in all other cases $e_\sigma = e'_\sigma$.

We let $e_{O_m} = e_{O'} \mid (\text{let}\ \ \mathtt{x}_i = e_i\ \text{in})_{i \in [1..n]}\ \text{send}\ c\ (\mathtt{x}_1, \ldots, \mathtt{x}_n)$.

Hence, we split the process for σ_s into two threads, passing control through a standard process calculus maneuver : the first part e_{σ_s} performs the session send and waits with a message handler for the next session receive ; in the meanwhile, the second subprocess of e_{O_m} performs some intermediate computations and sends the results to the message handler in e_{σ_s} over the channel c. Then $O_m = e_{O_m} \mid \prod_{\sigma \in K[\widetilde{s.p}, \sigma_s]} e_\sigma$ satisfies the induction hypothesis ; using RecvS and SendS, $\rho \uplus \{c, \omega'\}, P \mid O_m \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$. By using similar techniques, we extend O_m to the process O corresponding to $K[\widetilde{\sigma}]$, ρ by adding the corresponding internal session communications.

Using the process $O = e_O \mid \prod_{\sigma \in K} e_\sigma$ as constructed above, we can establish the ($\Rightarrow$) direction of the lemma. For every K, $\widetilde{n}$, ρ, P, ψ, if $K \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ and $\omega \notin fn(\psi)$, then the process $O_{K,c,\widetilde{n},\rho}$ defined as follows satisfies the lemma :

```
let c = new() in
```

```
let ω' = new() in
(let n_i = new () in)_{n_i∈ñ}
(let _ = fork e_σ in)_{σ∈K} e_O
```

It uses only pure functions in ρ, does not match on constructors, and does not contain ω; all its values are in $\mathsf{Val}(K_{\rho,O})[\omega = \omega']$ and hence the only values that are not in $\mathsf{Val}(K_{\rho,O})$ are undefined in ρ; finally, using Fresh and Fork, we have :
$\rho, P \,|\, O_{K,c,\widetilde{n},\rho} \rightarrow_{\mathsf{P}}{}^* \rho \uplus \{c, \omega', \widetilde{n}\}, P \,|\, O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

($\Leftarrow$) For every ρ, O, such that O only uses pure functions in ρ, does not match on constructors, does not contain ω, and whose values defined in ρ are contained in $\mathsf{Val}(K, \rho)$ we exhibit a K, $\widetilde{n}$, ψ such that the lemma holds. We define reduction contexts : $R[\cdot] ::= E[\cdot] \,||\, R[\cdot] \,|\, P \,||\, P \,|\, R[\cdot]$.

For a given ρ, O we define $K_{\rho,O}$ as the smallest set that satisfies the following :

- $\omega \in K_{\rho,O}$,
- for every pure expression e such that $O = R[e]$ and $\rho, e \longrightarrow_{\mathsf{e}}{}^* \rho, v$, $v \in K_{\rho,O}$, and
- for every session state σ in O : for all ρ_1, σ_1 such that $\rho_1, \sigma_1 \xrightarrow{\widetilde{\eta}}_{\mathsf{s}}{}^* \rho, \sigma$, $\sigma_1 \in K_{\rho,O}$; and for all ρ_2, σ_2 such that $\rho, \sigma \xrightarrow{\widetilde{\eta}}_{\mathsf{s}}{}^* \rho_2, \sigma_2$, $\sigma_2 \in K_{\rho,O}$.

We show that if $\rho \uplus \rho_l, P \,|\, O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$, where ρ_l consists of local type, session, and function definitions used only in O (and not in P), then there exists $\widetilde{n}, \psi$, such that $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$. We proceed by induction on the number of reductions in $\rightarrow_{\mathsf{P}}{}^*$.

Base Case : $\rightarrow_{\mathsf{P}}{}^*$ is empty. Hence, P sends on ω; and $K_{\rho,O}, \rho, P \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ using KRecv.

Inductive Case : $\rho \uplus \rho_l, P \,|\, O \rightarrow_{\mathsf{P}} \rho' \uplus \rho_l', P' \,|\, O' \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$. By the induction hypothesis, there exists $\widetilde{n}', \psi'$, such that $K_{\rho',O'} \uplus \{\widetilde{n}'\}, \rho' \uplus \{\widetilde{n}'\}, P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$. We exhibit $\widetilde{n}, \psi$ by case analysis on the first reduction $\rho, P \,|\, O \rightarrow_{\mathsf{P}} \rho', P' \,|\, O'$:

ParL The reduction step occurs within P : $\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', P'$ and $O = O'$. Then $K_{\rho,O} = K_{\rho',O'}$, $\widetilde{n} = \widetilde{n'}$, $\psi = \psi'$, and, using KStep, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \Rightarrow_{\mathsf{K}} K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.

