INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Sûreté globale des abstractions et sous-typage dans un langage distribué

Pierre-Malo Deniélou †
ENS Cachan, INRIA Rocquencourt

Stage du master MPRI
Sous la direction de James J. Leifer † et Jean-Jacques Lévy †
INRIA Rocquencourt

† {Prenom.Nom}@inria.fr

2 septembre 2005

Résumé : La préservation des abstractions, et notamment des types abstraits, est la tâche principale des systèmes de typage des langages de programmation. Or, en environnment distribué, lorsque les processus de sérialisation et de désérialisation permettent la communication de valeurs arbitraires, les abstractions ne sont pas garanties. Une solution à ce problème utilise le hachage du code source des modules (pour produire un espace unique de nommage des types abstraits) et des crochets colorés (pour délimiter dans la sémantique les expressions qui ont accès à l'implémentation concrète des types abstraits). Cependant, cette solution n'avait pas été validée en présence de sous-typage (et donc d'un polymorphisme borné) et d'abstractions partielles.

Nous avons donc développé des sémantiques statiques et dynamiques d'un langage combinant à la fois sérialisation et désérialisation, hachage et crochets colorés, sous-typage et abstractions partielles. Nous avons examiné et justifié les choix de conception qui s'offraient à nous. Nous prouvons enfin une partie des théorèmes de préservation, de progrès et de déterminisme.

Mots-clés : langage de programmation, ML, typage, sous-typage, types abstraits, sérialisation, modules, hachage, programmation distribuée, lambda-calcul

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Table des matières

Chapitre 1

Introduction

Les programmeurs attendent souvent de la part des langages de programmation typés une sûreté, une garantie contre un certain nombre d'erreurs à l'exécution. Cette propriété de sûreté pourrait se définir comme une propriété de préservation de l'abstraction.

Il s'agit, tout d'abord, d'une abstraction vis-à-vis de l'extérieur du langage, c'est-à-dire l'abstraction que constitue le langage de programmation par rapport aux détails de fonctionnement de la machine (pile, pointeurs, entiers et flottants, par exemple). Il existe cependant une seconde forme d'abstraction, celle-là interne au langage, qui peut être le résultat de l'utilisation de ses caractéristiques propres et de son système de types, comme les modules et les types abstraits. Ces derniers permettent de restreindre aux modules qui les déclarent, la manipulation d'un type de donnée. Le programmeur peut alors utiliser cette restriction pour assortir ses types abstraits d'invariants qu'il souhaite voir satisfaits.

Les systèmes de types à la ML ont pour objet de garantir la préservation de ces deux formes d'abstraction. Ils échouent cependant en environnement distribué, à cause des opérations de sérialisation et de la désérialisation. Ces fonctions sont utilisées lorsqu'il est nécessaire de transmettre par le réseau des valeurs arbitraires.

Une solution a pourtant été apportée par Leifer et al. [LPSW03] avec l'utilisation du hachage et des couleurs. Le comportement de cette solution en présence d'un système de type plus riche comportant notamment du *sous-typage et des abstractions partielles* n'avait pourtant pas encore été étudié.

Nous allons donc, dans cette introduction, présenter le problème que pose l'environnement distribué pour la préservation de l'abstraction, ainsi que la solution utilisant le hachage et les couleurs. Nous poursuivrons par une introduction au sous-typage et aux types partiellement abstraits, et une énonciation des problèmes qui pourraient survenir du fait de la présence du hachage et des couleurs. Enfin, nous préciserons les enjeux, les objectifs de ce stage, ainsi que les principales étapes du travail effectué.

1.1 Langages distribués et préservation de l'abstraction

1.1.1 Sûreté de la sérialisation

Lorsque l'environnement de programmation est distribué, la sérialisation et plus encore la désérialisation font peser une menace importante sur la sûreté du langage.

En effet, 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 par le réseau à un autre programme. Celui-ci va alors essayer de recréer la valeur grâce à l'opération inverse de désérialisation. Comme les deux programmes ne comportent pas toujours les mêmes modules, n'ont pas forcément accès aux mêmes versions de bibliothèques ou n'ont peut-être pas été compilées par le même compilateur, les échanges de valeurs ont de nombreuses raisons de ne pas être sûres.

Nous modéliserons l'environnement distribué 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, par exemple via un socket TCP. Ce ne sont aucunement des canaux typés mais de simples fonctions permettant de s'abstraire dans la suite des considérations relatives aux réseaux.

Nous noterons *`Programme1`* `||` *`Programme2`*, la mise sur deux machines distinctes reliées par le réseau des programmes *`Programme1`* et *`Programme2`* qui auront la possibilité de s'échanger des chaînes de caractères grâce aux fonctions `send` et `receive`.

Enfin, nous utiliserons les notations suivantes pour la sérialisation et la désérialisation :

```
marshall( _ : T )      unmarshall (_) : T
```

Notez la présence explicite des types d'envoi et de réception.

Une certaine correspondance entre ces deux types est nécessaire pour préserver l'abstraction vis-à-vis des détails de la machine, c'est-à-dire, en pratique, pour que le codage et le décodage de la valeur soient cohérents et que la communication réussisse. On peut par exemple exiger que ces types soient identiques, la vérification se faisant toujours à l'exécution.

Exemple 1.1.1 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 vérifier l'égalité des types, désérialiser la chaîne reçue comme entier, puis utiliser cette valeur (ici l'imprimer à l'écran).

```
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` après vérification que les deux types sont égaux : `int` = `int` .

Cependant, le type de la donnée envoyée peut ne pas être qu'une combinaison de types de base. Il peut par exemple dépendre, via des abréviations, d'autres parties du programme qui ne sont pas partagées avec le programme qui reçoit, ou, et c'est le cas qui nous intéresse, être un type abstrait. Ce type d'abstraction est alors difficile à préserver.

Les stratégies pour résoudre cette difficulté sont diverses. Certains langages vérifient simplement que les méthodes pour sérialiser et désérialiser correspondent bien au niveau des expressions concrètes, mais ne vérifient rien au niveau du typage : aucune abstraction n'est préservée. C'est le cas d'OCaml [OCa] qui ne fait que tester si les compilateurs de chacun des deux programmes ont le même numéro de version.

Exemple 1.1.2 L'exemple suivant cause un `Segmentation fault` en OCaml. Il s'agit juste de sérialiser une valeur d'un type particulier (ici un entier), et de désérialiser la chaîne obtenue en attendant un autre type (par exemple un flottant).

```
let s = Marshal.to_string 3 [] in
(Marshal.from_string s 0 : float);;
```

OCaml n'est donc pas sûr lorsque l'on utilise la sérialisation et la désérialisation.

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. En JoCaml [JoC] par exemple, l'établissement des canaux n'est sûr que s'il a lieu au sein d'un même programme ; deux programmes distincts doivent passer par un serveur de nom sans garantie de sûreté.

1.1.2 Préservation de l'abstraction

Nous avons vu qu'un des rôles d'un système de typage est la préservation de certaines abstractions à la fois vis-à-vis de l'extérieur du langage, mais aussi au sein-même de celui-ci. Nous allons nous intéresser ici au cas particulier des types abstraits.

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 peut seulement utiliser les fonctions qui le manipulent.

Exemple 1.1.3 Voici l'exemple d'un module définissant un 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. La déclaration abstraite ne permet ici l'utilisation du compteur que par les éléments du module qui sont déclarés dans la signature. Ici, on ne peut donc que créer un compteur en appelant `initial`, l'incrémenter et en extraire la valeur sous forme d'entier en utilisant `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 une donnée.

Certaines des difficultés vont cependant survenir en environnement distribué. 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). Notre objectif dans le chapitre 3 sera donc d'obtenir la préservation des abstractions de l'envoyeur, tout autant que la préservation des invariants du receveur.

1.1.3 La solution du hachage et des couleurs

Dans le papier [LPSW03], James J. Leifer et al. donnent une solution à cette difficulté en proposant d'affecter de manière reproductible à chaque module un nom unique sur toutes les machines qui sont susceptibles de communiquer entre elles, et d'enfermer les valeurs abstraites dans un coffret dont le nom du module d'origine est la clé. Cet espace unique de nom est réalisé en utilisant le hachage du code source de chaque module comme identifiant. Ainsi deux modules identiques compilés séparément sur deux machines différentes auront toujours le même nom.

Quelques remarques :

- Les identifiants des types abstraits (par exemple `Compteur.t`), qui faisaient référence au nom du module d'origine, sont remplacés à la compilation par une référence au résultat du hachage de ce module d'origine (i.e. `h.t` si `h` est le hachage 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 hachages correspondants. On ne fait donc que le hachage 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 hachage, 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 hachages auxquels ils font référence.
- Le nom du module est pris en compte lors du hachage. Cela permet ainsi au programmeur de forcer la distinction entre deux modules qui ont la même implémentation.
- Comme deux implémentations peuvent être identiques à α-renommage ou permutation près, le hachage ne peut se faire réellement sur le code source. Il se fait donc plutôt sur son arbre de syntaxe abstraite normalisé.
- Dans les langages habituels comme ML [MTH90], les valeurs abstraites ne sont pas distinguées à l'exécution de leur forme concrète, ce qui permet lors de la désérialisation de ne pas tenir compte de cette abstraction. La solution que nous présentons ici fait ainsi intervenir ici des crochets colorés qui viennent protéger la valeur d'une confusion avec un type manifeste lors de l'exécution (on a donc $[e]^T_{hm}$ au lieu de simplement e). Les crochets colorés sont annotés par une couleur, c'est-à-dire par un ensemble de hachages, précisément les hachages 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 [GMZ00].
- Les hachages ont lieu uniquement lors de la compilation. Les crochets colorés peuvent apparaître lors de la compilation comme de l'exécution du programme. L'utilisateur ne peut pas utiliser explicitement ni l'un ni l'autre dans son programme : ce sont seulement des outils sémantiques.

Nous reviendrons sur le hachage et les crochets colorés de façon plus détaillée dans les chapitres suivants. Un exposé détaillé sur le hachage peut aussi être trouvé dans Acute [SLW+05].

1.2 Sous-typage et enregistrements

1.2.1 Enregistrements

Les enregistrements sont une sorte de n-uplet dont les éléments sont étiquetés. Il s'agit d'une structure de donnée simple qui a la faculté de donner une base de modélisation à de nombreuses structures informatiques, comme les objets.

Exemple 1.2.1 Nous écrivons généralement un enregistrement à l'aide d'accolades, au sein desquelles on va associer des valeurs à des étiquettes. L'enregistrement ci-dessous associe respectivement aux étiquettes `nom`, `prenom` et `age`, des valeurs de types `string`, `string` et `int`.

```
{nom = "Dupont" ; prenom = "Jean" ; age = 33}
```

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 (comme des références) 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, et dont l'ordre des champs importe pour que notre sémantique opérationnelle soit déterministe. Les enregistrements ne servent en pratique dans notre langage que de prétexte minimal au sous-typage.

1.2.2 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 » ou de « moins précis ». Symétriquement, on peut parler du type plus petit comme d'un type « plus précis ».

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 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 correspondants sont identiques. C'est le sous-typage dit « en 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 « en profondeur ». Nous notons le sous-typage $T <: T'$ lorsque T est un sous-type de T'.

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

$$\frac{T_i <: T'_i \quad 1 \leqslant i \leqslant j}{\{l_1{:}T_1, ..., l_j{:}T_j\} <: \{l_1{:}T'_1, ..., l_j{:}T'_j\}} \text{ (Profondeur)}$$

Exemple 1.2.2 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. OCaml utilise une notation différente T `>:` T' lorsque T est plus petit que T'.

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

Le type enregistrement de gauche possède des champs supplémentaires par rapport au type enregistement 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 resultat 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 possèdent très souvent une notion de sous-typage ou ou une notion similaire (Java [Jav], Ocaml [OCa], ...). Les langages traitant nativement des schémas XML l'utilisent aussi [Cduce [BCF03], Xtatic [GLPS05]].

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 [Rém89] et sont en relation de sous-typage.

1.2.3 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 manipulation.

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 1.2.3 Donnons un exemple à l'aide d'un module schématisant la situation bancaire d'un patron « véreux ». 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 [CW85] et a été implémentée dans certains langages objets, comme Modula-3 [CDJ+89].

1.3 Problème, enjeux et plan du travail

Notre but dans ce travail a été de concevoir un langage de programmation avec sous-typage, types partiellement abstraits et préservation de l'abstraction en environnement distribué.

Comme le système de Leifer et al. [LPSW03] ne comporte pas de sous-typage, il était naturel d'essayer de l'appliquer dans notre cas plus général. Nous avons donc cherché à savoir quelles contraintes faisait peser la présence du sous-typage et des abstractions partielles sur la gestion des types abstraits par hachage et couleurs en environnement distribué.

En effet, la présence de sous-typage trouve son expression naturelle dans l'application des fonctions à des types plus petits, mais aussi dans la désérialisation dans un type plus grand que celui envoyé. Une première interrogation surgit alors. Un type abstrait peut-il être désérialisé en un autre type abstrait plus général ? Que signifie « plus général » pour des types abstraits ou partiellement abstraits ?

De plus, les crochets colorés enferment les valeurs concrètes dont le type est abstrait. Quelle influence possède la présence de types partiellement abstraits sur la gestion de ces crochets colorés ?

Trouver une solution à cette intégration permettrait à terme d'augmenter sensiblement l'expressivité des langages utilisant le hachage et les crochets colorés. Cela permettrait, de manière symétrique, d'appliquer la méthode du hachage et des crochets colorés à des langages possédant un système de type avec sous-typage, comme des langages objets ou XML.

Pour arriver à ce résultat, il est important de faire correspondre à un langage de base un système de types et une sémantique (si possible déterministe) dont on peut prouver la correction ; l'assurance

d'une adéquation du typage avec la sémantique peut nous venir de théorèmes de préservation du typage et de progrès. Une implémentation suivant la sémantique pas-à-pas, en typant chacune des étapes intermédiaires, donnerait enfin une garantie expérimentale.

Notre contribution est donc ici la définition d'un langage distribué sûr possédant sous-typage et abstractions partielles. Nous avons développé des sémantiques statiques et dynamiques pour s'assurer de la préservation des abstractions lors de la communication par sérialisation et désérialisation. Nous avons de plus prouvé une partie des théorèmes de préservation, de progrès et de déterminisme qui assurent le bon comportement de notre langage. Nous avons enfin justifié les choix de conceptions qui s'offraient à nous tout au long de ce développement.

Dans ce rapport, nous allons présenter dans le chapitre 2 le langage que nous allons utiliser à la base. Nous donnerons aussi un aperçu de son système de types et de sa sémantique opérationnelle. Dans le chapitre 3, nous aborderons le question de l'existence d'une éventuelle relation de sous-typage entre types abstraits. Ensuite, dans le chapitre 4, nous examinerons les contraintes que le sous-typage fait peser sur la gestion des crochets colorés. Nous donnerons enfin un aperçu des principaux résultats théoriques dans le chapitre 5. Il sera alors temps de conclure et d'évoquer les travaux actuels et futurs au sein du chapitre 6.

Nous avons mis en annexe l'intégralité de la syntaxe, du système de type et de la sémantique du langage définitif. Suivent alors les énoncés et les preuves, malheureusement incomplètes, des théorèmes.

Chapitre 2

Cadre de travail

Nous exposons ici les éléments essentiels du langage qui servira de modèle à notre étude des relations entre abstraction, hachage et sous-typage. Le lecteur pourra se rapporter aux annexes pour une référence complète et définitive.

Le langage dont nous étudierons les propriétés et que nous modifierons au fur et à mesure de nos progrès dans sa compréhension comprend un cœur de λ-calcul en appel par valeur avec produits et enregistrements, auquel on ajoute un langage de modules sans foncteurs. Le système de typage comportera signatures, sous-typage *a priori* implicite et types abstraits et partiellement abstraits. Enfin, nous donnerons une sémantique opérationnelle à petit pas. Ce sont ces trois éléments (syntaxe, système de typage, sémantique) que nous allons présenter dans ce chapitre.

2.1 Syntaxe

Nous exprimons nos exemples dans une syntaxe proche de celle d'OCaml, avec pourtant de grandes différences concernant le sous-typage et les types partiellement abstraits. Cependant, par concision, le système de types et la sémantique font appel à une syntaxe différente que nous allons présenter ici.

Cette syntaxe est volontairement incomplète et s'enrichira au fur et à mesure de nos avancées au cours des prochains chapitres. La syntaxe complète et définitive se trouve en annexe.

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 le hachage font partie de cette catégorie, et n'entrainent 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 reliées entre elles.

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

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 du module utilisé dans le hachage, un nom de variable de module (susceptible d'α-renommage), une structure et une signature.

$m ::= e$	expression
$\mid \textbf{module}\, N_U = M{:}S\ \textbf{in}\ m$	déclaration de module (U lié dans m)

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 ne va définir qu'un type et une valeur et une signature ne va donner qu'une sorte (*kind*), celle du type déclaré par la structure, et le type de la valeur. Retrouver la généralité des modules permettant la déclaration de plusieurs éléments recquiert un certain travail, principalement à cause des dépendances entre types ou valeurs (cf. Acute [SLW+05]). 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)

Exemple 2.1.1 Revenons à l'exemple de compte en banque 1.2.3 qui va nous servir sous une forme légèrement modifiée (on a ajouté une fonction permettant de mettre de l'argent en Suisse) à illustrer la correspondance entre la syntaxe qui se veut proche d'OCaml et notre nouvelle syntaxe. On y décrit de la façon suivante un module et une expression associée (on note `fst` et `snd` les première et seconde projections des couples) :

```
module Compte =
  struct
    type t = {france : int ; suisse : int}
    let v = ({france = 1 ; suisse = 10000000},
             (function (x:{france:int;suisse : int}) ->
               (function (y:int) -> {france = x.france ;
                                     suisse = y + x.suisse} )))
  end :
  sig
    type t <: { france : int }
    val v : t * (t -> int -> t)
  end in
marshall ((snd Compte.v) (fst Compte.v) 1000 : Compte.t)
```

Si l'on voulait réécrire la définition du module dans notre nouvelle syntaxe, cela donnerait :

$\textbf{module}\, \mathit{Compte}_U =$
$[\{\mathit{france} : \text{INT}; \mathit{suisse} : \text{INT}\}, (\{\mathit{france} = 1; \mathit{suisse} = 10000000\},$
$(\lambda x{:}\{\mathit{france} : \text{INT}; \mathit{suisse} : \text{INT}\}.(\lambda y{:}\text{INT}.\{\mathit{france} = x.\mathit{france}; \mathit{suisse} = y + x.\mathit{suisse}\}))] :$
$[X{:}\textbf{Le}(\{\mathit{france} : \text{INT}\}), X * (X{\to}\text{INT}{\to}X)]\ \textbf{in}$
$\textbf{mar}\,((\textbf{proj}_2\ U.\text{term})\,(\textbf{proj}_1\ U.\text{term})\,1000 : U.\text{TYPE})$

Examinons l'emploi des deux noms donnés au module : *Compte* est le nom qui sera utilisé dans le hachage pour distinguer éventuellement deux modules qui ont la même implémentation, alors que U est la variable qui lie le module aux appels U.term et U.TYPE dans le reste du programme. Cette variable U est susceptible d'alpha-renommage.

Types :
Les types de notre langage comportent un cœur de types usuels à la ML : `unit`, `string`, flèche, produit, enregistrement. On y ajoute un type $\top$ dont tous les autres types sont des sous-types. $\top$ nous permettra de modéliser les types abstraits et de modéliser certains jugements de correction. Nous noterons de plus $U.\text{TYPE}$ la référence à la déclaration de type du module correspondant à la variable U. Le type haché $h.\text{TYPE}$ est lui un type abstrait qui fait référence au module dont le hachage est h. Rappelons que les éléments situés sous la ligne pointillée ne peuvent être directement écrits dans un programme.

$T ::= \text{UNIT} \mid \text{INT} \mid \text{STRING}$		types de base
$\mid X \mid T \to T$		variable, fonction
$\mid \{l_1{:}T; ...; l_j{:}T\}$	$\mid T * ... * T$	enregistrement ($j > 0$), produit
$\mid U.\text{TYPE}$		type déclaré par le module U
.....		
$\mid \top$		type le plus général
$\mid h.\text{TYPE}$		type haché

Hachages et Couleurs :
Le hachage d'un module a lieu lorsque celui-ci déclare un type au moins partiellement abstrait. Nous ne précisons pas ici la méthode utilisée pour le hachage (cf. Acute [SLW+05]), mais un hachage $\mathbf{hash}(N, M{:}[X{:}\mathbf{Le}(T), T])$doit porter sur le nom déclaré du module N, sa structure M et sa signature $[X{:}\mathbf{Le}(T), T]$. Le résultat du hachage 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 le chapitre 3.

Une couleur sera définie comme une simple collection de hachage de modules. Ces modules correspondront aux modules pour lesquels on pourra associer au type abstrait son implémentation réelle.

$h ::= \mathbf{hash}(N, M{:}[X{:}\mathbf{Le}(T), T])$		hachage
$hm ::= \bullet$		couleur vide
$\mid \{h\}$		couleur avec un seul hachage
$\mid hm \cup hm$		union de couleurs

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 $\mathbf{Eq}(T)$ va être utilisée pour les types manifestes, c'est-à-dire les types qui sont explicites dans la signature. La sorte **Le** est dérivée des travaux de Cardelli et Wegner [CW85]. La sorte singleton **Eq** provient, quant à elle, des résultats de Leroy [Ler94], Harper et Lillibridge [HL94], Stone et Harper [SH00].

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

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 $U.\text{TYPE}$, l'expression $U.\text{term}$ 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 $[e]^T_{hm}$ délimitent les espaces dans lesquels on peut associer un module au résultat

de son hachage : à l'intérieur des crochets les hachages de hm sont supposés connus. Le type T représente alors le type qu'a l'expression e vue de l'extérieur.

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

Valeurs :
Parmi les expressions on distingue les valeurs, notées v^{hm}, qui correspondent aux expressions que l'on ne peut pas évaluer plus avant avec la sémantique. Elles sont annotées par une couleur 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 2.3.

La version définitive et complète est en annexe, mais nous en donnons un aperçu ici.

$v^{hm_0} ::= () \mid \underline{n}$	unit, entiers
$\mid (v_1^{hm_0}, ..., v_j^{hm_0})$	produit ($j \geqslant 2$)
$\mid \{l_1 = v_1^{hm_0}, ..., l_j = v_j^{hm_0}\}$	enregistrement ($j \geqslant 1$)
$\mid \lambda x{:}T.e$	abstraction (x lié dans e)
$\mid \mathbf{marshalled}\,(v^\bullet{:}T)$	valeurs sérialisées
$\mid [v^{hm_1}]^{h_1.\text{TYPE}}_{hm_1}$	crochets colorés où $h_1 \notin hm_0 \cap hm_1$

2.2 Système de typage

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

Le système de typage définit des jugements pour la correction des hachages, 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.

Voilà la forme des différents jugements que notre système de types va utiliser.

Jugements :

$\vdash h\,\mathrm{ok}$	$E \vdash_{hm} K == K'$	$E \vdash_{hm} K <: K'$	$E \vdash_{hm} K\,\mathrm{ok}$
$\vdash hm\,\mathrm{ok}$	$E \vdash_{hm} T == T'$	$E \vdash_{hm} T <: T'$	$E \vdash_{hm} T{:}K$
$E \vdash_{hm} \mathrm{ok}$	$E \vdash_{hm} S == S'$	$E \vdash_{hm} S <: S'$	$E \vdash_{hm} S\,\mathrm{ok}$
$E \vdash_\bullet m{:}T$	$E \vdash_{hm} e{:}T$	$E \vdash_{hm} M{:}S$	$E \vdash_{hm} U{:}S$
$\vdash n\,\mathrm{ok}$			

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 la totalité du système de typage définitif.

2.2.1 Correction

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

La correction d'un hachage correspond juste à l'adéquation de la structure à la signature associée.

$$\frac{\mathbf{nil} \vdash_\bullet M{:}[X{:}\mathbf{Le}(T),\, T']}{\vdash \mathbf{hash}(N, M{:}[X{:}\mathbf{Le}(T),\, T'])\,\mathrm{ok}}$$

La correction d'une couleur est simplement la correction des hachages qui la composent.

$$\frac{\vdash h\,\mathrm{ok}}{\vdash \{h\}\,\mathrm{ok}} \qquad \frac{\vdash hm_0\,\mathrm{ok} \quad \vdash hm_1\,\mathrm{ok}}{\vdash hm_0 \cup hm_1\,\mathrm{ok}} \qquad \frac{}{\vdash \bullet\,\mathrm{ok}}$$

La correction d'un environnement se fait en décomposant celui-ci élément par élément.

$$\frac{\vdash hm\,\mathrm{ok}}{\mathbf{nil} \vdash_{hm} \mathrm{ok}} \qquad \frac{E \vdash_{hm} T{:}\mathbf{Le}(\top) \quad x \notin \mathrm{dom}\, E}{E, x{:}T \vdash_{hm} \mathrm{ok}} \qquad \frac{E \vdash_{hm} K\,\mathrm{ok} \quad X \notin \mathrm{dom}\, E}{E, X{:}K \vdash_{hm} \mathrm{ok}} \qquad \frac{E \vdash_{hm} S\,\mathrm{ok} \quad U \notin \mathrm{dom}\, E}{E, U{:}S \vdash_{hm} \mathrm{ok}}$$

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

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

La correction d'un type T s'obtient en prouvant qu'il est un sous-type de $\top$, puis en utilisant la règle ci-dessous permettant de passer du sous-typage au sortage. Les règles pour prouver que tout type syntaxiquement correct est un sous-type de $\top$ sont excessivement simples : nous ne les présentons pas ici.

$$\frac{E \vdash_{hm} T <: T'}{E \vdash_{hm} T{:}\mathbf{Le}(T')}$$

Nous concluons par la correction d'une signature (correction de la sorte et du type) :

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

et d'un réseau (correction des machines qui le composent).

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

2.2.2 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 le hachage correspondant. 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.

$$\frac{E \vdash_{hm} \mathrm{ok}}{\begin{array}{l} E \vdash_{hm} h.\mathrm{TYPE} == T \\ \text{where } h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}\mathbf{Le}(T'), T'']) \quad \in \quad hm \end{array}}$$

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.

$$\frac{E \vdash_{hm} U{:}[X{:}K, T]}{E \vdash_{hm} U.\mathrm{TYPE}{:}K} \qquad \frac{E \vdash_{hm} T{:}\mathbf{Eq}(T')}{E \vdash_{hm} T == T'}$$

Toutes les autres équivalences de types, puis de sortes sont immédiates et ne sont donc pas reproduites ici.

2.2.3 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.

$$\frac{\begin{array}{l} E \vdash_{hm} \mathrm{ok} \qquad \vdash h \,\mathrm{ok} \\ \hline E \vdash_{hm} h.\mathrm{TYPE}{:}K \\ \text{where } h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}K, T']) \end{array}} \qquad \frac{E \vdash_{hm} T{:}\mathbf{Le}(T')}{E \vdash_{hm} T <: T'}$$

Le sous-typage peut aussi provenir de sa construction explicite à partir des enregistrements et des constructeurs du langage.

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

$$\frac{\begin{array}{c} E \vdash_{hm} T_0' <: T_0 \\ E \vdash_{hm} T_1 <: T_1' \end{array}}{E \vdash_{hm} T_0 \to T_1 <: T_0' \to T_1'} \qquad \frac{E \vdash_{hm} T_i <: T_i' \quad 1 \leqslant i \leqslant j}{E \vdash_{hm} T_1 * ... * T_j <: T_1' * ... * T_j'}$$

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

$$\frac{E \vdash_{hm} T == T'}{E \vdash_{hm} T <: T'}$$

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.

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

2.2.4 Typage

Les règles de typage comportent tout d'abord des éléments permettant d'utiliser le sous-typage et les équivalences de type.

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

Nous avons ensuite de quoi typer une variable ou un appel à une valeur de module.

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

Dans la seconde règle, ci-dessus, on impose que le type T ne fasse pas appel à la variable X, mais plutôt à $U.\text{TYPE}$. Cette contrainte est résolue par une règle de signaturage des variables de module que nous verrons plus loin dans la section 2.2.6.

Nous avons bien entendu les règles habituelles 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. On voit 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.

$$\frac{E \vdash_{hm} e{:}T}{E \vdash_{hm} \mathbf{mar}\,(e{:}T){:}\text{STRING}} \qquad \frac{E \vdash_{hm} \text{ok} \quad \mathbf{nil} \vdash_{\bullet} e{:}T}{E \vdash_{hm} \mathbf{marshalled}\,(e{:}T){:}\text{STRING}} \qquad \frac{E \vdash_{hm} T{:}\mathbf{Le}(\top) \quad E \vdash_{hm} e{:}\text{STRING}}{E \vdash_{hm} (\mathbf{unmar}\,e{:}T){:}T}$$

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.

$$\frac{E \vdash_{hm} T{:}\mathbf{Le}(\top)}{E \vdash_{hm} \mathbf{Unmarfailure}^T{:}T}$$

Il nous reste enfin les crochets colorés à typer.

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

Notez que e est typé dans la couleur hm'. Les crochets forment ainsi une frontière de connaissance : 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.

2.2.5 Sortage

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

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

$$\frac{E, X{:}K, E' \vdash_{hm} \text{ok}}{E, X{:}K, E' \vdash_{hm} X{:}K} \qquad \frac{E \vdash_{hm} U{:}[X{:}K, T]}{E \vdash_{hm} U.\text{TYPE}{:}K} \qquad \frac{E \vdash_{hm} \text{ok} \qquad \vdash h \,\text{ok}}{E \vdash_{hm} h.\text{TYPE}{:}K \quad \text{where } h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}K, T'])}$$

2.2.6 Modules

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

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

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 $U.\text{TYPE}$ lorsque l'on a pu associer U à $[X{:}K, T]$.

$$\frac{E \vdash_{hm} U{:}[X{:}K, T]}{E \vdash_{hm} U{:}[X{:}\mathbf{Eq}(U.\text{TYPE}), T]}$$

2.2.7 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.

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

2.3 Sémantique

La sémantique opérationnelle à petit pas est divisée en trois relations de réduction :
- $m \longrightarrow_{\mathsf{c}} m'$ pour la réduction des machines qui forme la compilation ;
- $e \longrightarrow_{hm} e'$ et $n \longrightarrow n'$ pour la réduction des expressions et des réseaux qui se fait pendant l'exécution.

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

2.3.1 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 le hachage du module pour remplacer les références au type du module $U.\text{TYPE}$ par le type haché qui, lui, sera clos. La partie terme du module utilise peut-être 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.

$$\begin{aligned}
\textbf{module}\, N_U \;&= [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''),\, T']\ \textbf{in}\ m \longrightarrow_{\mathsf{c}} \{U.\text{TYPE}{\leftarrow}T'',\, U.\text{term}{\leftarrow}v^\bullet\}m \\
\textbf{module}\, N_U \;&= [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''),\, T']\ \textbf{in}\ m \longrightarrow_{\mathsf{c}} \\
&\qquad \{U.\text{TYPE}{\leftarrow}h.\text{TYPE},\, U.\text{term}{\leftarrow}[v^\bullet]_{\{h\}}^{\{X \leftarrow h.\text{TYPE}\}\, T'}\}m \\
&\qquad \text{where} \quad h = \mathbf{hash}(N, ([T, v^\bullet]{:}[X{:}\mathbf{Le}(T''),\, T']))
\end{aligned}$$

2.3.2 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 donc nécessaire d'annoter la relation de réduction par la couleur courante de la réduction.

Les réductions des expressions seront intensément discutées dans le chapitre 4. Nous n'en présenterons donc ici que quelques unes.

Les valeurs, notées v^{hm}, sont conçues pour être les expressions irréductibles par ces règles. Leur utilisation permet d'avoir une sémantique déterministe.

Dans le cadre de notre λ-calcul en appel par valeur, certaines des règles sont simples :

$$\mathbf{proj}_i\,(v_1^{hm}, ..., v_j^{hm}) \longrightarrow_{hm} v_i^{hm} \qquad \text{if } 1 \leqslant i \leqslant j$$

$$\{l_1 = v_1^{hm}, ..., l_j = v_j^{hm}\}.l_i \longrightarrow_{hm} v_i^{hm} \qquad \text{if } 1 \leqslant i \leqslant j$$

tandis que la β-réduction ne l'est pas, puisqu'il faut ajouter des crochets interdisant à l'argument d'être placé dans une couleur différente de la couleur ambiante.

$$(\lambda x{:}T.e)\, v^{hm} \longrightarrow_{hm} \{x \leftarrow [v^{hm}]^{T}_{hm}\} e$$

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.

$$\mathbf{mar}\,(v^{hm}{:}T) \longrightarrow_{hm} \mathbf{marshalled}\,([v^{hm}]^{T}_{hm}{:}T)$$

La désérialisation va tester la correspondance entre les types envoyés et attendus. Cette correspondance est ici le sous-typage.

$$\mathbf{unmar}\,(\mathbf{marshalled}\,(v^{\bullet}{:}T){:}T') \longrightarrow_{hm} \begin{cases} v^{\bullet} & \text{if } \mathbf{nil} \vdash_{hm} T <: T' \\ \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

Comment se comportent les crochets colorés dans la sémantique ? Ceux-ci vont être poussés vers les feuilles des arbres syntaxiques pour qu'un crochet coloré ne puisse interférer avec les relations entre constructeurs et destructeurs.

$$[()]^{\text{UNIT}}_{hm'} \longrightarrow_{hm} ()$$

$$[(v^{hm'}, ..., v^{hm'}_{j})]^{T_1 * ... * T_j}_{hm'} \longrightarrow_{hm} ([v^{hm'}_{1}]^{T_1}_{hm'}, ..., [v^{hm'}_{j}]^{T_j}_{hm'})$$

$$[\{l_1 = v^{hm'}_{1}, .., l_j = v^{hm'}_{j}\}]^{\{l_1:T_1,...,l_j:T_j\}}_{hm'} \longrightarrow_{hm} \{l_1 = [v^{hm'}_{1}]^{T_1}_{hm'}, .., l_j = [v^{hm'}_{j}]^{T_j}_{hm'}\}$$

$$[\lambda x{:}T.e]^{T' \to T''}_{hm'} \longrightarrow_{hm} (\lambda x{:}T'.[\{x \leftarrow [x]^{T}_{hm'}\} e]^{T''}_{hm'})$$

$$[\mathbf{marshalled}\,(v^{\bullet}{:}T)]^{\text{STRING}}_{hm'} \longrightarrow_{hm} \mathbf{marshalled}\,(v^{\bullet}{:}T)$$

Il reste la question de la révélation d'un type abstrait. Voici la règle permettant de l'effectuer. Nous notons impl(h) le type concret dont h.TYPE est l'abstraction. Remarquez la condition $h \in hm \cap hm'$. Elle impose que le hachage h soit connu 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 autres règles qui permettront de les déconstruire.

$$[v^{hm'}]^{h.\text{TYPE}}_{hm'} \longrightarrow_{hm} [v^{hm'}]^{T}_{hm'} \qquad \text{when } h \in hm \cap hm' \text{ and impl}(h){=}T$$

Nous finissons cette présentation des règles de réduction des expressions par la réduction au sein d'un contexte qui permet, par exemple, de réduire à l'intérieur des crochets colorés. La définition exacte des contextes, notés $C^{hm'}_{hm}$, peut être trouvée en section 4.1.2 ou en annexe.

$$\frac{e \longrightarrow_{hm} e'}{C^{hm'}_{hm}.e \longrightarrow_{hm'} C^{hm'}_{hm}.e'}$$

Un certain nombre de ces règles seront modifiées dans les chapitres suivants pour atteindre un système cohérent.

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 typage impose comme une chaîne de caractères (le résultat de la sérialisation d'une valeur arbitraire).

$$CC^{\bullet}_{hm}.!\,v^{hm} \mid CC^{\bullet}_{hm'}.? \longrightarrow CC^{\bullet}_{hm}.() \mid CC^{\bullet}_{hm'}.v^{hm}$$

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

Chapitre 3

Abstraction et sous-typage

La première question que l'on peut se poser en ajoutant du sous-typage à un langage permettant la création de types abstraits, est l'éventuelle interférence entre ces deux éléments. Deux types abstraits peuvent-ils être sous-types l'un de l'autre ? Et s'ils ne sont que partiellement abstraits (i.e. de sorte $\mathbf{Le}(T)$ avec T différent de $\top$) ? Quelles contraintes impose l'environnement distribué ?

Dans ce chapitre nous allons donc essayer d'adapter notre langage en fonction des contraintes que nous mettrons au jour grâce à quelques exemples.

Exemple 3.0.1 Ce premier exemple servira de fil conducteur dans ce chapitre. Il propose ainsi deux programmes distincts qui correspondent, dans l'esprit, à deux versions successives mais compatibles d'une même bibliothèque de compteurs.

Le premier programme propose une implémentation par type abstrait d'un compteur que l'on peut initialiser, incrémenter et dont on peut récupérer la valeur. Le programme va tenter d'envoyer un compteur par le réseau.

