INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Global abstraction-safe marshalling with hash types

James J. Leifer[†] Gilles Peskine[†] Peter Sewell[‡] Keith Wansbrough[‡]

[†]INRIA Rocquencourt
{First.Last}@inria.fr

[‡]University of Cambridge
{First.Last}@cl.cam.ac.uk

N° 4851

25 June 2003

THÈME 1

Rapport de recherche

ISSN 0249-6399

Global abstraction-safe marshalling with hash types

James J. Leifer† Gilles Peskine† Peter Sewell‡ Keith Wansbrough‡

†INRIA Rocquencourt
{First.Last}@inria.fr

‡University of Cambridge
{First.Last}@cl.cam.ac.uk

Thème 1 — Réseaux et systèmes
Projet Moscova

Rapport de recherche n° 4851 — 25 June 2003 — 86 pages

Abstract: Type abstraction is a key feature of ML-like languages for writing large programs. Marshalling is necessary for writing distributed programs, exchanging values via network byte-streams or persistent stores. In this report we combine the two, developing compile-time and run-time semantics for marshalling, that guarantee abstraction-safety between separately-built programs.

We obtain a namespace for abstract types that is global, i.e. meaningful between programs, by hashing module declarations. We examine the scenarios in which values of abstract types are communicated from one program to another, and ensure, by constructing hashes appropriately, that the dynamic and static notions of type equality mirror each other. We use singleton kinds to express abstraction in the static semantics; abstraction is tracked in the dynamic semantics by coloured brackets. These allow us to prove preservation, erasure, and coincidence results. We argue that our proposal is a good basis for extensions to existing ML-like languages, pragmatically straightforward for language users and for implementors.

Key-words: programming languages, ML, type theory, abstract types, marshalling, serialisation, modules, singleton kinds, hashing, distributed programming, lambda calculus

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

Sérialisation avec sûreté globale des abstractions : utilisation des types de hachage

Résumé : L'abstraction de types est un trait essentiel des langages de la famille ML pour écrire les logiciels de taille importante. La sérialisation est un ingrédient indispensable des programmes distribués, de la transmission de valeurs sur un réseau, et du stockage persistant de données. Dans ce rapport, nous combinons les deux, en développant des sémantiques statique et dynamique de la sérialisation qui garantissent la sûreté des abstractions entre des programmes développés séparément.

Nous obtenons un espace de nommage pour les types abstraits qui est global, c'est-à-dire commun à tous les programmes, par hachage des déclarations de modules. Nous examinons les scénarios dans lesquels des valeurs de types abstraits sont transmises entre les programmes ; nous garantissons que les notions statique et dynamique d'égalité de types coïncident grâce à la construction de la fonction de hachage. Nous utilisons les *singleton kinds* pour exprimer les abstractions dans la sémantique statique ; ces abstractions sont suivies à la trace par des crochets colorés dans la sémantique dynamique. Cette méthode nous permet de prouver les théorèmes de préservation, d'effacement, et de correspondance. Nous proposons cette technique comme une base pour étendre les langages à la ML, car elle est simple et pratique pour l'utilisateur comme pour l'implémenteur.

Mots-clés : langages de programmation, ML, théorie des types, types abstraits, marshalling, sérialisation, modules, singleton kinds, hachage, programmation distribuée, lambda calcul

Contents

This technical report is a long version of the paper [LPSW03]: James J. Leifer, Gilles Peskine, Peter Sewell, Keith Wansbrough. “Global abstraction-safe marshalling with hash types”. *Proc. 8th ICFP*. 2003. (Available from `http://pauillac.inria.fr/~leifer/research.html`.)

1 Introduction

Problem Type abstraction is a basic tool for modular programming, allowing the programmer to separate the interface and the implementation of an abstract data type, and to limit the scope in which the implementation details are visible. Work on ML-style module systems, including [Mac84, MTH90, HL94, Ler94], has led to expressive language constructs for controlling abstraction, with modules (structures) that can export abstract types, and also parameterised modules (functors); they have rich notions of type equality to deal with generativity and sharing. This work has largely been in the non-distributed context, concerned only with isolated executions of single programs. There, build-time type checking suffices to guarantee both type-safety and *abstraction-safety* — the property that values of an abstract type can only be inspected or constructed by the code of its definition, and hence that any invariants of this code hold of all values. At run-time, type information can be erased.

In the distributed setting, abstraction-safety is more subtle. One may need to exchange values between multiple executions of the same build, between executions of different builds of the same sources, and between executions of builds of different sources (sharing some modules, perhaps, but not all). This interaction might be by network communication or via a persistent store; in either case, some run-time check is clearly needed to guarantee safety. For abstraction-safety, it does not suffice to check only the underlying representation type; intuitively, we need also that the sender and receiver have compatible invariants. This can be enforced by requiring that they have the same code, but in general, where there is only partial sharing, we shall see that the design of an appropriate check is delicate.

We focus in particular on language support for *marshalling* a value to a byte string and *unmarshalling* such strings back to values. With these, one can implement a variety of useful mechanisms above the standard (byte-string) primitives for network communication and persistence. For example: (1) In the existing distributed languages JoCaml [JoC] and Nomadic Pict [SWP99] a single program can dynamically distribute computations, which can then interact via typed channels, but unsafe “name servers” are required to bootstrap connections *between* programs. Type- and abstraction-safe marshalling would enable such name servers to be expressed in a safe way. (2) More generally, safe marshalling would enable one to code up a variety of communication abstractions, such as typed channels with differing behaviour (asynchronous, unicast, multicast,...), within a high-level language; they would then be automatically guaranteed to be safe.

Contribution We present a type system and semantics (both compile-time and run-time) for marshalling and unmarshalling values between separate programs. Our solution:

- covers modules that declare abstract types, and ancillary type-sharing constraints;
- involves a dynamic type-check at unmarshal time that guarantees both type-safety and abstraction-safety;
- ensures the resulting dynamic notion of type equality coincides with the usual static notion, so that distributed programming is a smooth extension of local programming;
- “just works” in standard cases for interaction between programs that share some modules, without requiring any shared data beyond the source code for these modules;
- supports controlled abstraction-breaking, where required; and
- is efficiently implementable.

It is therefore a good basis for extensions to existing ML-like languages, pragmatically straightforward for language users and for implementors.

Approach The basic idea of our solution is to construct a global namespace for abstract types, meaningful across all programs, by hashing module declarations. *Hash types* do not appear in source programs, but are constructed at compile-time. For example, consider a module called `N` with the body `struct type t=Trep let x=... end` and the published interface `sig type t val x:... end`. The hash h =

```
hash(module N=struct type t=Trep let x= ... end
              : sig type t      val x: ... end, t)
```

would be constructed to give a run-time analogue of the compile-time abstract type name `N.t`. By constructing hashes carefully, we ensure that a simple run-time syntactic type equality check, at unmarshal-time, corresponds to the compile-time notion of type equivalence used in type-checking.

The standard operational semantics for existentials forgets abstraction. In contrast, we give a run-time semantics that records which subterms can see through which abstractions using *coloured brackets*, adapting a device of [GMZ00]. For example, within the code of `N.x`, even after it has been substituted into its usage sites, the type equality h`==Trep` can be used. This enables us to prove type- and abstraction-preservation, progress, and the coincidence result mentioned above. We prove also that an implementation may safely erase all coloured brackets outside hashes at run-time.

Non-goals Our focus in this report is on what mechanisms are required to guarantee abstraction-safety. We do not address the full ML language; instead, we focus on a core language based on simply-typed λ-calculus with abstract and manifest modules, although we argue that our formal system may be cleanly extended. Dynamic *rebinding* of identifiers within marshalled values is considered in [BHS$^+$03]. Moreover, we are not here concerned with low-level representations of marshalled values; we assume some fixed scheme for marshalling simply-typed values. Finally, we protect against confusion, not malice.

Outline We begin in Section 2 by examining scenarios in which values of abstract types are communicated between programs, identifying the desired constructs and behaviour from the programmer's point of view. Section 3 outlines our solution informally, shows why it provides the desired behaviour, and shows it can be implemented efficiently. In Section 4 we present a formal calculus, λ_{hash}, that covers the novel aspects of our solution. It describes networks of interacting separately-built modular programs. In Section 5 and Section 6 we discuss related and future work and conclude.

2 Abstraction and interaction: the desired behaviour

In this section we discuss the desired behaviour of marshalling in a distributed setting, with a series of informal examples in an ML-like language. Our solution, in the following section, shows how this ideal can be achieved.

We consider an ML-like language in which a program consists of two parts: first a sequence of module declarations, each of which can introduce abstract types; then an expression (the main body of the program). We are concerned with interaction between whole programs, usually built separately. This interaction is via network communication, though it could equally be via a persistent store; in either case, the underlying mechanism simply transmits byte strings. For concreteness, most of our examples involve networks consisting of two machines, `pauillac` and `glia`, running programs, say *Pa* and *Pb*. These network configurations are written `pauillac`[*Pa*] | `glia`[*Pb*]. It then suffices to consider a single communication channel (such as a TCP connection between fixed ports); the language has communication primitives

```
send : string->unit    receive : unit->string
```

We can now explore the desired behaviour of `marshal`($e:T$) and `unmarshal`($e:T$), which marshal to and from `string`.

2.1 Communication

The simplest example is that of sending a value of a non-abstract type between separately-built programs. Consider the two programs

```
P1a =  send (marshal (5 : int))
P1b =  print_int (unmarshal (receive ():int))
```

If these are built and then executed on the two machines the communication and `unmarshal` should succeed:

`pauillac`[*P1a*] | `glia`[*P1b*] $\quad\surd$

2.2 Respecting types

On the other hand, if one machine sends a `string` that the other attempts to unmarshal as an `int` there should obviously be a run-time failure.

```
P2a  =  send (marshal ("five":string))
P2b  =  print_int (unmarshal (receive ():int))
```

pauillac[*P2a*] | glia[*P2b*]   ×

To ease debugging, it is desirable for that failure to occur as early as possible (at unmarshal-time rather than when the `string` is used later) and to be trapped cleanly, raising an exception rather than giving unpredictable behaviour. The implementation must therefore send some form of type representation. The following examples explore the constraints on what this must be.

2.3 Respecting abstractions

Now consider an example with an abstract type. Here the `EvenCounter` module declares a type `EvenCounter.t` which has a representation type of `int` but externally is abstract, as declared in its signature. The operations of `EvenCounter` enforce the invariant that values of `EvenCounter.t` are always represented by even integers. If we allowed an arbitrary integer to be unmarshalled as an `EvenCounter.t` then the abstraction, and this invariant, would be broken; the `unmarshal` should therefore fail.

```
P3a = send (marshal(5:int))

P3b = module EvenCounter =
        struct                  sig
          type t=int              type t
          let start=0      :      val start:t
          let get x = x           val get:t->int
          let up x = x+2          val up:t->t
        end                     end
      print_int (EvenCounter.get
          (unmarshal (receive ()):EvenCounter.t))
```

pauillac[*P3a*] | glia[*P3b*]   ×

Marshalling from a different abstract type — say a `TripleCounter.t` — to `EvenCounter.t` should fail similarly.

2.4 Communication between completely-shared sources

For communication between two instances of the same build, which therefore have identical source code, the problem is relatively simple. Below, *P4* declares an abstract type `IntSet.t` of sets of integers, representing them as binary search trees. It makes a run-time determination of which machine it is on and then sends or receives an `IntSet.t`; the unmarshal should succeed. We will develop this example later — suppose this first implementation orders subtrees by `<`, and has a `union` operation that does not remove duplicate entries.

```
P4 =
module IntSet =
  struct
    type t = int tree
    let singleton = singleton-code   IntSetStruct
    let mem = mem-code
    ...
  end : sig
    type t
    val singleton : int -> t
    val mem : int -> t -> bool       IntSetSig
    val empty : t
    val add : int -> t -> t
    val union : t -> t -> t
  end
if ...on-machine-pauillac... then
  send (marshal (IntSet.singleton 17 : IntSet.t))
else
  if IntSet.mem 17 (unmarshal(receive():IntSet.t))
  then print "y" else print "n"
```

pauillac[*P4*] | glia[*P4*] $\surd$

By default this should still succeed even if the two machines execute different builds of the same source.