ParR The reduction step occurs within O. By further case analysis :

Match,Mismatch,LetVal,Fork, LetFun, Type, Session
In these cases, $\rho = \rho'$, $O = O'$, $K_{\rho,O} = K_{\rho',O'}$, and no labeled transitions are needed.

Apply $O = R[l v_o \dots v_k]$, $O' = R[e\{x_0 = v_0; \dots; x_k = v_k\}]$, and $\rho = \rho'$, where l must be a pure function, and hence, e is a pure expression.

Apply $O = R[l v_0 \dots v_k]$, $O' = R[e\{x_0 = v_0; \dots; x_k = v_k\}]$, and $\rho = \rho'$, where l must be a pure function, and hence, e is a pure expression. $K_{\rho',O'}$ already includes the value v computed from the function application $(\rho, e\{x_0 = v_0; \dots; x_k = v_k\} \longrightarrow_{\mathsf{e}}{}^* \rho, v)$. $K_{\rho,O}$ contains $l, v_0, \dots, v_k$, hence by KApply, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \rightarrow_{\mathsf{K}} K_{\rho',O'} \uplus \{\widetilde{n}\}, \rho' \uplus \{\widetilde{n}\}, P$.

Fresh $\rho' = \rho \uplus \{n\}$, $O = R[new()]$, and $O' = R[n]$. Let $\widetilde{n} = \widetilde{n}' \uplus \{n\}$, then $K_{\rho,O} \uplus \{\widetilde{n}', n\} = K_{\rho,O'} \uplus \{\widetilde{n}'\}, \rho \uplus \{\widetilde{n}', n\}, P \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$

CommR,CommL In the non-session communication case, $\rho = \rho'$ and $K_{\rho,O} = K_{\rho,O'}$; no labeled transitions are needed. If it is a session communication on a session defined in ρ_l, again $K_{\rho,O} = K_{\rho',O'}$ and no transitions are needed. The remaining cases are when $O = R[\sigma_1 e_1] \,|\, R'[\sigma_2 e_2]$ and $O' = R[\sigma_1' e_1'] \,|\, R'[\sigma_2' e_2']$ Then σ_1, σ_2, σ_1', and σ_2' are all included in both $K_{\rho,O}$ and $K_{\rho',O'}$ and again no transitions are needed.

CommL O sends a message to P. If it is a channel communication $\overline{c}\,v$, then both c and v must be in $K_{\rho,O}$; and using KRecv, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{c\,v}{\Rightarrow}_{\mathsf{K}} K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.
If it is a session communication $s\overline{g}\,v$, then $O = R[\sigma(g(v), w)]$ and $\rho, \sigma \xrightarrow{\widetilde{s\overline{g}\,v}}_{\mathsf{s}} \rho', \sigma'$; using KRecvS, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \xrightarrow{sg\,v}_{\mathsf{K}} K', \rho'', P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.

CommR P sends a message to O. If it is a channel communication cv, then c must be in $K_{\rho,O}$; and using KSend, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\overline{c}\,v}{\Rightarrow}_{\mathsf{K}} K_{\rho,O} \uplus \{\widetilde{n}, v\}, \rho \uplus \{\widetilde{n}\}, P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.
If it is a session communication $s\overline{g}\,v$, then $O = R[\sigma(w)]$ and $\rho, \sigma \xrightarrow{s\overline{g}\,v}_{\mathsf{s}} \rho', \sigma'$; using KSendS, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \xrightarrow{s\overline{g}\,v}_{\mathsf{K}} K', \rho'', P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.

□

D.7 Proof of the integrity theorem

Auxiliary Lemma In the configuration $L\ \widetilde{S}\ UO$ we have that the libraries L and the session declarations $\widetilde{S}$ reduce deterministically. Hence, for any U and O, there exists $\rho_{L\widetilde{S}}$ (defining functions, sessions and values), substitutions σ and $\sigma^\bullet$ for values, the forwarder process `F` and the cache processes `C`, s.t. $\emptyset, L\ \widetilde{S}\ UO \rightarrow_{\mathsf{P}}{}^* \rho_{L\widetilde{S}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O\sigma^\bullet$.
Here $\rho_{L\widetilde{S}}$ records the declared sessions, plus the functions and types declared in the `Crypto` and `Prins` libraries as given in Appendix C.1, while σ records principals constants `prins` and $\sigma^\bullet$ records opponent accessible information : `prins`, `safe`, `vkey`, `psend`$^\bullet$, `chans`$^\bullet$, and `skeys`$^\bullet$, plus any information bound by U. Also, `F = forward ()` is the forwarder process that receives messages from opponent code to be sent to compliant principals and `C` represents the cache processes send cache_a $\mathit{content}_a$ (one per principal).