```
Programme1 :
  module CompteurA =
    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                          in
    let c = CompteurA.incremente (CompteurA.initial) in
      send (marshall (c : CompteurA.t))
```

Le second programme comporte la même implémentation des compteurs, à l'exception de la présence d'une nouvelle fonction qui permet de doubler en un appel la valeur du compteur. Le reste est identique. Ce deuxième programme va essayer de récupérer un compteur par le réseau et d'en afficher la valeur doublée.

```
Programme2 :
  module CompteurB =
    struct                        sig
```

```
  type t = int                  type t
  let initial = 0          :    val initial : t
  let incremente x = x + 1      val incremente : t -> t
  let double x = x * 2          val double : t -> t
  let valeur x = x              val valeur : t -> int
end                           end                          in
print_int (CompteurB.double (unmarshall (receive ()):CompteurB.t))
```

Mettons maintenant ces deux programmes sur deux machines reliées par le réseau.

Programme1 || *Programme2*

Quel devrait être le résultat ? Un échec, puisque les types sont abstraits, ou une réussite, puisqu'ils sont compatibles ? Toute la question revient à étudier le sous-typage entre types abstraits.

3.1 Une solution simple ?

Examinons toutes les possibilités de définir automatiquement une relation de sous-typage entre types abstraits. C'est-à-dire que nous cherchons à savoir s'il existe une relation entre les hachages qui engendrerait directement une relation de sous-typage de la manière suivante.

$$\frac{E \vdash_{hm} h <: h'}{E \vdash_{hm} h.\text{TYPE} <: h'.\text{TYPE}}$$

Tout d'abord, il est nécessaire de comprendre dans quelles conditions un tel test de sous-typage peut être amené à être effectué. C'est le cas par exemple à l'exécution, lors de la désérialisation, alors que l'on n'a pas accès aux modules mis en jeu, mais seulement à leur hachage. Or nous n'avons demandé aux hachages de ne comporter en clair (i.e. sous forme non hachée) que le type implémentation et sa sorte. Il parait raisonnable d'étendre cette présence en clair à la signature toute entière. Par contre, pour des raisons d'efficacité, il n'est pas souhaitable de fonder une vérification qui a lieu à l'exécution sur l'examen d'un quelconque code source. On peut rétorquer que toute preuve d'un jugement $E \vdash_{hm} T <: T'$ possède une sous-preuve de correction de la couleur qui va examiner l'intérieur des hachages (donc y compris le code source). Cette preuve n'a cependant aucune raison d'avoir lieu lors de l'exécution, et l'on peut imaginer aisément un moyen de vérification des hachages lors de la compilation, avec échange de certificats de conformité des hachages entre correspondants. La comparaison entre les hachages ne peut, elle, que se faire à l'exécution, et c'est pourquoi nous repoussons l'éventualité d'une comparaison des sources : nous n'autorisons que le test d'égalité.

Supposons donc, pour l'instant, qu'un type haché comporte, en clair, son type implémentation et sa signature. Nous allons maintenant étudier si la relation $h <: h'$ peut être liée aux relations déjà connues : sous-typage, sous-signaturage.

Étudions en premier si l'utilisation de la relation de sous-signaturage est compatible avec la préservation de l'abstraction. Pour bien séparer les différentes hypothèses nous supposons ici les implémentations (types et valeurs) identiques. Éliminons tout d'abord le cas le plus improbable : la relation de sous-typage se fait de manière contravariante par rapport au sous-signaturage, c'est-à-dire qu'un hachage est un sous-hachage d'un autre si leurs signatures sont dans une relation inverse

de sous-signaturage, les implémentations étant égales. On peut remarquer que cela correspond au cas mis en exergue par l'exemple introductif.

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_1, ([T_0, v^{hm}]{:}[X{:}K_0,\, T_0'])) \text{ ok} \\ \vdash \mathbf{hash}(N_2, ([T_0, v^{hm}]{:}[X{:}K_1,\, T_1'])) \text{ ok} \\ E \vdash_{hm} [X{:}K_1,\, T_1'] <: [X{:}K_0,\, T_0']\end{array}}{E \vdash_{hm} \mathbf{hash}(N_1, ([T_0, v^{hm}]{:}[X{:}K_0,\, T_0'])) <: \mathbf{hash}(N_2, ([T_0, v^{hm}]{:}[X{:}K_1,\, T_1']))}$$

Cette règle pose malheureusement un gros problème, puisqu'elle permet l'accès à un champ caché comme le montre l'exemple suivant.

Exemple 3.1.1 Nous avons deux représentations de comptes en banque similaires à celles de l'exemple 1.2.3. L'un cache la partie suisse, l'autre la laisse visible, cela grâce à des abstractions partielles. Un des programmes va sérialiser un compte et l'autre va essayer de le désérialiser et d'accéder au champ `suisse`.

```
Programme1 :
module Compte_1 =
  struct
    type t = { france : int ; suisse : int }
    let v = { france = 1 ; suisse = 10000000 }
  end :
  sig
    type t : Le ( { france : int } )
    val v : t
  end in
  send ( marshall (Compte_1.v : Compte_1.t) )

Programme2 :
module Compte_2 =
  struct
    type t = { france : int ; suisse : int }
    let v = { france = 10 ; suisse = 0 }
  end :
  sig
    type t : Le ( { france : int ; suisse : int } )
    val v : t
  end in
  print_int ((unmarshall ( receive ()) : Compte_2.t) .suisse )
```

Mettons maintenant ces deux programmes sur deux machines reliées par le réseau.

```
Programme1 || Programme2
```

La règle de sous-hachage proposée ci-dessus permet d'accepter la communication et de faire réussir la désérialisation. Or il n'est naturellement pas souhaitable que la partie suisse soit accessible au tout-venant.

Comme cette première proposition de sous-hachage permet de casser les abstractions, nous la rejetons.

Essayons donc maintenant dans le sens covariant avec la règle suivante :

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_1, ([T_0, v^{hm}]{:}[X{:}K_0,\, T_0'])) \text{ ok} \\ \vdash \mathbf{hash}(N_2, ([T_0, v^{hm}]{:}[X{:}K_1,\, T_1'])) \text{ ok} \\ E \vdash_{hm} [X{:}K_0,\, T_0'] <: [X{:}K_1,\, T_1']\end{array}}{E \vdash_{hm} \mathbf{hash}(N_1, ([T_0, v^{hm}]{:}[X{:}K_0,\, T_0'])) <: \mathbf{hash}(N_2, ([T_0, v^{hm}]{:}[X{:}K_1,\, T_1']))}$$

Rappelons la règle permettant d'établir le sous-signaturage $E \vdash_{hm} [X{:}K_0,\, T_0'] <: [X{:}K_1,\, T_1']$.

$$\frac{E \vdash_{hm} K <: K' \qquad E, X{:}K \vdash_{hm} T <: T'}{E \vdash_{hm} [X{:}K,\, T] <: [X{:}K',\, T']}$$

La correction est alors assurée : on ne pourra pas utiliser une valeur abstraite de manière contraire aux informations qui nous sont données. C'est là pourtant que le bât blesse : le programmeur peut avoir assorti son abstraction d'invariants qui ne sont pas exprimés dans le système de type. Examinons l'exemple suivant.

Exemple 3.1.2 Prenons la structure `M` suivante :

```
struct
  type t = { v : int }
  let v  = { initial = { v = 2 } ;
             double = function (x:{ v : int }) -> { v = x.v * 2 } ;
             incr = function (x:{ v : int }) -> { v = x.v + 1 } ;
             get = function (x:{ v : int }) -> x.v }
end
```

Elle est similaire à celle du `CompteurB` de l'exemple 3.0.1, tout en étant écrite avec l'utilisation d'une seule valeur. Elle définit un compteur et une valeur permettant son initialisation et sa manipulation. Donnons à cette structure deux signatures différentes et essayons de les faire communiquer.

```
Programme1 :
module Compteur = M :
  sig
    type t
    val v : { initial : t ;
              double : t -> t ;
              incr : t -> t ;
              get : t -> int
            }
  end in
send ( marshall (Compteur.v.incr (Compteur.v.initial) :Compteur.t))
```

Ce premier module permet toutes les manipulations codées.

```
Programme2 :
module Compteur_pair = M :
  sig
    type t
    val v : { initial : t ;
              double : t -> t ;
              get : t -> int
            }
  end in
print_int (Compteur_pair.v.get (unmarshall (receive ()) :Compteur_pair.t))
```

Ce second module ne permet plus l'incrémentation du compteur, mais permet seulement son doublement. Il vérifie donc l'invariant suivant : `Compteur_pair.v.get` ne donne que des valeurs paires. Cependant sa signature est plus générale que celle de `Compteur` : le sous-typage permet donc de désérialiser un compteur du module `Compteur` comme un compteur du module `Compteur_pair`. Mettons alors ces deux programmes sur deux machines en communication.

Programme1 || *Programme2*

Or à ce moment-là l'invariant ci-dessus est rompu, puisque le compteur du module `Compteur` avait une valeur impaire lors de l'envoi.

Cet exemple nous permet ainsi de voir en quoi une relation de sous-typage liée au sous-signaturage n'est pas désirable. On doit donc demander l'équivalence des signatures.

Une relation de sous-typage entre types abstraits fondée uniquement sur le type implémentation n'est pas plus raisonnable. On se voit donc forcé à renoncer à cette idée. De plus, l'exemple précédent montre bien que l'abstraction que le programmeur veut préserver n'est pas toute entière inscrite dans les types et les signatures.

3.2 Une solution : la spécification par l'utilisateur

On va donner au programmeur la possibilité de définir lui-même si son module peut importer de façon sûre des valeurs d'un module défini précédemment et, de manière symétrique, si les valeurs abstraites de son module peuvent être utilisées par d'autres modules. Il sera alors responsable des abstractions et invariants qu'il a lui-même créés.

Cependant on ne peut pas laisser l'utilisateur entièrement juge sur la compatibilité de deux types abstraits. Il est nécessaire d'imposer des contraintes au moins pour certifier l'absence d'erreur à l'exécution.

Il faut tout d'abord être sûr de la compatibilité effective des types abstraits, c'est-à-dire au niveau de l'implémentation, pour garantir l'absence d'erreur à l'exécution. On impose donc aux deux types abstraits qui sont déclarés être en relation de sous-typage d'avoir leur implémentation réelle dans la même relation de sous-typage, et ce de manière covariante.

La seconde contrainte est que l'information disponible (en cas d'abstraction partielle) soit plus faible du côté du type le plus général. En effet, si l'on peut utiliser une valeur complètement abstraite comme une valeur partiellement abstraite, la préservation de l'abstraction échoue (cf exemple 3.1.1).

Pour résumer, il est donc nécessaire de vérifier si une relation déclarée de sous-typage est correcte de la manière suivante.

$$\frac{\begin{array}{l}\vdash \mathbf{hash}(N_0, [T_0, v_0^\bullet]{:}[X{:}K_0,\, T_0']) \text{ ok} \\ \vdash \mathbf{hash}(N_1, [T_1, v_1^\bullet]{:}[X{:}K_1,\, T_1']) \text{ ok} \\ E \vdash_{hm} T_0 <: T_1 \\ E \vdash_{hm} K_0 <: K_1\end{array}}{E \vdash_{hm} \mathbf{hash}(N_0, [T_0, v_0^\bullet]{:}[X{:}K_0,\, T_0']).\text{TYPE} <: \mathbf{hash}(N_1, [T_1, v_1^\bullet]{:}[X{:}K_1,\, T_1']).\text{TYPE}}$$

Remarquons qu'une comparaison entre T_0' et T_1' aurait été superflue, puisqu'il n'est en aucun cas possible d'extraire d'un hachage ni le terme ni son type pour en produire un quelconque jugement. Les types hachés ne mettent en jeu que la déclaration de type du module et la sorte associée.

Nous pouvons donc maintenant ajouter à la syntaxe la possibilité de déclarer ces relations entre modules via les mots-clé **extends** M et **restricts** M que le programmeur peut ajouter lors de la déclaration. Utilisons la métavariable κ pour couvrir les hachages et les variables de module. Le programmeur ne peut utiliser directement que ces dernières, bien entendu.

Machines (programmes complets) :

$m ::= e$	expression
$\quad \mathbf{module}\, N_U\, \mathbf{extends}\, \kappa\, \mathbf{restricts}\, \kappa' = M{:}S\, \mathbf{in}\, m$	déclaration de module (U lié dans m)

On autorise même la référence à plusieurs modules avec la syntaxe **extends** $\kappa_1 ... \kappa_i$ et **restricts** $\kappa_1' ... \kappa_j'$. La présence de ces mots-clé dans la déclaration d'un module dont le type est manifeste sera ignorée.

3.3 Conséquences

Un jugement de sous-typage peut donc dépendre maintenant des déclarations explicites dans les modules. Nous devons donc enrichir nos jugements par l'annotation H qui rassemblera sous une forme non-ordonnée les couples de hachages et/ou variables de module qui ont été déclarés sous-types l'un de l'autre.

H pourra alors être utilisé pour prouver des jugements de sous-typage.

$$\frac{E \vdash_{hm}^H \text{ ok} \qquad \kappa <: \kappa' \in H}{E \vdash_{hm}^H \kappa.\text{TYPE} <: \kappa'.\text{TYPE}}$$

Plus généralement, les jugements deviennent :

Jugements :

$\vdash h$ ok	$E \vdash_{hm}^H K == K'$	$E \vdash_{hm}^H K <: K'$	$E \vdash_{hm}^H K$ ok
$\vdash hm$ ok	$E \vdash_{hm}^H T == T'$	$E \vdash_{hm}^H T <: T'$	$E \vdash_{hm}^H T{:}K$
$E \vdash_{hm}^H$ ok	$E \vdash_{hm}^H S == S'$	$E \vdash_{hm}^H S <: S'$	$E \vdash_{hm}^H S$ ok
$E \vdash_{\bullet}^H m{:}T$	$E \vdash_{hm}^H e{:}T$	$E \vdash_{hm}^H M{:}S$	$E \vdash_{hm}^H U{:}S$
$\vdash n$ ok			

Beaucoup des règles d'inférences sont cependant inchangées : une métavariable H annotant de façon identique les prémisses et la conclusion. Nous décrivons cependant ci-dessous toutes les règles modifiées.

Il faut d'abord réfléchir à certaines conséquences prévisibles, à commencer par la correction de cette relation H. Nous avons vu dans la section précédente les contraintes qui pesaient sur cette correction, et notamment qu'elle nécessitait la connaissance du type implémentation et de la sorte correspondante. Comme les variables de modules peuvent se retrouver dans H, il est nécessaire d'enrichir les déclarations des variables de module dans les environnements par le type implémentation correspondant. On verra donc des environnements comportant $U(T){:}S$ à la place du simple $U{:}S$.

Notons que cette présence du type concret dans l'environnement perturbe la notion usuelle de compilation séparée selon laquelle on peut compiler un module seulement avec les interfaces des modules dont il dépend. Ici, nous demandons en pratique que ces interfaces comportent, en plus de la signature, la réelle implémentation des éventuels types abstraits.

Pour illustrer ces premières modifications, voici la règle de typage de la déclaration des modules dans une machine. C'est ici que H est enrichie à la base.

$$\frac{\begin{array}{l} E \vdash^{H}_{\bullet} T{:}\mathbf{Le}(\top) \\ E \vdash^{H}_{\bullet} [T_0, v^{\bullet}]{:}S \\ E, U(T_0){:}S \vdash^{H\cup\{\kappa_1<:U\}\cup\ldots\cup\{\kappa_i<:U\}\cup\{U<:\kappa'_1\}\cup\ldots\cup\{U<:\kappa'_j\}}_{\bullet} m{:}T \end{array}}{E \vdash^{H}_{\bullet} (\mathbf{module}\, N_U\, \mathbf{extends}\, \kappa_1 \ldots \kappa_i\, \mathbf{restricts}\, \kappa'_1 \ldots \kappa'_j = [T_0, v^{\bullet}]{:}S\, \mathbf{in}\, m){:}T}$$

Nous rappelons que les mots-clés **extends** M et **restricts** M sont ignorés lorsque le type est déclaré manifeste. H n'est alors bien sûr pas enrichi.

Une seconde conséquence prévisible est la dépendance de la correction du hachage d'un module vis-à-vis de la relation H. Or les hachages doivent rester clos, la seule solution est donc d'enrichir les hachages avec une mention de la déclaration de sous-hachage dans laquelle ils ont été créés.

Hachages :
$h ::= \mathbf{hash}(N, H, M{:}[X{:}\mathbf{Le}(T), T])$ hachage

Exemple 3.3.1 Montrons dans un exemple la dépendance que peut posséder un hachage vis-à-vis de la relation H. Nous décrivons ci-dessous trois modules simples. Le second module étend le premier et se déclare compatible en utilisant la construction **extends** U . Le troisième module utilise l'implémentation du second module tout en déclarant son type comme sous-type du premier. La vérification de cette propriété fait évidemment appel à la relation H.

```
module ModuleA =
  struct                          sig
    type t = {l1:int}         :     type t <: {l1:int}
    let v = {l1=1}                  val v : t
  end                             end
in
module ModuleB extends ModuleA =
  struct                          sig
    type t = {l1:int;l2:int}  :     type t <: {l1:int}
    let v = {l1=1 ; l2=3}           val v : t
  end                             end
in
module ModuleC =
  struct                          sig
```

```
    type t = ModuleB.t        :    type t <: ModuleA.t
    let v = ModuleB.v              val v : t
  end                            end
in
(ModuleC.v).l1
```

Une fois la compilation effectuée, nous utilisons le résultat du hachage du troisième module, `ModuleC`, dans l'expression finale. Ainsi lorsque nous aurons à prouver un jugement au sujet de cette expression nous aurons à prouver la correction de ce hachage, et donc de l'adéquation entre la structure et la signature de ce troisième module. Or celle-ci dépend de la relation H. L'enrichissement du hachage par H est donc rendu nécessaire.

Notez que la définition des modules réclame que l'expression déclarée par le troisième module soit une valeur, ce qui n'est pas le cas ici pour des raisons de clarté. On peut résoudre facilement ce problème en utilisant une fonction pour envelopper l'expression.

L'ensemble H peut ainsi croître à chaque déclaration de module, il change donc potentiellement dans les jugements de typage de machines $E \vdash^H_\bullet m{:}T$. Mais H change-t-il dans les autres jugements ou y reste-t-il une annotation statique ?

Une première réponse peut-être apportée par les jugements de correction. En effet, dans un jugement du type $E \vdash^H_{hm}$ ok, on va déconstruire l'environnement pour en tester sa correction. Or H peut contenir des variables de modules, et dépend donc de l'environnement pour sa preuve de correction. De même la correction de l'environnement peut nécessiter certaines relations de sous-typage de H.

Nous sommes donc en présence d'une dépendance mutuelle.

Exemple 3.3.2 Reprenons l'exemple précédent et étudions le jugement de typage de la machine qui lui est associé. Réécrivons l'exemple dans la notation concise.

$$\begin{aligned}
\textbf{module}\,\mathtt{A}_{U0} &= [\{l_1{:}\text{INT}\},\{l_1=1\}]{:}[X{:}\mathbf{Le}(\{l_1{:}\text{INT}\}),X]\ \textbf{in}\\
\textbf{module}\,\mathtt{B}_{U1}\,\textbf{extends}\,U_0 &= [\{l_1{:}\text{INT},l_2{:}\text{INT}\},\{l_1=1,l_2=2\}]{:}[X{:}\mathbf{Le}(\{l_1{:}\text{INT}\}),X]\ \textbf{in}\\
\textbf{module}\,\mathtt{C}_V &= [U_1.\text{TYPE},U_1.\text{term}]{:}[X{:}\mathbf{Le}(U_0.\text{TYPE}),X]\ \textbf{in}\\
&\quad (V.\text{term}).l_1
\end{aligned}$$

La preuve du jugement de typage de la machine associé va débuter par le typage du premier module, ce qui va enrichir l'environnement pour qu'il devienne : $U_0(\{l_1{:}\text{INT}\}){:}[X{:}\mathbf{Le}(\{l_1{:}\text{INT}\}),X]$. Après le second module, il sera enrichi par la liaison : $U_1(\{l_1{:}\text{INT},l_2{:}\text{INT}\}){:}[X{:}\mathbf{Le}(\{l_1{:}\text{INT}\}),X]$, et H vaudra alors $\{U_1 <: U_0\}$. Enfin, au troisième module on mettra dans l'environnement $V(h_1.\text{TYPE}){:}[X{:}\mathbf{Le}(h_0.\text{TYPE}),X{\to}\text{UNIT}]$.

Ainsi, pour vérifier la correction de cet environnement faudra-t-il d'abord enlever la liaison du troisième module V (on ne peut dépouiller H puisque la correction de V en dépend), puis la déclaration de H (puisqu'il dépend de la présence de U_0 et U_1 dans l'environnement), puis les seconds et premiers modules. La dépendance mutuelle est donc bien présente.

Voici donc maintenant les règles permettant de vérifier la correction de l'environnement et de la relation de sous-typage H.

$$\frac{\vdash hm\,\mathrm{ok}}{\mathbf{nil} \vdash^{\mathbf{nil}}_{hm} \mathrm{ok}} \qquad \frac{E \vdash^H_{hm} T{:}\mathbf{Le}(\top) \quad x \notin \mathrm{dom}\, E}{E, x{:}T \vdash^H_{hm} \mathrm{ok}}$$

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_0, H_0, [T_0, v_0^\bullet]{:}[X{:}K_0,\, T_0']) \,\mathrm{ok} \\ \vdash \mathbf{hash}(N_1, H_1, [T_1, v_1^\bullet]{:}[X{:}K_1,\, T_1']) \,\mathrm{ok} \\ \mathbf{nil} \vdash^H_{hm} T_0 <: T_1 \\ \mathbf{nil} \vdash^H_{hm} K_0 <: K_1\end{array}}{\mathbf{nil} \vdash^{H \cup \{\mathbf{hash}(N_0,H_0,[T_0,v_0^\bullet]:[X:K_0,T_0']) <: \mathbf{hash}(N_1,H_1,[T_1,v_1^\bullet]:[X:K_1,T_1'])\}}_{hm} \mathrm{ok}}$$

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_0, H_0, [T_0, v_0^\bullet]{:}[X{:}K_0,\, T_0']) \,\mathrm{ok} \\ E, U(T_1){:}[X{:}K_1,\, T_1'] \vdash^H_{hm} T_0 <: T_1 \\ E, U(T_1){:}[X{:}K_1,\, T_1'] \vdash^H_{hm} K_0 <: K_1\end{array}}{E, U(T_1){:}[X{:}K_1,\, T_1'] \vdash^{H \cup \{\mathbf{hash}(N_0,H_0,[T_0,v_0^\bullet]:[X:K_0,T_0']) <: U\}}_{hm} \mathrm{ok}}$$

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_1, H_1, [T_1, v_1^\bullet]{:}[X{:}K_1,\, T_1']) \,\mathrm{ok} \\ E, U(T_0){:}[X{:}K_0,\, T_0'] \vdash^H_{hm} T_0 <: T_1 \\ E, U(T_0){:}[X{:}K_0,\, T_0'] \vdash^H_{hm} K_0 <: K_1\end{array}}{E, U(T_0){:}[X{:}K_0,\, T_0'] \vdash^{H \cup \{U <: \mathbf{hash}(N_1,H_1,[T_1,v_1^\bullet]:[X:K_1,T_1'])\}}_{hm} \mathrm{ok}}$$

$$\frac{\begin{array}{c}E_1, U_1(T_1){:}[X{:}K_1,\, T_1'], E_2, U_0(T_0){:}[X{:}K_0,\, T_0'] \vdash^H_{hm} T_0 <: T_1 \\ E_1, U_1(T_1){:}[X{:}K_1,\, T_1'], E_2, U_0(T_0){:}[X{:}K_0,\, T_0'] \vdash^H_{hm} K_0 <: K_1\end{array}}{E_1, U_1(T_1){:}[X{:}K_1,\, T_1'], E_2, U_0(T_0){:}[X{:}K_0,\, T_0'] \vdash^{H \cup \{U_0 <: U_1\}}_{hm} \mathrm{ok}}$$

$$\frac{\begin{array}{c}E_1, U_0(T_0){:}[X{:}K_0,\, T_0'], E_2, U_1(T_1){:}[X{:}K_1,\, T_1'] \vdash^H_{hm} T_0 <: T_1 \\ E_1, U_0(T_0){:}[X{:}K_0,\, T_0'], E_2, U_1(T_1){:}[X{:}K_1,\, T_1'] \vdash^H_{hm} K_0 <: K_1\end{array}}{E_1, U_0(T_0){:}[X{:}K_0,\, T_0'], E_2, U_1(T_1){:}[X{:}K_1,\, T_1'] \vdash^{H \cup \{U_0 <: U_1\}}_{hm} \mathrm{ok}}$$

$$\frac{E \vdash^H_{hm} K\,\mathrm{ok} \quad X \notin \mathrm{dom}\, E}{E, X{:}K \vdash^H_{hm} \mathrm{ok}} \qquad \frac{\begin{array}{c}E \vdash^H_{hm} T_0{:}K \\ E \vdash^H_{hm} [X{:}K,\, T]\,\mathrm{ok} \\ U \notin \mathrm{dom}\, E \quad U \notin \mathrm{dom}\, H\end{array}}{E, U(T_0){:}[X{:}K,\, T] \vdash^H_{hm} \mathrm{ok}}$$

Quelques remarques :

– Il n'y a pas d'ambiguité dans ces règles puisque l'on y réduit tour à tour l'environnement et la relation H en tenant compte des éventuelles liaisons de la première variable de l'environnement. A la fin il ne reste plus qu'à vérifier la partie close de H.

– Le jugement $U \notin \mathrm{dom}\, H$ est défini simplement par :

$$\frac{}{U \notin \mathrm{dom}\, \mathbf{nil}} \qquad \frac{U \notin \mathrm{dom}\, H \quad U \neq \kappa \quad U \neq \kappa'}{U \notin \mathrm{dom}\,(H \cup \{\kappa <: \kappa\})}$$

Concernant la question de l'éventuelle changement de l'annotation H, la réponse n'est pas encore complète. Examinons en effet le cas d'une relation de sous-typage déclarée par un programmeur sur une machine. Doit-elle être prise en compte par la machine à laquelle on va envoyer une valeur abstraite ?

Nous avons ici le choix de ne rien transmettre, ce qui revient à imposer que les déclarations soient identiques chez les deux programmeurs pour que H soit utile dans un cadre distribué. Nous avons fait le choix de le transmettre à chaque fois. Cela laisse ainsi toute liberté au receveur d'accepter ou non ces relations de sous-typage suivant les choix du programmeur (il peut vouloir protéger un module par exemple). Dans notre langage, nous avons supposé que le programmeur, par sa déclaration explicite, prenait ses responsabilités quant à la préservation de l'abstraction, et que donc la relation H était transmise, et imposée, chez le correspondant.

Pour effectuer cette propagation, il est alors nécessaire d'enrichir les objets sérialisés par la relation close H.

Expressions :

$e ::= ... \mid \mathbf{marshalled}_{H}(e{:}T) \mid ...$ résultat de la sérialisation

La vérification de type tient alors compte de cet ajout.

$$\frac{\begin{array}{c} E \vdash^{H_0}_{hm} \text{ ok} \\ \mathbf{nil} \vdash^{H_1}_{\bullet} e{:}T \end{array}}{E \vdash^{H_0}_{hm} \mathbf{marshalled}_{H_1}(e{:}T){:}\text{STRING}}$$

Cette nouvelle annotation H sera alors ajoutée à la relation courante lors de la désérialisation. Comme cette désérialisation a lieu à l'exécution, il est nécessaire d'enrichir les règles de réduction de notre sémantique à petits pas. Elles deviennent alors :

- $H, m \longrightarrow_{\mathsf{c}} H', m'$ pour la réduction des machines pendant la compilation ;
- $H, e \longrightarrow_{hm} H', e'$ et $n \longrightarrow n'$ pour la réduction des expressions et des réseaux pendant l'exécution.

Après avoir intégré tous ces changements, les règles de réduction des déclarations de modules dans les machines deviennent alors :

$$H, \mathbf{module}\, N_U = [T, v^{\bullet}]{:}[X{:}\mathbf{Eq}(T''), T'] \,\mathbf{in}\, m \longrightarrow_{\mathsf{c}} H, \{U.\text{TYPE}{\leftarrow}T'', U.\text{term}{\leftarrow}v^{\bullet}\}m$$

$$\begin{array}{l} H, \mathbf{module}\, N_U \,\mathbf{extends}\, h_1 ... h_i \,\mathbf{restricts}\, h'_1 ... h'_j = [T, v^{\bullet}]{:}[X{:}\mathbf{Le}(T''), T'] \,\mathbf{in}\, m \longrightarrow_{\mathsf{c}} \\ \qquad H \cup \{h_1 <: h\} \cup ... \cup \{h_i <: h\} \cup \{h <: h'_1\} \cup ... \cup \{h <: h'_j\}, \\ \qquad \{U{\leftarrow}h,\, U.\text{TYPE}{\leftarrow}h.\text{TYPE},\, U.\text{term}{\leftarrow}[v^{\bullet}]^{\{X \leftarrow h.\text{TYPE}\}\, T'}_{\{h\}}\}m \\ \qquad \text{where} \quad h = \mathbf{hash}(N, H, [T, v^{\bullet}]{:}[X{:}\mathbf{Le}(T''), T']) \end{array}$$

On notera la nouvelle substitution $\{U{\leftarrow}h\}$ qui a vocation à agir dans les autres déclarations d'extension ou restriction.

La transmission de H d'une machine à l'autre s'observe sur les règles de sérialisation et de désérialisation.

$$H, \mathbf{mar}\,(v^{hm}{:}T) \longrightarrow_{hm} H, \mathbf{marshalled}_{H}([v^{hm}]^{T}_{hm}{:}T)$$

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{H'}(v^\bullet{:}T){:}T') \longrightarrow_{hm} \begin{cases} H \cup H', v^\bullet & \text{if } \mathbf{nil} \vdash_{hm}^{H \cup H'} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases} \tag{3.1}$$

Lorsque la désérialisation réussit, H' est ainsi ajouté à la relation courante.

3.4 Retour sur l'exemple initial

Nous proposons donc une relation explicite de compatibilité entre les modules. Voyons donc comment celle-ci s'applique à l'exemple donné au tout début de ce chapitre.

```
Programme1 :
  module CompteurA =
    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                          in
    let c = CompteurA.incremente (CompteurA.initial) in
      send (marshall (c : CompteurA.t))
```

Le premier programme est inchangé : il ne possède que la version initiale des compteurs.

Le second programme connait l'implémentation initiale des compteurs, mais en a réalisé une meilleure version cependant entièrement compatible avec la première, bien que possédant des fonctions supplémentaires. Le programmeur prend alors la responsabilité de la préservation de ses invariants et va donc permettre l'importation de valeurs venant du `CompteurA` vers le `CompteurB`. Il utilise pour cela le mot-clé **restricts** .

```
Programme2 :
  module CompteurA =
    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                          in
  module CompteurB extends CompteurA restricts CompteurA =
    struct                        sig
      type t = int                  type t
      let initial = 0          :    val initial : t
      let incremente x = x + 1      val incremente : t -> t
      let double x = x * 2          val double : t -> t
      let valeur x = x              val valeur : t -> int
    end                           end                          in
    print_int
```

```
(CompteurB.valeur
  (CompteurB.double (unmarshall (receive ()):CompteurB.t)))
```

Mettons maintenant ces deux programmes sur deux machines reliées par le réseau.

Programme1 `||` *Programme2*

La désérialisation va donc réussir puisqu'il est prouvable, grâce à la relation explicitement déclarée par le programmeur, que `CompteurA.t <: CompteurB.t`. On verra donc s'afficher 2 sur l'écran !

Chapitre 4

Couleurs et sous-typage

Nous avons vu au chapitre 2 que les crochets colorés étaient les résultats de la β-réduction et surtout de la réduction des déclarations de modules. Ils ont pour fonction de délimiter les sous-expressions pour lesquelles on a accès à l'implémentation de certains types abstraits. Cependant, pour des questions de typage, une annotation est présente sur les crochets en plus de la couleur. L'interaction entre les crochés colorés et le sous-typage va donc se faire à ce niveau-là.

Exemple 4.0.1 Reprenons notre exemple de compte en banque. Le programme suivant devrait, selon toute logique, afficher la valeur du compte en banque, puisqu'il est connu par l'abstraction partielle que `Compte.v` a un champ `france`.