2.5 Communication between partially-shared sources

More generally, one may need communication between programs which share only some modules (perhaps ubiquitous standard libraries, or application-specific libraries). Here *P5a* and *P5b* share the `IntSet` module from before, but otherwise have different module declarations and main body expressions; their communication of an `IntSet.t` should succeed.

```
P5a =
module IntSet = IntSetStruct:IntSetSig
send (marshal (IntSet.singleton 17 : IntSet.t))

P5b =
module IntSet = IntSetStruct:IntSetSig
module M =
  struct let haszero x = IntSet.mem 0 x  end
  : sig  val haszero : IntSet.t -> bool  end
if M.haszero (unmarshal (receive () : IntSet.t))
then print "y" else print "n"
```

pauillac[*P5a*] | glia[*P5b*] $\surd$

2.6 Guaranteeing compatible invariants

In the previous example the two programs had syntactically identical `IntSet` implementations. Since `IntSet` does not depend on any other modules, this is a sufficient condition to guarantee that the two abstract types have compatible invariants, i.e. that any value of either will be correctly acted upon by the operations of the other. Moreover, it can be automatically checked, whereas compatibility of invariants cannot even be stated without specifing the behaviour of the two modules, and would then require general theorem-proving to verify. Note that it would not be sufficient to require that the two implementations use the same representation type, or even to require that the implementations are (in the absence of marshalling) observationally equivalent.

For example, suppose that *IntSetStructGt* is similar to *IntSetStruct* but orders subtrees with > rather than <.

```
P6a =
module IntSet = IntSetStructGt:IntSetSig
send (marshal (IntSet.add 0 (IntSet.add 1 (IntSet.add 2 IntSet.empty)) : IntSet.t))
```

When communicating with *P5b*, which contains the original *IntSetStruct*, the unmarshal should fail, as otherwise an erroneous result could be produced.

$$\texttt{pauillac}[P6a] \mid \texttt{glia}[P5b] \qquad \times$$

Later we shall see that a mechanism for intentionally circumventing this restriction, in a controlled way, is sometimes desirable.

2.7 Respecting names (when necessary)

In some cases one has modules with identical implementations that nonetheless provide conceptually different abstract types, for example in the Euro and Pound modules below. Unmarshalling should respect this difference, so the example should fail (just as, within a single ML program, types Euro.t and Pound.t would be incompatible).

```
P7a =
module Euro =
 struct type t=int  let of_int x = x      ... end
 : sig  type t      val of_int : int -> t ... end
send (marshal (Euro.of_int 17 : Euro.t))

P7b =
module Pound =
 struct type t=int  let of_int x = x      ... end
 : sig  type t      val of_int : int -> t ... end
unmarshal (receive (): Pound.t)
```

$$\texttt{pauillac}[P7a] \mid \texttt{glia}[P7b] \qquad \times$$

This restriction is not always useful (e.g. whether an integer set module is called IntSet or Set_Int is likely irrelevant), so the language should support some syntactic way of indicating whether a module name is significant or not.

2.8 Module dependencies

Consider now modules that depend on abstract types declared by other modules. In *P8a* below there is a module IntSet, providing an abstract type IntSet.t, followed by a module SummedIntSet, providing an abstract type of sets of integers augmented with a running sum. The expression part constructs, marshals and sends a value of the SummedIntSet.t abstract type. This SummedIntSet depends on IntSet in three ways: (1) IntSet.t occurs in its representation type IntSet.t $*$ int; (2) IntSet.t occurs in the type of an operation in its signature; and (3) operations from IntSet occur in the definitions of its operations. Any such dependency means that substantive changes to the definition of IntSet must propagate through to give distinct SummedIntSet.t types. On the other hand, any module declarations that are not (transitively) depended upon should have no effect on SummedIntSet.t.

For example, consider also *P8b* below. It has exactly the same text as SummedIntSet but a different implementation of IntSet — suppose *IntSetStruct'* has a different representation type from *IntSetStruct*, or the same representation but incompatible invariants, or different externally-observable behaviour. The *P8a* and *P8b* SummedIntSet.t types should be incompatible, so the unmarshal should fail.

```
P8a =
module IntSet = IntSetStruct:IntSetSig
module SummedIntSet =
  struct
    type t = IntSet.t * int
    let empty = (IntSet.empty,0)
    let sum (x,y) = y
    ...
  end : sig
    type t
    val empty : t
    val singleton : int -> t
    val sum : t -> int
    val to_intset : t -> IntSet.t
    ...
  end
send(marshal((SummedIntSet.singleton 2)
                    : SummedIntSet.t ))
P8b =
module IntSet = IntSetStruct':IntSetSig
module SummedIntSet = ...same text as above...
SummedIntSet.sum
  (unmarshal (receive () : SummedIntSet.t))
```

pauillac[*P8a*] | glia[*P8b*] ×

2.9 Mirroring local type sharing: manifest types, functors

The examples in this and subsequent subsections are not covered by the formal calculus of Section 4. Nonetheless we argue informally in Section 6.2 how they can be treated by straightforward extensions of our main techniques and earlier work.

ML module systems include parametric modules, known as *functors*, for large-scale software structuring and code reuse. In the single-program world there are a number of subtle type-equality issues, related to how generative functors are, and how one can express type sharing constraints [MTH90, Ler94, HL94, SH00]. Our marshalling primitives should correctly reflect these subtleties in inter-program communication.

For example, the module `SummedIntSet` above, which explicitly references `IntSet`, might be re-expressed in terms of a functor `F` which takes any argument structure `U` with interface *IntSetSig* and builds a `SummedIntSet`:

```
module IntSet =  IntSetStruct:IntSetSig
module F = functor (U:IntSetSig) ->
  struct type t=U.t*int ... end
  : sig  type t         ... end
module SummedIntSet = F(IntSet)
```

The functor `F` generates an abstract type, so we must consider when that type should be compatible with others. If two separate programs contain this preamble, they should be able to exchange values of their respective `SummedIntSet.t` types. This mirrors the behaviour of OCaml's *applicative functors* [Ler95], in which another instance of the application `F(IntSet)` within the same program would have a type compatible with `SummedIntSet.t`.

Should the functorised and non-functorised (*P8a*) `SummedIntSet.t` be compatible? Again following existing module systems, we should make them incompatible, as otherwise static type equality would depend on module substitution.

Type *sharing* allows functors to express type equalities between their argument and result; unmarshalling should respect these static type equalities. The example `F'` below constructs a type `t` but, in contrast to `F`, does not make that type abstract; instead it makes it manifestly equal to the product of its argument type `U.t` and `int`.

```
module F’ = functor (U:IntSetSig) ->
  struct type t=U.t*int ... end
  : sig  type t=U.t*int ... end
```

The application of `F’` to a module `IntSet` creates a static type equality `F’(IntSet).t==IntSet.t*int`, which should also be admitted at run-time.

2.10 Breaking abstractions (simple bidirectional case)

In ongoing software evolution, implementations of an abstract type may need to be changed, to fix bugs or add functionality, while values of that type exist on other machines or in a persistent store. It is often impractical to simultaneously upgrade all machines to a new implementation version.

A simple case is that in which the representation of the abstract type is unchanged and where the programmer asserts that the two versions have compatible invariants, so it is legitimate to exchange values in both directions. This may be the case even if the two are not identical, e.g. for an efficiency improvement or bug fix. Here there should be some mechanism for forcing the old and new types to be identical, breaking the Section 2.6 restriction.

For example, consider the improved *IntSetStructDeDup* implementation below, in which the operations are similar to *IntSetStruct*, the only difference being that `union` removes duplicates. The compiler cannot verify that *IntSetStructDeDup* has all the semantic properties that the programmer requires of *IntSetStruct*. Hence we provide a way of explicitly declaring that these modules provide compatible types. In *P10a* below, `IntSet’.t` is made equal to `IntSet.t` by the *strong coercion* `...with t =! IntSet.t`. The compiler checks only that the old and new types have compatible representations (here `int tree`), but should respect further abstractions *within* those representation types. This is based on our earlier work of [Sew01].

```
P10a =
module IntSet = IntSetStruct:IntSetSig
module IntSet’ =
  struct                      }
    type t = int tree         } IntSetStructDeDup
    ...improved operations... }
  end                         }
: IntSetSig with t =! IntSet.t
send (marshal (IntSet’.singleton 17 : IntSet’.t))

P10b =
module IntSet = IntSetStruct:IntSetSig
if IntSet.mem 0  (unmarshal(receive():IntSet.t))
then print "y" else print "n"
```

pauillac[*P10a*] | glia[*P10b*] $\surd$

2.11 Breaking abstractions (directed case)

In the more complex case where the old and new invariants are not compatible, or where the two representation types differ, the programmer will have to write an upgrade function. The same strong coercion can be used to make this possible.

For example, suppose we have a program that uses stored values of *IntSetStruct* and we wish to upgrade both the implementation and the stored values, changing the representation type from binary search trees to red-black trees. The new implementation would have a module declaration:

```
module IntSet2  =
  struct
    type t = int rbtree    } IntSetStructRBT
    ...
  end
  : IntSetSig
```

A program to upgrade the stored values can be expressed as below, with an `Upgrade` module that has both types, coerced respectively to be equal to the old and new abstract types. (We are not proposing machinery to *automatically* apply the `upgrade` function.)

```
module IntSet  = IntSetStruct:IntSetSig
module IntSet2  = IntSetStructRBT : IntSetSig
module Upgrade =
  struct
     type t1 = int tree
     type t2 = int rbtree
     let upgrade = ...
  end : sig
    type t1
    type t2
    val upgrade : t1 -> t2
  end
    with  t1 =! Intset.t and t2 =! Intset2.t
...map Upgrade.upgrade over the stored values...
```

Note that the coercion does not require the signature of `Upgrade` to coincide with those of `IntSet` and `IntSet2`. The compiler only checks that `IntSet.t` is represented by `int tree` and `IntSet2.t` by `int rbtree`.

2.12 Forcing generativity

Dually, sometimes it is desirable to *force* a type change between builds even when the code remains identical, to prevent confusion between old and new communicated values. For example, one may have several distributed deployments of the same application which should be kept logically isolated.

2.13 Effectful module initialisation

In our previous examples the components of modules are all values. Generalising this to arbitrary expressions (as ML does), an abstract type definition can be dependent on some computation with side effects.

For example, consider an `NCounter` module that reads its step value from standard input when initialised; the invariant of any instance is then that any value of its `NCounter.t` is a multiple of this step. Two instances of the module can obviously have different invariants, and so marshalling from one to another should fail. Thus each run of a program containing `NCounter` should have an incompatible type `NCounter.t`.

2.14 Marshalling functions and rebinding

In this report we deal only with marshalling of *closed* values; the semantics ensures that all module and expression declarations are substituted in before a `marshal` operation takes place. Marshalling of functions is therefore semantically straightforward.

A full language should, however, provide some form of *dynamic rebinding* of identifiers when they are unmarshalled, both to achieve the desired semantics where local resources have different behaviour in different contexts, and for performance reasons where much code is shared (and so should not be communicated). The paper [BHS+03] addresses dynamic rebinding, in the absence of type abstraction.

3 Solution: hash types as global names

This section introduces our solution informally, from both implementation and semantic viewpoints.

As we have seen, type-safe and abstraction-safe unmarshalling requires some run-time type representation in marshalled values, to permit a dynamic type comparison.

Our solution is based on the observation that hashing module definitions provides a global namespace for abstract types: if an identical module is hashed during builds of two different programs at different sites, the same hash will be obtained. Thus the programs share names for any abstract type provided they share the source code of the module that declares the type (and of its dependencies); no communication (e.g. of GUIDs) is needed at build time.

We regard hashes literally as types — hashes appear as a clause in the type grammar. They do not appear in source programs, but are inserted during compilation; as we shall see in more detail, the compiler replaces occurrences of an abstract type such as `IntSet.t` by the hash of the definition of `IntSet` that is in scope. Semantically, we work with ideal hashing, with a formal syntactic construction `hash(...)`. Implementations would realise this with an actual hash function; we discuss the low-level properties of hashes in Section 3.5.

At run-time, after this compile-time type substitution, the types in `marshal(`e`:`T`)` and `unmarshal(`e`:`T'`)` are closed, without free module identifiers or type variables. They can therefore be easily represented as byte strings, communicated across the network or stored in a persistent store, and can be compared with simple string equality.

We ensure that this dynamic equality precisely mirrors the static notion of provable type equality by carefully tuning the way in which hashes are generated and used; we show below that our system achieves this. Unmarshalling is therefore not only type-safe, but also abstraction-safe.

The standard operational semantics for abstract types forgets about abstraction as computation proceeds, substituting in representation types and operations. Here, in contrast, we need a run-time semantics that maintains abstraction throughout, both (1) so that our type preservation theorems tell us that abstractions are not broken; and (2) to support the proof that static and dynamic type equality coincide. After a module is reduced away, module code (which may see through the abstract type of that module) is intermixed with body code (which must treat the type as abstract). We therefore use a syntactic construct, *coloured brackets*, adapted from the work of [GMZ00], to delimit the regions in which different type equivalences hold. This is not purely a proof technique, however: in some subtle cases the coloured brackets within hashes are needed in compile-time hash generation to correctly distinguish abstract types that would otherwise be aliased. We show that implementations can erase coloured brackets outside hashes after compilation.