Similarly, for the implemented sessions $M_{\widetilde{S}}$, we have the deterministic reductions $\emptyset, L\ M_{\widetilde{S}}\ UO' \rightarrow_{\mathsf{P}}{}^* \rho_{LM_{\widetilde{S}}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O'\sigma^\bullet$. Here, in contrast to the session declarations in $\rho_{L\widetilde{S}}$, the environment $\rho_{LM_{\widetilde{S}}}$ contains role functions and types defined by the compiler, satisfying the relation : $[\![\rho_{L\widetilde{S}}]\!]_\emptyset = \rho_{LM_{\widetilde{S}}}$ which holds at the beginning, when no session has yet started, for the state empty $T = \emptyset$, that is, with empty *cache, nonce, path*, and *stuck* (see Section 3.8).

We also know that initial user code is independent of the compilation : $[\![U\sigma]\!]_\emptyset = U\sigma$

In addition, we let $K_0 \uplus K_h$ denote the initial values K_0 exported by L to opponent code, i.e. `prins`, `safe`, `vkey`, `psend`$^\bullet$, `chans`$^\bullet$, and `skeys`$^\bullet$, plus K_h that contains additional nonces $\mathcal{N}$ and signing keys from safe principals.

We then obtain that for all fresh $\widetilde{n}$: $K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}}, U$ is a valid configuration. Since the values $K_0 \uplus K_h$ exported by the libraries are the same in both high and low levels, modulo K_h, we get $[\![K_0 \uplus K_h \uplus \{\widetilde{n}\}]\!]_\emptyset = [\![K_0 \uplus K_h]\!]_\emptyset \uplus \{\widetilde{n}\} = K_0 \uplus \{\widetilde{n}\}$.

Gathering all of the above auxiliary results, we obtain the following lemma :

Lemma D.4 (Initial configuration) For the initial configuration $L\ \widetilde{S}\ UO$ it holds :

$$\emptyset, L\ \widetilde{S}\ UO \rightarrow_{\mathsf{P}}{}^* \rho_{L\widetilde{S}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O\sigma^\bullet \tag{D.4}$$

$$\emptyset, L\ M_{\widetilde{S}}\ UO' \rightarrow_{\mathsf{P}}{}^* \rho_{LM_{\widetilde{S}}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O'\sigma^\bullet \tag{D.5}$$

$$[\![\rho_{L\widetilde{S}}]\!]_\emptyset = \rho_{LM_{\widetilde{S}}} \tag{D.6}$$

$$[\![U\sigma]\!]_\emptyset = U\sigma \tag{D.7}$$

$$K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}}, U \text{ is a valid configuration} \tag{D.8}$$

$$[\![K_0 \uplus K_h \uplus \{\widetilde{n}\}]\!]_\emptyset = [\![K_0 \uplus K_h]\!]_\emptyset \uplus \{\widetilde{n}\} = K_0 \uplus \{\widetilde{n}\} \tag{D.9}$$

Restatement of Theorem 3.2. *If $L\,M_{\widetilde{S}}\,U\,O'$ may fail in* F *for some O' where ω does not occur, then $L\,\widetilde{S}\,U\,O$ may fail in* F+S *for some O where ω does not occur.*

Proof. We prove the theorem using the above fact. Since $L\ M_{\widetilde{S}}\ U\ O'$ fails, using Lemma D.4(D.5) we have that :

$$\emptyset, L\ M_{\widetilde{S}}\ U\ O' \rightarrow_{\mathsf{P}}{}^{*}\ \rho_{LM_{\widetilde{S}}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O'\sigma^{\bullet} \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$$

Let W be

$$K_0 \uplus \{\widetilde{n}\}, \rho_{LM_{\widetilde{S}}} \uplus \{\widetilde{n}\}, (U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C})$$

Since $O'\sigma^{\bullet}$ does not contain ω, by Lemma 3.4($\Leftarrow$) applied on $\rho := \rho_{LM_{\widetilde{S}}}$, $P := (U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C})$, $O := O'\sigma^{\bullet}$, there are fresh names $\widetilde{n}$ and φ s.t. $\omega \notin fn(\varphi)$ and $W \overset{\varphi\cdot\overline{\omega}()}{\Rightarrow}_{\mathsf{K}}$.