```
module Compte =
  struct
    type t = { france : int ; suisse : int }
    let v = { france = 1 ; suisse = 10000000 }
  end :
  sig
    type t : Le ( { france : int } )
    val v : t
  end in
 print_int ( (Compte.v).france )
```

Après la réduction du module `Compte`, `Compte.v` est un enregistrement protégé par des crochets. On a alors une interférence entre crochets et les relations constructeurs-destructeurs. La nouveauté est qu'il s'y mêle du sous-typage et des abstractions partielles.

Dans ce chapitre, nous allons partir de cet exemple et chercher une règle de réduction qui permette l'accès à au moins une partie de l'intérieur des crochets colorés lorsque l'abstraction n'est que partielle. Dans la première section, notre essai débouchera sur une nouvelle sémantique pour les crochets colorés. Dans la seconde, le déterminisme nous guidera vers le sous-typage explicite. Nous aborderons ensuite les conséquences de ces choix sur la sémantique opérationnelle. Nous terminerons ce chapitre avec un exemple de réduction utilisant l'exemple 4.0.1 ci-dessus.

4.1 Destructeur, sous-typage et couleurs

4.1.1 Énoncé d'une règle de réduction : premier essai

Revenons à l'exemple initial, et essayons d'écrire la règle de réduction correspondant à la situation d'un destructeur autour de crochets colorés dont le type est partiellement abstrait. Notons que cette situation ne peut pas survenir sans la présence de types partiellement abstraits qui seuls permettent de la typer correctement.

On veut donc que le destructeur entre dans les crochets pour réduire l'intérieur.

$$H, [v^{hm}]^{h.\text{TYPE}}_{hm}.l_i \longrightarrow_{hm_1} H, [v^{hm}.l_i]^{T}_{hm}$$

Cette règle est encore incomplète : il faut ajouter l'information que $h.\text{TYPE}$ est en quelque sorte un type enregistrement dans la couleur hm_1.

Remarquons cependant dans cette règle que le destructeur $.l_i$ est situé à l'extérieur des crochets dans le membre gauche et à l'intérieur des crochets dans le membre droit. Or si le typage du membre gauche impose que $[v^{hm}]^{h.\text{TYPE}}_{hm}$ soit connu comme un enregistrement dans la couleur hm_1 (comme supposé ci-dessus), ce n'est absolument pas clair que cela soit vrai de v^{hm} dans la couleur hm. C'est le cas si v^{hm} est elle-même un crochet coloré.

Example 4.1.1 Prenons deux hachages de modules que nous nommons respectivement h et h' :

$$h = \mathbf{hash}(N_U, \mathbf{nil}, [\{l{:}\text{INT}\}, \{l = 3\}]{:}[X{:}\mathbf{Le}(\top), X])$$
$$h' = \mathbf{hash}(N_V, \mathbf{nil}, [h.\text{TYPE}, [v^{\bullet}]^{h.\text{TYPE}}_{\{h\}}]{:}[Y{:}\mathbf{Le}(h.\text{TYPE}),\, Y])$$

Nous formons ensuite le terme suivant :

$$([[v^{\bullet}]^{h.\text{TYPE}}_{\{h\}}]^{h'.\text{TYPE}}_{\{h'\}}).l$$

Dans la couleur $\{h\}$, on peut prouver l'équivalence entre $h.\text{TYPE}$ et $\{l_1{:}\text{INT}\}$, et la relation $h'.\text{TYPE} <: h.\text{TYPE}$. En combinant les deux, on a la relation $h'.\text{TYPE} <: \{l_1{:}\text{INT}\}$. Cela a pour conséquence que le terme ci-dessus est bien typé dans la couleur $\{h\}$. Pourtant, lorsque nous appliquons la règle de réduction ci-dessus, le résultat n'est plus typable, car le jugement suivant n'est pas prouvable :

$$\vdash^{\mathbf{nil}}_{\{h'\}} [v^{\bullet}]^{h.\text{TYPE}}_{\{h\}}{:}\{l{:}\text{INT}\}$$

Cette situation est donc contre-exemple au théorème de préservation de type.

Comme cette règle est essentielle à notre langage, nous devons trouver une solution pour interdire ce contre-exemple.

Une des solutions possibles à cette situation est de rendre impossible une telle succession de crochets emboités en utilisant une règle de fusion des crochets comme celle-ci.

$$H, [[v^{hm_0}]^{h_0.\text{TYPE}}_{hm_0}]^{T}_{hm_1} \longrightarrow_{hm} H, [v^{hm_0}]^{T}_{hm_0 \cup hm_1}$$

Une autre solution est de modifier la sémantique des crochets colorés. En effet, ils sont utilisés jusqu'à présent comme frontière de connaissance, où une couleur *remplace* complètement une autre, c'est-à-dire que le passage au travers d'une couleur correspond à un gain, mais aussi à une perte de connaissance. Or seul le gain nous est utile pour distinguer les endroits où un type abstrait peut être révélé. On peut donc envisager de rendre les crochets additifs.

Nous verrons dans la section 4.3 que, de toutes les façons, la règle de fusion des crochets n'est pas suffisante dans certains cas. Donc la solution de l'additivité des crochets présente des avantages que la règle de fusion des crochets ne possède pas.

Remarquons enfin que dans les deux cas, il est nécessaire d'avoir la possibilité de former des couleurs contenant plusieurs hachages. Aucune règle ne formait, ni ne nécessitait de couleurs possédant plus d'un hachage.

4.1.2 Additivité des crochets colorés

Nous renonçons donc à la stricte séparation des crochets en espaces étanches : on les rend additifs, c'est-à-dire qu'ils vont simplement ajouter leur couleur à la couleur ambiante et non plus la remplacer complètement.

Ce choix de l'additivité va se retrouver dans les règles de typage et en particulier dans celle des crochets colorés. La couleur portée par le crochet est alors ajoutée à la couleur utilisée pour prouver le jugement.

$$\frac{E \vdash^{H}_{hm} T{:}\mathbf{Le}(\top) \quad E \vdash^{H}_{hm\cup hm'} e{:}T}{E \vdash^{H}_{hm} [e]^{T}_{hm'}{:}T}$$

On se souvient que les crochets tiennent un rôle dans la sérialisation puisqu'une valeur sérialisée doit pouvoir être décodée avec une couleur vide. Ce principe est rappelé par la règle de typage suivante et la règle de réduction de la désérialisation (toutes deux inchangées).

$$\frac{E \vdash^{H_0}_{hm} \text{ok} \quad \mathbf{nil} \vdash^{H_1}_{\bullet} e{:}T}{E \vdash^{H_0}_{hm} \mathbf{marshalled}_{H_1}(e{:}T){:}\text{STRING}}$$

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{H'}(v^{\bullet}{:}T){:}T') \longrightarrow_{hm} \begin{cases} H \cup H', v^{\bullet} & \text{if } \mathbf{nil} \vdash^{H\cup H'}_{hm} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

Une expression sérialisée doit être valide dans n'importe quel environnement et dans n'importe quelle couleur. C'est pourquoi, nous lui imposons la correction dans la couleur vide, et que la règle de réduction de **mar** protège la valeur sérialisée avec des crochets colorés. Cependant, ceux-ci ne sont pas toujours nécessaires et c'est l'éventuelle réduction au sein de **marshalled** qui le juge. Or, cette réduction doit se faire (si elle est possible) dans la couleur vide. Dans le contexte où toutes les frontières deviennent additives, celle-ce reste donc imperméable. Il faut donc être extrèmement attentif lors de la définition de la réduction au sein d'un contexte, et de la définition même des contextes. Voici la règle de réduction au sein des contextes. Elle prévoit donc un remplacement de la couleur hm' par hm lorsque l'on traverse un contexte $C^{hm'}_{hm}$.

$$\frac{H, e \longrightarrow_{hm} H', e'}{H, C^{hm'}_{hm}.e \longrightarrow_{hm'} H', C^{hm'}_{hm}.e'}$$

C'est donc la définition des contextes $C^{hm'}_{hm}$ qui est importante ici. Nous devons donc bien indiquer pour **marshalled** que $hm = \bullet$, et que pour les crochets l'additivité est de mise.

Contextes :

$C^{hm'}_{hm}$ ::=	$(v_1^{hm'}, .., v_{i-1}^{hm'}, \_, e_{i+1}, .., e_j)$	produit ($2 \leqslant j$ et $1 \leqslant i \leqslant j$), si $hm' = hm$
|	$\mathbf{proj}_i \_$	projection, si $hm' = hm$
|	$\{l_1 = v_1^{hm'}, ..., l_{i-1} = v_{i-1}^{hm'}, l_i = \_,$	enregistrement ($1 \leqslant j$
|	$\quad l_{i+1} = e_{i+1}, ..., l_j = e_j\}$	et $1 \leqslant i \leqslant j$), si $hm' = hm$
|	$\_.l_i$	accès à un champ, si $hm' = hm$
|	$\_\, e$	application gauche, si $hm' = hm$
|	$v^{hm'}\ \_$	application droite, si $hm' = hm$
|	$\mathbf{mar}\,(\_ : T)$	sérialisation, si $hm' = hm$
|	$\mathbf{marshalled}_H(\_ : T)$	sérialisation achevée, si $hm = \bullet$
|	$\mathbf{unmar}\ \_ : T$	désérialisation, si $hm' = hm$
|	$!\,\_$	envoi, si $hm' = hm$
|	$[\_]^T_{hm_1}$	crochets colorés, si $hm_1 \cup hm' = hm$

Le contexte des crochets montre bien l'additivité, alors que celui de **marshalled** marque une frontière.

De plus, comme les frontières ne sont pas toutes abolies (il reste **marshalled**), les crochets colorés restent nécessaires dans la β-réduction. En pratique, il ne le sont pas réellement car il n'y a pas de variables libres dans une expression $\mathbf{marshalled}_H(e)$ bien typée : on peut donc prouver que ces crochets sont inutiles et seront toujours déconstruits.

$$(\lambda x{:}T.e)\ v^{hm} \longrightarrow_{hm} \{x \leftarrow [v^{hm}]^T_{hm}\} e$$

Enfin, remarquons que l'additivité des crochets colorés induit un changement pour les valeurs au sein des crochets, puisque la définition des contextes a changé. Cela donne les règles mises à jour suivantes (la notation des valeurs $\widehat{v}^{hm}$ sera expliquée dans la section 4.3).

$$H, [(\widehat{v}_1^{hm'\cup hm}, ..., \widehat{v}_j^{hm'\cup hm})]^{T_1 * ... * T_j}_{hm'} \longrightarrow_{hm} H, ([\widehat{v}_1^{hm'\cup hm}]^{T_1}_{hm'}, ..., [\widehat{v}_j^{hm'\cup hm}]^{T_j}_{hm'})$$

$$H, [\{l_1 = \widehat{v}_1^{hm'\cup hm}, .., l_j = \widehat{v}_j^{hm'\cup hm}\}]^{\{l_1:T_1,...,l_j:T_j\}}_{hm'} \longrightarrow_{hm} H, \{l_1 = [\widehat{v}_1^{hm'\cup hm}]^{T_1}_{hm'}, .., l_j = [\widehat{v}_j^{hm'\cup hm}]^{T_j}_{hm'}\}$$

$$H, [\widehat{v}^{hm'\cup hm}]^{h.\mathrm{TYPE}}_{hm'} \longrightarrow_{hm} H, [\widehat{v}^{hm'\cup hm}]^{T}_{hm'} \qquad \text{when } h \in hm \text{ and } \mathrm{impl}(h){=}T$$

Dans ces règles, les valeurs pour hm' deviennent des valeurs pour $hm{\cup}hm'$ du fait de l'additivité.

4.2 Déterminisme et couleurs

4.2.1 Énoncé d'une règle de réduction : deuxième essai

Nous avons progressé dans l'étude de l'exemple initial en retenant la solution des crochets additifs et, pourtant, nous n'avons toujours pas résolu le problème complètement. Rappelons la proposition initiale de règle.

$$H, [v^{hm}]^{h.\mathrm{TYPE}}_{hm}.l_i \longrightarrow_{hm_1} H, [v^{hm}.l_i]^T_{hm}$$

Même en supposant que l'on ait l'information que $h.\mathrm{TYPE}$ est un enregistrement, il manque une information capitale : d'où vient le type T que l'on souhaite écrire sur le crochet coloré du membre droit ?

Le problème est encore plus criant lors la réduction d'un n-uplet.

$$H, \mathbf{proj}_i\, [v^{hm}]_{hm}^{h.\text{TYPE}} \longrightarrow_{hm_1} H, [\mathbf{proj}_i\, v^{hm}]_{hm}^{T}$$

Si l'on obtient une décomposition de $h.\text{TYPE}$ en n-uplet, c'est par le biais du sous-typage. Or, ce sous-typage, qui est en profondeur dans le cas des n-uplets et qui peut être contravariant pour un type fonctionnel, peut donner une décomposition de $h.\text{TYPE}$ en types complètement arbitraires. D'une part, le déterminisme des règles de réduction n'est plus du tout valable et, d'autre part, et c'est bien plus grave, la réduction vers des crochets d'un mauvais type peut faire échouer la préservation de type ou la normalisation du terme vers une valeur.

Cette situation nous oblige, pour préserver les théorèmes de préservation, de progrès et de déterminisme des réductions qui sont essentiels à la confiance que l'on peut porter à notre système, à adopter un sous-typage explicite.

4.2.2 Sous-typage explicite

Nous ajoutons donc une construction syntaxique pour le sous-typage explicite.

Expressions :

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

Dans nos exemples, nous noterons ${}_{(T_1 <: T_2)}e$ de la manière suivante :

```
(e : T1 <: T2)
```

Nous avons donc maintenant une règle de typage associée qui remplace le sous-typage implicite.

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

Aucun autre changement n'est nécessaire dans le système de typage, puisqu'il ne s'agit que de rendre syntaxiquement un phénomène qui était déjà présent.

La majeure partie des règles de réduction énoncées dans les chapitres précédents ne sont pas affectées par le sous-typage explicite. Cependant la sérialisation et la désérialisation, parce qu'elles comportaient des annotations de types, étaient déjà similaires à une déclaration de sous-typage explicite. Nous devons donc modifier légèrement la règle pour introduire le constructeur explicite là où il était implicite.

$$H, \mathbf{unmar}\,(\mathbf{marshalled}\,{}_{H'}(v^{\bullet} : T) : T') \longrightarrow_{hm} \begin{cases} H \cup H', {}_{(T <: T')}v^{\bullet} & \text{if } \mathbf{nil} \vdash_{hm}^{H \cup H'} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

Il reste maintenant à faire interagir ce sous-typage explicite avec le reste du système, et notamment les crochets colorés.

4.3 Sous-typage explicite et couleurs

Nous avons vu que les règles de réduction des crochets colorés avaient pour but de les repousser vers les feuilles de l'arbre syntaxique pour permettre aux constructeurs et destructeurs se réduire. La même contrainte va peser sur le sous-typage explicite. On aura donc les règles suivantes.

$$H,\ {}_{(\text{UNIT}<:\text{UNIT})}() \longrightarrow_{hm} H, ()$$

$$H,\ {}_{(\text{STRING}<:\text{STRING})}(\mathbf{marshalled}_{H'}(v^\bullet{:}T)) \longrightarrow_{hm} H, \mathbf{marshalled}_{H'}(v^\bullet{:}T)$$

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

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

Comme nous n'avons pas de sous-typage en profondeur pour les enregistrements, nous savons que l'on peut prouver les équivalences entre T_k et T'_k pour $0 \leqslant k \leqslant i$. Aucune annotation de sous-typage n'est donc nécessaire dans le membre droit. Cette règle suppose cependant que $i \leqslant j$.

$$H,\ {}_{(\{l_1:T'_1;\ldots;l_j:T'_j\}<:\{l_{\pi(1)}:T_{\pi(1)};\ldots;l_{\pi(i)}:T_{\pi(i)}\})}\{l_1 = v_1^{hm};\ldots;l_j = v_j^{hm}\} \longrightarrow_{hm} H, \{l_{\pi(1)} = v_{\pi(1)}^{hm};\ldots;l_{\pi(i)} = v_{\pi(i)}^{hm}\}$$

On remarque que ces règles nécessitent des types explicites sur les annotations de sous-typage, il est donc important de pouvoir révéler les types présents sur ces annotations.

$$H,\ {}_{(T_0<:h_0.\text{TYPE})}\widehat{v}^{hm} \longrightarrow_{hm} H,\ {}_{(T_0<:T_1)}\widehat{v}^{hm} \quad \text{when } h_0 \in hm \text{ and } \text{impl}(h_0){=}T_1$$

$$H,\ {}_{(h_0.\text{TYPE}<:TV^{hm})}\widehat{v}^{hm} \longrightarrow_{hm} H,\ {}_{(T_0<:TV^{hm})}\widehat{v}^{hm} \quad \text{when } h_0 \in hm \text{ and } \text{impl}(h_0){=}T_0$$

On introduit donc ici les types valeurs TV^{hm} qui possèdent des types en forme normale de tête pour la révélation par la couleur hm. C'est, encore une fois, uniquement utile au déterminisme de notre système. Leur définition exacte est la suivante.

Types valeurs :

TV^{hm} ::=	UNIT	unit
	$T_1 * \ldots * T_j$	produit ($2 \leqslant j$)
	$\{l_1{:}T_1;\ldots;l_j{:}T_j\}$	enregistrement ($1 \leqslant j$)
	$T{\to}T$	fonction
	STRING	chaîne de caractères
	h.TYPE	type haché si $h \notin hm$
	$\top$	Top

On a aussi introduit dans les deux règles de réduction précédentes les valeurs dites « de base ». Nous sommes obligés, pour accueillir le sous-typage explicite dans la grammaire des valeurs, de tenir compte de plusieurs contraintes. En particulier, la succession de crochets colorés et/ou d'annotations de sous-typage explicite ne sont plus permises comme vont le montrer les règles définies à la fin de cette section.

Valeurs :

$$
\begin{array}{rcll}
v^{hm_0} & ::= & \widehat{v}^{hm_0} & \text{valeurs de base} \\
& | & [\widehat{v}^{hm_1}]_{hm_1}^{h_1.\mathrm{TYPE}} & \text{crochets colorés où } h_1 \notin hm_0 \\
& | & {}_{(TV_1^{hm_0} <: TV_2^{hm_0})}\widehat{v}^{hm_0} & \text{sous-typage explicite où } TV_1^{hm_0} = h_0.\mathrm{TYPE} \\
& & & \quad \text{ou } TV_2^{hm_0} = h_2.\mathrm{TYPE} \\
\widehat{v}^{hm_0} & ::= & () & \text{unit} \\
& | & (v_1^{hm_0}, ..., v_j^{hm_0}) & \text{produit } (2 \leqslant j) \\
& | & \{l_1 = v_1^{hm_0}, ..., l_j = v_j^{hm_0}\} & \text{enregistrement } (1 \leqslant j) \\
& | & \lambda x{:}T.e & \text{abstraction } (x \text{ lié dans } e) \\
& | & \mathbf{marshalled}_{\,H}(v^{\bullet}{:}T) & \text{valeurs sérialisées closes (avec } H \text{ clos)}
\end{array}
$$

Maintenant que le sous-typage explicite permet les interactions entre constructeurs et destructeurs, il reste à étudier les interactions entre les crochets colorés et les annotations de sous-typage explicite.

Le premier choix qui s'offre à nous est de déterminer si le sous-typage explicite doit progresser vers ces mêmes feuilles plus vite que les crochets colorés.

Essayons de donner la primauté aux crochets colorés. Un essai donnerait par exemple ceci.

$$H, [\,{}_{(T' <: T'')}\widehat{v}^{hm' \cup hm}]_{hm'}^{T''} \longrightarrow_{hm} H, \,{}_{(T' <: T'')}([\widehat{v}^{hm' \cup hm}]_{hm'}^{T'})$$

Cette règle ne vérifie pas du tout la préservation de type, puisque la vérification du jugement de sous-typage a pu, dans le membre gauche, utiliser la couleur hm' qui n'est plus disponible dans le membre droit.

Il est ainsi clair que les annotations de sous-typage doivent aller vers les feuilles plus vite que les crochets colorés. Cela donne la règle suivante.

$$H, \,{}_{(T' <: T'')}([\widehat{v}^{hm' \cup hm}]_{hm'}^{h.\mathrm{TYPE}}) \longrightarrow_{hm} H, [\,{}_{(T' <: T'')}\widehat{v}^{hm' \cup hm}]_{hm'}^{T''} \quad \text{where } h \notin hm$$

Il reste enfin l'interaction du sous-typage avec lui-même.

$$H, \,{}_{(TV_0^{hm} <: TV_1^{hm})}v^{hm} \longrightarrow_{hm} H, \,{}_{(TV_2^{hm} <: TV_1^{hm})}\widehat{v}^{hm} \quad \text{where } v^{hm} = {}_{(TV_2^{hm} <: TV_3^{hm})}\widehat{v}^{hm}$$

Celle-ci est correcte car il est possible de prouver la transitivité du sous-typage.

4.4 Retour sur l'exemple initial

Revenons un instant sur l'exemple donné en exergue de ce chapitre.

```
module Compte =
  struct
    type t = { france : int ; suisse : int }
    let v = { france = 1 ; suisse = 10000000 }
  end :
  sig
    type t : Le ( {france : int} )
    val v : t
  end in
 print_int ( (Compte.v).france )
```

Réécrivons-le à l'aide de notre syntaxe définitive.