3.1 Simple examples

We illustrate the use of hashes in a simple case by referring back to the example of Section 2.4, in which a single program, *P4*, was built and run on the two machines. The build process is modelled in our semantics by type-checking, as usual, followed by reductions that substitute out module definitions, inserting hashes as required. Hash generation is deterministic, and hence the result of building *P4* on the two machines is identical. The program has a single module definition. It has a compile-time reduction as below, to an 'executable' *P4'*. (Note that for clarity of exposition, we omit coloured brackets from all reductions until Section 3.3; the example reductions as stated are not all type-preserving without them.)

```
P4 =
module IntSet = IntSetStruct:IntSetSig
if ...on-machine-pauillac... then
  send (marshal (IntSet.singleton 17 : IntSet.t))
else
  if IntSet.mem 17 (unmarshal(receive():IntSet.t))
  then print "y" else print "n"
```

$\longrightarrow_{c}$ (compilation)

```
if ...on-machine-pauillac... then
  send (marshal (singleton-code 17 : h ))
else
  if mem-code 17 (unmarshal (receive () : h))
  then print "y" else print "n"
= P4'
```

where
h = hash(module IntSet=*IntSetStruct* : *IntSetSig*,t).

Here the definitions of `IntSet.singleton` and `IntSet.mem` have been substituted for their occurrences, and the global name h has been substituted for type `IntSet.t`. Notice that h is constructed from the entire definition of `IntSet`, including the textual name `IntSet`, the implementation structure *IntSetStruct*, the interface *IntSetSig*, and the type field name `t`. In this simple example `IntSet` has no dependencies, so one can think of hashing its source text; we will discuss later the more interesting case of modules with dependencies, and also the question of exactly what form the hash function takes. Our liberal use of substitution is, of course, a semantic device — in practice compilation would use other representations.

At run-time, the two machines `pauillac` and `glia` execute their independently-compiled copies of *P4'*. Their shared knowledge of the hash h acts as a certificate that they may safely share values of their respective abstract types `IntSet.t` and `IntSet.t`.

```
  pauillac[P4'] | glia[P4']
⟶*      (local computation on pauillac and glia)
  pauillac[send(marshal(singleton-code 17:h))]
 | glia[if mem-code 17 (unmarshal(receive():h))
       then print "y" else print "n"]
⟶*      (local computation on pauillac, to get v)
  pauillac[send(marshalled( v:h ))]
 | glia[if mem-code 17 (unmarshal(receive():h))
       then print "y" else print "n"]
⟶       (communication)
  pauillac[ () ]
 | glia[if mem-code 17 (unmarshal(marshalled(v:h):h))
       then print "y" else print "n"]
⟶       (on glia: dynamic type check h=h, succeeds)
  pauillac[ () ]
 | glia[if mem-code 17 v
       then print "y" else print "n"]
⟶*      (on glia: computation, prints "y")
 pauillac[ () ] | glia[ () ]
```

Ultimately, only strings may be communicated across a network. The notation `marshalled` $(v:T)$ denotes a string literal containing representations of value v and its type T. This is only meaningful, and only used, where v and T are both closed.

Notice that the dynamic check is simple: just that the type h sent from `pauillac` is identical to the type h written into the `unmarshal` on `glia` at compile time. Yet, by virtue of the construction of these hashes, this is sufficient to guarantee both type-safety and abstraction-safety.

Consider now the programs of Section 2.1–2.7. How do hashes of modules provide the desired behaviour? In the case of concrete types, the comparison is obvious. For *P1*, `int`=`int`; for *P2*, `string`$\neq$`int`; for *P3*, for no hash h do we have `int`=h. As we have already seen, in the *P4* case the two programs share an identical hash h. For *P5*, in *P5a* and *P5b* the computed hash h for `IntSet.t` is identical (in fact the same h as above). Thus the h substituted for `IntSet.t` in *P5a*'s call to `marshal` will be identical to that in *P5b*'s call to `unmarshal`, and the communication will again succeed, exactly as we desire.

Although *P6a* (Section 2.6) contains a type `IntSet.t`, it is clear that the hash h' of the modified module `IntSet` differs from the hash h of the original module, correctly reflecting the difference in the modules' behaviour. The two programs will, correctly, be unable to communicate; an exception will be raised at the point of the `unmarshal`.

The example of Section 2.7 shows why one might wish the textual name (in general, the path) of a module to be included in its hash, along with the module body. The two modules `Euro` and `Pound` are identical in all but name, and so a hash that did not include the name would treat them as interchangeable, clearly leading to dangerous economic confusion, and furthermore differing from the usual semantics of ML-like languages. On the other hand, the programmer should also be able to specify that a name is *not* to be considered part of the module's identity. This can be done simply by having an additional form of module declaration, `module* N = ...` , for which hashing uses a canonical name `*`, not admissible in source programs, instead of the actual name N. In this simple scheme both sender and receiver must use the *'d form, of course.

3.2 Module dependencies

The example of Section 2.8 shows that the same module text defines a different abstract type if its dependencies change, which means that the hash of a module must depend on the hashes of its dependencies. In our substitutive reduction semantics, type dependencies are handled automatically: we have substituted hashes for any types of earlier modules before constructing the hash of a module that depends on them. We shall see how term dependencies are also automatically taken into account. Consider the following (a simplification of *P8a*):

```
module A=struct type t=bool    let x=true   end
         : sig type t          val x:t      end
module B=struct type t=A.t*int let x=(A.x,3) end
         : sig type t          val x:t       end
send (marshal (B.x : B.t))
```

$\longrightarrow_c$ (compilation)

```
module B=struct type t=h*int let x=(true,3) end
         : sig type t      val x:t        end
send (marshal (B.x : B.t))
```

$\longrightarrow_c$ (compilation)

```
send (marshal ((true,3) : h'))
```

where

```
h = hash (
module A=struct type t=bool   let x=true   end
         : sig type t        val x:t      end,t)
h' = hash (
module B=struct type t=h*int  let x=(true,3) end
         : sig type t        val x:t        end,t)
```

Here the hash h' for B is constructed after the hash h for A has been substituted for A.t, and after the term part true has been substituted for A.x. It is clear that if A changed, h would change, and so h' would change. This would still be true in the (unlikely) case that B mentions A.t but not A.x.

We must also ensure h' depends on h in the (common) case that B mentioned A.x but not A.t, i.e. where A is used in B only to implement an internal computation. The coloured brackets of the following section will conveniently suffice for this.

3.3 Abstraction-preserving reduction

Some reductions in Section 3.1, 3.2 require non-standard type equalities to make them type-preserving. For example, to type the intermediate state in Section 3.2 we must have (true,3) of type h*int, hence we need a type equality identifying h with its representation type bool.

We could allow this type equality to be used anywhere, but instead prefer to delimit more precisely which subterms can see through any particular abstraction. We introduce *coloured brackets*, adapted from the work of [GMZ00], during module reduction. In the previous example, the first reduction will actually replace $(\mathtt{A.x}, 3)$ by $([\mathtt{true}]^{h}_{h}$,3) instead of just (true,3). The brackets serve two purposes. First, the lower annotation (the *colour*) is a hash h, indicating that the additional type equivalence h == bool is available when typing the *inside* of the bracketed expression. This equivalence is drawn from the structure of h, viz. h=hash(module A= struct type t=bool...end:...,t). Thus, inside the brackets we have true:h. Second, the upper h annotation is the type of the bracketed expression as seen from the *outside*, thus reduction is type preserving. (One would often have a more complex type in the upper annotation, not just a hash, e.g. $[\mathtt{(true,3)}]^{h\mathtt{*int}}_{h}$.)

The reduction semantics of our formal system moves brackets around as required to ensure that abstraction is preserved throughout reduction, and so our type preservation result (Theorem 4.1) covers abstraction. If we did not use brackets but allowed hash type equalities to be used freely, abstraction would become invisible after reduction. The use of brackets also simplifies the statement of our result relating static and dynamic type equality (Theorem 4.7). Moreover, when compiling a module that refers to a term field of a previous one, the presence of brackets ensures that the hash of the later module does indeed depend on the hash of the earlier module.

3.4 Modest implementation demands

Few changes are required in an ML-like language to support the strategy outlined above.

Theorem 4.5 shows that type checking is decidable and that hashes play no role in compile-time type-checking of source code. In particular, we can use traditional type checking and inference algorithms essentially unchanged. Compile-time reduction only builds hashes, without ever looking inside one. Run-time reduction only ever compares hashes by string equality.

Theorem 4.6 shows that almost all coloured brackets and type information can be erased before run-time, with the exception of course of marshal and unmarshal type annotations, and brackets within hashes.

ML-like languages usually support separate compilation of modules. Typically, a compilation phase takes a module and the signatures of the modules it imports and generates code parameterised by these dependencies. For λ_{hash}, the compilation phase would also generate a hash parameterised by the hashes of the imported modules, in other words a hash-to-hash function. An appropriate compositional implementation of hashing must be used to make these efficiently representable. Typically, linking instantiates the parameterised code with jumps to the code of previous modules. For λ_{hash}, the linking phase would do two further things. First, it would patch the type annotations for `marshal` and `unmarshal` in the code by replacing references to module types by their hashes. Second, it would calculate the hash of the module by applying the hash-to-hash function (generated by compilation) to the hashes of previous modules.

3.5 Low-level details of hashes

In our semantics, we work with *ideal* hashing, taking a free constructor `hash(...)` which can be applied to elements of the abstract syntax. We can think of `hash` as a function whose injectivity guarantees abstraction-safety. To avoid communicating large quantities of source code, an implementation would reify `hash` with a fixed-length hash function, giving a safety guarantee that is only as strong as the probability of the absense of collisions.

This must be chosen so that (1) collisions are rare, and (2) hashes are not too costly to compute.

Both MD5 (RFC1321, 128-bit) and SHA-1 (RFC3174, 160-bit) are sufficiently cheap, and may be considered random functions for this application [Rob96]. Let us consider the likelihood of collisions. For n abstract types and N possible hash values, the probability of a collision is approximately $n^2/2N$. Pessimistically assuming 10^{10} programmers in the world, writing 300 lines of code per day with one abstract type per 100 loc, the probability of a collision in a century of abstract types (using MD5) would then be $(10^{15})^2/2^{129} \approx 10^{-9}$. This is much less than the probability of a cosmic-ray-induced processor error in this period.

It may be desirable to have an absolute guarantee of type-safety, while accepting probabilistic abstraction-safety. To achieve this, one could pair hashes with the corresponding underlying representation types. At the other extreme, one could accept a probabilistic guarantee even at simple types, by sending only a hash of the marshalled type. These choices must depend on a risk assessment.

Note that our proposal is aiming to protect only against accidental errors during programming and software deployment, not against malicious attack, and so we are not concerned with deliberate searches for collisions. Protecting against spoofed messages requires largely orthogonal techniques, e.g. message signatures and/or encryption, that are not in the scope of this report. Moreover, we do not address the problem of communication between untrusting peers, where one must check not just that the type advertised by the peer is compatible with the local type, but also the validity of the byte string's claim to represent a value of the advertised type (see, e.g., [PS00]).

We hash elements of the abstract syntax, not concrete syntax, for two reasons. Firstly, it ensures hashes are not dependent on, e.g., the choice of newline or newline/CR, or on comments. Secondly, it fits well with the rest of the semantics — recall we must calculate hashes of modules that are the results of module substitutions. In practice optimised calculations would be possible, without requiring the explicit construction of canonical representatives of abstract syntax elements.

Hashing abstract syntax, which we take up to alpha-equivalence, has the (benign) consequence that abstract type equality is not dependent on the names of function parameters. We have both internal (alpha-convertible) and external *module* names in the semantics; external names must be meaningful between programs.

4 Formal system

Our calculus describes networks of machines. Each machine executes a program; a program consists of a sequence of module declarations followed by an expression. The expression language consists of a simply-typed call-by-value λ-calculus with module field references, marshalling, and communication of strings.

Consider a program containing a module declaring an abstract type. There is an abstraction boundary between the module's body and the rest of the program. Inside the boundary, the type's representation is visible; thus the type is said to be *transparent*. Outside, the type's representation is not visible, thus the type is *opaque*. Our calculus tracks this abstraction boundary as reduction proceeds. Compilation replaces the abstract type by a *hash* h and wraps the code e that comes from inside the module definition with *coloured brackets* decorated by h, as in $[e]_h^T$. The distinction between opaque and transparent views is therefore witnessed by the brackets: inside the brackets, we view h as transparent; when outside h is opaque.