Let H be

$$K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}} \uplus \{\widetilde{n}\}, (U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C})$$

We have that W is a valid implementation of H, that is, $[\![H]\!]_{\emptyset} = W$, by Lemma D.4(D.7), Lemma D.4(D.8), and Lemma D.4(D.9). By Theorem 3.5, there is $W^{\circ}, H^{\circ}, W''$ s.t. $W \overset{\varphi\cdot\overline{\omega}()}{\Rightarrow}_{\mathsf{K}} W^{\circ} \rightarrow_{\mathsf{K}}^{*} W''$, $W' \rightarrow_{\mathsf{K}}^{*} W''$ and $H \overset{\psi\cdot\overline{\omega}()}{\Rightarrow}_{\mathsf{K}} H^{\circ}$, with φ a translation of ψ. Hence, $\omega \notin fn(\psi)$ neither. Expanding H and ignoring H°, we have :

$$K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}} \uplus \{\widetilde{n}\}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$$

Now, by Lemma 3.4($\Rightarrow$), since $\omega \notin fn(\psi)$, there exists O such that $\rho_{L\widetilde{S}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O\sigma^{\bullet} \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$; by Lemma D.4(D.4), then :

$$\emptyset, L\ \widetilde{S}\ U\ O \rightarrow_{\mathsf{P}}{}^{*} \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$$

i.e. , $L\ \widetilde{S}\ U\ O$ fails, establishing the theorem.

□

Annexe E

Table des exemples, théorèmes et définitions

Bibliographie

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses : The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2005.

[2] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 17–32, 2008.

[3] V. Benzaken and A. Frisch. CDuce : an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 51–63. ACM New York, NY, USA, 2003.

[4] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th Symposium on Principles of Programming Languages (POPL)*, pages 81–94. ACM New York, NY, USA, 1989.

[5] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[6] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 170–186. IEEE, July 2007.

[7] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5) :573–636, 2008.

[8] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Dénielou, Cédric Fournet, and James Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proc. 22th IEEE Symposium on Computer Security Foundations (CSF)*, IEEE, 2009.

[9] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop, (CSFW)*, pages 139–152, July 2006.

[10] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for ocaml. In *Proceedings of the 2006 workshop on ML*, pages 20–31, New York, NY, USA, 2006.

[11] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18(1) :47–64, 1982.

[12] Luís Caires and Hugo Torres Vieira. Conversation types. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.

[13] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Chris Hankin, editor, *Programming Languages and Systems, 16th European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[14] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4) :471–522, 1985.

[15] Luca Cardelli, Jim E. Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the 16th Symposium on Principles of Programming Languages (POPL)*, pages 202–212, Austin, Texas, 1989.

[16] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 197–213. Springer, 2006.

[17] G. Castagna and L. Padovani. Contracts for mobile processes. In *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR)*, number 5710 in LNCS, pages 211–228. Springer, 2009.

[18] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models : model checking message-passing programs. In *29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 45–57. ACM, January 2002.

[19] Ricardo Corin and Pierre-Malo Deniélou. A protocol compiler for secure sessions in ml. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing, Third Symposium (TGC'07)*, volume 4912 of *LNCS*, pages 276–293. Springer, 2007.

[20] M.J. Cox, R.S. Engelschall, S. Henson, B. Laurie, E.A. Young, and T.J. Hudson. Openssl, 2001.

[21] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34 :83–133, 1984.

[22] Pierre-Malo Deniélou and James J. Leifer. Abstraction preservation and subtyping in distributed languages. In *11th International Conference on Functional Programming (ICFP)*, pages 286–297. ACM, 2006.

[23] Pierre-Malo Deniélou. Page web de la thèse, 2009. `http://moscova.inria.fr/~denielou/these/`.

[24] Mariangiola Dezani-Ciancaglini, Dimitrios Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *20th European Conference for Object-Oriented Languages*, volume 4067 of *LNCS*, pages 328–352. Springer, July 2006.

[25] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A Distributed Object-Oriented language with Session types. In *International Symposium of Trustworthy Golbal Computing*, April 2005.

[26] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2) :198–208, 1983.

[27] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. *Rapport Techniques*, 154, 1993. At `http://coq.inria.fr`.

[28] J. Ellson, E. Gansner, L. Koutsofios, S.C. North, and G. Woodhull. Graphviz-open source graph drawing tools. *Lecture Notes in Computer Science*, pages 483–484, 2002.