```
module Compte =
  struct
    type t = { france : int ; suisse : int }
    let v = { france = 1 ; suisse = 10000000 }
  end :
  sig
    type t : Le ( {france : int} )
    val v : t
  end in
 print_int ((Compte.v) : Compte.t <: {france : int}) .france
```

La compilation va donner l'expression suivante (en abandonnant le `print_int`, et en notant h le hachage du module `Compte` et respectivement l_1 et l_2 les champs `france` et `suisse`).

$$(\,_{(h.\text{TYPE}<:\{l_1:\text{INT}\})}[\{l_1 = 1; l_2 = 10000000\}]^{h.\text{TYPE}}_{\{h\}}).l_1$$

Une seule règle s'applique, il s'agit de la règle échangeant crochets et sous-typage explicite. Le résultat est le suivant.

$$([\,_{(h.\text{TYPE}<:\{l_1:\text{INT}\})}\{l_1 = 1; l_2 = 10000000\}]^{\{l_1:\text{INT}\}}_{\{h\}}).l_1$$

Par le jeu des valeurs, une seule règle est encore possible et il s'agit de la règle permettant de révéler les types sur les annotations de sous-typage. Nous pouvons l'appliquer puisque l'intérieur des crochets possède bien une couleur contenant h.

$$([\,_{(\{l_1:\text{INT};l_2:\text{INT}\}<:\{l_1:\text{INT}\})}\{l_1 = 1; l_2 = 10000000\}]^{\{l_1:\text{INT}\}}_{\{h\}}).l_1$$

On doit donc ensuite appliquer la règle de réduction du sous-typage sur les enregistrements.

$$([\{l_1 = 1\}]^{\{l_1:\text{INT}\}}_{\{h\}}).l_1$$

On applique alors la réduction des crochets sur les enregistrements.

$$\{l_1 = [1]^{\text{INT}}_{\{h\}}\}.l_1$$

Puis la règle de réduction des crochets sur les types de base.

$$\{l_1 = 1\}.l_1$$

Nous concluons par l'appel de la valeur d'un champ dans un enregistrement.

$$1$$

Chapitre 5

Théorèmes

La confiance que nous pouvons avoir en notre langage et, notamment, en sa capacité à préserver les abstractions, vient d'un certain nombre de théorèmes qu'il est possible de prouver sur notre système de types et notre sémantique.

Nous avons rassemblé dans l'annexe B les preuves, malheureusement encore incomplètes, des théorèmes cités ci-dessous.

Nous résumons ici, avec quelques commentaires, les principaux résultats théoriques obtenus.

5.1 Système de type

On a ici juste quelques résultats concernant des propriétés attendues du système de typage.

Lemma 5.1.1 (cf. Lemma B.4.1 (reflexivity of kind equivalence))
Si $E \vdash^H_{hm} K$ ok alors $E \vdash^H_{hm} K == K$.

Lemma 5.1.2 (cf. Lemma B.4.2 (transitivity of kind equivalence))
Si $E \vdash^H_{hm} K == K'$ et $E \vdash^H_{hm} K' == K''$ alors $E \vdash^H_{hm} K == K''$.

Lemma 5.1.3 (cf. Lemma B.4.6 (signature equivalence is transitive))
Si $E \vdash^H_{hm} S == S'$ et $E \vdash^H_{hm} S' == S''$ alors $E \vdash^H_{hm} S == S''$.

Lemma 5.1.4 (cf. Lemma B.4.5 (transitivity of subtyping))
Si $E \vdash^H_{hm} T <: T'$ et $E \vdash^H_{hm} T' <: T''$ alors $E \vdash^H_{hm} T <: T''$.

5.2 Correction

Il est nécessaire de pouvoir garantir que toutes les parties des jugements sont correctes par des sous-preuves.

Lemma 5.2.1 (cf. Lemma B.1.6 (hashes have to be ok))
Si $E \vdash^H_{hm} J$ ou $\vdash hm$ ok est dérivable par une preuve Π et h est un sous-terme de $E \vdash^H_{hm} J$ ou $\vdash hm$ ok alors $\vdash h$ ok par une sous-preuve de Π.

Lemma 5.2.2 (cf. Lemma B.1.7 (environments and subhashes have to be ok))
Si $E \vdash^H_{hm} J$ alors $E \vdash^H_{hm}$ ok par une sous-preuve.

Lemma 5.2.3 (cf. Lemma B.5.7 (things have to be ok))

Si $E \vdash^H_{hm} T{:}K$ alors $E \vdash^H_{hm} K$ ok.
Si $E \vdash^H_{hm} T == T'$ ou $E \vdash^H_{hm} T <: T'$ alors $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ et $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$.
Si $E \vdash^H_{hm} K == K'$ ou $E \vdash^H_{hm} K <: K'$ alors $E \vdash^H_{hm} K$ ok et $E \vdash^H_{hm} K'$ ok.
Si $E \vdash^H_{hm} S == S'$ ou $E \vdash^H_{hm} S <: S'$ alors $E \vdash^H_{hm} S$ ok et $E \vdash^H_{hm} S'$ ok.
Si $E \vdash^H_{hm} e{:}T$ ou $E \vdash^H_{hm} m{:}T$ alors $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$.
Si $E \vdash^H_{hm} M{:}S$ ou $E \vdash^H_{hm} U{:}S$ alors $E \vdash^H_{hm} S$ ok.

5.3 Préservation

Pour démontrer la préservation du typage par les différentes réductions de notre sémantique, certains résultats, notamment sur les substitutions, sont requis.

Lemma 5.3.1 (cf. Lemma B.5.4 (type preservation by substitution))
Si $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$ par une preuve Π telle que $hm' \preccurlyeq \min\left(\mathrm{pvu}_\zeta\left(\Pi\right)\right)$ et $E_0 \vdash^{H_0}_{hm'} \eta{:}\tau$ avec $H_0 \subseteq H$ alors $E_0, \sigma E \vdash^H_{hm} \sigma J$ où $\sigma = \{\zeta{\leftarrow}\eta\}$ et $\zeta{:}\tau$ est une variable d'expression ou de type.

Lemma 5.3.2 (cf. Lemma B.5.12 (type preservation by module substitution in coloured judgements))
Supposons que $E_0, U{:}[X{:}K, T], E \vdash^H_{hm} J$, que z et Z sont des variables fraiches, que h est le résultat du hachage d'un module dont la signature est $[X{:}K, T]$, et enfin que $\sigma = \{U{\leftarrow}h, U.\textsc{type}{\leftarrow}Z, U.\mathrm{term}{\leftarrow}z\}$.
Si $J = U{:}S'$ pour un certain S' alors $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma S'$.
Sinon $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{hm} \sigma J$.

Theorem 5.3.3 (cf. Theorem B.7.1 (type preservation for expression reduction))

Si $\mathbf{nil} \vdash^H_{hm} e{:}T$ et $H, e \longrightarrow_{hm} H', e'$ alors $\mathbf{nil} \vdash^{H'}_{hm} e'{:}T$.

Corollary 5.3.4 (cf. Corollary B.7.3 (type preservation for network reduction))
Si $\vdash n$ ok et $n \longrightarrow n'$ alors $\vdash n'$ ok.

Theorem 5.3.5 (cf. Theorem B.7.4 (type preservation for machine reduction))
Si $\mathbf{nil} \vdash^H_\bullet m{:}T$ et $H, m \longrightarrow_{\mathsf{c}} H', m'$ alors $\mathbf{nil} \vdash^{H'}_\bullet m'{:}T$.

5.4 Progrès

La propriété de progrès garantit le bon comportement des termes bien typés.

Theorem 5.4.1 (cf. Theorem B.8.6 (progress of expressions))
Si $\mathbf{nil} \vdash^H_{hm} e{:}T$ alors de deux choses l'une :
- e est légalement bloquée dans la couleur hm.
- e peut se réduire, i.e. il existe e' et H' tels que $H, e \longrightarrow_{hm} H', e'$.

Corollary 5.4.2 (cf. Corollary B.8.7 (progress of networks))
Si $\vdash n$ ok alors nous sommes dans un des cas suivants :
- n est stoppé, i.e. il existe $n_{()}$ et n_{fail} tels que $n \equiv n_{()} \mid n_{\mathrm{fail}}$.
- n est en attente de lecture, i.e. il existe $n_{()}$, n_{fail} et $n_?$ tels que $n \equiv n_{()} \mid n_{\mathrm{fail}} \mid n_?$
- n est en attente d'écriture, i.e. il existe $n_{()}$, n_{fail} et $n_!$ tels que $n \equiv n_{()} \mid n_{\mathrm{fail}} \mid n_!$

– n peut se réduire, i.e. il existe n' tel que $n \longrightarrow n'$

où

$n_{()}$ ::=	$\mathbf{0}$	réseau vide
	$n_{()} \mid n_{()}$	mise en lien de deux réseaux
	$()$	unit
n_{fail} ::=	$\mathbf{0}$	réseau vide
	$n_{\text{fail}} \mid n_{\text{fail}}$	mise en lien de deux réseaux
	$CC^{\bullet}_{hm}.\mathbf{Unmarfailure}^{T}$	blocage
$n_{?}$ ::=	$n_{?} \mid n_{?}$	mise en lien de deux réseaux
	$CC^{\bullet}_{hm}.?$	attente de lecture
$n_{!}$ ::=	$n_{!} \mid n_{!}$	mise en lien de deux réseaux
	$CC^{\bullet}_{hm}.!\, v$	attente d'écriture

Theorem 5.4.3 (cf. Theorem B.8.8 (progress of machines))
Si $\mathbf{nil} \vdash^{H}_{\bullet} m{:}T$ alors de deux choses l'une : m est une expression ou H, m se réduit par $\longrightarrow_{\mathsf{c}}$.

5.5 Déterminisme

Le déterminisme est une garantie supplémentaire sur la sémantique du langage : un même programme se comportera toujours de la même façon.

Theorem 5.5.1 (cf. Theorem B.9.1 (determinism of machine reduction))
La réduction des machines est déterministe, i.e. si $H, m \longrightarrow_{\mathsf{c}} H_1, m_1$ et $H, m \longrightarrow_{\mathsf{c}} H_2, m_2$ alors $m_1 = m_2$, $H_1 = H_2$ et les deux réductions utilisent la même règle sur le même redex.

Theorem 5.5.2 (cf. Theorem B.9.3 (determinism of expression reduction))
La réduction des expressions et des machines est déterministe, i.e. si $H, e \longrightarrow_{hm} H', e'$ et $H, e \longrightarrow_{hm} H'', e''$ alors $e' = e''$ et $H' = H''$ et les deux réductions utilisent la même règle sur le même redex.

Chapitre 6

Conclusion et perspectives

Nous avons, tout au long de ce rapport, essayé de concilier les exigences de la préservation de l'abstraction avec les contraintes du hachage et des crochets colorés, du sous-typage et des abstractions partielles. Il est temps maintenant de jeter un regard sur ce qui vient d'être accompli, et de poser les jalons des travaux à venir.

6.1 Conclusion

La propriété de la préservation de l'abstraction n'est pas une caractéristique simple à étudier dans un langage, comme nous l'avons vu dans les précédents chapitres. Elle n'est pas non plus un résultat strict, mais plutôt un degré de confiance que l'on peut avoir dans le langage et ses sémantiques statiques et dynamiques, degré matérialisé par les théorèmes classiques de préservation, progrès et déterminisme dans notre cas.

Pour arriver à cela, il nous a fallu exposer longuement un langage modèle dans le chapitre 2, en détaillant sa syntaxe et ses sémantiques statiques et dynamiques, de manière partielle tout au moins. Ses caractéristiques comprenaient tous nos objectifs (hachage, crochets colorés, sous-typage, abstractions totales et partielles) sans toutefois en résoudre les difficultés.

Nous avons donc essayé, au chapitre 3, d'établir quelles conséquences le sous-typage avait sur les types abstraits et partiellement abstraits. Le langage s'est alors enrichi de nouvelles constructions permettant à l'utilisateur de spécifier directement l'éventuelle compatibilité entre deux types abstraits de deux modules distincts.

Au chapitre 4, nous avons enfin porté notre attention sur la gestion des crochets colorés. Il y avait en effet une interaction avec le sous-typage, et plus particulièrement avec les abstractions partielles. Il est donc devenu nécessaire de changer la sémantique des crochets colorés pour les rendre additifs et, dans un second temps, de supposer le sous-typage explicite pour sauvegarder la correction de notre système.

Enfin, dans le chapitre 5, nous avons examiné les principaux résultats théoriques qui ont été pour leur grande part prouvés sur notre langage. On retiendra notamment le théorème de préservation du type, le théorème de progrès, ainsi que le déterminisme de notre système de réduction.

6.2 Contributions par rapport aux solutions actuelles

Parmi les travaux actuels, le plus proche est le papier de Leifer et al. [LPSW03] qui présente l'utilisation du hachage pour la préservation des abstractions. Nous nous sommes servi de ce papier pour construire notre langage de base, présenté au chapitre 2.

Nous y avons cependant ajouté du sous-typage implicite et des déclarations de types partiellement abstraits qu'ils ne possédaient pas. Nous avons de plus enrichi ce langage de constructeurs que le programmeur peut utiliser pour exprimer la compatibilité qu'il désire entre deux types abstraits (chapitre 3). Nous avons alors choisi des règles de typage et de réduction convenables pour obtenir (au moins partiellement) les théorèmes de préservation du typage, de progrès et de déterminisme. Nous avons donc maintenant une grande confiance en la préservation des abstractions, tant au niveau des types abstraits que du maintien des invariants associés, dans notre langage. Les ajouts du sous-typage et des abstractions partielles permettent ainsi, par un polymorphisme borné, d'augmenter considérablement l'expressivité que l'on peut trouver dans un langage qui garantir la préservation des abstractions.

6.3 Travaux en cours d'achèvement

6.3.1 Sous-typage en profondeur

Nous avons fait le choix, dans le chapitre 1, de n'utiliser qu'un sous-typage en largeur dans un souci de simplification. Il semble, cependant, qu'à aucun moment le sous-typage en profondeur ne menace l'intégrité du système. Cet ajout parait donc assez immédiat.

6.3.2 Modules généraux

Les modules que nous avons considérés n'autorisaient la déclaration que d'un unique type et d'une unique valeur. Il serait donc intéressant de généraliser nos définitions de modules pour qu'elles acceptent plusieurs définitions de types et de valeurs. Un simple codage ne suffit pas à cause d'éventuelles dépendances entre types et valeurs. Acute [SLW+05], qui utilise le mécanisme du hachage et des crochets colorés, résoud cependant cette difficulté : il devrait donc être possible de la surmonter dans notre cas. La présence de foncteurs peut aussi être envisagée.

6.3.3 Preuves

Il reste encore quelques points techniques à prouver avant d'obtenir la preuve complète des théorèmes de préservation du typage, de progrès et déterminisme.

Certains lemmes portant sur la préservation du typage par certaines substitutions complexes manquent et les résultats, pour l'instant incomplets, portant sur la décomposition des types limitent la preuve de la préservation du typage par la réduction sur les expressions.

Un autre résultat liant les valeurs aux types bloque le théorème de préservation du typage par la réduction des réseaux et les théorèmes de progrès.

Enfin, quelques points techniques mineurs font obstacle à la preuve du déterminisme de la réduction des expressions.

Ces preuves restent donc à terminer pour justifier notre confiance dans le système.

6.3.4 Algorithmique

Nous ne nous sommes pas réellement intéressés à la partie algorithmique du système de typage de notre langage. Le sous-typage explicite et l'absence de polymorphisme militent pour la faisabilité de la vérification de type. L'inférence de type est-elle possible ? C'est probable car les annotations sont nombreuses, mais il y a là matière à réflexion.

6.3.5 Implémentation

Une implémentation prototype est actuellement en cours de réalisation. Elle tente de suivre au plus près la sémantique du langage tout en essayant de résoudre les problèmes concrets sous-jacents, comme la question du hachage et de la transmission des déclarations de compatibilité. La partie typage n'a pas encore été écrite.

Une implémentation pour un langage réellement utilisable nécessiterait au préalable quelques résultats théoriques. En particulier, il serait souhaitable de pouvoir inférer les annotations de sous-typage explicite pour que le programmeur n'ait à écrire qu'un minimum d'indications dans le code source. Ensuite, il faudrait s'intéresser à l'utilité des crochets et annotations de sous-typage lors de l'exécution. Il serait ainsi préférable de pouvoir les effacer

6.4 Perspectives

De nombreux problèmes nous attendent lorsqu'on s'attaque à la préservation de l'abstraction. Cependant, une fois étudiées les interactions avec le sous-typage, il est tentant de se projeter vers la question du polymorphisme plus général puisque le sous-typage étudié ici permet un polymorphisme borné. Le polymorphisme à la ML ou du système F (avec une inférence partielle) s'intègre-t-il facilement à notre langage ?

Comme les abstractions partielles sont présentes dans certains langages objets, et que le sous-typage entre enregistrements peut fournir une base de modélisation des objets, il semble intéressant d'évoquer la question d'une éventuelle présence d'objets dans le langage. La question notamment des types déclarés par les objets peut poser des problèmes.

Les types XML, notamment ceux présents dans CDuce [BCF03], constituent un défi important à cause de leur richesse. Ils s'associeraient pourtant bien à un langage destiné à l'utilisation en environnement distribué et autoriseraient un certain nombre d'applications intéressantes.

Enfin, l'intégration de certains de ces éléments (sous-typage simple et abstractions partielles comprises) dans le langage Acute peut être un objectif raisonnable, malgré la complexité d'Acute. Cela augmenterait ainsi l'expressivité du langage et rendrait plus aisée l'écriture de certains programmes.

6.5 Remerciements

Je remercie chaleureusement James Leifer pour sa présence et son amitié tout au long de ce stage. Je suis très reconnaissant aussi envers Jean-Jacques Lévy pour sa disponibilité et ses nombreux conseils qui, je n'en doute pas, me seront extrêmement utiles par la suite. Je ne peux oublier Francesco Zappa Nardelli et Gilles Peskine qui, par les nombreuses discussions communes et leurs remarques avisées, m'ont beaucoup aidé à faire ce travail. Il me reste enfin à saluer l'équipe Moscova et tous les gens du bâtiment 8 qui m'ont accueilli en me donnant des conditions de travail excellentes, et dont la bonne humeur va de pair avec l'extrême compétence. Je remercie en

particulier Thomasz Blanc qui a partagé son bureau avec moi, et les stagiaires du monde entier qui sont venus nous rendre visite.

Bibliographie

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce : an XML-centric general-purpose language. 38(9) :51–63, September 2003.

[CDJ+89] Luca Cardelli, Jim E. Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4) :471–522, 1985.

[GLPS05] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic compiler and runtime system, 2005.

[GMZ00] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6) :1037–1080, 2000.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st POPL*, 1994.

[Jav] The Java Language. `http://java.sun.com/`.

[JoC] JoCaml. `http://pauillac.inria.fr/jocaml/`.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st POPL*, pages 109–122, 1994.

[LPSW03] 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`.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[OCa] Objective Caml. `http://caml.inria.fr`.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. pages 77–88, Austin, Texas, January 1989.

[SH00] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Proc. 27th POPL*, pages 214–227, 2000.

[SLW+05] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute : High-level programming language design for distributed computation. In *Proceedings of ICFP 2005 : International Conference on Functional Programming (Tallinn)*, September 2005. To appear.

Annexe A

Syntaxe, Typage, Sémantique

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
$\lambda x{:}T.e$	abstraction (x binds in e)
$e\ e$	application
$\mathbf{mar}\,(e{:}T)$	dynamic
$\mathbf{marshalled}_{\,H}(e{:}T)$	closed, colour-independent dynamic
$\mathbf{unmar}\ e{:}T$	undynamic
$!\,e$	send
$?$	receive
U.term	term-part of a module
${}_{(T_1 <: T_2)}\,e$	explicit subtyping
$[e]^T_{hm}$	type colouring
$\mathbf{Unmarfailure}^T$	undyn failure

$v^{hm_0} ::=$	hm_0-value, i.e. value in the colour hm_0
$\widehat{v}^{hm_0}$	base values
${}_{(TV_1^{hm_0} <: TV_2^{hm_0})}\widehat{v}^{hm_0}$	explicit subtyping where $TV_1^{hm_0} = h_0.\text{TYPE}$ or $TV_2^{hm_0} = h_2.\text{TYPE}$
$[\widehat{v}^{hm_1 \cup hm_0}]^{h_1.\text{TYPE}}_{hm_1}$	type colouring where $h_1 \notin hm_0$

$\widehat{v}^{hm_0} ::=$	base values of the colour hm_0
$()$	unit
$(v_1^{hm_0}, ..., v_j^{hm_0})$	tuple $(2 \leqslant j)$
$\{l_1 = v_1^{hm_0}, ..., l_j = v_j^{hm_0}\}$	record $(1 \leqslant j)$
$\lambda x{:}T.e$	abstraction (x binds in e)
$\mathbf{marshalled}_H(v^\bullet{:}T)$	closed dynamic value (H is closed)
$C_{hm_2}^{hm_1} ::=$	single-level evaluation context
$(v_1^{hm_1}, .., v_{i-1}^{hm_1}, \_, e_{i+1}, .., e_j)$	tuple $(2 \leqslant j$ and $1 \leqslant i \leqslant j)$, if $hm_1 = hm_2$
$\mathbf{proj}_i\ \_$	projection, if $hm_1 = hm_2$
$\{l_1 = v_1^{hm_1}, ..., l_{i-1} = v_{i-1}^{hm_1}, l_i = \_,$ $l_{i+1} = e_{i+1}, ..., l_j = e_j\}$	tuple $(1 \leqslant j$ and $1 \leqslant i \leqslant j)$, if $hm_1 = hm_2$
$\_.l_i$	projection, if $hm_1 = hm_2$
$\_\ e$	application left, if $hm_1 = hm_2$
$v^{hm_1}\ \_$	application right, if $hm_1 = hm_2$
$(T_1 <: T_2)\ \_$	explicit subtyping
$\mathbf{mar}\ (\_{:}T)$	dynamic, if $hm_1 = hm_2$
$\mathbf{marshalled}_H(\_{:}T)$	colour-independent dynamic, if $hm_2 = \bullet$
$\mathbf{unmar}\ \_{:}T$	undynamic, if $hm_1 = hm_2$
$!\ \_$	send, if $hm_1 = hm_2$
$[\_]_{hm}^T$	coloured bracket where $hm \cup hm_1 = hm_2$
$CC_{hm_2}^{hm_1} ::=$	coloured evaluation context
$CC_{hm'}^{hm_1}.C_{hm_2}^{hm'}$	extra level
$\_$	identity, if $hm_1 = hm_2$
$T ::=$	type
UNIT	unit
$T_1 * ... * T_j$	tuple $(2 \leqslant j)$
$\{l_1{:}T_1; ...; l_j{:}T_j\}$	record $(1 \leqslant j)$
X	variable
$T \to T$	function
STRING	dynamic
U.TYPE	type-part of a module
h.TYPE	hash
$\top$	Top

TV^{hm} ::=	types in head normal form in color hm
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
STRING	dynamic
h.TYPE	hash if $h \notin hm$
$\top$	Top
TC ::=	first-level constructed type context
UNIT	unit
STRING	dynamic
$_{-1} \rightarrow _{-2}$	function
$_{-1} * ... * _{-j}$	tuple ($2 \leqslant j$)
$\{l_1{:}_{-1}; ...; l_j{:}_{-j}\}$	record ($1 \leqslant j$)
hm ::=	hash sets
$\bullet$	empty
$\{h\}$	some hash (we may sometimes forget the braces)
$hm \cup hm$	union of hash sets
h ::=**hash**$(N, H, M{:}[X{:}\mathbf{Le}(T), T'])$	hash
H ::=	subhash relationship
nil	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 ::=$[T, v^{\bullet}]$	module structure (type-part, term-part)
S ::=$[X{:}K, T]$	module signature (X binds in T)

$m ::=$	machine
$\quad e$	expression
$\quad \mathbf{module}\ N_U\ \mathbf{extends}\ \kappa_1 ... \kappa_i\ \mathbf{restricts}\ \kappa'_1 ... \kappa'_j$	module declaration (U binds in m), $0 \leqslant i$, $0 \leqslant j$.
$\quad\quad = M{:}S\ \mathbf{in}\ m$	
$n ::=$	network
$\quad \mathbf{0}$	null
$\quad n \mid n$	parallel composition
$\quad e$	expression (on one machine)
$\kappa ::=$	elements of the subhash relationship
$\quad U$	module name
$\quad h$	hash
$\zeta ::=$	variable
$\quad x$	expression variable
$\quad X$	type variable
$\quad U$	module variable
$\chi ::=$	substitutable entity
$\quad X$	type variable
$\quad U.\textsc{type}$	type-part of a module
$\quad x$	expression variable
$\quad U.\text{term}$	term-part of a module
$\quad U$	module variable
$\ddot{X} ::=$	type substitutable entity
$\quad X$	type variable
$\quad U.\textsc{type}$	type-part of a module
$\dddot{x} ::=$	expression substitutable entity
$\quad x$	expression variable
$\quad U.\text{term}$	term-part of a module
$E ::=$	environment
$\quad E, x{:}T$	expression variable binding
$\quad E, X{:}K$	type variable binding
$\quad E, U(T){:}S$	module variable binding
$\quad \mathbf{nil}$	empty

$J ::=$		colourable statement
	ok	environment correctness
	K ok	kind correctness
	$K == K'$	kind equivalence
	$K <: K'$	subkinding
	$T{:}K$	kind of a type
	$T == T'$	type equivalence
	$T <: T'$	subtyping
	S ok	signature correctness
	$S == S'$	signature equivalence
	$S <: S'$	subsignaturing
	$e{:}T$	type of an expression
	$M{:}S$	signature of a module expression
	$U{:}S$	signature of a module variable
	$m{:}T$	type of a machine
CJ ::=	$E \vdash^H_{hm} J$	couloured judgement
MJ ::=		monochrome judgement
	$\vdash hm$ ok	hash set and subhash correctness
	$\vdash h$ ok	hash correctness
	$\vdash h \in hm$	hash set membership
	$\vdash n$ ok	network correctness
AJ ::=		judgement
	$\zeta \notin \mathrm{dom}\, E$	non-clash judgement
	$E \vdash_{hm} J$	coloured judgement
	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
τ ok $::=$	$T{:}\mathbf{Le}(\top)$	$\mid$	K ok	$\mid$	S ok	sort is correct

$=$	syntactic equality
$\in$	syntactic membership
σ, μ	substitutions
π	permutation
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.TYPE, U.term, X, x) (if U.TYPE $\in$ fse $\aleph$ or U.term $\in$ fse $\aleph$ then $U \in$ fse $\aleph$)

A.2 Typing rules

A.2.1 $\boxed{\zeta \notin \mathrm{dom}\, E}$ non-clash in environments

$$\frac{}{\zeta \notin \mathrm{dom}\, \mathbf{nil}}\ (\mathsf{clash.nil}) \qquad \frac{\zeta \notin \mathrm{dom}\, E \quad \zeta \neq \zeta'}{\zeta \notin \mathrm{dom}\, (E, \zeta' {:} \tau)}\ (\mathsf{clash.cons})$$

For any two distinct variables ζ and ζ', $\zeta \neq \zeta'$ is an axiom.

A.2.2 $\boxed{U \notin \mathrm{dom}\, H}$ non-clash in the subhash relationship

$$\frac{}{U \notin \mathrm{dom}\, \mathbf{nil}}\ (\mathsf{clash.hash.nil}) \qquad \frac{U \notin \mathrm{dom}\, H \quad U \neq \kappa \quad U \neq \kappa'}{U \notin \mathrm{dom}\, (H \cup \{\kappa <: \kappa\})}\ (\mathsf{clash.hash.cons})$$

For any two distinct variables U and U', $U \neq U'$ is an axiom.

A.2.3 $\boxed{\vdash h\,\mathrm{ok}}$ hash correctness

$$\frac{\mathbf{nil} \vdash^{H}_{\bullet} M{:}[X{:}\mathbf{Le}(T),\, T']}{\vdash \mathbf{hash}(N, H, M{:}[X{:}\mathbf{Le}(T),\, T'])\,\mathrm{ok}}\ (\mathsf{hok.hash})$$

Note that N may be any external name.

Note that we demand that a module have an opaque type (at least partially) in order to take its hash.

A.2.4 $\boxed{\vdash hm\,\mathrm{ok}}$ hash sets correctness

$$\frac{\vdash h\,\mathrm{ok}}{\vdash \{h\}\,\mathrm{ok}}\ (\mathsf{hmok.sing}) \qquad \frac{\vdash hm_0\,\mathrm{ok} \quad \vdash hm_1\,\mathrm{ok}}{\vdash hm_0 \cup hm_1\,\mathrm{ok}}\ (\mathsf{hmok.union}) \qquad \frac{}{\vdash \bullet\,\mathrm{ok}}\ (\mathsf{hmok.zero})$$

A.2.5 $\boxed{E \vdash^{H}_{hm}\ \mathrm{ok}}$ environment correctness

$$\frac{\vdash hm\,\mathrm{ok}}{\mathbf{nil} \vdash^{\mathbf{nil}}_{hm}\ \mathrm{ok}}\ (\mathsf{envok.nil}) \qquad \frac{E \vdash^{H}_{hm} T{:}\mathbf{Le}(\top) \quad x \notin \mathrm{dom}\, E}{E, x{:}T \vdash^{H}_{hm}\ \mathrm{ok}}\ (\mathsf{envok.x})$$

$$\frac{\begin{array}{l}\vdash \mathbf{hash}(N_0, H_0, [T_0, v_0^{\bullet}]{:}[X{:}K_0,\, T_0'])\,\mathrm{ok} \\ \vdash \mathbf{hash}(N_1, H_1, [T_1, v_1^{\bullet}]{:}[X{:}K_1,\, T_1'])\,\mathrm{ok} \\ \mathbf{nil} \vdash^{H}_{hm} T_0 <: T_1 \\ \mathbf{nil} \vdash^{H}_{hm} K_0 <: K_1\end{array}}{\mathbf{nil} \vdash^{H \cup \{\mathbf{hash}(N_0, H_0, [T_0, v_0^{\bullet}]{:}[X{:}K_0, T_0']) <: \mathbf{hash}(N_1, H_1, [T_1, v_1^{\bullet}]{:}[X{:}K_1, T_1'])\}}_{hm}\ \mathrm{ok}}\ (\mathsf{envok.hashhash})$$

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_0, H_0, [T_0, v_0^\bullet]{:}[X{:}K_0,\, T_0']) \,\mathrm{ok} \\ E,\, U(T_1){:}[X{:}K_1,\, T_1'] \vdash_{hm}^{H}\ T_0 <: T_1 \\ E,\, U(T_1){:}[X{:}K_1,\, T_1'] \vdash_{hm}^{H}\ K_0 <: K_1\end{array}}{E,\, U(T_1){:}[X{:}K_1,\, T_1'] \vdash_{hm}^{H\cup\{\mathbf{hash}(N_0,H_0,[T_0,v_0^\bullet]:[X:K_0,T_0'])<:U\}}\ \mathrm{ok}}\ \textsf{(envok.hashU)}$$

$$\frac{\begin{array}{c}\vdash \mathbf{hash}(N_1, H_1, [T_1, v_1^\bullet]{:}[X{:}K_1,\, T_1']) \,\mathrm{ok} \\ E,\, U(T_0){:}[X{:}K_0,\, T_0'] \vdash_{hm}^{H}\ T_0 <: T_1 \\ E,\, U(T_0){:}[X{:}K_0,\, T_0'] \vdash_{hm}^{H}\ K_0 <: K_1\end{array}}{E,\, U(T_0){:}[X{:}K_0,\, T_0'] \vdash_{hm}^{H\cup\{U<:\mathbf{hash}(N_1,H_1,[T_1,v_1^\bullet]:[X:K_1,T_1'])\}}\ \mathrm{ok}}\ \textsf{(envok.Uhash)}$$

$$\frac{\begin{array}{c}E_1,\, U_1(T_1){:}[X{:}K_1,\, T_1'], E_2,\, U_0(T_0){:}[X{:}K_0,\, T_0'] \vdash_{hm}^{H}\ T_0 <: T_1 \\ E_1,\, U_1(T_1){:}[X{:}K_1,\, T_1'], E_2,\, U_0(T_0){:}[X{:}K_0,\, T_0'] \vdash_{hm}^{H}\ K_0 <: K_1\end{array}}{E_1,\, U_1(T_1){:}[X{:}K_1,\, T_1'], E_2,\, U_0(T_0){:}[X{:}K_0,\, T_0'] \vdash_{hm}^{H\cup\{U_0<:U_1\}}\ \mathrm{ok}}\ \textsf{(envok.mod)}$$

$$\frac{\begin{array}{c}E_1,\, U_0(T_0){:}[X{:}K_0,\, T_0'], E_2,\, U_1(T_1){:}[X{:}K_1,\, T_1'] \vdash_{hm}^{H}\ T_0 <: T_1 \\ E_1,\, U_0(T_0){:}[X{:}K_0,\, T_0'], E_2,\, U_1(T_1){:}[X{:}K_1,\, T_1'] \vdash_{hm}^{H}\ K_0 <: K_1\end{array}}{E_1,\, U_0(T_0){:}[X{:}K_0,\, T_0'], E_2,\, U_1(T_1){:}[X{:}K_1,\, T_1'] \vdash_{hm}^{H\cup\{U_0<:U_1\}}\ \mathrm{ok}}\ \textsf{(envok.modopp)}$$

$$\frac{E \vdash_{hm}^{H}\ K\,\mathrm{ok} \qquad X \notin \mathrm{dom}\, E}{E, X{:}K \vdash_{hm}^{H}\ \mathrm{ok}}\ \textsf{(envok.X)} \qquad \frac{\begin{array}{c}E \vdash_{hm}^{H}\ T_0{:}K \\ E \vdash_{hm}^{H}\ [X{:}K,\, T]\,\mathrm{ok} \\ U \notin \mathrm{dom}\, E \qquad U \notin \mathrm{dom}\, H\end{array}}{E,\, U(T_0){:}[X{:}K,\, T] \vdash_{hm}^{H}\ \mathrm{ok}}\ \textsf{(envok.U)}$$

An alternate way of stating (envok. ?) could be :

$$\frac{E \vdash_{hm}^{H}\ \tau\,\mathrm{ok} \qquad \zeta \notin \mathrm{dom}\, E}{E, \zeta{:}\tau \vdash_{hm}^{H}\ \mathrm{ok}}\ \textsf{(envok. ?)}$$

A.2.6 $\boxed{E \vdash_{hm}^{H}\ K\,\mathrm{ok}}$ kind correctness

$$\frac{E \vdash_{hm}^{H}\ T{:}\mathbf{Le}(\top)}{E \vdash_{hm}^{H}\ \mathbf{Le}(T)\,\mathrm{ok}}\ \textsf{(Kok.Le)} \qquad \frac{E \vdash_{hm}^{H}\ T{:}\mathbf{Le}(\top)}{E \vdash_{hm}^{H}\ \mathbf{Eq}(T)\,\mathrm{ok}}\ \textsf{(Kok.Eq)}$$

A.2.7 $\boxed{E \vdash_{hm}^{H}\ K == K'}$ kind equality

$$\frac{E \vdash_{hm}^{H}\ T == T'}{E \vdash_{hm}^{H}\ \mathbf{Le}(T) == \mathbf{Le}(T')}\ \textsf{(Keq.Le)} \qquad \frac{E \vdash_{hm}^{H}\ T == T'}{E \vdash_{hm}^{H}\ \mathbf{Eq}(T) == \mathbf{Eq}(T')}\ \textsf{(Keq.Eq)}$$

A.2.8 $\boxed{E \vdash^H_{hm} K <: K'}$ subkinding

$$\frac{E \vdash^H_{hm} T{:}\mathbf{Le}(\top)}{E \vdash^H_{hm} \mathbf{Eq}(T) <: \mathbf{Le}(T)}\ (\mathsf{Ksub.Eq}) \qquad \frac{E \vdash^H_{hm} T <: T'}{E \vdash^H_{hm} \mathbf{Le}(T) <: \mathbf{Le}(T')}\ (\mathsf{Ksub.Le})$$

$$\frac{E \vdash^H_{hm} K == K'}{E \vdash^H_{hm} K <: K'}\ (\mathsf{Ksub.refl}) \qquad \frac{E \vdash^H_{hm} K <: K' \quad E \vdash^H_{hm} K' <: K''}{E \vdash^H_{hm} K <: K''}\ (\mathsf{Ksub.tran})$$

Note that (Ksub.tran) is currently derivable.

A.2.9 $\boxed{E \vdash^H_{hm} T{:}K}$ kind of a type

$$\frac{E \vdash^H_{hm} T{:}K \quad E \vdash^H_{hm} K <: K'}{E \vdash^H_{hm} T{:}K'}\ (\mathsf{TK.sub}) \qquad \frac{E \vdash^H_{hm} T == T'}{E \vdash^H_{hm} T{:}\mathbf{Eq}(T')}\ (\mathsf{TK.Eq}) \qquad \frac{E \vdash^H_{hm} T <: T'}{E \vdash^H_{hm} T{:}\mathbf{Le}(T')}\ (\mathsf{TK.Le})$$