In order to express this distinction in our inference rules, we decorate each judgement with a *colour* hm, as in $E \vdash_{hm} e{:}T$. The colour has one of two forms: it can be a hash h, in which case h is transparent and all other hashes are opaque; or it can be the empty colour $\bullet$, in which case all hashes are opaque.

The appendices present a systematic development of the formal definitions and theorems complete with proofs.

4.1 Relation to the informal discussion

For brevity, we take a module language in which structures are type/term pairs, rather than general dependent records from the earlier informal development. The following table summarises the correspondence between the informal and formal module syntax.

`struct type t =` T_0 `let y =` $v^\bullet$ `end`	$\leftrightarrow$	$[T_0, v^\bullet]$
`sig type t val y :` T `end`	$\leftrightarrow$	$[X{:}\mathbf{Type}, T]$
`sig type t =` T_1 `val y :` T `end`	$\leftrightarrow$	$[X{:}\mathbf{Eq}(T_1), T]$

We split module names into two parts, an external name N and an alpha-convertible name U. We write module declarations as $\mathbf{module}\, N_U = M{:}S \;\mathbf{in}\; m$, where U binds in m and N neither binds nor is subject to binding. The user would write only one identifier, which would be used for both. External names play no role in the static type system; they are used in hash construction and hence in dynamic type checks.

The formal system omits `=!` coercions and run-time generativity, which should be straightforward extensions. Functors are also omitted, though we include most of the technical machinery they require, expressing abstract and manifest types in signatures using singleton kinds. In Section 6.2 we propose extensions for treating these omissions.

4.2 Syntax

We let x, X and U range over expression, type and module variables.

Networks:
$n ::= 0 \;\mid\; m \;\mid\; n|n$

Machines (whole programs):
$m ::= e \mid \mathbf{module}\, N_U = M{:}S \;\mathbf{in}\; m$ (U binds in m)

Modules:
$M ::= [T, v^\bullet]$ structure ($v^\bullet$ is a value)
$S ::= [X{:}K, T]$ signature (X binds in T)

Types:

$T ::= \mathrm{UNIT} \mid \mathrm{INT} \mid \mathrm{STRING}$	base types
$\mid X \mid T \to T \mid T * \ldots * T$	variable, function, product
$\mid U.\mathrm{TYPE}$	type part of a module
$\mid h$	hash

Hashes:
$h ::= \mathbf{hash}(N, M{:}[X{:}\mathbf{Type}, T])$ hash
$hm ::= h \mid \bullet$ colour ("hash maybe")

Kinds:

$K ::= \mathbf{Type}$	kind of all types
$\mid \mathbf{Eq}(T)$	kind of types statically equal to T

Expressions:

$e ::= () \mid \underline{n}$	unit, integers
$\mid (e, ..., e) \mid \mathbf{proj}_i\, e$	tuple, projection
$\mid x \mid \lambda x{:}T.e \mid e\ e$	lambda calculus (x binds in e)
$\mid U.\mathrm{term}$	value part of a module
$\mid \mathbf{mar}\,(e{:}T)$	marshalling primitive
$\mid \mathbf{unmar}\ e{:}T$	unmarshalling primitive
$\mid !\, e \mid ?$	send and receive
....	
$\mid \mathbf{marshalled}\,(e{:}T)$	result of marshalling
$\mid \mathbf{UnmarFailure}$	exception caused by **unmar**
$\mid [e]^T_{hm}$	coloured bracket

User source programs are closed terms of the m grammar which do not contain any of the constructs below the dotted lines.

Values v^{hm} are indexed by a colour. They are defined formally below; they include usual λ-calculus values, $\mathbf{marshalled}\,(v^\bullet{:}T)$ and "necessary" brackets around values. For closed $v^\bullet$, the value $\mathbf{marshalled}\,(v^\bullet{:}T)$ is a string, the sequence of bits that represents the value $v^\bullet$ and the type T.

We work up to alpha-conversion. We write substitutions as follows: $\{x{\leftarrow}e\}A$ replaces x by e in A; we also define substitutions on module components, as in $\{U.\textsc{type}{\leftarrow}T,\, U.\mathrm{term}{\leftarrow}e\}A$.

4.3 Static and dynamic semantics

The static type system for programs has judgements for subkinding, type equality, and subsignaturing relations. Module structures M and names U have signatures S, expressions and machines have types T, and types have kinds K. The system also defines correctness of colours hm, environments E, kinds K, and signatures S.

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

The typing rules are largely standard; the novel rules will be explained below. Recall that judgements are annotated by a colour hm, i.e. an optional hash — the idea being that derivations of judgements annotated by a hash h can make use of the equality between the abstract type h and its implementation.

Type environments may contain bindings for module, type and expression variables. Earlier variables bind in later types, kinds and signatures.

$E ::= \ \mathbf{nil} \mid E, x{:}T \mid E, X{:}K \mid E, U{:}S$

Static typing of networks, $\vdash n\,\mathrm{ok}$, simply means that all machines are well-formed.

We define compile-time reductions $m \longrightarrow_{\mathsf{c}} m'$ of machines (performed after type checking), and run-time reductions $e \longrightarrow_{hm} e'$ and $n \longrightarrow n'$ for expressions and networks.

4.3.1 Singleton kinds

Following [Ler94, HL94, SH00], we use singleton kinds to handle abstract and concrete signatures in a uniform way. We have two families of kinds: $\mathbf{Type}$ is the kind of all types; and, for any type T, $\mathbf{Eq}(T)$ is the *singleton kind* of all types that are provably equal to T.

A module consists of a structure $[T_0, v^\bullet]$ and a signature $[X{:}K,\, T]$. The structure has a representation type T_0 and a value $v^\bullet$ — think of a tuple of operations. This $v^\bullet$ must have the type $\{X{\leftarrow}T_0\}\,T$, and the implementation type T_0 must have the kind K. This is made precise by the following rule:

$$\frac{\begin{array}{ll} E \vdash_{hm} T_0{:}K & E, X{:}K \vdash_{hm} T{:}\mathbf{Type} \\ E \vdash_{hm} v^\bullet{:}T' & E, X{:}\mathbf{Eq}(T_0) \vdash_{hm} T' == T \end{array}}{E \vdash_{hm} [T_0, v^\bullet]{:}[X{:}K,\, T]}\ (\mathsf{MS.struct})$$

For an abstract module we have $K = \mathbf{Type}$, revealing no information about the representation type, whereas for a concrete module, commonly $K = \mathbf{Eq}(T_0)$, revealing it. This is captured with the type equality relation: in the context of a module declaration $\mathbf{module}\, N_U = M{:}[X{:}K,\, T]\ \mathbf{in}\ \_$, one can use the path $U.\mathrm{TYPE}$ to refer to the type part of the module. If it is concrete, with $K = \mathbf{Eq}(T_0)$, one can further use the type equality $U.\mathrm{TYPE} == T_0$, whereas if it is abstract $U.\mathrm{TYPE}$ is typically not equal to any other type.

The subkinding relation $K <: K'$ places $\mathbf{Type}$ above all singleton kinds. This is used to define subsignaturing and hence, using subsumption, allows a concrete module can be used as if it had an abstract signature.

4.3.2 Hash formation, type equality of hashes

At run-time, we need globally meaningful type names for abstract types, corresponding to the $U.\mathrm{TYPE}$ paths used in compile-time type checking. We construct these global names by hashing (well-typed) closed abstract modules, together with the associated external name.

$$\frac{\vdash_{\bullet} [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T]}{\vdash \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T])\ \mathrm{ok}}\ \textsf{(hmok.hash)}$$

As explained informally earlier, judgements annotated by a hash permit an additional type equality: under the colour $h = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T])$, h is equal to its implementation T_0:

$$\frac{E \vdash_h \mathrm{ok}}{E \vdash_h h == T_0}\ \textsf{(Teq.hash)}$$

These two rules examine the internal structure of hashes, which might be thought to be computationally problematic. However, while they are semantically necessary, they play no role in user program type-checking (Theorem 4.5) or in execution.

4.3.3 Compile-time reduction and coloured brackets

Module reduction constructs the type representations that will be used at run-time in marshalling and unmarshalling. Reducing a concrete module is simple: we replace references to its type component by its manifest type, and references to its term component by the value inside the module.

$$\mathbf{module}\, N_U = [T_0, v^\bullet]{:}[X{:}\mathbf{Eq}(T_1),\, T]\ \mathbf{in}\ m \quad \longrightarrow_{\mathsf{c}} \quad \{U.\mathrm{TYPE}{\leftarrow}T_1,\, U.\mathrm{term}{\leftarrow}v^\bullet\}m \quad \textsf{(mred.Eq)}$$

When it comes to abstract types, things are more interesting. Given an abstract module declaration $\mathbf{module}\, N_U = [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T]$, we normally have no way of referring to its type other than by name, i.e. $U.\mathrm{TYPE}$. However U is not meaningful on other machines, which motivates the introduction of the *hash* of the module, i.e. $h = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T])$. Then module reduction replaces references to the type component by h. References to the term component are replaced by the value suitably protected by *h-coloured brackets*, which embody the abstraction boundary around the module's body as discussed above.

$$\mathbf{module}\, N_U = [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T]\ \mathbf{in}\ m \quad \longrightarrow_{\mathsf{c}} \quad \{U.\mathrm{TYPE}{\leftarrow}h,\, U.\mathrm{term}{\leftarrow}[v^\bullet]_h^{\{X\leftarrow h\}T}\}m \quad \textsf{(mred.Type)}$$

In general, in a bracket expression $[e]_{hm}^T$, the lower annotation hm is a colour that indicates what type equalities may be used to type e. If $hm = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type},\, T])$, then the equality $hm == T_0$ is available when typing e, through (Teq.hash) (Section 4.3.2). If $hm = \bullet$, e is typable without any extra equalities. The upper annotation T is the externally visible type of e. The following rule (the only typing rule that mentions brackets) shows this colour change formally.

$$\frac{E \vdash_{hm'} T{:}\mathbf{Type} \qquad E \vdash_{hm} e{:}T}{E \vdash_{hm'} [e]_{hm}^T{:}T}\ \textsf{(eT.col)}$$

4.3.4 Expression reduction

Expression reduction is based on a standard call-by-value λ-calculus semantics. In this subsection, we give the function application rule and bracket-pushing rules. In later subsections, we show the rules for marshalling and communication.

As we show in Theorem 4.6, brackets can be erased before run-time reduction. However, the brackets' presence is necessary for type preservation (Theorem 4.1). Given their presence, we need reduction rules to "push" them inwards so that the brackets do not interfere with computationally significant reductions (Theorem 4.3). To describe the bracket pushing rules, and to achieve type preservation, it is necessary to index the reduction relation, class of values, and reduction contexts by colours.

We write v^{hm} for a value of colour hm. Brackets may appear in a value when used to build a value of an abstract type out of a value of the corresponding implementation type, for example $[3]^{h}_{h}$, where the implementation type of h is INT.

$$\begin{array}{rl} v^{hm} ::= & \underline{n} \mid () \mid (v^{hm}, ..., v^{hm}) \mid \lambda x{:}T.e \\ \mid & \mathbf{marshalled}\,(v^{\bullet}{:}T) \mid [v^{h_1}]^{h_1}_{h_1} \text{ where } h_1 \neq hm \end{array}$$

The following bookkeeping rules push brackets with manifestly decomposable types inside expressions, and remove them where not necessary.

$$\begin{array}{rlll} [\underline{n}]^{\text{INT}}_{hm'} & \longrightarrow_{hm} & \underline{n} & \text{(ered.col.int)} \\ [()]^{\text{UNIT}}_{hm'} & \longrightarrow_{hm} & () & \text{(ered.col.unit)} \\ [(v_1^{hm}, ..., v_j^{hm})]^{T_1 * ... * T_j}_{hm'} & \longrightarrow_{hm} & ([v_1^{hm}]^{T_1}_{hm'}, ..., [v_j^{hm}]^{T_j}_{hm'}) & \text{(ered.col.tuple)} \\ [\lambda x{:}T.e]^{T' \to T''}_{hm'} & \longrightarrow_{hm} & \lambda x{:}T'.[\{x \leftarrow [x]^{T}_{hm'}\}e]^{T''}_{hm'} & \text{(ered.col.fun)} \\ [\mathbf{marshalled}\,(v^{\bullet}{:}T)]^{\text{STRING}}_{hm'} & \longrightarrow_{hm} & \mathbf{marshalled}\,(v^{\bullet}{:}T) & \text{(ered.col.marshalled)} \\ [[v^{h_1}]^{h_1}_{h_1}]^{h_1}_{h_2} & \longrightarrow_{hm} & [v^{h_1}]^{h_1}_{h_1} \quad \text{if } h_1 \neq h_2 \wedge h_2 \neq hm & \text{(ered.col.col)} \\ [v^{hm_1}]^{h_2}_{hm_1} & \longrightarrow_{hm} & v^{hm_1} \quad \text{if } hm_1 = hm \vee hm_1 = \bullet & \text{(ered.col.le)} \end{array}$$

Function application introduces brackets to protect the argument, since the formal parameter may itself be used under a bracket in the body of the function. This is a variant of [GMZ00], where the formal parameter has to be used at the colour of the function itself.