[29] M. F
"ahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J.R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. volume 40, page 190. ACM, 2006.

[30] C Fournet and G Gonthier. The reflexive chemical abstract machine and the joincalculus. In *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[31] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml : A language for concurrent distributed and mobile programming. *Lecture Notes in Computer Science*, 2638 :129–158, 2003. At `http://jocaml.inria.fr`.

[32] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml : a language for concurrent distributed and mobile programming. In *Advanced Functional Programming, 4th International School, Oxford, August 2002*, volume 2638 of *LNCS*, pages 129–158. Springer, 2003.

[33] V. Gapeyev, M.Y. Levin, B.C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on programming language technologies for XML (PLAN-X)*, pages 04–24, 2005.

[34] J. Garrigue. Private row types : abstracting the unnamed. *Lecture Notes in Computer Science*, 4279 :44, 2006.

[35] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP)*, pages 74–90, 1999.

[36] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(6) :1037–1080, 2000.

[37] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st Symposium on Principles of Programming Languages (POPL)*, page 137. ACM, 1994.

[38] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# language specification*, 2003. At `http://msdn.microsoft.com/vcsharp/`.

[39] Grégoire Henry, Michel Mauny, and Emmanuel Chailloux. Typer la dé-sérialisation sans sérialiser les types. In *Journée francophone des langages applicatifs (JFLA)*, pages 133–146, Pauillac France, 01 2006. INRIA.

[40] Kohei Honda. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*, number 715 in LNCS, pages 509–523. Springer, 1993.

[41] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *Programming Languages and Systems, 7th European Symposium on Programming (ESOP)*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[42] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008.

[43] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *Proceedings of the 22nd European conference on Object-Oriented Programming (ECOOP)*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] The Java Language. http://java.sun.com/.

[45] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL)*, pages 173–184, New York, NY, USA, 2007. ACM Press.

[46] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003. Available from http://pauillac.inria.fr/~leifer/research.html.

[47] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Symposium on Principles of Programming Languages (POPL)*, pages 109–122. ACM New York, NY, USA, 1994.

[48] X. Leroy. The OCaml programming language. At http://caml.inria.fr, 1998.

[49] R Milner. A calculus of communicating systems. In *LNCS 76*. Springer-Verlag, 1980.

[50] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[51] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i. *I and II. Information and Computation*, 100, 1989.

[52] James H. Morris, Jr. *Lambda-Calculus Models of Programming Languages*. Ph. D. dissertation, MIT, December 1968. Report No. MAC–TR–57.

[53] M. Myers, C. Adams, D. Solo, and D. Kemp. Internet X.509 Certificate Request Message Format. RFC 2511 (Proposed Standard), March 1999. Obsoleted by RFC 4211.

[54] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *In PADL, volume 3057 of LNCS*, pages 56–70. Springer, 2004.

[55] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[56] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. *EPFL*, 2004.

[57] G. Peskine. *Abstract types in collaborative programs*. PhD thesis, Université Paris VII–Denis Diderot, 2008.

[58] Frank Pfenning and Carsten Schürmann. System description : Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16*, pages 202–206. Springer-Verlag LNAI, 1999.

[59] Benjamin C. Pierce. *Types and Programming Languages*. MIT press Cambridge, MA, 2002.

[60] Andrew M. Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 245–290. MIT Press, 2005.

[61] J. Planul, R. Corin, and C. Fournet. Secure Enforcement for Global Process Specifications. In *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR)*, page 526. Springer, 2009.

[62] G. Plotkin, M. Abadi, and L. Cardelli. Subtyping and parametricity. In *Ninth IEEE Symposium on Logic in Computer Science (LICS)*, pages 310–319, 1994.

[63] D. Remy. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th Symposium on Principles of Programming Languages (POPL)*, pages 77–88. ACM New York, NY, USA, 1989.

[64] A. Schmitt and J.B. Stefani. The Kell Calculus : A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2004.

[65] P. Sewell, J.J. Leifer, K. Wansbrough, F.Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute : high-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4-5) :547–612, 2007.

[66] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL)*, pages 214–227. ACM New York, NY, USA, 2000.

[67] D. Syme. *F#*, 2005. At `http://research.microsoft.com/fsharp/`.

[68] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2) :64–87, 2006.

[69] N. Wirth. The programming language Oberon. *Software - Practice and Experience*, 18(7) :671–690, 1988.

[70] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy (S&P)*, pages 178–104, Washington, DC, USA, 1993. IEEE Computer Society.

[71] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types, October 2009. Submitted to FOSSACS'09.