$$\frac{E, X{:}K, E' \vdash^H_{hm} \mathrm{ok}}{E, X{:}K, E' \vdash^H_{hm} X{:}K}\ (\mathsf{TK.var}) \qquad \frac{E \vdash^H_{hm} U{:}[X{:}K, T]}{E \vdash^H_{hm} U.\mathrm{TYPE}{:}K}\ (\mathsf{TK.mod})$$

$$\frac{E \vdash^H_{hm} \mathrm{ok} \qquad \vdash h\,\mathrm{ok}}{E \vdash^H_{hm} h.\mathrm{TYPE}{:}K}\ (\mathsf{TK.hash})$$
where $h = \mathbf{hash}(N, H', [T, v^\bullet]{:}[X{:}K, T'])$

A.2.10 $\boxed{E \vdash^H_{hm} T == T'}$ type equivalence

$$\frac{E \vdash^H_{hm} T{:}\mathbf{Eq}(T')}{E \vdash^H_{hm} T == T'}\ (\mathsf{Teq.Eq})$$

$$\frac{E \vdash^H_{hm} \mathrm{ok}}{E \vdash^H_{hm} h.\mathrm{TYPE} == T}\ (\mathsf{Teq.hash})$$
where $h = \mathbf{hash}(N, H', [T, v^\bullet]{:}[X{:}\mathbf{Le}(T'), T'']) \quad \in \quad hm$

The rule (Teq.hash) is the rule that introduces type equivalences (see Lemma B.6.6 (type decomposition)). Other rules only serve to propagate equivalences. Of course another way to obtain a type equivalence is to have an explicit $\mathbf{Eq}(T)$ in the judgement and use (Teq.Eq).

$$\frac{E \vdash^H_{hm} T{:}\mathbf{Le}(\top)}{E \vdash^H_{hm} T == T}\ (\mathsf{Teq.refl}) \qquad \frac{E \vdash^H_{hm} T == T'}{E \vdash^H_{hm} T' == T}\ (\mathsf{Teq.sym}) \qquad \frac{E \vdash^H_{hm} T == T' \quad E \vdash^H_{hm} T' == T''}{E \vdash^H_{hm} T == T''}\ (\mathsf{Teq.tran})$$

$$\frac{E \vdash^H_{hm} T_0 == T'_0 \quad E \vdash^H_{hm} T_1 == T'_1}{E \vdash^H_{hm} T_0 \to T_1 == T'_0 \to T'_1}\ (\mathsf{Teq.cong.fun}) \qquad \frac{E \vdash^H_{hm} T_i == T'_i \quad \forall i.1 \leqslant i \leqslant j}{E \vdash^H_{hm} T_1 * \ldots * T_j == T'_1 * \ldots * T'_j}\ (\mathsf{Teq.cong.tuple})$$

$$\frac{E \vdash^H_{hm} T_i == T'_i \quad \forall i.1 \leqslant i \leqslant j}{E \vdash^H_{hm} \{l_1{:}T_1, ..., l_j{:}T_j\} == \{l_1{:}T'_1, ..., l_j{:}T'_j\}} \text{ (Teq.cong.rec)}$$

$$\frac{E \vdash^H_{hm} T_i{:}\mathbf{Le}(\top) \quad 1 \leqslant i \leqslant j}{E \vdash^H_{hm} \{l_1{:}T_1, ..., l_j{:}T_j\} == \{l_{\pi(1)}{:}T_{\pi(1)}, ..., l_{\pi(j)}{:}T_{\pi(j)}\}} \text{ (Teq.cong.rec.perm)}$$

A.2.11 $\boxed{E \vdash^H_{hm} T <: T'}$ subtyping

$$\frac{E \vdash^H_{hm} T{:}\mathbf{Le}(T')}{E \vdash^H_{hm} T <: T'} \text{ (Tsub.Le)} \qquad \frac{E \vdash^H_{hm} T == T'}{E \vdash^H_{hm} T <: T'} \text{ (Tsub.Equi)}$$

$$\frac{E \vdash^H_{hm} \text{ ok} \qquad \kappa <: \kappa' \in H}{E \vdash^H_{hm} \kappa.\text{TYPE} <: \kappa'.\text{TYPE}} \text{ (Tsub.Subhash)}$$

.

We have the choice between width and depth subtyping for records. For now we choose to ignore depth subtyping for simplicity.

$$\frac{E \vdash^H_{hm} T_i{:}\mathbf{Le}(\top) \quad 1 \leqslant i \leqslant k}{E \vdash^H_{hm} \{l_1{:}T_1, ..., l_j{:}T_j, ..., l_k{:}T_k\} <: \{l_1{:}T_1, ..., l_j{:}T_j\}} \text{ (Tsub.cong.record.width)}$$

$$\frac{E \vdash^H_{hm} T_i <: T'_i \quad 1 \leqslant i \leqslant j}{E \vdash^H_{hm} \{l_1{:}T_1, ..., l_j{:}T_j\} <: \{l_1{:}T'_1, ..., l_j{:}T'_j\}} \text{ (Tsub.cong.record.depth)}$$

$$\frac{\begin{array}{c}E \vdash^H_{hm} T'_0 <: T_0 \\ E \vdash^H_{hm} T_1 <: T'_1\end{array}}{E \vdash^H_{hm} T_0 \to T_1 <: T'_0 \to T'_1} \text{ (Tsub.cong.fun)} \qquad \frac{E \vdash^H_{hm} T_i <: T'_i \quad 1 \leqslant i \leqslant j}{E \vdash^H_{hm} T_1 * ... * T_j <: T'_1 * ... * T'_j} \text{ (Tsub.cong.tuple)}$$

The following rules have to be present to ensure that if T is well-formed then T is a subtype of $\top$. This allows (with (TK.Le)) to prove that any well-formed T has kind $\mathbf{Le}(\top)$.

$$\frac{\begin{array}{c}\vdash h \text{ ok} \\ E \vdash^H_{hm} \text{ ok}\end{array}}{E \vdash^H_{hm} h.\text{TYPE} <: \top} \text{ (Tsub.hash)} \qquad \frac{E \vdash^H_{hm} \text{ ok}}{E \vdash^H_{hm} \top <: \top} \text{ (Tsub.top)}$$

$$\frac{E \vdash^H_{hm} \text{ ok}}{E \vdash^H_{hm} \text{UNIT} <: \top} \text{ (Tsub.unit)} \qquad \frac{E \vdash^H_{hm} \text{ ok}}{E \vdash^H_{hm} \text{STRING} <: \top} \text{ (Tsub.dyn)}$$

$$\frac{\begin{array}{c}E \vdash^H_{hm} T <: \top \\ E \vdash^H_{hm} T' <: \top\end{array}}{E \vdash^H_{hm} T \to T' <: \top} \text{ (Tsub.fun)} \qquad \frac{E \vdash^H_{hm} T_i <: \top \quad \forall i.1 \leqslant i \leqslant j}{E \vdash^H_{hm} T_1 * ... * T_j <: \top} \text{ (Tsub.tuple)}$$

$$\frac{E \vdash^H_{hm} T_i <: \top \quad \forall i.1 \leqslant i \leqslant j}{E \vdash^H_{hm} \{l_1{:}T_1, ..., l_j{:}T_j\} <: \top} \text{ (Tsub.rec)}$$

A.2.12 $\boxed{E \vdash^H_{hm} S\,\mathrm{ok}}$ signature correctness

$$\frac{E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)}{E \vdash^H_{hm} [X{:}K, T]\,\mathrm{ok}}\ (\mathsf{Sok})$$

A.2.13 $\boxed{E \vdash^H_{hm} S == S'}$ signature equivalence

Note that we never use signature equivalence.

$$\frac{E \vdash^H_{hm} K == K' \qquad E, X{:}K \vdash^H_{hm} T == T'}{E \vdash^H_{hm} [X{:}K, T] == [X{:}K', T']}\ (\mathsf{Seq.struct})$$

Signature equivalence is an equivalence relation since kind equivalence and type equivalence are.

A.2.14 $\boxed{E \vdash^H_{hm} S <: S'}$ subsignaturing

$$\frac{\begin{array}{c}E \vdash^H_{hm} K <: K' \\ E, X{:}K \vdash^H_{hm} T <: T'\end{array}}{E \vdash^H_{hm} [X{:}K, T] <: [X{:}K', T']}\ (\mathsf{Ssub.struct})$$

$$\frac{E \vdash^H_{hm} S\,\mathrm{ok}}{E \vdash^H_{hm} S <: S}\ (\mathsf{Ssub.refl}) \qquad \frac{\begin{array}{c}E \vdash^H_{hm} S <: S' \\ E \vdash^H_{hm} S' <: S''\end{array}}{E \vdash^H_{hm} S <: S''}\ (\mathsf{Ssub.tran})$$

Note that (Ssub.refl) and (Ssub.tran) are derivable.

A.2.15 $\boxed{E \vdash^H_{hm} e{:}T}$ type of an expression

$$\frac{\begin{array}{c}E \vdash^H_{hm} e{:}T \\ E \vdash^H_{hm} T <: T'\end{array}}{E \vdash^H_{hm\ (T<:T')} e{:}T'}\ (\mathsf{eT.sub}) \qquad \frac{\begin{array}{c}E \vdash^H_{hm} e{:}T \\ E \vdash^H_{hm} T == T'\end{array}}{E \vdash^H_{hm} e{:}T'}\ (\mathsf{eT.eq})$$

$$\frac{E, x{:}T, E' \vdash^H_{hm} \mathrm{ok}}{E, x{:}T, E' \vdash^H_{hm} x{:}T}\ (\mathsf{eT.var}) \qquad \frac{\begin{array}{c}E \vdash^H_{hm} U{:}[X{:}K, T] \\ E \vdash^H_{hm} T{:}\mathbf{Le}(\top)\end{array}}{E \vdash^H_{hm} U.\mathrm{term}{:}T}\ (\mathsf{eT.mod})$$

Note that in (eT.mod), the condition $E \vdash_{hm} T{:}\mathbf{Le}(\top)$ guarantees that $X \notin \mathrm{fv}\ T$.

$$\frac{\begin{array}{c}E \vdash^H_{hm} e'{:}T \to T' \\ E \vdash^H_{hm} e{:}T\end{array}}{E \vdash^H_{hm} e'\ e{:}T'}\ (\mathsf{eT.ap}) \qquad \frac{E, x{:}T \vdash^H_{hm} e{:}T'}{E \vdash^H_{hm} \lambda x{:}T.e{:}T \to T'}\ (\mathsf{eT.fun})$$

$$\frac{E \vdash^H_{hm} e_i{:}T_i \quad \forall i.1 \leqslant i \leqslant j}{E \vdash^H_{hm} (e_1, .., e_j){:}T_1 * ... * T_j}\ (\mathsf{eT.tuple}) \qquad \frac{E \vdash^H_{hm} e{:}T_1 * ... * T_j \quad 1 \leqslant i \leqslant j}{E \vdash^H_{hm} \mathbf{proj}_i\, e{:}T_i}\ (\mathsf{eT.proj})$$

$$\frac{E \vdash^H_{hm} e_i{:}T_i \quad \forall i.1 \leqslant i \leqslant j}{E \vdash^H_{hm} \{l_1 = e_1, .., l_j = e_j\}{:}\{l_1{:}T_1, ..., l_j{:}T_j\}} \text{ (eT.record)}$$

$$\frac{E \vdash^H_{hm} e{:}\{l_1{:}T_1, ..., l_j{:}T_j\} \quad 1 \leqslant i \leqslant j}{E \vdash^H_{hm} e.l_i{:}T_i} \text{ (eT.field)}$$

$$\frac{E \vdash^H_{hm} \text{ ok}}{E \vdash^H_{hm} ():\text{UNIT}} \text{ (eT.unit)} \qquad \frac{E \vdash^H_{hm} e{:}\text{STRING}}{E \vdash^H_{hm} !\, e{:}\text{UNIT}} \text{ (eT.send)} \qquad \frac{E \vdash^H_{hm} \text{ ok}}{E \vdash^H_{hm} ?{:}\text{STRING}} \text{ (eT.recv)}$$

$$\frac{E \vdash^H_{hm} e{:}T}{E \vdash^H_{hm} \mathbf{mar}\,(e{:}T){:}\text{STRING}} \text{ (eT.mar)} \qquad \frac{E \vdash^{H_0}_{hm} \text{ ok} \quad \mathbf{nil} \vdash^{H_1}_{\bullet} e{:}T}{E \vdash^{H_0}_{hm} \mathbf{marshalled}_{H_1}(e{:}T){:}\text{STRING}} \text{ (eT.marred)}$$

$$\frac{E \vdash^H_{hm} T{:}\mathbf{Le}(\top) \quad E \vdash^H_{hm} e{:}\text{STRING}}{E \vdash^H_{hm} (\mathbf{unmar}\, e{:}T){:}T} \text{ (eT.unmar)} \qquad \frac{E \vdash^H_{hm} T{:}\mathbf{Le}(\top)}{E \vdash^H_{hm} \mathbf{Unmarfailure}^T{:}T} \text{ (eT.Undynfailure)}$$

For (eT.unmar), the condition $E \vdash_{hm} T{:}\mathbf{Le}(\top)$ 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.

$$\frac{E \vdash^H_{hm} T{:}\mathbf{Le}(\top) \quad E \vdash^H_{hm \cup hm'} e{:}T}{E \vdash^H_{hm} [e]^T_{hm'}{:}T} \text{ (eT.col)}$$

Note that the colors are additive here.

A.2.16 $\boxed{E \vdash^H_{hm} M{:}S}$ signature of a module expression

$$\frac{E \vdash^H_{hm} T{:}K \quad E, X{:}K \vdash^H_{hm} T'{:}\mathbf{Le}(\top) \quad E, X{:}\mathbf{Eq}(T) \vdash^H_{hm} T'' <: T' \quad E \vdash^H_{hm} v^{hm}{:}T''}{E \vdash^H_{hm} [T, v^{hm}]{:}[X{:}K, T']} \text{ (MS.struct)} \qquad \frac{E \vdash^H_{hm} M{:}S \quad E \vdash^H_{hm} S <: S'}{E \vdash^H_{hm} M{:}S'} \text{ (MS.sub)}$$

A.2.17 $\boxed{E \vdash^H_{hm} U{:}S}$ signature of a module variable

$$\frac{E, U(T){:}S, E' \vdash^H_{hm} \text{ ok}}{E, U(T){:}S, E' \vdash^H_{hm} U{:}S} \text{ (US.var)} \qquad \frac{E \vdash^H_{hm} U{:}S \quad E \vdash^H_{hm} S == S'}{E \vdash^H_{hm} U{:}S'} \text{ (US.eq)} \qquad \frac{E \vdash^H_{hm} U{:}[X{:}K, T]}{E \vdash^H_{hm} U{:}[X{:}\mathbf{Eq}(U.\text{TYPE}), T]} \text{ (US.self)}$$

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.

$$\frac{\begin{array}{l}E \vdash^H_\bullet T{:}\mathbf{Le}(\top)\\ E \vdash^H_\bullet [T_0, v^\bullet]{:}S\\ E, U(T_0){:}S \vdash^{H\cup\{\kappa_1<:U\}\cup\ldots\cup\{\kappa_i<:U\}\cup\{U<:\kappa'_1\}\cup\ldots\cup\{U<:\kappa'_j\}}_\bullet m{:}T\end{array}}{E \vdash^H_\bullet (\mathbf{module}\, N_U\, \mathbf{extends}\, \kappa_1 \ldots \kappa_i\, \mathbf{restricts}\, \kappa'_1 \ldots \kappa'_j = [T_0, v^\bullet]{:}S\ \mathbf{in}\ m){:}T}\ (\mathsf{mT.letext})$$

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

$$\frac{\vdash n_i\,\mathrm{ok} \quad i=1,2}{\vdash n_1 \mid n_2\,\mathrm{ok}}\ (\mathsf{nok.par}) \qquad \frac{}{\vdash \mathbf{0}\,\mathrm{ok}}\ (\mathsf{nok.zero}) \qquad \frac{\mathbf{nil} \vdash^{\mathbf{nil}}_\bullet e{:}\mathrm{UNIT}}{\vdash e\,\mathrm{ok}}\ (\mathsf{nok.expr})$$

A.3 Semantics

A.3.1 $\boxed{H, m \longrightarrow_{\mathsf{c}} H', m'}$ compile-time reduction

$$H, \mathbf{module}\, N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \mathbf{in}\ m \longrightarrow_{\mathsf{c}} H, \{U.\mathrm{TYPE}{\leftarrow}T'', U.\mathrm{term}{\leftarrow}v^\bullet\}m \qquad (\mathsf{mred.Eq})$$

$$\begin{array}{l}H, \mathbf{module}\, N_U\, \mathbf{extends}\, h_1 \ldots h_i\, \mathbf{restricts}\, h'_1 \ldots h'_j = [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T']\ \mathbf{in}\ m \longrightarrow_{\mathsf{c}}\\ \qquad H \cup \{h_1 <: h\} \cup \ldots \cup \{h_i <: h\} \cup \{h <: h'_1\} \cup \ldots \cup \{h <: h'_j\},\\ \qquad \{U{\leftarrow}h,\, U.\mathrm{TYPE}{\leftarrow}h.\mathrm{TYPE},\, U.\mathrm{term}{\leftarrow}[v^\bullet]^{\{X\leftarrow h.\mathrm{TYPE}\}T'}_{\{h\}}\}m \qquad (\mathsf{mred.Le})\\ \qquad \text{where} \quad h = \mathbf{hash}(N, H, [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T'])\end{array}$$

A.3.2 $\boxed{H, e \longrightarrow_{hm} H', e'}$ expression reduction

Expression

$$H, \mathbf{proj}_i\,(v_1^{hm}, \ldots, v_j^{hm}) \longrightarrow_{hm} H, v_i^{hm} \qquad \text{if } 1 \leqslant i \leqslant j \qquad (\mathsf{ered.proj})$$

$$H, \{l_1 = v_1^{hm}, \ldots, l_j = v_j^{hm}\}.l_i \longrightarrow_{hm} H, v_i^{hm} \qquad \text{if } 1 \leqslant i \leqslant j \qquad (\mathsf{ered.field})$$

$$H, (\lambda x{:}T.e)\, v^{hm} \longrightarrow_{hm} H, \{x{\leftarrow}[v^{hm}]^T_{hm}\}e \qquad (\mathsf{ered.ap})$$

Marshalling

$$H, \mathbf{mar}\,(v^{hm}{:}T) \longrightarrow_{hm} H, \mathbf{marshalled}_{\,H}([v^{hm}]^{T}_{hm}{:}T) \qquad \text{(ered.mar)}$$

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{\,H'}(v^{\bullet}{:}T){:}T') \longrightarrow_{hm} \begin{cases} H \cup H', {}_{(T<:T')}v^{\bullet} & \text{if } \mathbf{nil} \vdash^{H\cup H'}_{hm} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases} \qquad \text{(ered.unmar)}$$

Subtyping

$$H, {}_{(TV_0^{hm}<:TV_1^{hm})}v^{hm} \longrightarrow_{hm} H, {}_{(TV_2^{hm}<:TV_1^{hm})}\widehat{v}^{hm} \quad \text{where } v^{hm} = {}_{(TV_2^{hm}<:TV_3^{hm})}\widehat{v}^{hm} \qquad \text{(ered.sub.sub)}$$

$$H, {}_{(T_0<:h_0.\mathrm{TYPE})}\widehat{v}^{hm} \longrightarrow_{hm} H, {}_{(T_0<:T_1)}\widehat{v}^{hm} \quad \text{when } h_0 \in hm \text{ and } \mathrm{impl}(h_0){=}T_1 \qquad \text{(ered.sub.typeright)}$$

$$H, {}_{(h_0.\mathrm{TYPE}<:TV^{hm})}\widehat{v}^{hm} \longrightarrow_{hm} H, {}_{(T_0<:TV^{hm})}\widehat{v}^{hm} \quad \text{when } h_0 \in hm \text{ and } \mathrm{impl}(h_0){=}T_0 \qquad \text{(ered.sub.typeleft)}$$

$$H, {}_{(T_1*\ldots*T_j<:T'_1*\ldots*T'_j)}(v_1^{hm}, \ldots, v_j^{hm}) \longrightarrow_{hm} H, ({}_{(T_1<:T'_1)}v_1^{hm}, \ldots, {}_{(T_j<:T'_j)}v_j^{hm}) \qquad \text{(ered.sub.tuple)}$$

$$H, {}_{(\{l_1:T_1;\ldots;l_j:T_j\}<:\{l_{\pi(1)}:T_{\pi(1)};\ldots;l_{\pi(i)}:T_{\pi(i)}\})}\{l_1 = v_1^{hm}; \ldots; l_j = v_j^{hm}\} \longrightarrow_{hm} H, \{l_{\pi(1)} = v_{\pi(1)}^{hm}; \ldots; l_{\pi(i)} = v_{\pi(i)}^{hm}\} \qquad \text{(ered.sub.record)}$$

We suppose here that $i \leqslant j$. Note that we don't have depth subtyping.

$$H, {}_{(T_1\to T_2<:T'_1\to T'_2)}(\lambda x{:}T_0.e) \longrightarrow_{hm} H, (\lambda x{:}T'_1.\,{}_{(T_2<:T'_2)}\{x \leftarrow {}_{(T'_1<:T_1)}x\}e) \qquad \text{(ered.sub.fun)}$$

$$H, {}_{(\mathrm{STRING}<:\mathrm{STRING})}\mathbf{marshalled}_{\,H'}(v^{\bullet}{:}T) \longrightarrow_{hm} H, \mathbf{marshalled}_{\,H'}(v^{\bullet}{:}T) \qquad \text{(ered.sub.marshalled)}$$

$$H, {}_{(\mathrm{UNIT}<:\mathrm{UNIT})}() \longrightarrow_{hm} H, () \qquad \text{(ered.sub.unit)}$$

$$H, {}_{(T'<:T'')}([\widehat{v}^{hm'\cup hm}]^{h.\mathrm{TYPE}}_{hm'}) \longrightarrow_{hm} H, [{}_{(T'<:T'')}\widehat{v}^{hm'\cup hm}]^{T''}_{hm'} \quad \text{where } h \notin hm \qquad \text{(ered.sub.col)}$$

Bracket pushing

$$H, [\widehat{v}^{hm'\cup hm}]^{h.\text{TYPE}}_{hm'} \longrightarrow_{hm} H, [\widehat{v}^{hm'\cup hm}]^{T}_{hm'} \qquad \text{when } h \in hm \text{ and } \text{impl}(h){=}T \qquad \text{(ered.col.type)}$$

$$H, [(\widehat{v}_1^{hm'\cup hm}, ..., \widehat{v}_j^{hm'\cup hm})]^{T_1 * ... * T_j}_{hm'} \longrightarrow_{hm} H, ([\widehat{v}_1^{hm'\cup hm}]^{T_1}_{hm'}, ..., [\widehat{v}_j^{hm'\cup hm}]^{T_j}_{hm'}) \qquad \text{(ered.col.tuple)}$$

$$H, [\{l_1 = \widehat{v}_1^{hm'\cup hm}, .., l_j = \widehat{v}_j^{hm'\cup hm}\}]^{\{l_1:T_1,...,l_j:T_j\}}_{hm'} \longrightarrow_{hm} H, \{l_1 = [\widehat{v}_1^{hm'\cup hm}]^{T_1}_{hm'}, .., l_j = [\widehat{v}_j^{hm'\cup hm}]^{T_j}_{hm'}\} \qquad \text{(ered.col.record)}$$

$$H, [\lambda x{:}T.e]^{T'\to T''}_{hm'} \longrightarrow_{hm} H, (\lambda x{:}T'.[\{x \leftarrow [x]^{T}_{hm'}\}e]^{T''}_{hm'}) \qquad \text{(ered.col.fun)}$$

$$H, [\textbf{marshalled}_{H'}(v^{\bullet}{:}T)]^{\text{STRING}}_{hm'} \longrightarrow_{hm} H, \textbf{marshalled}_{H'}(v^{\bullet}{:}T) \qquad \text{(ered.col.marred)}$$

$$H, [()]^{\text{UNIT}}_{hm'} \longrightarrow_{hm} H, () \qquad \text{(ered.col.unit)}$$

Congruence

$$\frac{H, e \longrightarrow_{hm} H', e'}{H, C^{hm'}_{hm}.e \longrightarrow_{hm'} H', C^{hm'}_{hm}.e'} \text{ (ered.cong)}$$

A.3.3 $n \equiv n'$ network structural congruence

$$\frac{}{\mathbf{0} \mid n \equiv n} \text{ (nsc.id)} \qquad \frac{}{n_1 \mid n_2 \equiv n_2 \mid n_1} \text{ (nsc.commut)} \qquad \frac{}{n_1 \mid (n_2 \mid n_3) \equiv (n_1 \mid n_2) \mid n_3} \text{ (nsc.assoc)}$$

Plus reflexivity, symmetry and transitivity of $\equiv$.

A.3.4 $n \longrightarrow n'$ network reduction

$$\frac{e \longrightarrow_{\bullet} e'}{e \longrightarrow e'} \text{ (nred.expr)} \qquad \frac{n \longrightarrow n'}{n \mid n'' \longrightarrow n' \mid n''} \text{ (nred.par)} \qquad \frac{n \equiv n_0 \longrightarrow n_0' \equiv n'}{n \longrightarrow n'} \text{ (nred.strcong)}$$

$$CC^{\bullet}_{hm}.!\, v^{hm} \mid CC^{\bullet}_{hm'}.? \longrightarrow CC^{\bullet}_{hm}.() \mid CC^{\bullet}_{hm'}.v^{hm} \qquad \text{(nred.comm)}$$

Annexe B

Théorèmes et preuves

We use the words "proof" and "derivation" indifferently, to mean a natural deduction-style tree of inference steps leading to a judgement.

Definition B.0.1 (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.1 Correctness

Definition B.1.1 (domain of an environment) The domain of an environment E, written $\mathrm{dom}\, E$, is a set of variables defined by induction as follows :

- $\mathrm{dom}\, \mathbf{nil} = \varnothing$
- $\mathrm{dom}\, (E, \zeta{:}\tau) = \mathrm{dom}\, E \cup \{\zeta\}$

Lemma B.1.2 (non-membership in domain is interpreted trivially) The judgement $\zeta \notin \mathrm{dom}\, E$ is provable iff ζ is not a member of the domain of E.

Proof. Induct on the derivation of $\zeta \notin \mathrm{dom}\, E$. The rules (clash.nil) and (clash.cons) trivially maintain this property. □

Definition B.1.3 (domain of a subhash relation) The domain of a subhash relation H, written $\mathrm{dom}\, H$, is a set of variables defined by induction as follows :

- $\mathrm{dom}\, \mathbf{nil} = \varnothing$
- $\mathrm{dom}\, (H \cup \{U <: V\}) = \mathrm{dom}\, H \cup \{U; V\}$
- $\mathrm{dom}\, (H \cup \{U <: h\}) = \mathrm{dom}\, H \cup \{U\}$
- $\mathrm{dom}\, (H \cup \{h <: V\}) = \mathrm{dom}\, H \cup \{V\}$
- $\mathrm{dom}\, (H \cup \{h <: h'\}) = \mathrm{dom}\, H$

Lemma B.1.4 (non-membership in subhash domain is interpreted trivially) The judgement $U \notin \mathrm{dom}\, H$ is provable iff U is not a member of the domain of H.

Proof. Induct on the derivation of $U \notin \text{dom}\, H$. 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.1.5 (colours have to be ok) If $E \vdash^H_{hm} J$ then $\vdash hm$ ok by a proper subproof.

Proof. Induct on the derivation of $E \vdash^H_{hm} J$. 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 rules (envok.nil), which have $\vdash hm$ ok as a premise. □

Lemma B.1.6 (hashes have to be ok) If $E \vdash^H_{hm} J$ or $\vdash hm$ ok is derivable by a proof Π and h is a subterm of $E \vdash^H_{hm} J$ or $\vdash hm$ ok then $\vdash h$ ok by a subproof of Π.

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 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 $\vdash \mathbf{hash}(N, H', M{:}S)$ ok. If $h = \mathbf{hash}(N, H', M{:}S)$ 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 $\mathbf{hash}(N_0, H_0, [T_0, v_0^{hm}]{:}S_0) <: \mathbf{hash}(N_1, H_1, [T_1, v_1^{hm}]{:}S_1)$ in (envok.hashhash) Two of the premises are $\vdash \mathbf{hash}(N_0, H_0, [T_0, v_0^{hm}]{:}S_0)$ ok and $\vdash \mathbf{hash}(N_1, H_1, [T_1, v_1^{hm}]{:}S_1)$ ok, giving the desired result.

Case subterm of $\mathbf{hash}(N_0, H_0, [T_0, v_0^{hm}]{:}S_0) <: U$ in (envok.hashU) One of the premises is $\vdash \mathbf{hash}(N_0, H_0, [T_0, v_0^{hm}]{:}S_0)$ ok, giving the desired result.

Case subterm of $U <: \mathbf{hash}(N_1, H_1, [T_1, v_1^{hm}]{:}S_1)$ in (envok.Uhash) One of the premises is $\vdash \mathbf{hash}(N_1, H_1, [T_1, v_1^{hm}]{:}S_1)$ ok, giving the desired result.

Case subterm of $\mathbf{hash}(N, H', M{:}S)$ in (TK.hash) : One premise is $\vdash \mathbf{hash}(N, H', M{:}S)$ ok. If $h = \mathbf{hash}(N, H', M{:}S)$ we have the desired result. Otherwise induction gives the desired result.

Case subterm of $E \vdash^H_{hm} h.\text{TYPE} == T$ in (Teq.hash) : Since h **in** hm, induction gives the desired result.

Case h in $E \vdash^H_{hm} h.\text{TYPE} <: \top$ in (Tsub.hash) : One of the premises is $\vdash h$ ok, giving the desired result.

Case N_U in (mT.letext) : Trivial ($\text{fv}\, N = \varnothing$).

□

Lemma B.1.7 (environments and subhashes have to be ok) If $E \vdash^H_{hm} J$ then $E \vdash^H_{hm}$ ok by a subproof.

Proof. Simultaneously with the following lemma. □

Lemma B.1.8 (prefixes of ok environments are ok) If $E, b \vdash^H_{hm}$ ok then there exists a subset H' of H such that $E \vdash^{H'}_{hm}$ ok by a subproof. In particular, if b isnot in the domain of H, we can take $H' = H$.

Proof. Induct on the derivation of $E, b \vdash^H_{hm}$ ok. Note that if $b \notin \operatorname{dom} H$, $E, b \vdash^H_{hm}$ ok can only be derived from a judgement of the form $E \vdash^H_{hm} J$, hence the result of the second lemma follows by induction from the first lemma. If $b \in \operatorname{dom} H$, $E, b \vdash^H_{hm}$ ok can only be derived from some judgements $E, b \vdash^{H'}_{hm} J$ with H' strictly smaller than 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 $E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. By induction, we get $E, X{:}K \vdash^H_{hm}$ ok by a subproof. We conclude by the second lemma with $E \vdash^H_{hm}$ 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 $E, x{:}T \vdash^H_{hm} e{:}T'$. Since x binds an expression variable, by induction, $E, x{:}T \vdash^H_{hm}$ ok by a subproof, hence $E \vdash^H_{hm}$ ok by the second lemma, as desired.

□

Lemma B.1.9 (environments and subhashes are ok in the empty colour) If $E \vdash^H_{hm}$ ok, then $E \vdash^H_{\bullet}$ ok.

Proof. To do. □

Lemma B.1.10 (ok environments have no repetition in the domain)
If $E, E' \vdash^H_{hm}$ ok then $\operatorname{dom} E \cap \operatorname{dom} E' = \varnothing$.

Proof. Induct 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 :

- $\zeta \notin \operatorname{dom} H$ $E, E' \vdash^H_{hm}$ ok must have been derived by the appropriate (envok.*) rule, with the premises $E, E'' \vdash^H_{hm} \tau$ ok and $\zeta \notin \operatorname{dom}(E, E'')$. From the first premise, by Lemma B.1.7 (environments and subhashes have to be ok), we get $E, E'' \vdash^H_{hm}$ 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.
- $\zeta \in \operatorname{dom} H$ $E, E' \vdash^H_{hm}$ ok must have been derived from some judgements of the form $E, E' \vdash^{H'}_{hm}$ ok with H' strictly smaller than H. By induction we have $\operatorname{dom} E \cap \operatorname{dom} E' = \varnothing$.

□

Lemma B.1.11 (free variables of a judgement come from the environment)
If $E \vdash^H_{hm} J$ then $\operatorname{fv}(J) \cup \operatorname{fv}(H) \subseteq \operatorname{dom}(E)$. For completeness's sake : if $\vdash hm$ ok then $\operatorname{fv}(hm) \subseteq \varnothing$; 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.1.2 (non-membership in domain is interpreted trivially) and Lemma B.1.4 (non-membership in subhash domain is interpreted trivially).

Induct on the size of the derivation Π of the judgement.

Most rules whose conclusion is a coloured judgement have the following property : every meta-variable (including H) in the right-hand side of the conclusion $E \vdash^H_{hm} J$ 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 $\vdash \mathbf{hash}(N, H, M{:}S)\,\mathrm{ok}$ and the premise is $\mathbf{nil} \vdash^H_\bullet M{:}S$. By induction we have $\mathrm{fv}\,(\mathbf{hash}(N, \lambda,)M{:}S) = \mathrm{dom}\,\mathbf{nil} = \varnothing$ 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 $E \vdash^H_{hm} \mathbf{hash}(N, H', M{:}[X{:}K, T'])\,{:}K$. One of the premises is $\vdash \mathbf{hash}(N, H', M{:}[X{:}K, T'])\,\mathrm{ok}$. By induction $\mathrm{fv}\,(\mathbf{hash}(N, H', M{:}[X{:}K, T'])) = \varnothing$, and in particular $\mathrm{fv}\,(K) = \varnothing \subseteq \mathrm{dom}\,E$ as desired.

Case (Teq.hash) : The conclusion is $E \vdash^H_{hm} h == T$ where $h = \mathbf{hash}(N, H', [T, v^{hm}]{:}S)$ and $h \in hm$. The premise is $E \vdash^H_{hm}$ ok. By Lemma B.1.5 (colours have to be ok), $\vdash hm\,\mathrm{ok}$ by a proper subproof. By induction, we have $\mathrm{fv}\,hm = \varnothing$. As $h \in hm$, we have $\mathrm{fv}\,h = \varnothing$ and in particular $\mathrm{fv}\,T = \varnothing$, so $\mathrm{fv}\,(h == T) = \varnothing \subseteq \mathrm{dom}\,E$.

Case (Tsub.Subhash) : The conclusion is $E \vdash^H_{hm} \kappa.\mathrm{TYPE} <: \kappa'.\mathrm{TYPE}$ with $E \vdash^H_{hm}$ 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 $\mathrm{fv}\,\kappa.\mathrm{TYPE} <: \kappa'.\mathrm{TYPE} \subseteq \mathrm{dom}\,E$.

Case h in (Tsub.hash) : One of the premises is $\vdash h\,\mathrm{ok}$. 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 $E, \zeta{:}\tau \vdash^H_{hm} J'$ 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 $E, \zeta{:}\aleph \vdash^H_{hm} J'$ where E and hm are the environment and the colour of the conclusion and $\aleph$ is the metavariable under consideration. By Lemma B.1.7 (environments and subhashes have to be ok) and reversing the appropriate (envok.*) rule, we have, by a proper subproof, respectively, $E \vdash^H_{hm} K\,\mathrm{ok}$, $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$, $E \vdash^H_{hm} M{:}S$. 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.1.6 (hashes have to be ok), for any hash h that is a subterm of T', we have $\vdash h\,\mathrm{ok}$ 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 $E \vdash_{hm} [e]^T_{hm'}{:}T$. All that remains to be shown is that $\mathrm{fv}\,hm' \subseteq \mathrm{dom}\,E$. One premise of the rule is $E \vdash_{hm'} e{:}T$. By Lemma B.1.5 (colours have to be ok), we have $\vdash hm'\,\mathrm{ok}$ by a proper subproof, so by induction $\mathrm{fv}\,hm' = \varnothing$ whence the desired result.

Case (eT.marred) The type T, the expression e and the subhash relations H_0 and H_1 from the conclusion are present in one of the premises with an empty environment. By induction we can then state that they do not have any free variable.

Case N (in (mT.letext)) : Trivial as $\mathrm{fv}\, N = \varnothing$.

Case (nok.expr) : The conclusion is $\vdash e\,\mathrm{ok}$ and the premise is $\mathbf{nil} \vdash_\bullet e{:}\textsc{unit}$. By induction we have $\mathrm{fv}\, e \subseteq \mathrm{dom}\,\mathbf{nil} = \varnothing$ as desired.

□

Definition B.1.12 (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 $U{:}S \vdash_{hm} U.\textsc{type}{:}\mathbf{Le}(\top)$, one has to use (TK.mod), with a premise of the form $U{:}S \vdash_{hm} U{:}S'$.

Definition B.1.13 (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.2 Variables and colours

Definition B.2.1 (hashes in something) The hashes in a syntactic entity are the subterms that are $\mathbf{hash}(N, H, M{:}S)$ and that are not themselves subterms of a hash.

Definition B.2.2 (partial order on colours) We call *topped colours* the set formed by all colours (i.e. hm).

We define a partial order on topped colours as follows : $hm \preccurlyeq hm'$ iff $hm \subseteq hm'$. We name $\mathfrak{H}$ the set of all hashes.

Note that $\preccurlyeq$ defines a poset with all greatest lower bounds, with $\varnothing$ as the minimum element and $\mathfrak{H}$ as the maximum element.

Definition B.2.3 (pvu) Let Π be a proof of a coloured judgement $E \vdash^H_{hm} J$, and let ζ be any variable. The set of colours at which the variable ζ is used in the proof Π, written $\mathrm{pvu}_\zeta(\Pi)$, is defined as follows.

Consider the last rule used in the proof. The environment of its conclusion is an environment pattern. Take all the metavariables $\aleph$ in this pattern whose instantiation in the last step of Π contains ζ as a variable bound by the environment. Then $\mathrm{pvu}_\zeta(\Pi)$ is the union over all $\aleph$ of the following sets :

- If $\aleph$ occurs in the right-hand side of the conclusion of the rule, anywhere but under an explicit bracket, then $\{hm\}$, else $\varnothing$.
- For every occurence of $\aleph$ in the right-hand side of the conclusion of the rule under an explicit bracket, the colour that the subscript of the bracket instantiates to.

– For every occurence of $\aleph$ in the environment part of a premise which is a coloured judgement, $\mathrm{pvu}_\zeta(\Pi')$ where Π' is the subproof of Π that leads to said premise.

Definition B.2.4 (alternate, informal definition of pvu) Consider a proof Π of a coloured judgement of the form $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$. The set of colours at which the variable ζ is pulled from the environment in Π, written $\mathrm{pvu}_\zeta(\Pi)$, is the set of colours hm such that the judgement $E_0, \zeta{:}\tau, E' \vdash^H_{hm} \zeta{:}\tau$ appears in the proof as a conclusion of a var-rule ((eT.var), (TK.var) or (US.var)).

If ζ is not in the domain of the conclusion of Π, then we define $\mathrm{pvu}_\zeta(\Pi)$ to be the empty set.

Lemma B.2.5 (monotonicity of pvu)
If Π' is a subproof of Π then $\mathrm{pvu}_\zeta(\Pi') \subseteq \mathrm{pvu}_\zeta(\Pi)$. Hence $\min\big(\mathrm{pvu}_\zeta(\Pi)\big) \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi')\big)$.

Proof. Trivial from the first definition. □

Definition B.2.6 (substitution) Some potentially interesting cases :

$$\begin{aligned}
\sigma(\mathbf{mar}\,(e_0{:}T_0)) &= \mathbf{mar}\,(\sigma e_0{:}\sigma T_0)\\
\sigma(\mathbf{marshalled}_H(e_0{:}T_0)) &= \mathbf{marshalled}_H(\sigma e_0{:}\sigma T_0)\\
\sigma(\mathbf{unmar}\,e_0{:}T_0) &= \mathbf{unmar}\,\sigma e_0{:}\sigma T_0\\
\sigma([e_0]^T_{hm_0}) &= [\sigma e_0]^{\sigma T}_{\sigma hm_0}\\
\{U.\textsc{type}{\leftarrow}T\}U.\textsc{type} &= T\\
\{U.\mathrm{term}{\leftarrow}e\}U.\mathrm{term} &= e\\
\sigma(\kappa <: \kappa') &= \sigma\kappa <: \sigma\kappa'
\end{aligned}$$

Note that substitution performs all necessary alpha-conversions. We generally leave alpha-conversion implicit.

Lemma B.2.7 (stability of values by substitution) Let σ be any substitution, hm be a colour and v^{hm} be any hm-value with correct hashes. Then σv^{hm} is an hm-value.

Proof. Induct on the structure of v^{hm}. As per the syntax of values, the only places where a value may contain free variables are inside hashes or under a λ. Since the hashes in v^{hm} are assumed to be correct, they are closed by Lemma B.1.11 (free variables of a judgement come from the environment). As for λ's, they allow an arbitrary expression, hence they are stable by substitution. □