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

4.3.5 Marshalling

As in [ACPP91], $\mathbf{mar}\,(e{:}T)$ "tags" the value of e with a type annotation T, producing a result of type STRING. The dual construct $\mathbf{unmar}\; e{:}T$ produces a value of type T, which the type tag in e must (dynamically) match.

$$\frac{E \vdash_{hm} e{:}T}{E \vdash_{hm} \mathbf{mar}\,(e{:}T){:}\text{STRING}} \quad \text{(eT.mar)}$$

$$\frac{E \vdash_{hm} T{:}\mathbf{Type} \qquad E \vdash_{hm} e{:}\text{STRING}}{E \vdash_{hm} (\mathbf{unmar}\; e{:}T){:}T} \quad \text{(eT.unmar)}$$

There is a subtlety here: in the conclusion of (eT.mar), the fact that e has the type T may require the extra type equality provided by hm. Hence we introduce $\mathbf{marshalled}\,(e'{:}T)$, which requires the argument to be not only closed but typable in $\bullet$, i.e. everywhere. Reduction transforms $\mathbf{mar}\,(v^{hm}{:}T)$ into $\mathbf{marshalled}\,([v^{hm}]^{T}_{hm}{:}T)$, where the brackets serve to ensure that any type equality provided by hm is always available to type v^{hm} (even after sending the marshalled value to another machine). Note that before reducing $\mathbf{mar}\,(v^{hm}{:}T)$, both v^{hm} and T will have been closed by substitution.

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

$$\frac{E \vdash_{hm} \text{ok} \qquad \vdash_{\bullet} e{:}T}{E \vdash_{hm} \mathbf{marshalled}\,(e{:}T){:}\text{STRING}} \quad \text{(eT.marshalled)}$$

The unmarshalling of a string first extracts the type tag T from the string and compares it with the tag for the expected type T'. Since T is a valid type for $v^{\bullet}$ in $\bullet$, it is also one in hm. The type tags T and T' are compared by *syntactic equality*: if the types match, the original value is extracted from the string; otherwise an exception is raised. This dynamic type equivalence is closely related to static equivalence (Theorem 4.7).

$$\begin{array}{lll} \mathbf{unmar}\,(\mathbf{marshalled}\,(v^{\bullet}{:}T){:}T') & & \\ \qquad \longrightarrow_{hm} v^{\bullet} & & \text{if } T = T' \\ \qquad \longrightarrow_{hm} \mathbf{UnmarFailure} & & \text{otherwise} \end{array}$$

4.3.6 Programs and networks

A machine consists of a series of module declarations followed by an expression. Each module declaration may refer to the previous ones.

$$\frac{E \vdash_\bullet T{:}\mathbf{Type} \qquad E \vdash_\bullet M{:}S \qquad E, U{:}S \vdash_\bullet m{:}T}{E \vdash_\bullet (\mathbf{module}\, N_U \ = M{:}S \ \mathbf{in}\ m){:}T} \ (\mathsf{mT.let})$$

A network is a parallel juxtaposition of machines. Note that each machine has its own environment: there is no explicit scope that encompasses more than one machine.

$$\frac{\vdash n_1 \,\mathrm{ok} \quad \vdash n_2 \,\mathrm{ok}}{\vdash n_1 \mid n_2 \,\mathrm{ok}} \ (\mathsf{nok.par}) \qquad \frac{\vdash_\bullet m{:}\mathrm{UNIT}}{\vdash m \,\mathrm{ok}} \ (\mathsf{nok.mach})$$

We assume that there is a single channel, which carries values of type STRING. The expression $!\,e$ sends the value of e over that channel, and ? reads a value from that channel. Communication is straightforward as all the work required to make values and types intercomprehensible is done by the marshalling apparatus; for suitable evaluation contexts $CC^\bullet_{hm_1}$ and $CC^\bullet_{hm_2}$ we have just the rule below, writing context application with a dot.

$$CC^\bullet_{hm_1}.!\, v^{hm_1} \mid CC^\bullet_{hm_2}.? \longrightarrow CC^\bullet_{hm_1}.() \mid CC^\bullet_{hm_2}.v^{hm_1} \quad (\mathsf{nred.comm})$$

4.4 Results

First, our calculus enjoys type preservation and progress properties.

Theorem 4.1 (type preservation for compile-time, expression, and network reduction) (See Theorem G.15 (type preservation for expression reduction), Theorem G.18 (type preservation for machine reduction), Corollary G.17 (type preservation for network reduction).)

- if $m \longrightarrow_{\mathsf{c}} m'$ and $\vdash_\bullet m{:}T$ then $\vdash_\bullet m'{:}T$;
- if $e \longrightarrow_{hm} e'$ and $\vdash_{hm} e{:}T$ then $\vdash_{hm} e'{:}T$; and
- if $n \longrightarrow n'$ and $\vdash n \,\mathrm{ok}$ then $\vdash n' \,\mathrm{ok}$.

Theorem 4.2 (progress for compile-time reduction)
(See Theorem H.8 (progress of machines).)
If $\vdash_\bullet m{:}\mathrm{UNIT}$ then either

- m is an expression; or
- m reduces, i.e. there exists m' such that $m \longrightarrow_{\mathsf{c}} m'$.

Moreover, compile-time reduction is terminating.

Theorem 4.3 (progress for expressions)
(See Theorem H.6 (progress of expressions).)
If $\vdash_{hm} e{:}T$ then one of the following holds:

- e is a value, i.e. there exists v^{hm} such that $e = v^{hm}$;
- e reduces, i.e. there exists e' such that $e \longrightarrow_{hm} e'$;
- e is blocked waiting for I/O, i.e. there exists $CC^{hm}_{hm_2}$ and e' such that $e = CC^{hm}_{hm_2}.!\, e'$ or $e = CC^{hm}_{hm_2}.?$; or
- e has thrown an exception, i.e. there exists $CC^{hm}_{hm_2}$ such that $e = CC^{hm}_{hm_2}.\mathbf{UnmarFailure}$.

In addition, we have proved a normalisation result for expressions, showing that the rules for coloured brackets do not introduce any divergencies.

Both compile-time machine reduction and run-time expression reduction are deterministic (network reduction is not, of course):

Theorem 4.4 (determinacy for compile-time and expression reduction)
(See Theorem H.11 (determinism of expression reduction), Discussion H.12 (strength of determinism), Theorem H.9 (determinism of machine reduction).)

If $m \longrightarrow_{\mathsf{c}} m'$ and $m \longrightarrow_{\mathsf{c}} m''$ then $m' = m''$ and both reductions apply the same rule to the same redex. if $e \longrightarrow_{hm} e'$ and $e \longrightarrow_{hm} e''$ then $e' = e''$ and both reductions apply the same rule to the same redex.

For static type checking:

Theorem 4.5 (decidability of type checking)
(See Theorem I.14 (decidability of type checking), Discussion I.16 (decidability of type checking for user programs).)
Type checking is decidable. Furthermore, user source programs can be typed by derivations involving no hashes or coloured brackets.

At run-time, all type annotations except those on $\mathbf{mar}$, $\mathbf{marshalled}$, and $\mathbf{unmar}$ can be erased. Moreover, all coloured brackets can be erased except for those that occur within a hash within one of those remaining annotations. More precisely, we define $\mathrm{erase}(e)$ to be e with all type annotations and brackets erased except that the type annotations on $\mathbf{mar}$, $\mathbf{marshalled}$, and $\mathbf{unmar}$ are left unchanged. We define $\xrightarrow[\mathrm{erase}]{}$ to be like $\longrightarrow_{hm}$ by taking the erase-image of the left- and right-hand sides of each rule (and removing rules that would become $e \xrightarrow[\mathrm{erase}]{} e$).

Theorem 4.6 (erasure preserves reduction outcomes)
(See Theorem I.30 (type erasure).)
Assume $\vdash_\bullet e{:}T$. We have that $e \longrightarrow_\bullet e'$ implies $\mathrm{erase}(e) \xrightarrow[\mathrm{erase}]{}{}^{\leqslant 1}\mathrm{erase}(e')$. Conversely, $\mathrm{erase}(e) \xrightarrow[\mathrm{erase}]{} e_0$ implies that there exists e' such that $\mathrm{erase}(e') = e_0$ and $e \longrightarrow_\bullet^{\geqslant 1} e'$.

Note that brackets are needed in module reduction, to keep track of a module's ancestors as we build its hash.

Finally we show that, under reasonable conditions, static and dynamic type equality coincide. Let D be a module declaration context:

$$\mathbf{module}\, N_{0\,U_0} = M_0{:}S_0 \;\mathbf{in}\; ...\mathbf{module}\, N_{j\,U_j} = M_j{:}S_j \;\mathbf{in}\; \_$$

in the user source language (with no brackets or hashes). Consider a machine $D.C.e$ for some expression context C and an expression $e = (\mathbf{unmar}\,(\mathbf{mar}\,(e_0{:}T_0){:}T_1))$. One would like this dynamic type check to succeed if and only if T_0 and T_1 are statically provably equal, i.e. iff $U_0{:}S_0, ..., U_j{:}S_j \vdash_\bullet T_0 == T_1$.

Write σ_D for the accumulated substitution defined by the module reduction rules for D (we omit an explicit definition for lack of space). The dynamic check is then $\sigma_D\, T_0 = \sigma_D\, T_1$. We have:

Theorem 4.7 (coincidence between dynamic and static type checking)
(See Corollary J.21 (coincidence between undyn-time and static type checking with non-repeated external names).)
Suppose that $D.C.e$ is well formed (i.e. $\vdash_\bullet D.C.e{:}\mathrm{UNIT}$), that it contains no hashes, and that its external names $N_0, ..., N_j$ are distinct. Let $E = U_0{:}S_0, ..., U_j{:}S_j$ be the associated environment. Assume that T_0 and T_1 contain no hashes and $E \vdash_\bullet T_i{:}\mathbf{Type}$ for $i = 0, 1$. Then $E \vdash_\bullet T_0 == T_1$ iff $\sigma_D\; T_0 = \sigma_D\; T_1$.

The requirement that the external names $N_0, ..., N_j$ be distinct rules out the rather pathological programs in which there are two module definitions with the same name, one shadowing the other, which have identical structures, signatures, and dependencies. The exclusion of hashes is automatic for user source programs.

One can imagine stronger theorems, relating type equality between two programs that share a common (DAG-)prefix of module definitions, but their statements become rather elaborate.

5 Related work

Modules and generativity There is an extensive literature on ML-style modules, including [Mac84, MTH90, HL94, Ler94, SH00, DCH03], much of it discussing subtle questions of generativity versus applicativity. To our knowledge, however, none deals with the inter-program case. In [Sew01], fresh type names are generated during call-by-value module reduction, with ν-binders that can extrude across distributed scope. This allows inter-program sharing, and also a `with!` coercion, but at the pragmatically-awkward cost of requiring particular object files to be shared.

Type dynamic Our marshal and unmarshal operations are essentially constructors and destructors for values of dynamic type; $\mathbf{mar}$ is just `dynamic`, and $\mathbf{unmar}$ is a restricted form of `typecase`. Our dynamic values have type STRING, emphasising that they may be communicated readily. Type `Dynamic` was first formalised by Abadi et al. [ACPP91, ACPR95], who also gives a historical survey. Intensional polymorphism [HM95, Wei00] permits run-time type analysis of *all* values.

Marshalling abstract types The problem of marshalling values of abstract (existential [MP88]) type has not been satisfactorily addressed theoretically before. In several systems, abstract types are run- or build-time generative, so that two executions or builds of the same source will yield distinct types. While communication within such a program can be abstraction-safe, successful communication between builds can only be at the representation type, and hence abstraction-unsafe. This is true, for instance, of [ACPR95], TMAL [Dug02], Modula-3 [B^+94, BNOW95], Alice [Ali03], and the typed-channel languages listed below.

Weirich [Wei02] exposes an existential's representation type to type analysis, permitting a type-safe polytypic marshalling function to be written. As future work we hope to expose our global type names at term level (cf. [HWC00]), permitting an *abstraction-safe* polytypic marshalling function to be written. Furuse and Weis [FW00] argue for ignoring abstraction, checking representation types only.

A number of programming languages feature some form of built-in marshalling (pickling, serialisation, etc.): for example Modula-3, Alice, Java, .NET, and OCaml. Most of these languages serialise the type along with the value in order to permit a check at unmarshal time, and represent the type by a hash. Languages differ, however, in exactly what is hashed — i.e., in what is considered when deciding type equality.

In Modula-3, abstract types are made opaque by *branding*, which may be either by a literal string (analogous to an external name) or a compiler-generated unique identifier. The latter are unique within a program but not necessarily related between programs, so explicit brands must be used for inter-program communication; however, they do not guarantee abstraction-safety for that case. *Revelation* can be used to make an abstraction transparent.

In Alice, abstract type creation is run-time generative, meaning that abstract types from different executions are always distinct. This vacuous abstraction-safety forces the use of representation types for pickling between different programs.

In Java serialisation [Sun02], class equivalence is on fully-qualified class name, the representation type of all fields, and the types of all non-private methods; the implementation is not considered in type equality. A strong coercion (Section 2.10) is provided (although compatibility of representation types is not checked until unmarshal time).

In .NET serialisation [Mic01], class equivalence is on the textual name along with the implementation of the entire assembly in which it is defined (a single DLL or EXE, which may comprise many source files). This guarantees data structure invariants are maintained, as in our approach; however, we work on the much finer scale of individual modules, and furthermore we require only source code to be shared, not object files.

Java and .NET both support controlled relaxation of type equivalence checking, the use of which clearly voids any abstraction-safety guarantees made by the type system.

OCaml [OCa] does no typechecking for marshalling at all, and hence is not even type-safe. When unmarshalling a function, it verifies (by a hash) that the communicating builds are identical, thus allowing the code pointers of all closures to be communicated literally.

Coloured brackets Coloured brackets were introduced in [ZGM99, GMZ00]; we differ in that we permit a variable to occur in a colour other than the one where it is defined. Our proofs are harder, our β-rule has to introduce extra brackets, but our brackets carry only a single optional hash, rather than a list of hashes. Rossberg [Ros02], like us, is concerned to preserve the opacity of abstract types under reduction due to the presence of typecase. His coercions serve the same purpose as our brackets, but his use of the closed-scope `open` construct instead of dot notation [CL90] prevents any possibility of sharing values of abstract type between instances.

Typed channels Several languages, e.g. JoCaml [JoC], Nomadic Pict [SWP99], Facile [TLK96, Kna95], implement *typed channels*. These permit type- and abstraction-safe communication once the channel is established. Establishing a channel at an abstract type, however, requires the endpoints somehow to share the type already; in the case that the endpoints reside in different programs or instances, this requires an unsafe cast, usually performed (outside the language) by a name server.

6 Conclusions and future work

6.1 Summary

We have proposed a novel and expressive design for guaranteeing type- and abstraction-safe marshalling of data sent between distributed ML programs, that can uniformly treat manifest, abstract, and generative types. The key technical idea is to use hashes of module declarations as globally-meaningful type names, which are inserted at compile-time and then compared dynamically when unmarshalling. We add coloured brackets to delimit the "abstraction boundary" within which hashes are transparent, tracking these brackets through the reductions so as to achieve type and abstraction preservation. Our proposal is a smooth extension of existing ML-like languages: type checking is unchanged, most type information can be erased before run-time, and the dynamic type check closely mirrors static ML type equivalence.

6.2 Future work

In the future, we aim to broaden our solution to be applicable to full-scale languages.

The following extensions will be required to cope with the examples in Section 2.9–2.13. The *strong coercion* (Section 2.10 and 2.11) used for forcing an abstract type to have the same hash as an earlier module, has a simple compile-time implementation: check the representation types of the two are provably equal, then simply reuse the hash of the earlier module as the type name for the new. This requires the compiler to keep a mapping from hashes to representation types, which is straightforward. *Programmer-requested generativity* (Section 2.12) can be dealt with in an implementation by generating a fresh global name (say a random bit string of the same length as hashes) at compile time; its semantics can be modelled by ν-binding. Both this and the strong coercion are very similar to the constructs in our earlier work [Sew01]. *Side-effect-induced generativity* (Section 2.13) requires a way to identify simple pure computations in structure bodies that the programmer can easily understand; abstract types of structures with pure computations should be hashes, whereas those of structures with effectful computations should have freshly-generated names. *Functors* (Section 2.9) are a more substantial extension, but, at least for a restricted but useful class, should be straightforward. Consider first-order applicative functors [Ler95] and module expressions that are either (i) an explicit structure, possibly multiply-abstracted, or (ii) *pure*, i.e. constructed from module identifiers, abstraction and application. These give rise to *functions* from hashes to hashes; applying these functions gives run-time representations of the compile-time path-based type names.

Other substantial extensions also need to be considered. *Dependent record structures*, i.e. module structures with multiple fields also appear in this report's informal examples; they should be conceptually straightforward. *Parametric and substructuring polymorphism* within the dynamic check would allow receivers to accept a more general type than that offered by the sender. This is a more substantial extension; it will be a challenge to minimise the transmitted type information required for these dynamic "subtype" checks. One may want to *rebind* (Section 2.14) identifiers within a transmitted value to avoid the overhead of sending code already available at the other end, or to obtain location-specific behaviour; here we aim to integrate hash types with [BHS$^+$03]. Marshalling *reference cells* exhibits related problems: should the reference be rebound, made remote, or duplicated? More generally, one must consider values mentioning other machine resources: screens, files...

We wish to integrate our work with existing systems for distributed programming which have statically typed channels for normal operation but no safe way of initiating communication, such as JoCaml [JoC] and Nomadic Pict [SWP99]. We also wish to test the expressiveness of our marshalling primitives by using them to write libraries for safe distributed communication and persistence.

Acknowledgments We acknowledge support from a Royal Society University Research Fellowship (Sewell), EPSRC grant GRN24872 (Wansbrough), EC FET-GC project IST-2001-33234 PEPITO, and APPSEM 2. The authors thank Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and the anonymous referees for their suggestions.

A Introduction to the complete definitions and proofs

A.1 Differences between the main body and the appendices

The following appendices present the full formal system with all proofs. There are slight differences between these and the main body of the report:

Type system Some admissible rules were omitted from the main body. They are (Ssub.refl) and (Ssub.tran) (section C.11). We now also formalise the $\notin \mathrm{dom}$ relation (section C.1), and define a kind equality (section C.5) which is used in the rules for subkinding. For the sake of completeness, we define a signature equality relation (section C.10).

Integer constants We now omit the rules for the INT type and integer constants. They (and other base types) can be trivially added to the system on the model given by UNIT and $()$.

A.2 Correspondence between the theorems in the main body and in the appendices

Main body	Appendices
Theorem 4.1	Theorem G.15 (type preservation for expression reduction), Theorem G.18 (type preservation for machine reduction), Corollary G.17 (type preservation for network reduction)
Theorem 4.2	Theorem H.8 (progress of machines)
Theorem 4.3	Theorem H.6 (progress of expressions)
Theorem 4.4	Theorem H.11 (determinism of expression reduction), Discussion H.12 (strength of determinism), Theorem H.9 (determinism of machine reduction)
Theorem 4.5	Theorem I.14 (decidability of type checking), Discussion I.16 (decidability of type checking for user programs)
Theorem 4.6	Theorem I.30 (type erasure)
Theorem 4.7	Corollary J.21 (coincidence between undyn-time and static type checking with non-repeated external names)

B Syntax

$e ::=$	expression
$()$	unit
$(e_1, ..., e_j)$	tuple ($2 \leqslant j$)
$\mathbf{proj}_i\ e$	projection
x	variable
$\lambda x{:}T.e$	abstraction (x binds in e)
$e\ e$	application
$\mathbf{mar}\,(e{:}T)$	dynamic
$\mathbf{marshalled}\,(e{:}T)$	closed, colour-independent dynamic
$\mathbf{unmar}\ e{:}T$	undynamic
$!\,e$	send
$?$	receive
$U.\mathrm{term}$	term-part of a module
$[e]^T_{hm}$	type colouring
$\mathbf{UnmarFailure}^T$	undyn failure

v^{hm_0} ::=	hm_0-value, i.e. value in the colour hm_0
$()$	unit
$(v_1^{hm_0}, ..., v_j^{hm_0})$	tuple ($2 \leqslant j$)
$\lambda x{:}T.e$	abstraction (x binds in e)
$\mathbf{marshalled}\,(v^{\bullet}{:}T)$	closed dynamic value
$[v^{h_1}]_{h_1}^{h_1}$	type colouring, where $h_1 \neq hm_0$

We sometimes omit the colour annotation on values, when it is irrelevant.

$C_{hm_2}^{hm_1}$ ::=	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 $hm_1 = hm_2$
$\_\ e$	application left, if $hm_1 = hm_2$
$v^{hm_1}\ \_$	application right, if $hm_1 = hm_2$
$\mathbf{mar}\,(\_{:}T)$	dynamic, if $hm_1 = hm_2$
$\mathbf{marshalled}\,(\_{:}T)$	colour-independent dynamic, if $hm_2 = 0$
$\mathbf{unmar}\ \_{:}T$	undynamic, if $hm_1 = hm_2$
$!\ \_$	send, if $hm_1 = hm_2$
$[\_]_{hm_2}^{T}$	coloured bracket

$CC_{hm_2}^{hm_1}$ ::=	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 ($2 \leqslant j$)
X	variable
$T \to T$	function
STRING	dynamic
U.TYPE	type-part of a module
h	hash

TC ::=	first-level constructed type context
UNIT	unit
STRING	dynamic
$\_1 \to \_2$	function
$\_1 * ... * \_j$	tuple ($2 \leqslant j$)

hm ::=	hash option
$\bullet$	none
h	some hash

h ::= $\mathbf{hash}(N, M{:}[X{:}\mathbf{Type}, T])$	hash

N	external name

K ::=	kind
$\quad$ **Type**	opaque
$\quad$ $\mathbf{Eq}(T)$	singleton
M ::= $[T, v]$	module structure (type-part, term-part)
S ::= $[X{:}K, T]$	module signature (X binds in T)
m ::=	machine
$\quad$ e	expression
$\quad$ **module** $N_U = M{:}S$ **in** m	module declaration (U binds in m)
n ::=	network
$\quad$ $\mathbf{0}$	null
$\quad$ $n \mid n$	parallel composition
$\quad$ e	expression (on one machine)
ζ ::=	variable
$\quad$ x	expression variable
$\quad$ X	type variable
$\quad$ U	module variable
χ ::=	substitutable entity
$\quad$ X	type variable
$\quad$ $U.\text{TYPE}$	type-part of a module
$\quad$ x	expression variable
$\quad$ $U.\mathrm{term}$	term-part of a module
$\quad$ U	module variable
$\dddot{X}$ ::=	type substitutable entity
$\quad$ X	type variable
$\quad$ $U.\text{TYPE}$	type-part of a module
$\dddot{x}$ ::=	expression substitutable entity
$\quad$ x	expression variable
$\quad$ $U.\mathrm{term}$	term-part of a module
E ::=	environment
$\quad$ $E, x{:}T$	expression variable binding
$\quad$ $E, X{:}K$	type variable binding
$\quad$ $E, U{:}S$	module variable binding
$\quad$ **nil**	empty

$J ::=$	colourable statement
$\quad \mathrm{ok}$	environment correctness
$\quad K\ \mathrm{ok}$	kind correctness
$\quad K == K'$	kind equivalence
$\quad K <: K'$	subkinding
$\quad T{:}K$	kind of a type
$\quad T == T'$	type equivalence
$\quad S\ \mathrm{ok}$	signature correctness
$\quad S == S'$	signature equivalence
$\quad S <: S'$	subsignaturing
$\quad e{:}T$	type of an expression
$\quad M{:}S$	signature of a module expression
$\quad U{:}S$	signature of a module variable
$\quad m{:}T$	type of a machine
$\mathrm{CJ} ::= E \vdash_{hm} J$	couloured judgement
$\mathrm{MJ} ::=$	monochrome judgement
$\quad \vdash hm\ \mathrm{ok}$	hash correctness
$\quad \vdash n\ \mathrm{ok}$	network correctness
$\mathrm{AJ} ::=$	judgement
$\quad \zeta \notin \mathrm{dom}\, E$	non-clash judgement
$\quad E \vdash_{hm} J$	coloured judgement
$\quad \mathrm{MJ}$	monochrome judgement

Additionally, we use the following notations:

$\zeta{:}\tau ::=$	$x{:}T \mid X{:}K \mid U{:}S$	variable has sort
$\eta{:}\tau ::=$	$e{:}T \mid T{:}K \mid M{:}S$	term has sort
$\tau\ \mathrm{ok} ::=$	$T{:}\mathbf{Type} \mid K\ \mathrm{ok} \mid S\ \mathrm{ok}$	sort is correct

$=$	syntactic equality
σ, μ	substitutions
Π	derivation (i.e. proof tree)
$\{\chi{\leftarrow}\eta\}\aleph$	substitution: replace χ by η in $\aleph$
fv	free variables (U, X, x)
fse	free substitutable entities (U, $U.\mathrm{TYPE}$, $U.\mathrm{term}$, X, x) (if $U.\mathrm{TYPE} \in \mathrm{fse}\,\aleph$ or $U.\mathrm{term} \in \mathrm{fse}\,\aleph$ then $U \in \mathrm{fse}\,\aleph$)

(and more, introduced as they are defined...)