Lemma B.2.8 (computing the pvu of a type world judgement) Let Π be a proof of a type world judgement $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$. Then $hm \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi)\big)$.

Proof. Induct on the structure of Π. We freely use Lemma B.1.7 (environments and subhashes have to be ok) and Lemma B.1.10 (ok environments have no repetition in the domain).

Consider first the rules whose conclusion is a type world judgement but for which $J \neq$ ok. Each premise has one of the following properties :

1. The premise is a type world judgement whose colour is the same as in the conclusion and whose environment contains the conclusion's.
2. The premise is either not a coloured judgement or a coloured judgement with an empty environment.

Suppose that the conclusion of Π is obtained by such a rule. Then consider any subproof Π' of Π leading to a premise of the aforementioned conclusion. One of the alternatives holds :

Case 1 : By induction, $hm \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi')\big)$.

Case 2 : Then $\mathrm{pvu}_\zeta(\Pi') = \varnothing$ whence $hm \preccurlyeq \mathfrak{H} = \min\big(\mathrm{pvu}_\zeta(\Pi')\big)$.

The set $\mathrm{pvu}_\zeta(\Pi)$ is the union over all the premises of the $\mathrm{pvu}_\zeta(\Pi')$'s, plus hm if the rule under consideration is (TK.var) or (US.var) introducing ζ. Hence $hm \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi)\big)$.

Let us turn to the (envok.*) rules. We distinguish as to whether there is a variable added to the environment and wheter ζ is this variable.

Case (envok.*) not adding any variable Each premise has one of the properties 1 and 2. We can conclude as before.

Case (envok.*) not adding ζ : There exist E', ζ', and τ' such that the conclusion is $E_0, \zeta{:}\tau, E', \zeta'{:}\tau' \vdash^H_{hm}$ ok, and the premises are $\zeta' \notin \mathrm{dom}(E_0, \zeta{:}\tau, E')$ and $E_0, \zeta{:}\tau, E' \vdash^H_{hm} \tau\,\mathrm{ok}$. Let Π' be the subproof leading to the latter judgement. Then by induction $hm \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi')\big)$, whence the desired result as $\mathrm{pvu}_\zeta(\Pi) = \mathrm{pvu}_\zeta(\Pi')$.

Case (envok.*) adding ζ : The conclusion is $E_0, \zeta{:}\tau \vdash^H_{hm}$ ok. The only premise that is a coloured judgement is $E_0 \vdash^H_{hm} \tau\,\mathrm{ok}$, which does not contain ζ. So $\mathrm{pvu}_\zeta(\Pi) = \varnothing$ whence $hm \preccurlyeq \top = \min\big(\mathrm{pvu}_\zeta(\Pi)\big)$.

The only remaining rule is (eT.col). Suppose that the conclusion of Π is obtained by such a rule. There exist e, T and hm' such that the conclusion is $E_0, \zeta{:}\tau, E \vdash^H_{hm} [e]^T_{hm'}{:}T$. The only specific subproof is the one concluding by $E_0, \zeta{:}\tau, E \vdash^H_{hm\cup hm'} e{:}T$. By induction we get $hm \cup hm' \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi')\big)$ from which we deduce $hm \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi')\big)$. As twhe set $\mathrm{pvu}_\zeta(\Pi)$ is the union over all the premises of the $\mathrm{pvu}_\zeta(\Pi')$'s in this case, we get $hm \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi)\big)$. □

Lemma B.2.9 (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 $U.\textsc{type} \in \mathrm{fse}\,\aleph$ then $U \in \mathrm{fv}\,\aleph$.

We may use this lemma implicitly.

Proof. Trivial from the definition of fv and fse. □

Lemma B.2.10 (types do not contain free expression variables)
If $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ then $\mathrm{fse}\,T$ does not contain any expression substitutable entity (i.e. $\ddot{x}$). Also, if $E \vdash^H_{hm} K\,\mathrm{ok}$ (respectively $E \vdash^H_{hm} S\,\mathrm{ok}$) 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, $U.\textsc{type}$) or obvious by induction (constructed type). The only non-trivial case is a hash type $h.\textsc{type}$. By Lemma B.1.6 (hashes have to be ok), $\vdash h\,\mathrm{ok}$. By Lemma B.1.11 (free variables of a judgement come from the environment), $\mathrm{fv}\,h = \varnothing$, whence by Lemma B.2.9 (connection between fv and fse) $\mathrm{fse}\,h = \varnothing$.

If $E \vdash^H_{hm} K\,\mathrm{ok}$, then by reversing (Kok.Eq) or(Kok.Le) we get $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$, whence by the first part of this lemma $\mathrm{fse}\,K = \mathrm{fse}\,T$ has the desired property.

If $E \vdash^H_{hm} [X{:}K, T]\,\mathrm{ok}$, then by reversing (Sok) we get $E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. 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.2.11 (environments do not contain free expression variables)
If $E_0, E_1 \vdash^H_{hm} \mathrm{ok}$ 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.1.8 (prefixes of ok environments are ok), we have $H' \subseteq H$ and $E_0, E_1' \vdash^{H'}_{hm} \mathrm{ok}$ by a subproof. By induction, we have $\ddot{x} \notin \mathrm{fse}\, E_1'$. Also, by Lemma B.2.10 (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.2.12 (expression substitution in environment) If $E_0, E_1 \vdash^H_{hm} \mathrm{ok}$ and $\ddot{x}$ is an expression substitutable entity (i.e. an expression variable or U.term for some U) then $\{\ddot{x} \leftarrow \eta\} E_1 = E_1$.

Proof. Trivial consequence of Lemma B.2.11 (environments do not contain free expression variables). □

B.3 Weakening

Lemma B.3.1 ("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.3.2 (colour stripping judgements)
If $E \vdash^H_{hm} J$ and $\vdash hm'\,\mathrm{ok}$ and $hm \preccurlyeq hm'$ then $E \vdash^H_{hm'} J$ for all coloured statements J other than "type of a machine".

Proof. Induct on the derivation of $E \vdash^H_{hm} J$. 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.3.1 ("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 hm by hm' does yield an instance of the rule again), we have $E \vdash^H_{hm'} J$. We list the remaining cases.

Case (envok.nil) : Trivial ($\vdash hm'\,\mathrm{ok}$ is assumed in this lemma).

Case (Teq.hash) : $h \in hm \preccurlyeq hm'$, so $h \in hm'$.

Case (eT.marred) : Trivial.

Case (eT.col) : There exist e, T and hm_0 such that the conclusion is $E \vdash^H_{hm} [e]^T_{hm_0} : T$. Since we have $hm \cup hm_0 \preccurlyeq hm' \cup hm_0$ we can apply the induction hypothesis to the premise $E \vdash^H_{hm \cup hm_0} e : T$.

□

Lemma B.3.3 (subhash weakening)
If $E \vdash^H_{hm} J$ and $E \vdash^{H'}_{hm}$ ok and $H \subseteq H'$ then $E \vdash^{H'}_{hm} J$.

Proof. Induct on the derivation of $E \vdash^H_{hm} J$. 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 $J =$ ok **:** The second hypothesis gives us the result.

Case (eT.fun) : The premise has an additional expression variable binding. We know that $E \vdash^{H'}_{hm}$ ok and that $E \vdash^H_{hm} \lambda x : T.e : T \to T'$ and $E, x : T \vdash^H_{hm} e : T'$. By Lemma B.1.7 (environments and subhashes have to be ok), we get $E, x : T \vdash^H_{hm}$ ok by a subproof. By reversing (envok.x) (x cannot be in $\operatorname{dom} H$), we have $E \vdash^H_{hm} T : \mathbf{Le}(\top)$. By induction, we get $E \vdash^{H'}_{hm} T : \mathbf{Le}(\top)$ and then by (envok.x) $E, x : T \vdash^{H'}_{hm}$ ok. We conclude by induction and (eT.fun).

Cases (Sok), (Ssub.struct), (MS.struct) : Similar to (eT.fun).

Case (mT.letext) : One of the premises is $E, U(T_0) : S \vdash^{H \cup \{\kappa_1 <: U\} \cup \ldots \cup \{\kappa_i <: U\} \cup \{U <: \kappa'_1\} \cup \ldots \cup \{U <: \kappa'_j\}}_\bullet m : T$. By Lemma B.1.7 (environments and subhashes have to be ok) we get $E, U(T_0) : S \vdash^{H \cup \{\kappa_1 <: U\} \cup \ldots \cup \{\kappa_i <: U\} \cup \{U <: \kappa'_1\} \cup \ldots \cup \{U <: \kappa'_j\}}_\bullet$ ok by a subproof. By reversing $i + j$ times (envok.Uhash) and (envok.hashU) we get $E, U(T_0) : S \vdash^H_\bullet$ ok. Then we reverse (envok.U), apply the induction hypothesis and reapply (envok.U) to get $E, U(T_0) : S \vdash^{H'}_\bullet$ ok. By induction and (envok.Uhash) and (envok.hashU) we have $E, U(T_0) : S \vdash^{H' \cup \{\kappa_1 <: U\} \cup \ldots \cup \{\kappa_i <: U\} \cup \{U <: \kappa'_1\} \cup \ldots \cup \{U <: \kappa'_j\}}_\bullet$ ok and we conclude by induction and (mT.letext).

□

Lemma B.3.4 (weakening)
If $E, E', E'' \vdash^H_{hm}$ ok and $E, E'' \vdash^H_{hm} J$ then $E, E', E'' \vdash^H_{hm} J$.

Furthermore, if $\zeta \in \operatorname{dom} E'$ and $E, E', E'' \vdash^H_{hm}$ ok is derived by a proof Π such that $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, then there is a proof Π' of $E, E', E'' \vdash^H_{hm} J$ such that $\operatorname{pvu}_\zeta(\Pi') = \varnothing$.

Proof. We freely use Lemma B.1.2 (non-membership in domain is interpreted trivially) and Lemma B.1.4 (non-membership in subhash domain is interpreted trivially).

Consider the variables that appear in the derivation of $E, E'' \vdash^H_{hm} J$ but not in the judgement $E, E'' \vdash^H_{hm} J$ itself. We can alpha-convert them to variables that are not present in $\operatorname{dom}(E, E', E'', H)$.

Induct on the derivation of $E, E'' \vdash^H_{hm} J$.

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 $\hat{E}$ 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 a possibly bigger subhash relationship.
   (b) The premise is $\hat{\zeta} \notin \operatorname{dom} \hat{E}$ for some $\hat{\zeta}$ that is in the domain of the conclusion.
   (c) The premise does not mention $\hat{E}$.

Suppose that $E, E'' \vdash^{H}_{hm} J$ was derived by an instance α of such a rule. There are two cases, depending on whether the instantiation of $\hat{E}$ (as per condition 1) includes the whole of E or not.

Case $\hat{E}$ 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 $\hat{E}$ 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 $E, E'', E_i \vdash^{H_i}_{hm_i} J_i$, where E_i is of length at most one. By Lemma B.1.7 (environments and subhashes have to be ok), $E, E'', E_i \vdash^{H_i}_{hm_i}$ ok by a subproof.

If E_i is empty, then we have $E, E', E'', E_i \vdash^{H_i}_{hm_i}$ ok by induction. Otherwise there exist ζ' and τ such that $E_i = \zeta' {:} \tau$. The judgement $E, E'', \zeta' {:} \tau$ must have been derived by the appropriate (envok.*) rules from $E, E'' \vdash^{H'_i}_{hm_i} \tau$ ok and $\zeta' \notin \operatorname{dom}(E, E'', H'_i)$. By induction, we have $E, E', E'' \vdash^{H'_i}_{hm_i} \tau$ ok, whence by the same (envok.*) rules as previously : $E, E', E'', \zeta' {:} \tau \vdash^{H_i}_{hm_i}$ ok (recalling that we performed alpha-conversion on the proof so that $\zeta' \notin \operatorname{dom} E''$ whence $\zeta' \notin \operatorname{dom}(E, E', E'', H_i)$). By Lemma B.3.3 (subhash weakening) we finally get $E, E', E'', \zeta' {:} \tau \vdash^{H_i}_{hm_i}$ ok.

If furthermore $\zeta \in \operatorname{dom} E'$ and $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, then the induction gives a proof Π'' of $E, E', E'', \zeta' {:} \tau \vdash^{H_i}_{hm_i}$ ok such that $\operatorname{pvu}_\zeta(\Pi'') = \varnothing$.

In any case, $E, E', E'', E_i \vdash^{H_i}_{hm_i}$ ok, so we can apply induction, getting $E, E', E'', E_i \vdash^{H_i}_{hm_i} J_i$ as desired.

If $\zeta \in \operatorname{dom} E'$ and $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, then we have obtained a proof Π'' of $E, E', E'', E_i \vdash^{H_i}_{hm_i}$ ok such that $\operatorname{pvu}_\zeta(\Pi'') = \varnothing$, and the last induction gives a proof Π' of $E, E', E'', E_i \vdash^{H_i}_{hm_i} J_i$ such that $\operatorname{pvu}_\zeta(\Pi') = \varnothing$.

Case 2b : The premise is $\zeta' \notin \operatorname{dom}(E, E''')$ with ζ' in $\operatorname{dom}(E, E''', E'''')$. Hence $\zeta' \in \operatorname{dom} E''''$. Since $E, E', E''', E'''' \vdash^{H}_{hm}$ ok, by Lemma B.1.10 (ok environments have no repetition in the domain), we have $\zeta \notin \operatorname{dom}(E, E', E''')$ as desired.

Case 2c : The premise in α is exactly the premise in ω.

We have a derivation of all the premises in ω, so we get a proof Π' of its conclusion. Note further that if $\zeta \in \operatorname{dom} E'$ and $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, we do get that $\operatorname{pvu}_\zeta(\Pi') = \varnothing$. (If ω is an instance of a (*.var) rule, then ζ is not the variable being introduced by $\zeta \in \operatorname{dom} E'$ and Lemma B.1.10 (ok environments have no repetition in the domain).)

Case $\hat{E}$ 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 $E''', \zeta':\tau, E'''', E', E'' \vdash^H_{hm}$ ok. By (*.var) we get a proof Π' of $E''', \zeta':\tau, E'''', E', E'' \vdash^H_{hm} \zeta':\tau$ as desired.

If $\zeta \in \operatorname{dom} E'$ and $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, then $\operatorname{pvu}_\zeta(\Pi') = \varnothing$, because $\zeta \neq \zeta'$ since $\zeta \in \operatorname{dom} E'$ and Lemma B.1.10 (ok environments have no repetition in the domain).

The only rule whose conclusion is a coloured judgement that does not match the conditions above is (envok.nil). If the last step of the derivation uses this rule, then its conclusion is $\mathbf{nil} \vdash^{\mathbf{nil}}_{hm}$ ok, and we desire a proof of $E' \vdash^{\mathbf{nil}}_{hm}$ ok, which holds by assumption : take $\Pi' = \Pi$. □

Lemma B.3.5 (merging environments)
If $E, E' \vdash^H_{hm}$ ok and $E, E'' \vdash^H_{hm}$ ok and $\operatorname{dom} E' \cap \operatorname{dom} E'' = \varnothing$ then $E, E', E'' \vdash^H_{hm}$ ok.

Furthermore, if $\zeta \in \operatorname{dom} E'$ and $E, E' \vdash^H_{hm}$ ok is derived by a proof Π such that $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, then there is a proof Π' of $E, E', E'' \vdash^H_{hm}$ ok such that $\operatorname{pvu}_\zeta(\Pi') = \varnothing$.

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.1.8 (prefixes of ok environments are ok) $E \vdash^H_{hm}$ ok.

We freely use Lemma B.1.2 (non-membership in domain is interpreted trivially) and Lemma B.1.4 (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 $E, E'', \zeta':\tau \vdash^H_{hm}$ 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 $E, E'' \vdash^H_{hm} \tau$ ok. By Lemma B.1.7 (environments and subhashes have to be ok), we have $E, E'' \vdash^H_{hm}$ ok. By induction, we get $E, E', E'' \vdash^H_{hm}$ ok. Then by applying the appropriate (envok.{x,X,U}) rule we get $E, E', E'', \zeta':\tau \vdash^H_{hm}$ ok as desired.

Suppose furthermore that $\zeta \in \operatorname{dom} E'$ and $\operatorname{pvu}_\zeta(\Pi) = \varnothing$. Then the proof of $E, E', E'' \vdash^H_{hm}$ ok obtained above by induction and that of $E, E', E'' \vdash^H_{hm} \tau$ ok obtained by Lemma B.3.4 (weakening) have an empty pvu for ζ, so the proof Π' of $E, E', E'', \zeta:\tau$ that we build satisfies $\operatorname{pvu}_\zeta(\Pi') = \varnothing$. □

Lemma B.3.6 (combined weakening)
If $E, E' \vdash^H_{hm}$ ok and $E, E'' \vdash^H_{hm} J$ and $\operatorname{dom} E' \cap \operatorname{dom} E'' = \varnothing$ then $E, E', E'' \vdash^H_{hm} J$.

Furthermore, if $\zeta \in \operatorname{dom} E'$ and $E, E' \vdash^H_{hm}$ ok is derived by a proof Π such that $\operatorname{pvu}_\zeta(\Pi) = \varnothing$, then there is a proof Π' of $E, E', E'' \vdash^H_{hm} J$ such that $\operatorname{pvu}_\zeta(\Pi') = \varnothing$.

Proof. Trivial combination of Lemma B.3.5 (merging environments) and Lemma B.3.4 (weakening). □

Lemma B.3.7 (environment and subhash weakening) If $E \vdash^H_{hm} J$ and $E, E' \vdash^{H'}_{hm}$ ok with $H \subseteq H'$ then $E, E' \vdash^{H'}_{hm} J$.

Proof. Induction on the derivation of $E, E' \vdash^{H'}_{hm}$ ok.

Case (envok.nil) : Trivial.

Cases (envok.x), (envok.X), (envok.U) : The conclusion is $E, E', \zeta:\tau \vdash^{H'}_{hm}$ ok, and one premise is $E, E' \vdash^{H'}_{hm} \tau$ ok. By Lemma B.1.7 (environments and subhashes have to be ok) we have $E, E' \vdash^{H'}_{hm}$ ok by a subproof. By induction we get $E, E' \vdash^{H'}_{hm} J$ and we conclude by Lemma B.3.4 (weakening).

Cases remaining (envok.*) rules : The conclusion is $E, E' \vdash_{hm}^{H' \cup \{\kappa <: \kappa'\}}$ ok, and one premise is $E, E' \vdash_{hm}^{H'} \kappa <: \kappa'$. By Lemma B.1.7 (environments and subhashes have to be ok) we have $E, E' \vdash_{hm}^{H'}$ ok by a subproof. By induction we get $E, E' \vdash_{hm}^{H'} J$ and we conclude by Lemma B.3.3 (subhash weakening).

□

B.4 Type system

Lemma B.4.1 (reflexivity of kind equivalence) If $E \vdash_{hm}^{H} K$ ok then $E \vdash_{hm}^{H} K == K$.

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 $E \vdash_{hm}^{H} T{:}\mathbf{Le}(\top)$. By using (Teq.refl) and (Keq.Le), we have the desired $E \vdash_{hm}^{H} K == K$. □

Lemma B.4.2 (transitivity of kind equivalence) If $E \vdash_{hm}^{H} K == K'$ and $E \vdash_{hm}^{H} K' == K''$ then $E \vdash_{hm}^{H} K == K''$.

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.4.3 (discreteness of subkinding) If $E \vdash_{hm}^{H} K <: \mathbf{Eq}(T)$ then $E \vdash_{hm}^{H} K == \mathbf{Eq}(T)$ by a subproof.

Proof. Induct on the derivation of $E \vdash_{hm}^{H} K <: \mathbf{Eq}(T)$. If the last rule in the proof is (Ksub.tran), then the result holds by induction and Lemma B.4.2 (transitivity of kind equivalence). Otherwise the last rule is (Ksub.refl) and the premise is the desired result. □

Lemma B.4.4 (kinds are smaller than top) If $E \vdash_{hm}^{H} K$ ok then $E \vdash_{hm}^{H} K <: \mathbf{Le}(\top)$.

Proof. If there exists T such that $K = \mathbf{Le}(T)$, then by reversing (Kok.Le) we get $E \vdash_{hm}^{H} T{:}\mathbf{Le}(\top)$. To have $E \vdash_{hm}^{H} \mathbf{Le}(T) <: \mathbf{Le}(\top)$, 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 $E \vdash_{hm}^{H} \mathbf{Eq}(T) <: \mathbf{Le}(\top)$. □

Lemma B.4.5 (transitivity of subtyping) If $E \vdash_{hm}^{H} T <: T'$ and $E \vdash_{hm}^{H} T' <: T''$ then $E \vdash_{hm}^{H} T <: T''$.

Proof. By (Ksub.Le) we have $E \vdash_{hm}^{H} \mathbf{Le}(T') <: \mathbf{Le}(T'')$. On the other side, we know that $E \vdash_{hm}^{H} T{:}\mathbf{Le}(T')$ by (TK.Le). We use (TK.sub) to get $E \vdash_{hm}^{H} T{:}\mathbf{Le}(T'')$. We conclude by (Tsub.Le). □

Lemma B.4.6 (signature equivalence is transitive) If $E \vdash_{hm}^{H} S == S'$ and $E \vdash_{hm}^{H} S' == S''$ then $E \vdash_{hm}^{H} S == S''$.

Proof. Trivial by transitivity of type and kind equivalence. □

Lemma B.4.7 (components of modules are ok) If $E \vdash_{hm}^{H} [T, v^{hm}]{:}[X{:}K, T']$ then there exists T'' such that $E \vdash_{hm}^{H} T{:}K$ and $E, X{:}K \vdash_{hm}^{H} T'{:}\mathbf{Le}(\top)$ and $E, X{:}\mathbf{Eq}(T) \vdash_{hm}^{H} T'' <: T'$ and $E \vdash_{hm}^{H} v^{hm}{:}T''$ and $E \vdash_{hm}^{H} K$ ok.

Proof. Reverse (MS.struct) to get the first four judgements. As for the last one, since $E, X{:}K \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, by Lemma B.1.7 (environments and subhashes have to be ok) and reversing (envok.X) (X cannot be in $\mathrm{dom}\, H$), we get $E \vdash_{hm} K$ ok. □

Lemma B.4.8 (bindings in an ok environment are ok)] If $E, \zeta{:}\tau, E' \vdash^H_{hm}$ ok then we have a proof of $E, \zeta{:}\tau, E' \vdash^H_{hm} \tau$ ok or $E, \zeta{:}\tau, E' \vdash^H_{hm} \tau{:}\mathbf{Le}(\top)$ if τ is a type.

Proof. Induct on the derivation of $E, \zeta{:}\tau, E' \vdash^H_{hm}$ 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 $E, \zeta{:}\tau, E' \vdash^{H'}_{hm} J$ with $H' \subseteq H$. Then by Lemma B.1.7 (environments and subhashes have to be ok), we have $E, \zeta{:}\tau, E' \vdash^{H'}_{hm}$ ok by a subproof. By induction we get $E, \zeta{:}\tau, E' \vdash^{H'}_{hm} \tau$ ok. By Lemma B.3.3 (subhash weakening), we finally have $E, \zeta{:}\tau, E' \vdash^H_{hm} \tau$ ok.

Cases (envok.x),(envok.X),(envok.U) : If $E' = \mathbf{nil}$, then a premise will be $E \vdash^H_{hm} \tau$ ok or $E \vdash^H_{hm} \tau{:}\mathbf{Le}(\top)$ if τ is a type. We conclude by Lemma B.3.4 (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 $E, \zeta{:}\tau, E'' \vdash^H_{hm} \tau'$ ok. By Lemma B.1.7 (environments and subhashes have to be ok), we get $E, \zeta{:}\tau, E'' \vdash^H_{hm}$ ok and by induction $E, \zeta{:}\tau, E'' \vdash^H_{hm} \tau$ ok. We conclude by Lemma B.3.4 (weakening).

□

Lemma B.4.9 (types are ok provided their hashes are) $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ iff $\mathrm{fv}\, T \subseteq \mathrm{dom}\, E$ and $E \vdash^H_{hm}$ ok and all the hashes in T are ok.

Proof. Through (Tsub.Le) and (TK.Le), we have $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ iff $E \vdash^H_{hm} T <: \top$. We will use freely this equivalence in the following.

Suppose that $\mathrm{fv}\, T \subseteq E$ and all the hashes in T are ok. We prove that $E \vdash^H_{hm} T <: \top$ by induction on the syntax of T.

Case $T = \text{UNIT}$ or $T = \text{STRING}$ or $T = \top$: Trivial by (Tsub.unit) or (Tsub.dyn) or (Tsub.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, $E \vdash^H_{hm} T_i <: \top$ for $1 \leqslant i \leqslant j$. By (Tsub.tuple), we have $E \vdash^H_{hm} T_1 * ... * T_j <: \top$.

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, $E \vdash^H_{hm} T_i <: \top$ for $1 \leqslant i \leqslant j$. By (Tsub.rec), we have $E \vdash^H_{hm} \{l_1{:}T_1; ...; l_j{:}T_j\} <: \top$.

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, $E \vdash^H_{hm} T_i$ for $1 \leqslant i \leqslant 2$. By (Tsub.fun), we have $E \vdash^H_{hm} T_1 \to T_2 <: \top$.

Case T is a hash h.TYPE : The hash h is ok by assumption, so we get $E \vdash^H_{hm} h.\text{TYPE} <: \top$ 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 $E \vdash^H_{hm} X{:}K$. Then we just apply Lemma B.4.8 (bindings in an ok environment are ok) to get $E \vdash^H_{hm} K$ ok. By Lemma B.4.4 (kinds are smaller than top), we have $E \vdash^H_{hm} K <: \mathbf{Le}(\top)$, whence by (TK.sub) $E \vdash^H_{hm} X{:}\mathbf{Le}(\top)$ as desired.

Case there exists U such that $T = U.\text{TYPE}$: Since $\{U\} = \text{fv}\, T \subseteq \text{dom}\, E$, 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 $E \vdash^H_{hm} U.\text{TYPE}{:}K$. Then we just apply Lemma B.4.8 (bindings in an ok environment are ok) to get $E \vdash^H_{hm} [X{:}K, T']$ ok. By reversing (Sok), we have $E, X{:}K \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$. By Lemma B.1.7 (environments and subhashes have to be ok), we have $E, X{:}K \vdash^H_{hm}$ ok. By reversing (envok.X), we get $E \vdash^H_{hm} K$ ok. By Lemma B.4.4 (kinds are smaller than top), we have $E \vdash^H_{hm} K <: \mathbf{Le}(\top)$, whence by (TK.sub) and (Tsub.Le) $E \vdash^H_{hm} U.\text{TYPE} <: \top$ as desired.

Now suppose $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. Then $\text{fv}\, T \subseteq \text{dom}\, E$ by Lemma B.1.11 (free variables of a judgement come from the environment). Also all the hashes in T are ok by Lemma B.1.6 (hashes have to be ok). □

Lemma B.4.10 (colour and subhash change preserves type okedness)
If $\mathbf{nil} \vdash^{H_0}_{hm_0} T{:}\mathbf{Le}(\top)$ and $\mathbf{nil} \vdash^{H_1}_{hm_1}$ ok then $\mathbf{nil} \vdash^{H_1}_{hm_1} T{:}\mathbf{Le}(\top)$.

Proof. Trivial application of Lemma B.4.9 (types are ok provided their hashes are). □

Lemma B.4.11 (colour change preserves type okedness)
If $E \vdash^H_{hm_0} T{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm_1}$ ok then $E \vdash^H_{hm_1} T{:}\mathbf{Le}(\top)$.

Proof. Since $E \vdash^H_{hm_0}$ ok and $\vdash hm_1$ ok implies $E \vdash^H_{hm_1}$ ok, this is a trivial application of Lemma B.4.9 (types are ok provided their hashes are). □

Lemma B.4.12 (relating type-is-kind and subkinding) If $E \vdash^H_{hm} T{:}K$ then $E \vdash^H_{hm} \mathbf{Eq}(T) <: K$.

Proof. If there exists T' such that $K = \mathbf{Le}(T')$, then by applying (Tsub.Le) we get $E \vdash^H_{hm} T <: T'$. From Lemma B.1.6 (hashes have to be ok) and Lemma B.1.7 (environments and subhashes have to be ok) and Lemma B.4.9 (types are ok provided their hashes are), we know that $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$. Then we use (Ksub.Eq) and (Ksub.Le) and (Ksub.tran) to get the desired $E \vdash^H_{hm} \mathbf{Eq}(T) <: \mathbf{Le}(T')$.

If there exists T' such that $K = \mathbf{Eq}(T')$. Then we have a proof of $E \vdash^H_{hm} T == T'$ by (Teq.Eq). We conclude by (Keq.Eq) and (Ksub.refl). □

Lemma B.4.13 (type equivalence is a congruence)
If $E \vdash^H_{hm} T' == T''$ and $E \vdash^H_{hm} T' <: T'''$ and $E \vdash^H_{hm} T'' <: T'''$ and $E, X{:}\mathbf{Le}(T''') \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ then $E \vdash^H_{hm} \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$.

In particular, by using Lemma B.1.6 (hashes have to be ok) and Lemma B.4.9 (types are ok provided their hashes are), if $E \vdash^H_{hm} T' == T''$ and $E, X{:}\mathbf{Le}(\top) \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ then $E \vdash^H_{hm} \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$.

Proof. Induct on the structure of T.

Case $T = \text{UNIT}$ or $T = \text{STRING}$ or $T = U.\text{TYPE}$ or $T = Y \neq X$ or $T = h.\text{TYPE}$: Then $X \notin \text{fv}\, T$ (if $T = h.\text{TYPE}$, this is because $\text{fv}\, T = \varnothing$ by Lemma B.1.6 (hashes have to be ok) and Lemma B.1.11 (free variables of a judgement come from the environment)). By Lemma B.4.9 (types are ok provided their hashes are), the hashes of T are ok and $\text{fv}\, T \subseteq \text{dom}\, E \cup \{X\}$. By Lemma B.4.9 (types are ok provided their hashes are) in the other direction, since $\text{fv}\, T \subseteq E$, we have $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. By (Teq.refl), we get $E \vdash^H_{hm} T == T$ which is the desired result.

Case $T = X$: We have $E \vdash^H_{hm} T' == T''$ as desired.

Case $T = T_1 \to T_2$ **:** By induction, we have $E \vdash^H_{hm} \{X \leftarrow T'\}T_i == \{X \leftarrow T''\}T_i$ for $i = 1, 2$. By (Teq.cong.fun), we get $E \vdash^H_{hm} \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$ as desired.

Case $T = T_1 * ... * T_j$ **:** By induction, we have $E \vdash^H_{hm} \{X \leftarrow T'\}T_i == \{X \leftarrow T''\}T_i$ for $i = 1, ..., j$. By (Teq.cong.tuple), we get $E \vdash^H_{hm} \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$ as desired.

Case $T = \{l_1 : T_1, ..., l_j : T_j\}$ **:** By induction, we have $E \vdash^H_{hm} \{X \leftarrow T'\}T_i == \{X \leftarrow T''\}T_i$ for $i = 1, ..., j$. By (Teq.cong.rec), we get $E \vdash^H_{hm} \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$ as desired.

□

Lemma B.4.14 (type substitution in equivalence)
If $E \vdash^H_{hm} \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$ and $E = E_0, X{:}\mathbf{Eq}(T_0), E_1$ then $E \vdash^H_{hm} \mathrm{ok}\, T == T'$.

Proof. By (TK.var) and (Teq.Eq), we have $E \vdash^H_{hm} X == T_0$. Let Y be a fresh variable, i.e. $Y \notin \mathrm{dom}\, E$. By Lemma B.1.7 (environments and subhashes have to be ok) and (Kok.Le) and (envok.X), we have $E, Y{:}\mathbf{Le}(\top) \vdash^H_{hm}$ ok. By Lemma B.4.9 (types are ok provided their hashes are) applied once in each direction, we get that $E, Y{:}\mathbf{Le}(\top) \vdash^H_{hm} \{X \leftarrow Y\}T{:}\mathbf{Le}(\top)$. Then, by Lemma B.4.13 (type equivalence is a congruence), we get $E \vdash^H_{hm} \{Y \leftarrow X\}\{X \leftarrow Y\}T == \{Y \leftarrow T_0\}\{X \leftarrow Y\}T$, i.e. $E \vdash^H_{hm} T == \{X \leftarrow T_0\}T$.

Similarly, we have $E \vdash^H_{hm} T' == \{X \leftarrow T_0\}T'$. By (Teq.sym), we get $E \vdash^H_{hm} \{X \leftarrow T_0\}T' == T'$.

Finally, by two applications of (Teq.tran), we get $E \vdash^H_{hm} T == T'$ as desired. □

B.5 Type preservation by substitution

Definition B.5.1 (unresolved free variables of an environment) The unresolved free variables of an environment, written $\mathrm{ufv}\, E$, are defined as follows :

$$\mathrm{ufv}\, \mathbf{nil} = \varnothing$$
$$\mathrm{ufv}\, (\zeta{:}\tau, E) = (\mathrm{ufv}\, (E) \setminus \{\zeta\}) \cup \mathrm{fv}\, \tau$$

It is immediate that $\mathrm{ufv}\, E \subseteq \mathrm{ufv}\, (E, E')$.

Lemma B.5.2 (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.5.3 (ok environments have no unresolved free variables) If $E \vdash^H_{hm}$ ok then $\mathrm{ufv}\, E = \varnothing$.

Proof. We prove by induction on the derivation size that $E \vdash^H_{hm} J$ implies $\mathrm{ufv}\, E = \varnothing$. Most rules whose conclusion is a coloured judgement $E \vdash^H_{hm} J$ 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 $\mathbf{nil} \vdash^H_{hm} J$ are trivial.

The remaining rules are (envok.{x,X,U}). If we write the conclusion as $E, \zeta{:}\tau \vdash^H_{hm}$ ok, one premise is $E \vdash^H_{\bullet} \tau\, \mathrm{ok}$. We have $\mathrm{ufv}\, E = \varnothing$ by induction. By Lemma B.1.11 (free variables of a judgement come from the environment), we have $\mathrm{fv}\, \tau \subseteq \mathrm{dom}\, E$. By Lemma B.5.2 (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.5.4 (type preservation by substitution) If $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$ by a proof Π such that $hm' \preccurlyeq \min\left(\mathrm{pvu}_\zeta\,(\Pi)\right)$ and $E_0 \vdash^{H_0}_{hm'} \eta{:}\tau$ with $H_0 \subseteq H$ then $E_0, \sigma E \vdash^H_{hm} \sigma J$ where $\sigma = \{\zeta \leftarrow \eta\}$ and $\zeta{:}\tau$ is an expression or type binding.