Note that in particular external names (N) are not variables or substitutable entities, so they are never “free” nor can substitution touch them: $\mathrm{fv}\, N = \mathrm{fse}\, N = \varnothing$ and $\sigma N = N$.

C Static judgements

C.1 $\boxed{\zeta \notin \mathrm{dom}\, E}$ non-clash

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

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

C.2 $\boxed{\vdash hm \,\mathrm{ok}}$ hash correctness

$$\frac{\mathbf{nil} \vdash_{\bullet} M{:}[X{:}\mathbf{Type}, T]}{\vdash \mathbf{hash}(N, M{:}[X{:}\mathbf{Type}, T])\,\mathrm{ok}}\ (\mathsf{hmok.hash}) \qquad \frac{}{\vdash \bullet\,\mathrm{ok}}\ (\mathsf{hmok.zero})$$

Note that N may be any external name.
Note that we demand that a module have an opaque type in order to take its hash.

C.3 $\boxed{E \vdash_{hm} \mathrm{ok}}$ environment correctness

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

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

$$\frac{E \vdash_{hm} \tau\,\mathrm{ok} \quad \zeta \notin \mathrm{dom}\,E}{E, \zeta{:}\tau \vdash_{hm} \mathrm{ok}}$$

C.4 $\boxed{E \vdash_{hm} K\,\mathrm{ok}}$ kind correctness

$$\frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} \mathbf{Type}\,\mathrm{ok}}\ (\mathsf{Kok.Type}) \qquad \frac{E \vdash_{hm} T{:}\mathbf{Type}}{E \vdash_{hm} \mathbf{Eq}(T)\,\mathrm{ok}}\ (\mathsf{Kok.Eq})$$

C.5 $\boxed{E \vdash_{hm} K == K'}$ kind equality

$$\frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} \mathbf{Type} == \mathbf{Type}}\ (\mathsf{Keq.Type}) \qquad \frac{E \vdash_{hm} T == T'}{E \vdash_{hm} \mathbf{Eq}(T) == \mathbf{Eq}(T')}\ (\mathsf{Keq.Eq})$$

C.6 $\boxed{E \vdash_{hm} K <: K'}$ subkinding

$$\frac{E \vdash_{hm} T{:}\mathbf{Type}}{E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Type}}\ (\mathsf{Ksub.Eq})$$

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

Note that (Ksub.tran) is currently derivable.

C.7 $\boxed{E \vdash_{hm} T{:}K}$ kind of a type

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

$$\frac{\vdash h\,\mathrm{ok} \quad E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} h{:}\mathbf{Type}}\ (\mathsf{TK.hash}) \qquad \frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} \mathrm{UNIT}{:}\mathbf{Type}}\ (\mathsf{TK.unit}) \qquad \frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} \mathrm{STRING}{:}\mathbf{Type}}\ (\mathsf{TK.mar})$$

$$\frac{E \vdash_{hm} T{:}\mathbf{Type} \quad E \vdash_{hm} T'{:}\mathbf{Type}}{E \vdash_{hm} T \to T'{:}\mathbf{Type}}\ (\mathsf{TK.fun}) \qquad \frac{E \vdash_{hm} T_i{:}\mathbf{Type} \quad \forall i.1 \leqslant i \leqslant j}{E \vdash_{hm} T_1 * \ldots * T_j{:}\mathbf{Type}}\ (\mathsf{TK.tuple})$$

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

C.8 $\boxed{E \vdash_{hm} T == T'}$ type equivalence

$$\frac{E \vdash_{hm} T{:}\mathbf{Eq}(T')}{E \vdash_{hm} T == T'} \text{ (Teq.Eq)} \qquad \frac{E \vdash_{h} \text{ok}}{E \vdash_{h} h == T} \text{ (Teq.hash)}$$
where $h = \mathbf{hash}(N, [T, v^{hm}]{:}[X{:}\mathbf{Type}, T'])$

The rule (Teq.hash) is the rule that introduces type equivalences (see Lemma G.11 (type decomposition)). Other rules 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).

$$\frac{E \vdash_{hm} T{:}\mathbf{Type}}{E \vdash_{hm} T == T} \text{ (Teq.refl)} \qquad \frac{E \vdash_{hm} T == T'}{E \vdash_{hm} T' == T} \text{ (Teq.sym)} \qquad \frac{E \vdash_{hm} T == T' \quad E \vdash_{hm} T' == T''}{E \vdash_{hm} T == T''} \text{ (Teq.tran)}$$

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

C.9 $\boxed{E \vdash_{hm} S \text{ ok}}$ signature correctness

$$\frac{E, X{:}K \vdash_{hm} T{:}\mathbf{Type}}{E \vdash_{hm} [X{:}K, T] \text{ ok}} \text{ (Sok)}$$

C.10 $\boxed{E \vdash_{hm} S == S'}$ signature equivalence

Note that we never use signature equivalence.

$$\frac{E \vdash_{hm} K == K' \qquad E, X{:}K \vdash_{hm} T == T'}{E \vdash_{hm} [X{:}K, T] == [X{:}K', T']} \text{ (Seq.struct)}$$

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

C.11 $\boxed{E \vdash_{hm} S <: S'}$ subsignaturing

$$\frac{E \vdash_{hm} K <: K' \quad E, X{:}K \vdash_{hm} T == T'}{E \vdash_{hm} [X{:}K, T] <: [X{:}K', T']} \text{ (Ssub.struct)}$$

$$\frac{E \vdash_{hm} S \text{ ok}}{E \vdash_{hm} S <: S} \text{ (Ssub.refl)} \qquad \frac{E \vdash_{hm} S <: S' \quad E \vdash_{hm} S' <: S''}{E \vdash_{hm} S <: S''} \text{ (Ssub.tran)}$$

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

C.12 $\boxed{E \vdash_{hm} e{:}T}$ type of an expression

$$\frac{E \vdash_{hm} e{:}T \quad E \vdash_{hm} T == T'}{E \vdash_{hm} e{:}T'} \text{ (eT.eq)} \qquad \frac{E, x{:}T, E' \vdash_{hm} \text{ok}}{E, x{:}T, E' \vdash_{hm} x{:}T} \text{ (eT.var)} \qquad \frac{E \vdash_{hm} U{:}[X{:}K, T] \quad E \vdash_{hm} T{:}\mathbf{Type}}{E \vdash_{hm} U.\text{term}{:}T} \text{ (eT.mod)}$$

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

$$\frac{E \vdash_{hm} e'{:}T \to T' \quad E \vdash_{hm} e{:}T}{E \vdash_{hm} e'\ e{:}T'} \text{ (eT.ap)} \qquad \frac{E, x{:}T \vdash_{hm} e{:}T'}{E \vdash_{hm} \lambda x{:}T.e{:}T \to T'} \text{ (eT.fun)}$$

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

$$\frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} (){:}\text{UNIT}}\ (\mathsf{eT.unit}) \qquad \frac{E \vdash_{hm} e{:}\text{STRING}}{E \vdash_{hm} !\ e{:}\text{UNIT}}\ (\mathsf{eT.send}) \qquad \frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} ?{:}\text{STRING}}\ (\mathsf{eT.recv})$$

$$\frac{E \vdash_{hm} e{:}T}{E \vdash_{hm} \mathbf{mar}\,(e{:}T){:}\text{STRING}}\ (\mathsf{eT.mar}) \qquad \frac{\begin{array}{c}E \vdash_{hm} \mathrm{ok}\\ \mathbf{nil} \vdash_{\bullet} e{:}T\end{array}}{E \vdash_{hm} \mathbf{marshalled}\,(e{:}T){:}\text{STRING}}\ (\mathsf{eT.marred})$$

$$\frac{\begin{array}{l}E \vdash_{hm} T{:}\mathbf{Type}\\ E \vdash_{hm} e{:}\text{STRING}\end{array}}{E \vdash_{hm} (\mathbf{unmar}\ e{:}T){:}T}\ (\mathsf{eT.unmar}) \qquad \frac{E \vdash_{hm} T{:}\mathbf{Type}}{E \vdash_{hm} \mathbf{UnmarFailure}^T{:}T}\ (\mathsf{eT.Undynfailure})$$

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

$$\frac{\begin{array}{l}E \vdash_{hm} T{:}\mathbf{Type}\\ E \vdash_{hm'} e{:}T\end{array}}{E \vdash_{hm} [e]^T_{hm'}{:}T}\ (\mathsf{eT.col})$$

Note that, when reading the rule backwards, hm' **replaces** hm: the brackets denote a colour *change*.

C.13 $\boxed{E \vdash_{hm} M{:}S}$ signature of a module expression

$$\frac{\begin{array}{l}E \vdash_{hm} T{:}K\\ E, X{:}K \vdash_{hm} T'{:}\mathbf{Type}\\ E, X{:}\mathbf{Eq}(T) \vdash_{hm} T'' == T'\\ E \vdash_{hm} v^{hm}{:}T''\end{array}}{E \vdash_{hm} [T, v^{hm}]{:}[X{:}K, T']}\ (\mathsf{MS.struct})$$

Remark: in (MS.struct), the type T'' and the premise $E \vdash_{hm} v^{hm}{:}T''$ are there to ensure that X is not free in v^{hm}, even though its interesting type T' may contain X.

The premise $E, X{:}K \vdash_{hm} T'{:}\mathbf{Type}$ is implied by the premise $E, X{:}\mathbf{Eq}(T) \vdash_{hm} T'' == T'$, since the correctness of T' only requires that the environment provide X, not that it be given any particular kind. (See Lemma F.9 (types are ok provided their hashes are).)

We might be worried that there should be a (MS.sub), but this would be doing subsignaturing at compile time, and we have subsignaturing at run time via (US.sub) which seems to subsume it.

We require the expression component of the module to be a value. This is to dispel any concern over the time at which a module is initialised. Note that if an arbitrary expression e is desired, it can be encoded as $\lambda x{:}\text{UNIT}.e$, with the type of the module changed from T' to $\text{UNIT}{\rightarrow}T'$, and its uses changed from $U.\mathrm{term}$ to $U.\mathrm{term}\,()$.

C.14 $\boxed{E \vdash_{hm} U{:}S}$ signature of a module variable

$$\frac{E, U{:}S, E' \vdash_{hm} \mathrm{ok}}{E, U{:}S, E' \vdash_{hm} U{:}S}\ (\mathsf{US.var}) \qquad \frac{\begin{array}{l}E \vdash_{hm} U{:}S\\ E \vdash_{hm} S <: S'\end{array}}{E \vdash_{hm} U{:}S'}\ (\mathsf{US.sub}) \qquad \frac{E \vdash_{hm} U{:}[X{:}K, T]}{E \vdash_{hm} U{:}[X{:}\mathbf{Eq}(U.\text{TYPE}), T]}\ (\mathsf{US.self})$$

The way we prove things about the type of $U.\mathrm{term}$ is quite interesting. Here is an example: what are the possible types T such that $E \vdash_{\bullet} U.\mathrm{term}{:}T$ with $E = (U{:}[X{:}\mathbf{Type}, X])$? Let us look at all the possible derivations (arguing slightly informally, i.e. without verifying everything, on the "the proof has to be of this form" bit).

$$\dfrac{\dfrac{\alpha}{E \vdash_{\bullet} T{:}\mathbf{Type}} \qquad \dfrac{\dfrac{\dfrac{\beta}{E \vdash_{\bullet} U{:}[X{:}K_4, T_3]}}{E \vdash_{\bullet} U{:}[X{:}K_3, T_3]}\,(\mathsf{US.self}) \qquad \dfrac{\dfrac{\gamma}{E \vdash_{\bullet} K_3 <: K_2} \quad \dfrac{\delta}{E, X{:}K_3 \vdash_{\bullet} T_3 == T}}{E \vdash_{\bullet} [X{:}K_3, T_3] <: [X{:}K_2, T]}\,(\mathsf{Seq.struct})}{E \vdash_{\bullet} U{:}[X{:}K_2, T]}\,(\mathsf{US.sub})}{E \vdash_{\bullet} U.\mathrm{term}{:}T}\,(\mathsf{eT.mod})$$

We can check that any proof of $E \vdash_{\bullet} U.\mathrm{term}{:}T$ can be shortened to a proof of this form. The only alternative is not to use (US.self), but we can verify that this does not lead to any proof of $E \vdash_{\bullet} U{:}[X{:}K_3, T_3]$. (If the signature of U was $[X{:}\mathbf{Eq}(T_1), X]$, we could usefully use (US.var) there, and (US.sub) would rewrite the X into T_1, so we would get $T = T_1$). At this point, we still get to choose T, K_2, K_3, K_4 and T_3, but the derivations of (α), (β), (γ) and (δ) constrain these choices.