Proof. Induct on the derivation Π of $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$.

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 $\hat{E}$ 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 $\hat{E}$ in the leftmost position, and in no other place.
   (b) The premise is $\hat{\zeta} \notin \mathrm{dom}\,\hat{E}$ for some $\hat{\zeta}$.
   (c) The premise is a judgement with an empty environment and does not mention $\hat{E}$.

Suppose that $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$ 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 $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$ except where shown ; furthermore ζ is not in a binding position in any premise either, except in instances of $\hat{E}$ 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 $\hat{E}$ 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 $\hat{E}$ 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 $\hat{\zeta} \notin \mathrm{dom}\,\hat{E}$. Let us write ζ' for the instantiation of $\hat{\zeta}$. Then $\zeta' \neq \zeta$ as $\zeta' \notin \mathrm{dom}\,(E_0, \zeta{:}\tau, E'')$ is derivable.

Note furthermore that well-typed values are stable by substitution, as per Lemma B.2.7 (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 $\sigma E_0, \sigma E'' \vdash^H_{hm} \sigma J$, which is also $E_0, \sigma E'' \vdash^H_{hm} J$ as $\zeta \notin \mathrm{fv}\,E_0$ by Lemma B.1.7 (environments and subhashes have to be ok), Lemma B.1.8 (prefixes of ok environments are ok) and Lemma B.5.3 (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 $E_0, \zeta{:}\tau, E'', E'_i \vdash^H_{hm_i} J_i$. Given Lemma B.2.5 (monotonicity of pvu), we can apply induction, getting $E_0, \sigma E'', \sigma E'_i \vdash^H_{hm_i} \sigma J_i$, which is the corresponding premise in ω (recall that $\sigma E_0 = E_0$).

[1] This is needed for (MS.struct), which requires a value in one place.

Case 2b : The premise is of the form $\zeta' \notin \mathrm{dom}\,(E_0, \zeta{:}\tau, E'')$, and ζ' is not ζ (see (†) above). We need to prove that $\zeta' \notin \mathrm{dom}\,(E_0, \sigma E'')$. This follows easily, given that $\mathrm{dom}\,(E_0, \sigma E'') = \mathrm{dom}\,(E_0, E'') \subseteq \mathrm{dom}\,(E_0, \zeta{:}\tau, E'')$.

Case 2c : The premise is a judgement AJ with an empty environment, so by Lemma B.1.11 (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 $\hat{E}$, so it is in fact exactly the premise needed in ω.

As all the premises of ω are derivable, its conclusion holds. It reads : $E_0, \sigma E'', \sigma E' \vdash^H_{hm} \sigma J$, which is what we set out to prove.

Special cases : The instance was obtained by instantiating the metavariable $\hat{E}$ 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,X}) : Then $E = \mathbf{nil}$ and $E_2 = \mathbf{nil}$. The proof obligation is $E_1 \vdash^H_{hm}$ ok, i.e. $E_0 \vdash^{H_0}_{hm}$ ok, which holds by Lemma B.1.8 (prefixes of ok environments are ok).

Cases (eT.var), (TK.var) : α is of the form

$$\frac{E_0, \zeta{:}\tau, E' \vdash^H_{hm} \mathrm{ok}}{E_0, \zeta{:}\tau, E' \vdash^H_{hm} \zeta{:}\tau}$$

and $\sigma = \{\zeta{\leftarrow}\eta\}$, and we have $E_0 \vdash^{H_0}_{hm'} \eta{:}\tau$ with $hm' \preccurlyeq \min\big(\mathrm{pvu}_\zeta(\Pi)\big) \preccurlyeq hm$. By induction (which we can apply thanks to Lemma B.2.5 (monotonicity of pvu)), we have $E_0, \sigma E' \vdash^H_{hm}$ ok.

Furthermore, since $\vdash hm\,\mathrm{ok}$ by Lemma B.1.5 (colours have to be ok), $E_0 \vdash^{H_0}_{hm} \eta{:}\tau$ by Lemma B.3.2 (colour stripping judgements). By Lemma B.3.4 (weakening) we get $E_0, \sigma E' \vdash^H_{hm} \eta{:}\tau$ as desired.

Every remaining rule is inapplicable because the environment in the conclusion must be empty. □

Lemma B.5.5 (strengthening) If $E_0, \zeta{:}\tau, E \vdash^H_{hm} J$ and $\zeta \notin \mathrm{fv}\,E \cup \mathrm{fv}\,J$ and $\zeta{:}\tau$ is a type or expression variable binding then $E_0, E \vdash^H_{hm} J$.

Proof. Since $\zeta{:}\tau$ is a type or expression variable binding, we know that $\zeta \notin \mathrm{dom}\,H$. By Lemma B.1.7 (environments and subhashes have to be ok) and Lemma B.1.8 (prefixes of ok environments are ok), we have $E_0, \zeta{:}\tau \vdash^{H'}_{hm}$ ok with $H' \subseteq H$. By Lemma (environments and subhashes are ok in the empty colour), we get $E_0, \zeta{:}\tau \vdash^{H'}_{\bullet}$ ok. Let us then reverse the rule (envok.{x,X}) that was applied to obtain this latter judgement :

Case $\zeta{:}\tau$ is $X{:}K$: Then we have $E_0 \vdash^{H'}_{\bullet} K\,\mathrm{ok}$. If 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 $E_0 \vdash^{H'}_{\bullet} \eta{:}\mathbf{Le}(\top)$. By (Teq.refl) and (TK.Eq) or (Tsub.Equi) and (TK.Le), we get $E_0 \vdash^{H'}_{\bullet} \eta{:}K$.

Case $\zeta{:}\tau$ is $x{:}T$: Then we have $E_0 \vdash^{H'}_{\bullet} T{:}\mathbf{Le}(\top)$. Let η be $\mathbf{Unmarfailure}^T$. By (eT.Undynfailure), we have $E_0 \vdash^{H'}_{\bullet} \eta{:}T$.

In any case, we have $E_0 \vdash^{H'}_{\bullet} \eta{:}\tau$. By Lemma B.5.4 (type preservation by substitution), we have $E_0, \sigma E \vdash^H_{hm} \sigma J$ where $\sigma = \{\zeta{\leftarrow}\eta\}$. However, by assummption, $\zeta \notin \mathrm{fv}\,E \cup \mathrm{fv}\,J$. Hence $\sigma E = E$ and $\sigma J = J$, so we get $E_0, E \vdash^H_{hm} J$ as desired. □

Lemma B.5.6 (weakening kind to ok kind in the environment) If $E_0 \vdash^H_{hm} K <: K'$ and $E_0 \vdash^H_{hm} K$ ok and $E_0, X{:}K', E_1 \vdash^{H'}_{hm} J$ and $H \subseteq H'$ and J is a type world judgement right-hand side then $E_0, X{:}K, E_1 \vdash^{H'}_{hm} J$.

Note that the hypothesis $E_0 \vdash^H_{hm} K$ ok is in fact superfluous (see Lemma (weakening kind in the environment) below).

Proof. Since $E_0 \vdash^H_{hm} K$ ok, by (envok.X), $E_0, Y{:}K \vdash^H_{hm}$ ok where Y is fresh. By Lemma B.3.4 (weakening), from $E_0, Y{:}K \vdash^H_{hm}$ ok and $E_0 \vdash^H_{hm} K'$ ok, we get $E_0, Y{:}K \vdash^H_{hm} K'$ ok. By (envok.X), $E_0, Y{:}K, X{:}K' \vdash^H_{hm}$ ok.

By Lemma B.3.7 (environment and subhash weakening), $E_0, Y{:}K, X{:}K', E_1 \vdash^{H'}_{hm} J$.

From $E_0, Y{:}K \vdash^H_{hm}$ ok, by (TK.var), we get $E_0, Y{:}K \vdash^H_{hm} Y{:}K$. By Lemma B.3.4 (weakening), we also get $E_0, Y{:}K \vdash^H_{hm} K <: K'$. By (TK.sub), we get $E_0, Y{:}K \vdash^H_{hm} Y{:}K'$. By Lemma B.5.4 (type preservation by substitution), using Lemma B.2.8 (computing the pvu of a type world judgement) $E_0, Y{:}K', \{X \leftarrow Y\}E_1 \vdash^{H'}_{hm} \{X \leftarrow Y\}J$. By alpha-conversion, we have $E_0, X{:}K', E_1 \vdash^{H'}_{hm} J$ as desired. □

Lemma B.5.7 (things have to be ok)
If $E \vdash^H_{hm} T{:}K$ then $E \vdash^H_{hm} K$ ok.
If $E \vdash^H_{hm} T == T'$ or $E \vdash^H_{hm} T <: T'$ then $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$.
If $E \vdash^H_{hm} K == K'$ or $E \vdash^H_{hm} K <: K'$ then $E \vdash^H_{hm} K$ ok and $E \vdash^H_{hm} K'$ ok.
If $E \vdash^H_{hm} S == S'$ or $E \vdash^H_{hm} S <: S'$ then $E \vdash^H_{hm} S$ ok and $E \vdash^H_{hm} S'$ ok.
If $E \vdash^H_{hm} e{:}T$ or $E \vdash^H_{hm} m{:}T$ then $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$.
If $E \vdash^H_{hm} M{:}S$ or $E \vdash^H_{hm} U{:}S$ then $E \vdash^H_{hm} S$ 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 $E \vdash^H_{hm} \mathbf{Le}(T) == \mathbf{Le}(T')$. The premise is $E \vdash^H_{hm} T == T'$. By induction we get $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, whence by (Kok.Le), $E \vdash^H_{hm} \mathbf{Le}(T)$ ok and $E \vdash^H_{hm} \mathbf{Le}(T')$ ok.

Case (Keq.Eq) : The conclusion is $E \vdash^H_{hm} \mathbf{Eq}(T) == \mathbf{Eq}(T')$. The premise is $E \vdash^H_{hm} T == T'$. By induction we get $E \vdash^H_{hm} T{:}\mathbf{Type}$ and $E \vdash^H_{hm} T'{:}\mathbf{Type}$, whence by (Kok.Eq), $E \vdash^H_{hm} \mathbf{Eq}(T)$ ok and $E \vdash^H_{hm} \mathbf{Eq}(T')$ ok.

Case (Ksub.Eq) : The conclusion is $E \vdash^H_{hm} \mathbf{Eq}(T) <: \mathbf{Le}(T)$. The premise is $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. From this, by (Kok.Eq), we get $E \vdash^H_{hm} \mathbf{Eq}(T)$ ok and by (Kok.Le), we get $E \vdash^H_{hm} \mathbf{Le}(T)$ ok.

Case (Ksub.Le) : The conclusion is $E \vdash^H_{hm} \mathbf{Le}(T) <: \mathbf{Le}(T')$. The premise is $E \vdash^H_{hm} T <: T'$. By induction we know that $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, whence by (Kok.Le), $E \vdash^H_{hm} \mathbf{Le}(T)$ ok and $E \vdash^H_{hm} \mathbf{Le}(T')$ ok.

Cases (Ksub.refl), (Ksub.tran) : Trivial by induction.

Case (TK.sub) : The conclusion is $E \vdash^H_{hm} T{:}K'$ and one of the premises is $E \vdash^H_{hm} K <: K'$. By induction, we get $E \vdash^H_{hm} K'$ ok.

Case (TK.Eq) : The conclusion is $E \vdash^H_{hm} T{:}\mathbf{Eq}(T')$. The premise is $E \vdash^H_{hm} T == T'$. By induction we have $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, hence $E \vdash^H_{hm} \mathbf{Eq}(T')$ ok by (Kok.Eq).

Case (TK.Le) : The conclusion is $E \vdash^H_{hm} T{:}\mathbf{Le}(T')$. The premise is $E \vdash^H_{hm} T <: T'$. By induction we have $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, hence $E \vdash^H_{hm} \mathbf{Le}(T')$ ok by (Kok.Le).

Case (TK.var) : The conclusion is of the form $E, X{:}K, E' \vdash^H_{hm} X{:}K$. The premise is $E, X{:}K, E' \vdash^H_{hm}$ ok. By Lemma B.4.8 (bindings in an ok environment are ok), we get $E, X{:}K, E' \vdash^H_{hm} K$ ok as desired.

Case (TK.mod) : The conclusion is $E \vdash^H_{hm} U.\text{TYPE}{:}K$. The premise is $E \vdash^H_{hm} U{:}[X{:}K, T]$. By Lemma B.1.11 (free variables of a judgement come from the environment), we know that $\operatorname{dom} H \subseteq \operatorname{dom} E$. By induction we have $E \vdash^H_{hm} [X{:}K, T]$ ok. This must have been obtained by applying (Sok), with the premise $E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)$, and by α-conversion, we know that $X \notin \operatorname{dom} H$. By Lemma B.1.7 (environments and subhashes have to be ok), we have $E, X{:}K \vdash^H_{hm}$ ok that is smaller. Hence, by reversing (envok.X), we get a proof of $E \vdash^H_{hm} K$ ok.

Case (TK.hash) : The conclusion is $E \vdash^H_{hm} h.\text{TYPE}{:}K$ where $h = \mathbf{hash}(N, H', [T, v^{hm}]{:}[X{:}K, T'])$. One of the premises is $\vdash \mathbf{hash}(N, H', [T, v^{hm}]{:}[X{:}K, T'])$ ok. By reversing (hok.hash) we have $\mathbf{nil} \vdash^{H'}_{\bullet} [T, v^{hm}]{:}[X{:}K, T']$. Note that by Lemma B.1.11 (free variables of a judgement come from the environment), we know that $\operatorname{dom} H = \varnothing$. If the last rule is (MS.struct) we know that $X{:}K \vdash^{H'}_{\bullet} T'{:}\mathbf{Le}(\top)$. By Lemma B.1.7 (environments and subhashes have to be ok), $X{:}K \vdash^{H'}_{\bullet}$ ok. If the last rule is (MS.sub), one of the premises is $\mathbf{nil} \vdash^{H'}_{\bullet} S <: [X{:}K, T']$. By a trivial induction on the number of (Ssub.tran) and the application of the Lemma B.1.7 (environments and subhashes have to be ok) to one of the premises of (Sok) or (Ssub.struct), we get $X{:}K \vdash^{H'}_{\bullet}$ ok. In all cases we reverse (envok.X) to get $\mathbf{nil} \vdash^{H'}_{\bullet} K$ ok. By reversing (Kok.Le) or (Kok.Eq), and by the Lemma B.4.10 (colour and subhash change preserves type okedness) followed by another instance of (Kok.Le) or (Kok.Eq), we get $\mathbf{nil} \vdash^H_{hm} K$ ok. We conclude by Lemma B.3.4 (weakening).

Case (Teq.Eq) : The conclusion is $E \vdash^H_{hm} T == T'$. The premise is $E \vdash^H_{hm} T{:}\mathbf{Eq}(T')$. By induction we have $E \vdash^H_{hm} \mathbf{Eq}(T')$ ok, which must have been derived by (Kok.Eq) from $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$. From this, (Tsub.Le) and (Ksub.Le), we get $E \vdash^H_{hm} \mathbf{Le}(T') <: \mathbf{Le}(\top)$. On the other side we can use (Ksub.Eq) to get $E \vdash^H_{hm} \mathbf{Eq}(T') <: \mathbf{Le}(T')$. By (Ksub.tran) and (TK.sub), we have $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. Note that this may well be the shortest way to obtain $E \vdash^H_{hm} T{:}\mathbf{Type}$ (take $T = U.\text{TYPE}$).

Case (Teq.hash) : The conclusion is $E \vdash^H_{hm} h.\text{TYPE} == T$, and $h = \mathbf{hash}(N, H', [T, v^{hm_1}]{:}S) \in hm$. The premise is $E \vdash^H_{hm}$ ok. By Lemma B.1.6 (hashes have to be ok), we get $\vdash \mathbf{hash}(N, H', [T, v^{hm_1}]{:}S)$ ok. Note that by Lemma B.1.11 (free variables of a judgement come from the environment), we know that $\operatorname{dom} H' = \varnothing$. By reversing (hok.hash) we have $\mathbf{nil} \vdash^{H'}_{\bullet} [T, v^{hm_1}]{:}S$. By a trivial induction on the number of (MS.sub), we can reverse (MS.struct) to get $\mathbf{nil} \vdash^{H'}_{\bullet} T{:}\mathbf{Le}(\top)$. By the Lemma B.4.10 (colour and subhash change preserves type okedness), we have $\mathbf{nil} \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. We conclude by Lemma B.3.4 (weakening). We also have $E \vdash^H_{hm} h.\text{TYPE} <: \top$ by (Tsub.hash) using Lemma B.1.7 (environments and subhashes have to be ok). Then we know that $E \vdash^H_{hm} h.\text{TYPE}{:}\mathbf{Le}(\top)$ by (TK.Le).

Case (Teq.refl) : Trivial.

Cases (Teq.sym), (Teq.tran) : Trivial by induction.

Case (Teq.cong.fun) : The conclusion is $E \vdash^H_{hm} T_0 \to T_1 == T'_0 \to T'_1$. By induction on the premises, we get $E \vdash^H_{hm} T_j{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'_j{:}\mathbf{Le}(\top)$ for $j = 0, 1$. By (Tsub.Le) and (Tsub.fun), we get $E \vdash^H_{hm} T_0 \to T_1 <: \top$ and $E \vdash^H_{hm} T'_0 \to T'_1 <: \top$. We conclude by (TK.Le).

Case (Teq.cong.tuple) and (Teq.cong.rec) : Similar to case (Teq.cong.fun).

Case (Teq.cong.rec.perm) : Trivial by using the premises with (Tsub.rec).

Case (Tsub.Le) : The conclusion is $E \vdash^H_{hm} T <: T'$. The premise is $E \vdash^H_{hm} T{:}\mathbf{Le}(T')$. By induction, we have $E \vdash^H_{hm} \mathbf{Le}(T')$ ok, which must have been derived by (Kok.Le) from $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$.

From this, (Tsub.Le) and (Ksub.Le), we get $E \vdash^H_{hm} \mathbf{Le}(T') <: \mathbf{Le}(\top)$. Then we use (TK.sub) to get $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$.

Case (Tsub.Equi) : Trivial by induction.

Case (Tsub.Subhash) : The conclusion is $E \vdash^H_{hm} \kappa.\text{TYPE} <: \kappa'.\text{TYPE}$ with $\kappa <: \kappa' \in H$. The premise is $E \vdash^H_{hm}$ ok. If κ or κ' is a hash, we use the Lemma B.1.6 (hashes have to be ok) and (Tsub.hash) and (TK.Le). If κ or κ' is a module variable U, then by Lemma B.1.11 (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 $E \vdash^H_{hm} U{:}S$. By Lemma B.4.7 (components of modules are ok) we know that $E \vdash^H_{hm} K$ ok, and also by (TK.mod) we get $E \vdash^H_{hm} U.\text{TYPE}{:}K$. We conclude with Lemma B.4.4 (kinds are smaller than top) and (TK.sub).

Case (Tsub.cong.record.width) : Trivial.

Case (Tsub.cong.fun) : The conclusion is $E \vdash^H_{hm} T_0 \to T_1 <: T'_0 \to T'_1$. By induction on the premises, we get $E \vdash^H_{hm} T_j{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'_j{:}\mathbf{Le}(\top)$ for $j = 0, 1$. By (Tsub.Le) and (Tsub.fun), we get $E \vdash^H_{hm} T_0 \to T_1 <: \top$ and $E \vdash^H_{hm} T'_0 \to T'_1 <: \top$. 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 $E \vdash^H_{hm} T <: \top$ for some T. By Lemma B.1.7 (environments and subhashes have to be ok), $E \vdash^H_{hm}$ ok, and we apply (Tsub.top) and (TK.Le) to get $E \vdash^H_{hm} \top{:}\mathbf{Le}(\top)$. On the other side we just have to apply (TK.Le) to have $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$.

Case (Seq.struct) : The conclusion is $E \vdash^H_{hm} [X{:}K, T] == [X{:}K', T']$. One premise is $E, X{:}K \vdash^H_{hm} T == T'$. By induction, we get $E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ and $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, 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 $E \vdash^H_{hm} {}_{(T <: T')} e{:}T'$. One of the premises is $E \vdash^H_{hm} T <: T'$. By induction we have $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$.

Case (eT.eq) : The conclusion is $E \vdash^H_{hm} e{:}T'$. One premise is $E \vdash^H_{hm} T == T'$. By induction we get $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$.

Case (eT.var) : Similar to case (TK.var).

Case (eT.mod) : The conclusion is $E \vdash^H_{hm} U.\text{term}{:}T$. One premise is $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$.

Case (eT.ap) : The conclusion is $E \vdash^H_{hm} e\ e'{:}T'$. One premise is $E \vdash^H_{hm} e'{:}T \to T'$. By induction we get $E \vdash^H_{hm} T \to T'{:}\mathbf{Le}(\top)$. By Lemma B.4.9 (types are ok provided their hashes are) applied to $E \vdash^H_{hm} T \to T'{:}\mathbf{Le}(\top)$ then to get the desired $E \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$.

Case (eT.fun) : The conclusion is $E \vdash^H_{hm} \lambda x{:}T.e{:}T \to T'$. The premise is $E, x{:}T \vdash^H_{hm} e{:}T'$. By induction, we get $E, x{:}T \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$ by a proof Π. By Lemma B.2.10 (types do not contain free expression variables), $x \notin \text{fv}\ T'$. By Lemma B.1.7 (environments and subhashes have to be ok), we have $E \vdash^H_{hm}$ ok. By Lemma B.4.9 (types are ok provided their hashes are) applied once in each direction, we get first that the hashes in T' are ok and $\text{fv}\ T' \subseteq \text{dom}\, E \cup \{x\}$ (hence $\text{fv}\ T' \subseteq \text{dom}\, E$), then that $E \vdash^H_{hm} T'$ ok. Also, by Lemma B.1.7 (environments and subhashes have to be ok), we get $E, x{:}T \vdash^H_{hm}$ ok, whence $E \vdash^H_{hm} T{:}\mathbf{Le}(\top)$ by reversing (envok.x). By (Tsub.Le), (Tsub.fun) and (TK.Le), we get $E \vdash^H_{hm} T \to T'{:}\mathbf{Le}(\top)$.

Cases (eT.tuple), (eT.record) : The conclusion is $E \vdash^H_{hm} (e_1, ..., e_j) : T_1 * ... * T_j$ and $E \vdash^H_{hm} \{l_1 = e_1, .., l_j = e_j\} : \{l_1 : T_1, ..., l_j : T_j\}$. The premises are $E \vdash^H_{hm} e_i : T_i$ for $1 \leqslant i \leqslant j$. By induction, we have $E \vdash^H_{hm} T_i : \mathbf{Le}(\top)$ for all i, whence by (Tsub.Le), (Tsub.tuple) or (Tsub.rec), and (TK.Le) : $E \vdash^H_{hm} T_1 * ... * T_j : \mathbf{Le}(\top)$ and $E \vdash^H_{hm} \{l_1 : T_1, ..., l_j : T_j\} : \mathbf{Le}(\top)$.

Cases (eT.proj), (eT.field) : Similar to (eT.ap).

Cases (eT.send), (eT.recv), (eT.mar), (eT.marred), (eT.unit) : By Lemma B.1.7 (environments and subhashes have to be ok), we have $E \vdash^H_{hm}$ 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 $E \vdash^H_{hm} M : [X : K, T']$. One premise is $E, X : K \vdash^H_{hm} T' : \mathbf{Type}$, whence by (Sok) : $E \vdash^H_{hm} [X : K, T']$ ok.

Case (MS.sub) : Trivial by induction.

Case (US.var) : Similar to case (TK.var).

Case (US.self) : The conclusion is $E \vdash^H_{hm} U : [X : \mathbf{Eq}(U.\text{TYPE}), T]$. The premise is $E \vdash^H_{hm} U : [X : K, T]$. By (TK.mod), we have $E \vdash^H_{hm} U.\text{TYPE} : K$. And by induction we get $E \vdash^H_{hm} [X : K, T]$ ok, whence by reversing (Sok) : $E, X : K \vdash^H_{hm} T : \mathbf{Le}(\top)$.

By Lemma B.1.7 (environments and subhashes have to be ok) and reversing (envok.X), we have $E \vdash^H_{hm} K$ ok. By Lemma B.4.4 (kinds are smaller than top), we get $E \vdash^H_{hm} K <: \mathbf{Le}(\top)$. By (TK.sub), given that $E \vdash^H_{hm} U.\text{TYPE} : K$, we have $E \vdash^H_{hm} U.\text{TYPE} : \mathbf{Le}(\top)$, whence by (Kok.Eq) : $E \vdash^H_{hm} \mathbf{Eq}(U.\text{TYPE})$ ok.

If there exists a type T such that $K = \mathbf{Le}(T)$, then we have $E \vdash^H_{hm} \mathbf{Le}(U.\text{TYPE}) <: \mathbf{Le}(T)$ by (Tsub.Le) and (Ksub.Le). With (Ksub.Eq), and (Ksub.tran), we get $E \vdash^H_{hm} \mathbf{Eq}(U.\text{TYPE}) <: \mathbf{Le}(T)$. Otherwise there exists T such that $K = \mathbf{Eq}(T)$. From $E \vdash^H_{hm} U.\text{TYPE} : \mathbf{Eq}(T)$, by (Teq.Eq), (Keq.Eq) and (Ksub.refl), we get $E \vdash^H_{hm} \mathbf{Eq}(U.\text{TYPE}) <: \mathbf{Eq}(T)$. In either case we have $E \vdash^H_{hm} \mathbf{Eq}(U.\text{TYPE}) <: K$.

By Lemma B.5.6 (weakening kind to ok kind in the environment), $E, X : \mathbf{Eq}(U.\text{TYPE}) \vdash^H_{hm} T : \mathbf{Le}(\top)$. Hence by (Sok) we have $E \vdash^H_{hm} [X : \mathbf{Eq}(U.\text{TYPE}), T]$ ok.

Cases (mT.expr), (mT.letext) : Trivial by induction.

□

Lemma B.5.8 (weakening kind in the environment)
If $E_0 \vdash^{H_0}_{hm} K <: K'$ and $E_0, X : K', E_1 \vdash^{H_1}_{hm} J$ and $H_0 \subseteq H_1$ and J is a type world judgement right-hand side then $E_0, X : K, E_1 \vdash^{H_1}_{hm} J$.

Proof. By Lemma B.5.7 (things have to be ok), $E_0 \vdash^{H_0}_{hm} K$ ok. By Lemma B.5.6 (weakening kind to ok kind in the environment), we get the desired result. □

Lemma B.5.9 (type preservation by guarded expression variable substitution)
If $E_0, x : T, E \vdash^H_{hm} J$ and $E_0 \vdash^{H_0}_{hm'} e : T$ and $E_0 \vdash^{H_0}_{\bullet}$ ok then $E_0, \sigma E \vdash^H_{hm} \sigma J$ where $\sigma = \{x \leftarrow [e]^T_{hm'}\}$.

Proof. $E_0 \vdash^{H_0}_{hm'} T : \mathbf{Le}(\top)$ by Lemma B.5.7 (things have to be ok). By Lemma B.4.9 (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 $E_0 \vdash^{H_0}_{hm'} e : T$ yields $E_0 \vdash^{H_0}_{\bullet} [e]^T_{hm'} : T$. We can now apply Lemma B.5.4 (type preservation by substitution) to get the desired result. □

Lemma B.5.10 (reversing subsignaturing judgement)
If $E \vdash^H_{hm} [X{:}K, T] <: [X{:}K', T']$ then $E \vdash^H_{hm} K <: K'$ and $E, X{:}K \vdash^H_{hm} T <: T'$.

Proof. Induct on the derivation of $E \vdash^H_{hm} [X{:}K, T] <: [X{:}K', T']$.

Case (Ssub.refl) : The premise is $E \vdash^H_{hm} [X{:}K, T]$ ok, which must have been derived by (Sok) from $E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. By (Teq.refl) and (Tsub.Equi), we have $E, X{:}K \vdash^H_{hm} T <: T$. By Lemma B.1.7 (environments and subhashes have to be ok), we have $E, X{:}K \vdash^H_{hm}$ ok. By reversing (envok.X), we get $E \vdash^H_{hm} K$ ok whence $E \vdash^H_{hm} K <: K$ by Lemma B.4.1 (reflexivity of kind equivalence) and (Ksub.refl).

Case (Ssub.tran) : The premises are $E \vdash^H_{hm} [X{:}K, T] <: [X{:}K'', T'']$ and $E \vdash^H_{hm} [X{:}K'', T''] <: [X{:}K', T']$. By induction twice, we have : $E \vdash^H_{hm} K <: K''$, $E, X{:}K \vdash^H_{hm} T <: T''$, $E \vdash^H_{hm} K'' <: K'$ and $E, X{:}K'' \vdash^H_{hm} T'' <: T'$. By (Ksub.tran), we have $E \vdash^H_{hm} K <: K'$. By Lemma B.5.8 (weakening kind in the environment) and Lemma B.4.5 (transitivity of subtyping), we get $E, X{:}K \vdash^H_{hm} T <: T'$ as desired.

Case (Ssub.struct) : The premises are the desired judgements.

□

Lemma B.5.11 (reversing module value variable typing judgement)
If $E \vdash^H_{hm} U{:}S$ then there exist E_0, E_1, K, T such that $E = E_0, U{:}[X{:}K, T], E_1$ and $E \vdash^H_{hm} [X{:}\mathbf{Eq}(U.\text{TYPE}), T] == S$.

Proof. Induct on the derivation of $E \vdash^H_{hm} U{:}S$.

Case (US.var) : Then there exist E_0, E_1, K, T such that $E = E_0, U{:}[X{:}K, T], E_1$ and $S = [X{:}K, T]$. The premise is $E \vdash^H_{hm}$ ok. By (US.self) and (TK.mod), we have $E \vdash^H_{hm} U.\text{TYPE}{:}K$. By Lemma B.4.12 (relating type-is-kind and subkinding), we get $E \vdash^H_{hm} \mathbf{Eq}(U.\text{TYPE}) <: K$. By Lemma B.5.7 (things have to be ok) and reversing (Sok), we get $E, X{:}K \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. By Lemma B.4.9 (types are ok provided their hashes are) applied once in each direction, given that $E, X{:}\mathbf{Eq}(U.\text{TYPE}) \vdash^H_{hm}$ ok by Lemma B.5.7 (things have to be ok) and (envok.X), we get $E, X{:}\mathbf{Eq}(U.\text{TYPE}) \vdash^H_{hm} T{:}\mathbf{Le}(\top)$. By (Teq.refl), (Tsub.Equi) and (Ssub.struct), we get $E \vdash^H_{hm} [X{:}\mathbf{Eq}(U.\text{TYPE}), T] <: S$ as desired.

Case (US.eq) : There exists S' such that the premises are $E \vdash^H_{hm} U{:}S'$ and $E \vdash^H_{hm} S' == S$. By induction, we have $E = E_0, U{:}[X{:}K, T], E_1$ and $E \vdash^H_{hm} [X{:}\mathbf{Eq}(U.\text{TYPE}), T] == S'$. We conclude by Lemma B.4.6 (signature equivalence is transitive).

Case (US.self) : There exist K' and T' such that $S = [\mathbf{Eq}(U.\text{TYPE}), T']$ and the premise is $E \vdash^H_{hm} U{:}[X{:}K', T']$. By induction we have $E = E_0, U{:}[X{:}K, T], E_1$ and $E \vdash^H_{hm} [X{:}\mathbf{Eq}(U.\text{TYPE}), T] <: [X{:}K', T']$. By Lemma B.5.10 (reversing subsignaturing judgement), we have $E, X{:}\mathbf{Eq}(U.\text{TYPE}) \vdash^H_{hm} T == T'$. Given that we also have $E \vdash^H_{hm} \mathbf{Eq}(U.\text{TYPE}) <: \mathbf{Eq}(U.\text{TYPE})$ (by Lemma B.5.7 (things have to be ok), (Teq.refl), (Keq.Eq) and (Ksub.refl)), by (Ssub.struct), we get $E \vdash^H_{hm} [X{:}\mathbf{Eq}(U.\text{TYPE}), T] <: S$ as desired.

□

Lemma B.5.12 (type preservation by module substitution in coloured judgements)
Suppose $E_0, U(T_0){:}[X{:}K, T], E \vdash^H_{hm} J$ and z and Z are fresh and h is a hash of a module with $[X{:}K, T]$ as signature and $\sigma = \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z, U.\text{term}{\leftarrow}z\}$.

If $J = U{:}S'$ for some S' then $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma S'$.

Otherwise $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash^{\sigma H}_{hm} \sigma J$.

Proof. To do. □

Lemma B.5.13 (type world judgements do not contain free expression variables) If $E \vdash^H_{hm} J$ is a derivable type world judgement then $\text{fse}\, E \cup \text{fse}\, H \cup \text{fse}\, J$ does not contain any free expression substitutable entity.

Proof. Given Lemma B.5.7 (things have to be ok), every free substitutable entity in J is free in a type, kind or signature that is correct in E under hm. Thus, by Lemma B.2.10 (types do not contain free expression variables), no free substitutable entity in J is an expression substitutable entity. Also, by Lemma B.1.7 (environments and subhashes have to be ok), $E \vdash^H_{hm}$ ok, so by Lemma B.2.11 (environments do not contain free expression variables), E does not have any free substitutable entity. H does only includes hashes and module variables. □

Lemma B.5.14 (type preservation by module substitution in coloured judgements for type world judgements) Suppose $E_0, U(T_0){:}[X{:}K, T], E \vdash^H_{hm} J$ 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 $\sigma = \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}$. Then $E_0, Z{:}K, \sigma E \vdash^{\sigma H}_{hm} \sigma J$.

Proof. By Lemma B.5.12 (type preservation by module substitution in coloured judgements), for Z and z fresh, we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, E' \vdash^{H'}_{hm} J'$ where $E' = \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z, U.\text{term}{\leftarrow}z\}E$ and $J' = \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z, U.\text{term}{\leftarrow}z\}J$ and $H' = \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z, U.\text{term}{\leftarrow}z\}H$.

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.1.6 (hashes have to be ok), any hash in E' or J' or H' is ok, and by Lemma B.1.11 (free variables of a judgement come from the environment), none of these hashes has a free occurence of z. Therefore $z \notin \text{fv}\, E'$ and $z \notin \text{fv}\, J'$ and $z \notin \text{fv}\, H'$.

Thus we can write the judgement in question as

$E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}E \vdash^{\{U\leftarrow h, U.\text{TYPE}\leftarrow Z\}H}_{hm} \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}J$.

Given that $z \notin \text{fv}\,\{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}E \cup \text{fv}\,\{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}J \cup \text{fv}\,\{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}H$, by Lemma B.5.5 (strengthening), we get $E_0, Z{:}K, \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}E \vdash^{\{U\leftarrow h, U.\text{TYPE}\leftarrow Z\}H}_{hm} \{U{\leftarrow}h, U.\text{TYPE}{\leftarrow}Z\}J$. □

Lemma B.5.15 (simplified module and type equality substitution for type world judgements) Suppose $U(T_0){:}[X{:}\mathbf{Eq}(T), T'], E \vdash^H_{hm} J$ and $\chi = U.\text{TYPE}$, or $X{:}\mathbf{Eq}(T), E \vdash^H_{hm} J$ 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, then $\{\chi{\leftarrow}T\}E \vdash^{H'}_{hm} \{\chi{\leftarrow}T\}J$, with $H' = \{U{\leftarrow}h\}H$ if $\chi = U.\text{TYPE}$, and $H' = H$ otherwise.

Proof. We consider each case.

$U(T_0){:}[X{:}\mathbf{Eq}(T), T'], E \vdash^H_{hm} J$ **:** By Lemma B.5.14 (type preservation by module substitution in coloured judgements for type world judgements), this case reduces to the other case.

$X{:}\mathbf{Eq}(T), E \vdash^H_{hm} J$ **by some proof** Π **:** By Lemma B.1.7 (environments and subhashes have to be ok), $X{:}\mathbf{Eq}(T), E \vdash^H_{hm}$ ok. By Lemma B.1.8 (prefixes of ok environments are ok), $X{:}\mathbf{Eq}(T) \vdash^{H_0}_{hm}$ ok with $H_0 \subseteq H$. By (TK.var) and (Teq.Eq), $\mathbf{nil} \vdash^{H_0}_{hm} T{:}\mathbf{Eq}(T)$. By Lemma B.2.8 (computing the pvu of a type world judgement) applied to Π, we have $hm \preccurlyeq \min(\text{pvu}_X(\Pi))$. By Lemma B.5.4 (type preservation by substitution), $\{X{\leftarrow}T\}E \vdash^H_{hm} \{X{\leftarrow}T\}J$, as desired.

□

Lemma B.5.16 (type preservation by fully carried out module substitution)
If $U(T)$:$[X$:$\mathbf{Le}(T_0), T_1], E \vdash^H_\bullet J$ and J is not U:S for any S and $\mathbf{nil} \vdash^\varnothing_\bullet [T, v^\bullet]$:$[X$:$\mathbf{Le}(T_0), T_1]$, then $\sigma E \vdash^{\sigma H}_\bullet \sigma J$, where $\sigma = \{U \leftarrow h, U.\text{TYPE} \leftarrow h, U.\text{term} \leftarrow [v^\bullet]^{\{X \leftarrow h\} T_1}_h\}$, where $h = \mathbf{hash}(N, H, [T, v^\bullet]$:$[X$:$\mathbf{Le}(T_0), T_1])$, for any N.

Proof. To do. □

Lemma B.5.17 (signature rewriting in a type world judgement) If $E_0, U(T_0)$:$[X$:$K, T], E_1 \vdash^{H_1}_{hm} J$ is a derivable type world judgement and J is not of the form U:S' for any S' and $E_0 \vdash^{H_0}_{hm} [X$:$K, T']$ ok with $H_0 \subseteq H_1$ then $E_0, U(T_0)$:$[X$:$K, T'], E_1 \vdash^{H_1}_{hm} J$.

Proof. To do. □

B.6 Type decomposition

Lemma B.6.1 (shortening typing proof)
If $E \vdash^H_{hm} e$:T then there exists T' such that $E \vdash^H_{hm} e$:T' by a subproof that does not have (eT.eq) as the last rule used and $E \vdash^H_{hm} T' == T$.

Proof. Induct on the structure of the derivation Π of $E \vdash^H_{hm} e$:T.

If the proof $E \vdash^H_{hm} e$:T does not have an instance of (eT.eq) as the last step, then we have the desired result, given that $E \vdash^H_{hm} T$:$\mathbf{Le}(\top)$ by Lemma B.5.7 (things have to be ok), whereupon we can apply (Teq.refl).

Otherwise there is T' such that $E \vdash^H_{hm} e$:T is derived from $E \vdash^H_{hm} e$:T' and $E \vdash^H_{hm} T' == T$. By applying induction to the (proper) subproof Π' leading to $E \vdash^H_{hm} e$:T', we get that there is T'' such that $E \vdash^H_{hm} e$:T'' by a subproof of Π' that does not have (eT.eq) as the last step used, and $E \vdash^H_{hm} T'' == T'$. By (Teq.tran), we have $E \vdash^H_{hm} T'' == T$, which completes our proof obligation. □

Lemma B.6.2 (reversing typing proof through a context)
If $\mathbf{nil} \vdash^H_{hm'} CC^{hm'}_{hm}.e$:$T'$ then there exists T such that $\mathbf{nil} \vdash^H_{hm} e$:$T$. If furthermore $\mathbf{nil} \vdash^H_{hm} e_1$:$T$ then $\mathbf{nil} \vdash^H_{hm'} CC^{hm'}_{hm}.e_1$:$T'$.

Proof. Induct on the structure of $CC^{hm'}_{hm}$. By Lemma B.6.1 (shortening typing proof), there exists T_0 such that $\mathbf{nil} \vdash^H_{hm'} T_0 == T'$, and $\mathbf{nil} \vdash^H_{hm'} CC^{hm'}_{hm}.e$:$T_0$ by a proof Π that does not end with (eT.eq).

Case $CC^{hm'}_{hm} = \_$ **:** Trivial.

Case $CC^{hm'}_{hm} = C^{hm'}_{hm_1}.CC^{hm_1}_{hm}$ **:** Then Π ends with an application of (eT.tuple), (eT.record), (eT.field), tuple(eT.proj), (eT.ap), (eT.mar), (eT.marred), (eT.unmar), (eT.send) or (eT.col) (depending on $C^{hm'}_{hm_1}$). In any case, one premise is $\mathbf{nil} \vdash^H_{hm_1} CC^{hm_1}_{hm}.e$:$T_1$ for some T_1. By induction we get $\mathbf{nil} \vdash^H_{hm} e$:$T$ for some T.

If furthermore $\mathbf{nil} \vdash^H_{hm} e_1$:$T$, then we have $\mathbf{nil} \vdash^H_{hm_1} CC^{hm_1}_{hm}.e_1$:$T_1$ by induction. Each of the (eT.*) rules considered above is linear with respect to the expression metavariable instantiated by $CC^{hm_1}_{hm}.e$, with exactly one occurence above the line and one below. Instantiating this metavariable by $CC^{hm_1}_{hm}.e_1$ yields another instance of the rule. By replacing in Π the derivation leading to $\mathbf{nil} \vdash^H_{hm_1} CC^{hm_1}_{hm}.e$:$T_1$ by that leading to $\mathbf{nil} \vdash^H_{hm_1} CC^{hm_1}_{hm}.e_1$:$T_1$, we

get a derivation of $\mathbf{nil} \vdash^H_{hm'} CC^{hm'}_{hm}.e_1{:}T_0$. By (eT.eq), since $\mathbf{nil} \vdash^H_{hm'} T_0 == T'$, we have $\mathbf{nil} \vdash^H_{hm'} CC^{hm'}_{hm}.e_1{:}T'$ as desired.

□

Definition B.6.3 (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.6.4 (purely abstract environment) A purely abstract environment is one that contains only bindings of the form $U{:}[X{:}\mathbf{Le}(T_0), T_1]$ or $X{:}\mathbf{Le}(T)$.

Lemma B.6.5 (equality kinding in an uncontributing environment)
If $E \vdash^H_{hm} T{:}\mathbf{Eq}(T')$ and E is a bare bones environment then $E \vdash^H_{hm} T == T'$ 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 $T = T' = U.\text{TYPE}$, with a proof whose last steps are (US.var), (US.self) and lastly (TK.mod). To prove that $E \vdash_{hm} U.\text{TYPE} == U.\text{TYPE}$, we can't do any better than starting at $E \vdash_{hm}$ 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 $E \vdash^H_{hm} T{:}K$ and $E \vdash^H_{hm} K <: \mathbf{Eq}(T')$. By Lemma B.4.3 (discreteness of subkinding), $E \vdash^H_{hm} K == \mathbf{Eq}(T')$ by a subproof, whence by reversing (Keq.Eq) there exists T'' such that $K = \mathbf{Eq}(T'')$ and $E \vdash^H_{hm} T'' == T'$, the latter being derived by a proper subproof of the original proof. By induction on $E \vdash^H_{hm} T{:}\mathbf{Eq}(T'')$, we get $E \vdash^H_{hm} T == T''$ by a smaller proof. By (Teq.tran), we get $E \vdash^H_{hm} T == T'$, 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.1.11 (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.6.6 (type decomposition) Let E be a purely abstract environment.

1. If $E \vdash^H_{hm} T_0 == TC(T_1, ..., T_j)$ or $E \vdash^H_{hm} TC(T_1, ..., T_j) == T_0$ then there exist $T'_1, ..., T'_j$ such that $E \vdash^H_{hm} T'_i == T_i$ for $1 \leqslant i \leqslant j$ and either $T_0 = TC(T'_1, ..., T'_j)$ or there exist v and T' and T'' and N and H_1 such that $T_0 = h.\text{TYPE}$ with $h = \mathbf{hash}(N, H_1, [T'_0, v]{:}[X{:}\mathbf{Le}(T'), T'']) \in hm$.
2. If $E \vdash^H_{hm} T_0 == h_1.\text{TYPE}$ or $E \vdash^H_{hm} h_1.\text{TYPE} == T_0$ then one of the following cases holds :
   (a) There are T and T' and v and N and H_1 such that $h_1 = \mathbf{hash}(N, H_1, [T'_0, v]{:}[X{:}\mathbf{Le}(T), T']) \in hm$ and $E \vdash^H_{hm} T_0 == T'_0$.
   (b) There are T and T' and v and N and H_1 such that $T_0 = h.\text{TYPE}$ with $h = \mathbf{hash}(N, H, [T'_0, v]{:}[X{:}\mathbf{Le}(T), T']) \in hm$ and $E \vdash^H_{hm} T'_0 == h_1.\text{TYPE}$.
   (c) $T_0 = h_1.\text{TYPE}$.

Proof. To do. □

Lemma B.6.7 (decomposition of type equivalence)
If $\mathbf{nil} \vdash^H_{hm} TC(T_1, ..., T_j) == TC(T_1, ..., T'_j)$ then $\mathbf{nil} \vdash^H_{hm} T_i == T'_i$ for $1 \leqslant i \leqslant j$.

Proof. Trivial consequence of part 1 of Lemma B.6.6 (type decomposition). □

Lemma B.6.8 (triviality of type equivalence in a trivial environment)
If $\mathbf{nil} \vdash^H_\bullet T == T'$ then $T = T'$.

Proof. Induct on the structure of T.

Case T is of the form $TC(T_1, ..., T_j)$: By Lemma B.6.6 (type decomposition) and the trivial colour of the hypothesis, there exist $T'_1, ..., T'_j$ such that $\mathbf{nil} \vdash^H_\bullet T_i == T'_i$ 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 $\mathbf{nil} \vdash^H_\bullet T_i == T'_i$ for $1 \leqslant i \leqslant j$ we have $T_i = T'_i$ hence $T = T'$ as desired.

Case T is of the form h.TYPE : By Lemma B.6.6 (type decomposition) and the trivial colour of the hypothesis, $T = T'$.

Case T is of the form U.TYPE or X : Impossible by Lemma B.1.11 (free variables of a judgement come from the environment).

□

I need the same set of lemmas for subtyping and especially :

Lemma B.6.9 (decomposition of subtyping)
If $\mathbf{nil} \vdash^H_{hm} TC(T_1, ..., T_j) <: TC(T_1, ..., T'_j)$ then $\mathbf{nil} \vdash^H_{hm}$ some relation between T_i and T'_i for $1 \leqslant i \leqslant j$ depending on TC.

Proof. To do. □

B.7 Type preservation by reduction

Theorem B.7.1 (type preservation for expression reduction)
If $\mathbf{nil} \vdash^H_{hm} e{:}T$ and $H, e \longrightarrow_{hm} H', e'$ then $\mathbf{nil} \vdash^{H'}_{hm} e'{:}T$.

Proof. Note that by Lemma B.6.1 (shortening typing proof), there exists T' such that $\mathbf{nil} \vdash^H_{hm} T' == T$ and $\mathbf{nil} \vdash^H_{hm} e{:}T'$ 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.5.7 (things have to be ok), we have $\mathbf{nil} \vdash^H_{hm} T'{:}\mathbf{Le}(\top)$, and by Lemma B.1.7 (environments and subhashes have to be ok), we have $\mathbf{nil} \vdash^H_{hm}$ ok.

We induct on the derivation of the reduction and use the notations of each of the rules as much as possible.

Case (ered.proj) :

$$H, \mathbf{proj}_i\,(v_1^{hm}, ..., v_j^{hm}) \longrightarrow_{hm} H, v_i^{hm} \qquad \text{if } 1 \leqslant i \leqslant j \qquad \text{(ered.proj)}$$

Let $e = \mathbf{proj}_i\,(v_1^{hm}, ..., v_j^{hm})$. Then $\mathbf{nil} \vdash^H_{hm} e{:}T'$ must have been derived by (eT.proj), and the ith premise is $\mathbf{nil} \vdash^H_{hm} v_i^{hm}{:}T'$, i.e. $\mathbf{nil} \vdash^H_{hm} e'{:}T'$.

Case(ered.field) :

$$H, \{l_1 = v_1^{hm}, ..., l_j = v_j^{hm}\}.l_i \longrightarrow_{hm} H, v_i^{hm} \qquad \text{if } 1 \leqslant i \leqslant j \qquad \text{(ered.field)}$$

Let $e = \{l_1 = v_1^{hm}; ...; l_j = v_j^{hm}\}.l_i$. Then $\mathbf{nil} \vdash_{hm}^{H} e : T'$ must have been derived by (eT.field), and the ith premise is $\mathbf{nil} \vdash_{hm}^{H} v_i^{hm} : T'$, i.e. $\mathbf{nil} \vdash_{hm}^{H} e' : T'$.

Case (ered.ap) :

$$H, (\lambda x{:}T.e)\, v^{hm} \longrightarrow_{hm} H, \{x \leftarrow [v^{hm}]_{hm}^{T}\}e \qquad \text{(ered.ap)}$$

Let $e' = \{x \leftarrow v^{hm}\}e$. $\mathbf{nil} \vdash_{hm}^{H} (\lambda x{:}T.e) v^{hm} : T'$ must have been derived by (eT.ap), with the premises $\mathbf{nil} \vdash_{hm}^{H} v^{hm} : T_0$ and $\mathbf{nil} \vdash_{hm}^{H} (\lambda x{:}T.e) : T_0 \to T'$.

By Lemma B.6.1 (shortening typing proof), there exists T'' such that $\mathbf{nil} \vdash_{hm}^{H} (\lambda x{:}T.e) : T''$ by a proof that does not end in (eT.eq), and $\mathbf{nil} \vdash_{hm}^{H} T'' == T_0 \to T'$. By reversing (eT.fun), there exists T_1 such that $T'' = T \to T_1$ and $x{:}T \vdash_{hm}^{H} e : T_1$. By Lemma B.6.7 (decomposition of type equivalence), we have $\mathbf{nil} \vdash_{hm}^{H} T_1 == T'$.

By Lemma B.5.4 (type preservation by substitution), we have $\mathbf{nil} \vdash_{hm}^{H} \{x \leftarrow v^{hm}\}e : T_1$. By (eT.eq), we get $\mathbf{nil} \vdash_{hm}^{H} e' : T'$.

Case (ered.mar) :

$$H, \mathbf{mar}\,(v^{hm}{:}T) \longrightarrow_{hm} H, \mathbf{marshalled}_{H}([v^{hm}]_{hm}^{T}{:}T) \qquad \text{(ered.mar)}$$

Let $e = \mathbf{mar}\,(v^{hm}{:}T)$, and $e' = \mathbf{marshalled}_{H}([v^{hm}]_{hm}^{T}{:}T)$. $\mathbf{nil} \vdash_{hm}^{H} e : T'$ must have been derived by (eT.mar), with the premise $\mathbf{nil} \vdash_{hm}^{H} v^{hm} : T$; also $T' = \text{STRING}$.

By Lemma B.5.7 (things have to be ok) and Lemma B.4.11 (colour change preserves type okedness), $\mathbf{nil} \vdash_{\bullet}^{H} T : \mathbf{Le}(\top)$. By (eT.col), we get $\mathbf{nil} \vdash_{\bullet}^{H} [v^{hm}]_{hm}^{T} : T$. By Lemma B.1.7 (environments and subhashes have to be ok), we have $\mathbf{nil} \vdash_{hm}^{H} \text{ok}$. By (eT.marred), we get $\mathbf{nil} \vdash_{hm}^{H} e' : \text{STRING}$.

Case (ered.unmar) :

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{H'}(v^{\bullet}{:}T){:}T') \longrightarrow_{hm} \begin{cases} H \cup H', {}_{(T <: T')}v^{\bullet} & \text{if } \mathbf{nil} \vdash_{hm}^{H \cup H'} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

(ered.unmar)

Let $e = \mathbf{unmar}\,(\mathbf{marshalled}_{H'}(v^{\bullet}{:}T)){:}T'$. By reversing (eT.unmar), we get $\mathbf{nil} \vdash_{hm}^{H} T' : \mathbf{Le}(\top)$ and $\mathbf{nil} \vdash_{hm}^{H} \mathbf{marshalled}\,(v^{\bullet}{:}T) : \text{STRING}$; also T' rightly corresponds to the T' from the introduction of this proof. There are two possible outcomes.

Case e' is $\mathbf{Unmarfailure}^{T'}$: By Lemma B.5.7 (things have to be ok), $\mathbf{nil} \vdash_{hm}^{H} T' : \mathbf{Le}(\top)$, hence $\mathbf{nil} \vdash_{hm}^{H} \mathbf{Unmarfailure}^{T'} : T'$ by (eT.Undynfailure). We conclude by Lemma B.3.4 (weakening).

Case e' is the value ${}_{(T <: T')}v^{\bullet}$: We know $\mathbf{nil} \vdash_{hm}^{H \cup H'} T <: T'$:]. By Lemma B.6.1 (shortening typing proof) and reversing (eT.marred), we get $\mathbf{nil} \vdash_{\bullet}^{H} v^{hm} : T$. By Lemma B.3.2 (colour stripping judgements), this implies that $\mathbf{nil} \vdash_{hm}^{H} v^{hm} : T$. By Lemma B.3.4 (weakening), we have then $\mathbf{nil} \vdash_{hm}^{H \cup H'} v^{hm} : T$. We conclude by (eT.sub).

Case (ered.sub.sub) :

$$H, {}_{(TV_0^{hm} <: TV_1^{hm})}v^{hm} \longrightarrow_{hm} H, {}_{(TV_2^{hm} <: TV_1^{hm})}\widehat{v}^{hm} \quad \text{where } v^{hm} = {}_{(TV_2^{hm} <: TV_3^{hm})}\widehat{v}^{hm}$$

(ered.sub.sub)

$\mathbf{nil} \vdash^H_{hm} {}_{(TV_0^{hm} <: TV_1^{hm})}({}_{(TV_2^{hm} <: TV_3^{hm})}\widehat{v}^{hm}) : TV_1^{hm}$ must have been derived by (eT.sub). Then the premises are $\vdash^H_{hm} TV_0^{hm} <: TV_1^{hm}$ and $\vdash^H_{hm} {}_{(TV_2^{hm} <: TV_3^{hm})}\widehat{v}^{hm} : TV_1^{hm}$. By Lemma B.6.1 (shortening typing proof), and by reversing (eT.sub), we get $\vdash^H_{hm} TV_3^{hm} == TV_0^{hm}$ and $\vdash^H_{hm} {}_{(TV_2^{hm} <: TV_3^{hm})}\widehat{v}^{hm} : TV_3^{hm}$ and $\vdash^H_{hm} TV_2^{hm} <: TV_3^{hm}$ and $\vdash^H_{hm} \widehat{v}^{hm} : TV_2^{hm}$. By Lemma B.4.5 (transitivity of subtyping) and (Tsub.Equi), we have $\vdash^H_{hm} TV_2^{hm} <: TV_1^{hm}$. By (eT.sub) we get then $\vdash^H_{hm} {}_{(TV_2^{hm} <: TV_1^{hm})}\widehat{\widehat{v}}^{hm} : TV_1^{hm}$.

Case (ered.sub.typeright) :

$$H, {}_{(T_0 <: h_0.\text{TYPE})}\widehat{v}^{hm} \longrightarrow_{hm} H, {}_{(T_0 <: T_1)}\widehat{v}^{hm} \quad \text{when } h_0 \in hm \text{ and } \text{impl}(h_0) = T_1 \qquad \text{(ered.sub.typeright)}$$

$\mathbf{nil} \vdash^H_{hm} {}_{(T_0 <: h_0.\text{TYPE})} v^{hm} : h_0.\text{TYPE}$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash^H_{hm} T_0 <: h_0.\text{TYPE}$ and $\mathbf{nil} \vdash^H_{hm} v^{hm} : T_0$. From Lemma B.1.7 (environments and subhashes have to be ok) and since $h_0 \in hm$ and $\text{impl}(h_0) = T_1$, we know that $\mathbf{nil} \vdash^H_{hm} h_0.\text{TYPE} == T_1$ by (Teq.hash). By (Tsub.Equi) and the Lemma B.4.5 (transitivity of subtyping), we get $\mathbf{nil} \vdash^H_{hm} T_0 <: T_1$. By (eT.sub), we know that $\mathbf{nil} \vdash^H_{hm} {}_{(T_0 <: T_1)} v^{hm} : T_1$. We conclude by (eT.eq).

Case (ered.sub.typeleft) :

$$H, {}_{(h_0.\text{TYPE} <: TV^{hm})}\widehat{v}^{hm} \longrightarrow_{hm} H, {}_{(T_0 <: TV^{hm})}\widehat{v}^{hm} \quad \text{when } h_0 \in hm \text{ and } \text{impl}(h_0) = T_0 \qquad \text{(ered.sub.typeleft)}$$

$\mathbf{nil} \vdash^H_{hm} {}_{(h_0.\text{TYPE} <: TV^{hm})}\widehat{v}^{hm} : TV^{hm}$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash^H_{hm} h_0.\text{TYPE} <: TV^{hm}$ and $\mathbf{nil} \vdash^H_{hm} \widehat{v}^{hm} : h_0.\text{TYPE}$. From Lemma B.1.7 (environments and subhashes have to be ok) and since $h_0 \in hm$ and $\text{impl}(h_0) = T_0$, then we know that $\vdash^H_{hm} h_0.\text{TYPE} == T_0$ by (Teq.hash). By (Tsub.Equi) and the Lemma B.4.5 (transitivity of subtyping), we get $\mathbf{nil} \vdash^H_{hm} T_0 <: TV^{hm}$. By (eT.eq) we have a proof of $\vdash^H_{hm} \widehat{\widehat{v}}^{hm} : T_0$ which leads to know that $\mathbf{nil} \vdash^H_{hm} {}_{(T_0 <: TV^{hm})}\widehat{v}^{hm} : TV^{hm}$ by (eT.sub).

Case (ered.sub.tuple) :

$$H, {}_{(T_1 * ... * T_j <: T'_1 * ... * T'_j)}(v_1^{hm}, ..., v_j^{hm}) \longrightarrow_{hm} H, ({}_{(T_1 <: T'_1)} v_1^{hm}, ..., {}_{(T_j <: T'_j)} v_j^{hm}) \qquad \text{(ered.sub.tuple)}$$

$\vdash^H_{hm} {}_{(T_1 * ... * T_j <: T'_1 * ... * T'_j)}(v_1^{hm}, ..., v_j^{hm}) : (T'_1 * ... * T'_j)$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash^H_{hm} (v_1^{hm}, ..., v_j^{hm}) : (T_1 * ... * T_j)$ and $\vdash^H_{hm} (T_1 * ... * T_j) <: (T'_1 * ... * T'_j)$. By some Lemma B.6.9 (decomposition of subtyping) we know that for $1 \leqslant i \leqslant j$, $\vdash^H_{hm} T_i <: T'_i$. We are now able to use (eT.sub) to prove, for $1 \leqslant i \leqslant j$, $\vdash^H_{hm} {}_{(T_i <: T'_i)} v_i^{hm} : T'_i$. We then use (eT.tuple) to prove the desired $\vdash^H_{hm} ({}_{(T_1 <: T'_1)} v_1^{hm}, ..., {}_{(T_j <: T'_j)} v_j^{hm}) : (T'_1 * ... * T'_j)$.

Case (ered.sub.record) :

$$H, {}_{(\{l_1 : T_1; ...; l_j : T_j\} <: \{l_{\pi(1)} : T_{\pi(1)}; ...; l_{\pi(i)} : T_{\pi(i)}\})}\{l_1 = v_1^{hm}; ...; l_j = v_j^{hm}\} \longrightarrow_{hm} H, \{l_{\pi(1)} = v_{\pi(1)}^{hm}; ...; l_{\pi(i)} = v_{\pi(i)}^{hm}\} \qquad \text{(ered.sub.record)}$$

$\vdash^H_{hm} {}_{(\{l_1 : T_1; ...; l_j : T_j\} <: \{l_{\pi(1)} : T_{\pi(1)}; ...; l_{\pi(i)} : T_{\pi(i)}\})}\{l_1 = v_1^{hm}; ...; l_j = v_j^{hm}\} : \{\{l_{\pi(1)} : T_{\pi(1)}; ...; l_{\pi(i)} : T_{\pi(i)}\}\}$ must have been derived by (eT.sub). The premises are then $\mathbf{nil} \vdash^H_{hm} \{l_1 = v_1^{hm}; ...; l_j = v_j^{hm}\} : \{l_1 : T_1; ...; l_j : T_j\}$ and $\vdash^H_{hm} \{l_1 : T_1; ...; l_j : T_j\} <: \{\{l_{\pi(1)} : T_{\pi(1)}; ...; l_{\pi(i)} : T_{\pi(i)}\}\}$. As we have only width subtyping, we know that, for $1 \leqslant k \leqslant i \leqslant j$, $\vdash^H_{hm} T_k == T'_{\pi(k)}$. By Lemma B.6.1 (shortening typing proof) we have $\mathbf{nil} \vdash^H_{hm} \{l_1 = v_1^{hm}; ...; l_j = v_j^{hm}\} : \{l_1 : T'_1; ...; l_j : T'_j\}$ whose last rule is (eT.record) and $\mathbf{nil} \vdash^H_{hm} \{l_1 : T_1; ...; l_j : T_j\} == \{l_1 : T'_1; ...; l_j : T'_j\}$. The premises are

then, for $1 \leqslant k \leqslant j$, $\vdash^H_{hm} v^{hm}_k : T'_k$. By Lemma B.6.7 (decomposition of type equivalence) we know that for $1 \leqslant k \leqslant j$, $\vdash^H_{hm} T_k == T'_k$. We can now prove that, for $1 \leqslant k \leqslant j$, $\vdash^H_{hm} v^{hm}_k : T_k$ using (eT.eq).By (eT.record) we now have $\vdash^H_{hm} \{l_1 = v^{hm}_1; ...; l_i = v^{hm}_i\} : \{l_1 : T_1; ...; l_i : T_i\}$.

Case (ered.sub.fun) :

$$H, {}_{(T_1 \to T_2 <: T'_1 \to T'_2)}(\lambda x{:}T_0.e) \longrightarrow_{hm} H, (\lambda x{:}T'_1.{}_{(T_2 <: T'_2)}\{x \leftarrow {}_{(T'_1 <: T_1)}x\}e) \qquad \text{(ered.sub.fun)}$$

To do.

Cases (ered.sub.*) : To do.

Cases (ered.col.*) : To do.

In all cases, we have $\mathbf{nil} \vdash_{hm} e' : T'$, whence by (eT.eq), $\mathbf{nil} \vdash_{hm} e' : T$. □

Lemma B.7.2 (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 $\vdash n$ ok, we get $\vdash n'$ 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 $\vdash n$ ok, we get $\vdash n_i$ ok for $1 \leqslant i \leqslant 3$, whence by (nok.par) twice $\vdash n'$ ok.

Reflexivity, symmetry, transitivity : Trivial (the latter two, by induction).

□

Corollary B.7.3 (type preservation for network reduction) If $\vdash n$ ok and $n \longrightarrow n'$ then $\vdash n'$ ok.

Proof. Induct on the derivation of the reduction $n \longrightarrow n'$.

Case (nred.expr) : Trivial by Theorem B.7.1 (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 $n_0 \longrightarrow n_1$. By reversing (nok.par), we have $\vdash n_0$ ok and $\vdash n_2$ ok. By induction we have $\vdash n_1$ ok. By (nok.par), we have $\vdash n'$ ok.

Case (nred.strcong) : There exist n_0 and n_1 such that $n \equiv n_0 \longrightarrow n_1 \equiv n'$. By Lemma B.7.2 (type preservation for network structural congruence), we get $\vdash n_0$ ok. By induction we get $\vdash n_1$ ok. By Lemma B.7.2 (type preservation for network structural congruence), we get $\vdash n'$ ok.

Case (nred.comm) : To do.

□

Theorem B.7.4 (type preservation for machine reduction) If $\mathbf{nil} \vdash^H_\bullet m : T$ and $H, m \longrightarrow_{\mathsf{c}} H', m'$ then $\mathbf{nil} \vdash^{H'}_\bullet m' : T$.

Proof. Consider each rule.

The Lemma B.5.12 (type preservation by module substitution in coloured judgements) and Lemma B.5.16 (type preservation by fully carried out module substitution) needs some work before getting a correct proof.

Case (mred.Le) : To do.

Case (mred.Eq) : To do.

□

B.8 Progress

Definition B.8.1 (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 $CC^{hm'}_{hm}$ and v^{hm} such that $e = CC^{hm'}_{hm}.!\,v^{hm}$
- e is ready to input, i.e. there exists $CC^{hm'}_{hm}$ such that $e = CC^{hm'}_{hm}.?$

Definition B.8.2 (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 $CC^{hm'}_{hm}$ and T such that $e = CC^{hm'}_{hm}.\mathbf{Unmarfailure}^T$.

Lemma B.8.3 (dormancy in context) If e is dormant and $CC^{hm'}_{hm}$ is a coloured evaluation context then $CC^{hm'}_{hm}.e$ is dormant.

Proof. Composing coloured evaluation contexts yields a coloured evaluation context. □

Lemma B.8.4 (reduction in context) If $e \longrightarrow_{hm}$ and $CC^{hm'}_{hm}$ is an evaluation context then $CC^{hm'}_{hm}.e \longrightarrow_{hm'}$.

Proof. Apply (ered.cong) as many times as the size of $CC^{hm'}_{hm}$ requires. □

Definition B.8.5 (legitimately stuck expressions) An expression e is *legitimately stuck* in hm iff one of the following cases holds :
- e is a hm-value
- e is dormant.

Theorem B.8.6 (progress of expressions) If $\mathbf{nil} \vdash^H_{hm} e{:}T$ then one of the following cases holds :
- e is legitimately stuck in hm.
- e can reduce, i.e. there exists e' and H' such that $H, e \longrightarrow_{hm} H', e'$.

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.1.11 (free variables of a judgement come from the environment) since the environment is empty.

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 $\mathbf{nil} \vdash^H_{hm} e_0{:}T_1 \to T$ and $\mathbf{nil} \vdash^H_{hm} e_1{:}T_1$. Apply the inductive hypothesis to e_0.

Case e_0 can reduce : there exists e'_0 and H' such that $H, e_0 \longrightarrow_{hm} H', e'_0$. By (ered.cong), $H, e \longrightarrow_{hm} H', e'_0\ e_1$.

Case e_0 is ready to output : there exist $CC^{hm}_{hm'}$ and $v^{hm'}$ such that $e_0 = CC^{hm}_{hm'}.!\,v^{hm'}$, thus $e = (\_\ e_1).CC^{hm}_{hm'}.!\,v^{hm'}$ 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** hm**-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 $H, e_1 \longrightarrow_{hm} H', e_1'$. By (ered.cong), $H, e \longrightarrow_{hm} H', e_0\ e_1'$.

Case e_1 **is ready to output :** there exist $CC^{hm}_{hm'}$ and $v^{hm'}$ such that $e_1 = CC^{hm}_{hm'}.!\, v^{hm'}$, thus $e = (e_0\ \_).CC^{hm}_{hm'}.!\, v^{hm'}$ 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** hm**-value :** To do.

Case (eT.fun) : e is a value.

Case (eT.send) : e is ready to output.

Case (eT.recv) : e is ready ot input.

Case (eT.mar) : There exist an e_0 and a T_0 such that $e = \mathbf{mar}\,(e_0{:}T_0)$, and $T = \text{STRING}$. If e_0 is dormant or reduces then the same holds for e by Lemma B.8.4 (reduction in context) and Lemma B.8.3 (dormancy in context). Otherwise, by the inductive hypothesis on e_0, e_0 is a hm-value, so by (ered.mar), $H, e \longrightarrow_{hm} H, \mathbf{marshalled}_{\,H}([e_0]^T_{hm}{:}T)$.

Case (eT.marred) : There exist an e_0 and a T_0 such that $e = \mathbf{marshalled}_{\,H_0}(e_0{:}T_0)$, and $T = \text{STRING}$. If e_0 is dormant or reduces then the same holds for e by Lemma B.8.4 (reduction in context) and Lemma B.8.3 (dormancy in context), since $\mathbf{marshalled}\,(\_{:}T_0)$ is an evaluation context. Otherwise, by the inductive hypothesis on e_0, e_0 is a hm-value, so e is an hm-value.

Case (eT.unmar) : To do.

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.8.3 (dormancy in context) and Lemma B.8.4 (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) : To do.

Case (eT.field) : Similar to eT.proj.

Case (eT.col) : There is an e_0 and an hm_0 such that $e = [e_0]^T_{hm_0}$. Apply the inductive hypothesis to e_0 ; if e_0 is a value, the discussion depends on its form and that of hm_0.

By Lemma B.6.1 (shortening typing proof), there is a type T' such that $\mathbf{nil} \vdash_{hm_0} e_0{:}T'$ by a smaller proof that does not use (eT.eq) as its last step and $\mathbf{nil} \vdash_{hm_0} T' == T$.

Case e_0 **is dormant :** e is dormant.

Case e_0 **reduces :** There is an e_0' such that $e_0 \longrightarrow_{hm_0} e_0'$. By (ered.cong), $e \longrightarrow_{hm} [e_0']^T_{hm_0}$.

Case e_0 **is an** hm_0**-value that is not a bracket expression :** To do.

Case $e_0 = [v^{h_1}]^{h_1}_{h_1}$ **for some** h_1 **and** $hm_0 = \bullet$ **:** To do.

Case $e_0 = [v^{h_1}]_{h_1}^{h_1}$ **for some** h_1 **and** $hm_0 \neq \bullet$ **:** To do.

Case (eT.sub) : To do.

□

Corollary B.8.7 (progress of networks) If $\vdash n$ 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 $n \longrightarrow n'$

where

$n_{()} ::=$	$\mathbf{0}$	null
	$n_{()} \mid n_{()}$	parallel composition
	$()$	unit
$n_{\text{fail}} ::=$	$\mathbf{0}$	null
	$n_{\text{fail}} \mid n_{\text{fail}}$	parallel composition
	$CC^{\bullet}_{hm}.\mathbf{Unmarfailure}^T$	dead
$n_? ::=$	$n_? \mid n_?$	parallel composition
	$CC^{\bullet}_{hm}.?$	waiting to input
$n_! ::=$	$n_! \mid n_!$	parallel composition
	$CC^{\bullet}_{hm}.!\, v$	waiting to output

Proof. Induct on the derivation of $\vdash n$ 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) : By Theorem B.8.6 (progress of expressions), one of the following cases holds :

Case n **is a** $\bullet$**-value :** To do.

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.8.8 (progress of machines) If $\mathbf{nil} \vdash^H_{\bullet} m{:}T$ then either m is an expression or it reduces under $\longrightarrow_{\mathsf{c}}$.

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.9 Determinism of reduction

Theorem B.9.1 (determinism of machine reduction) Reduction of machines is deterministic, i.e. if $m \longrightarrow_{\mathsf{c}} m_1$ and $m \longrightarrow_{\mathsf{c}} m_2$ 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 $m = \mathbf{module}\, N_U = M{:}[X{:}\mathbf{Le}(T'), T]\ \mathbf{in}\ m'$: The only applicable rule is (mred.Le).

Case $m = \mathbf{module}\, N_U = M{:}[X{:}\mathbf{Eq}(T'), T]\ \mathbf{in}\ m'$: The only applicable rule is (mred.Eq).

□

Lemma B.9.2 (values do not reduce) If $H, e \longrightarrow_{hm} H', e'$ then e is not an hm-value.

Proof. We prove that if e is an hm-value then e does not reduce in hm. We induct on the structure of values.

Case $v^{hm} = ()$: No reduction rule applies.

Case $v^{hm} = (v_1^{hm}, ..., v_j^{hm})$: 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 $(v_1^{hm}, ..., v_{i-1}^{hm}, \_, v_{i+1}^{hm}, ..., v_j^{hm})$. But then $v_i^{hm} \longrightarrow_{hm}$, which is impossible by induction.

Case $v^{hm} = \{l_1 = v_1^{hm}; ...; l_j = v_j^{hm}\}$: 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 $\{l_1 = v_1^{hm_1}, .., l_{i-1} = v_{i-1}^{hm_1}, l_i = \_, l_{i+1} = e_{i+1}, .., l_j = e_j\}$. But then $v_i^{hm} \longrightarrow_{hm}$, which is impossible by induction.

Case $v^{hm} = (\lambda x{:}T.e)$: No reduction rule applies.

Case $v^{hm} = \mathbf{marshalled}\,(v^{\bullet}{:}T)$: 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 $\mathbf{marshalled}\,(\_{:}T)$. But then $v^{\bullet} \longrightarrow_{\bullet}$, which is impossible by induction.

Case $v^{hm} = [\widehat{v}^{hm_1 \cup hm}]_{hm_1}^{h_1.\mathrm{TYPE}}$ where $h_1 \notin hm$: The rule (ered.col.type) requires $h_1 \in hm$: contradiction. The other rules (ered.col.*) does not apply as it requires the type annotation on the bracket not to be a hash. If (ered.cong) applies, then it is with a context of the form $[\_]_{hm_1}^{h_1.\mathrm{TYPE}}$. But then $\widehat{v}^{hm_1 \cup hm} \longrightarrow_{hm_1 \cup hm}$, which is impossible by induction.

Case $v^{hm} = {}_{(TV_1^{hm} <: TV_2^{hm})}\widehat{v}^{hm}$: Since $TV_1^{hm} = h_0.\mathrm{TYPE}$ or $TV_2^{hm} = h_0.\mathrm{TYPE}$ (ered.sub.{tuple,record.*, fun,marshalled,unit}) do not apply. The rule (ered.sub.sub) requires $\widehat{v}^{hm}$ 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 hm which is impossible by definition of TV^{hm}.

□

Theorem B.9.3 (determinism of expression reduction) Reduction of expressions and machines is deterministic, i.e. if $H, e \longrightarrow_{hm} H', e'$ and $H, e \longrightarrow_{hm} H'', e''$ 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 $e = x$, $e = U.\text{term}$, $e = \mathbf{Unmarfailure}^T$ **:** No reduction is possible.

Cases $e = ()$, $e = (v_1^{hm}, ..., v_j^{hm})$, $e = \lambda x{:}T.e_0$, $e = \mathbf{marshalled}\,(v^\bullet{:}T)$ **:** No reduction is possible, by Lemma B.9.2 (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 *hm*-values. The case $i = j + 1$ has already been treated. Given Lemma B.9.2 (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 *hm*-value, given Lemma B.9.2 (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 *hm*-values, given Lemma B.9.2 (values do not reduce), the only possibility of reduction is (ered.ap). If e_1 is an *hm*-value and e_2 is not an *hm*-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 *hm*-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 *hm*-value, then e_0 does not reduce by Lemma B.9.2 (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 $e = \mathbf{marshalled}\,(e_0{:}T)$ **:** The only possibility of reduction is (ered.cong) with the context $\mathbf{marshalled}\,(\_{:}T)$, 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.

Cases $e = !\, e_0$, $e = ?$ **:** No reduction is possible (communication happens at the network level).

Case $e = [e_1]^T_{hm_1}$ **:** To do.

Case $e = {}_{(T_1 <: T_2)}\, e_1$ **:** To do.

□