In α we see that fv $T \subseteq \{U\}$. In (β), all we have left to do (assuming as stated a minimal proof) is apply (US.var), so $K_4 = \mathbf{Type}$ and $T_3 = X$. In (β) we have that K_3 is $\mathbf{Eq}(U.\mathrm{TYPE})$. In (γ) we have the only constraint on K_2, and we might as well take $K_2 = \mathbf{Type}$. In (δ), we need to prove that $X == T$, and since X does not appear in T, the equivalence must come from the environment, hence $K_3 = \mathbf{Eq}(T)$. Unifying the constraints on K_3 yields $\boldsymbol{T = U.}\mathrm{TYPE}$ as the one possible type for $U.\mathrm{term}$.

C.15 $\boxed{E \vdash_{hm} m{:}T}$ type of a machine

$$\frac{E \vdash_{\bullet} e{:}T}{E \vdash_{\bullet} e{:}T}\ (\mathsf{mT.expr}) \qquad \frac{\begin{array}{c} E \vdash_{\bullet} T{:}\mathbf{Type} \\ E \vdash_{\bullet} M{:}S \\ E, U{:}S \vdash_{\bullet} m{:}T \end{array}}{E \vdash_{\bullet} (\mathbf{module}\ N_U = M{:}S\ \mathbf{in}\ m){:}T}\ (\mathsf{mT.let})$$

In (mT.expr), the premise is a "type of an expression" judgement, while the conclusion is a "type of a machine" judgement.

Note: in (mT.let), the first premise is saying that U is not free in T.

C.16 $\boxed{\vdash n\ \mathrm{ok}}$ network correctness

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

D Reduction rules and structural congruence

D.1 $\boxed{m \longrightarrow_{\mathsf{c}} m'}$ compile-time reduction

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

where $h = \mathbf{hash}(N, [T, v^{\bullet}]{:}[X{:}\mathbf{Type}, T'])$

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

D.2 $\boxed{e \longrightarrow_{hm} e'}$ expression reduction

Note that we don't rewrite types at all (unlike Grossman et al., p13 rule 7).

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

(otherwise stuck)

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

(Note that (as Grossman et al.) we introduce a bracket around the argument of the function. This is needed e.g., because if $h = \mathbf{hash}(N, [\mathrm{INT}, 3]{:}[X{:}\mathbf{Type}, X])$ then $\mathbf{nil} \vdash_h (\lambda x{:}\mathrm{INT}.[x]_{\bullet}^{\mathrm{INT}})[3]_h^h{:}\mathrm{INT}$ is derivable, yet $[[3]_h^h]_{\bullet}^{\mathrm{INT}}$ is not well-typed.)

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

$$\mathbf{unmar}\,(\mathbf{marshalled}\,(v^{\bullet}{:}T){:}T') \longrightarrow_{hm} \begin{cases} v^{\bullet} & \text{if } T = T' \\ \mathbf{UnmarFailure}^{T} & \text{otherwise} \end{cases} \qquad (\mathsf{ered.unmar})$$

Note that we use syntactic equality (up to alpha-conversion (inside hashes)) to compare the types (which are closed). In practice, we could use a md5-style checksum of the alpha-normalized type expression.

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

(and all base types when we have them)

$$[(v_1^{hm'}, ..., v_j^{hm'})]^{T_1 * ... * T_j}_{hm'} \longrightarrow_{hm} ([v_1^{hm'}]^{T_1}_{hm'}, ..., [v_j^{hm'}]^{T_j}_{hm'}) \qquad \text{(ered.col.tuple)}$$

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

In the left-hand side, the brackets represent a boundary at which the equation hm' may be used. This boundary appears in the right-hand side at each input site (the occurences of x) and at the output site (around the body of the function).

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

$$[[v^{h_0}]^{h_0}_{h_0}]^{h_0}_{h_1} \longrightarrow_{hm} [v^{h_0}]^{h_0}_{h_0} \qquad \text{(ered.col.col)}$$

for $h_0 \neq h_1$ and $h_1 \neq hm$.
We used to have this rule for an arbitrary type T, but in fact it is only ever necessary for $T = h_0$.

$$[v^{hm_1}]^{h_2}_{hm_1} \longrightarrow_{hm} v^{hm_1} \qquad \text{(ered.col.le)}$$

for $hm_1 \preccurlyeq hm$. (See Definition E.13 (partial order on colours).)
We used to have this rule with an arbitrary type T rather than h_2, but the other (ered.col.*) rules deal with the case when T is a constructed type.

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

D.3 $\boxed{n \equiv n'}$ 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 $\equiv$.

D.4 $\boxed{n \longrightarrow n'}$ network reduction

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

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

Note that the value that is sent is a value in the domain from which it is sent, but just by looking at this rule, it might not be a value in the domain where it is received. In fact, typing will ensure that $v^{hm} = \textbf{marshalled}\,(v^\bullet{:}T)$ and this is a value in any domain.

E General definitions and lemmas

E.1 Proofs

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

Definition E.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).

E.2 Correctness of parts

Definition E.2 (domain of an environment) The domain of an environment E, written $\operatorname{dom} E$, is a set of variables defined by induction as follows:

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

Lemma E.3 (non-membership in domain is interpreted trivially) The judgement $\zeta \notin \operatorname{dom} E$ is provable iff ζ is not a member of the domain of E.

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

We will freely make use of this lemma in the remainder of this section.

Lemma E.4 (colours have to be ok) If $E \vdash_{hm} J$ then $\vdash hm\,\mathrm{ok}$ by a proper subproof.

Proof. Induct on the derivation of $E \vdash_{hm} J$. 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 rule (envok.nil), which has $\vdash hm\,\mathrm{ok}$ as a premise. □

Lemma E.5 (hashes have to be ok) If $E \vdash_{hm} J$ or $\vdash hm\,\mathrm{ok}$ is derivable by a proof Π and h is a subterm of $E \vdash_{hm} J$ or $\vdash hm\,\mathrm{ok}$ then $\vdash h\,\mathrm{ok}$ 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. 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 subterm of $\mathbf{hash}(N, M{:}S)$ in (TK.hash): One premise is $\vdash \mathbf{hash}(N, M{:}S)\,\mathrm{ok}$. If $h = \mathbf{hash}(N, M{:}S)$ we have the desired result. Otherwise induction gives the desired result.

Case hm in (envok.nil): Trivial by induction.

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

Case (hmok.hash): The conclusion is $\vdash \mathbf{hash}(N, M{:}S)\,\mathrm{ok}$. If $h = \mathbf{hash}(N, M{:}S)$ we have the desired result. Otherwise induction gives the desired result.

Case (hmok.zero): Trivial.

□

Lemma E.6 (environments have to be ok) If $E \vdash_{hm} J$ then $E \vdash_{hm} \mathrm{ok}$ by a subproof.

Proof. Simultaneously with the following lemma. □

Lemma E.7 (prefixes of ok environments are ok) If $E, b \vdash_{hm} \mathrm{ok}$ then $E \vdash_{hm} \mathrm{ok}$ by a subproof.

Proof. Induct simultaneously on the derivation of $E \vdash_{hm} J$ and $E, b \vdash_{hm} \mathrm{ok}$. Note that $E, b \vdash_{hm} \mathrm{ok}$ can only be derived from a judgement of the form $E \vdash_{hm} J$, hence the result of the second lemma follows by induction from the first lemma. 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 environment, and so we apply the induction hypothesis to that premise. We list the remaining cases.

Cases (envok.nil), (envok.x), (envok.X), (envok.U): Trivial.

Case (eT.fun): There exist x, T, T', e such that $J = \lambda x{:}T.e{:}T \to T'$. The premise is $E, x{:}T \vdash_{hm} e{:}T'$. By induction, $E, x{:}T \vdash_{hm} \mathrm{ok}$ by a subproof, hence $E \vdash_{hm} \mathrm{ok}$ by the second lemma, as desired.

□

Lemma E.8 (ok environments have no repetition in the domain) If $E, E' \vdash_{hm} \mathrm{ok}$ then $\operatorname{dom} E \cap \operatorname{dom} E' = \varnothing$.

Proof. Induct on the length of E'. If $E' = \mathbf{nil}$, then $\mathrm{dom}\, E' = \varnothing$, so $\mathrm{dom}\, E \cap \mathrm{dom}\, E' = \varnothing$. Otherwise write $E' = (E'', \zeta{:}\tau)$. Then $E, E' \vdash_{hm}$ ok must have been derived by the appropriate (envok.*) rule, with the premises $E, E'' \vdash_{hm} \tau$ ok and $\zeta \notin \mathrm{dom}\,(E, E'')$. From the first premise, by Lemma E.6 (environments have to be ok), we get $E, E'' \vdash_{hm}$ ok by a subproof, whence $\mathrm{dom}\, E \cap \mathrm{dom}\, E'' = \varnothing$ by induction. Then $\mathrm{dom}\, E \cap \mathrm{dom}\, E' = (\mathrm{dom}\, E \cap \mathrm{dom}\, E'') \cup (\mathrm{dom}\, E \cap \{\zeta\}) = \varnothing \cup \varnothing = \varnothing$ as desired. □

Lemma E.9 (free variables of a judgement come from the environment) If $E \vdash_{hm} J$ then $\mathrm{fv}\,(J) \subseteq \mathrm{dom}\,(E)$. For completeness's sake: if $\vdash hm$ ok then $\mathrm{fv}\,(hm) \subseteq \varnothing$; if $\vdash n$ ok then $\mathrm{fv}\,(n) \subseteq \varnothing$.

Proof. We freely use Lemma E.3 (non-membership in 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 metavariable in the right-hand side of the conclusion $E \vdash_{hm} J$ 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 (TK.hash): The conclusion is $E \vdash_{hm} \mathbf{hash}(N, M{:}S){:}\mathbf{Type}$. One premise is $\vdash \mathbf{hash}(N, M{:}S)$. By induction we have $\mathrm{fv}\,(\mathbf{hash}(N, M{:}S)) = \varnothing$, so $\mathrm{fv}\,(\mathbf{hash}(N, M{:}S)) \subseteq \mathrm{dom}\, E$.

Case (Teq.hash): The conclusion is $E \vdash_{h} h == T$ where $h = \mathbf{hash}(N, [T, v^{hm}]{:}S)$. The premise is $E \vdash_{h}$ ok. By Lemma E.4 (colours have to be ok), $\vdash h$ by a proper subproof. By induction, we have $\mathrm{fv}\, h = \varnothing$ and in particular $\mathrm{fv}\, T = \varnothing$, so $\mathrm{fv}\,(h == T) = \varnothing \subseteq \mathrm{dom}\, E$.

Case (hmok.hash): The conclusion is $\vdash \mathbf{hash}(N, M{:}S)$ ok and the premise is $\mathbf{nil} \vdash_{\bullet} M{:}S$. By induction we have $\mathrm{fv}\,(\mathbf{hash}(N, M{:}S)) = \mathrm{dom}\,\mathbf{nil} = \varnothing$ as desired.

Case (hmok.zero): Trivial.

Cases T in (Sok); T and T' (Seq.struct) and (Ssub.struct); e in (eT.fun); m in (mT.let): 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 $E, \zeta{:}\tau \vdash_{hm} J'$ 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 K in (Sok); T in (eT.fun): In each case, there is a premise of the form $E, \zeta{:}\aleph \vdash_{hm} J'$ where E and hm are the environment and the colour of the conclusion and $\aleph$ is the metavariable under consideration. By Lemma E.6 (environments have to be ok) and reversing the appropriate (envok.*) rule, we have, by a proper subproof, respectively, $E \vdash_{hm} K$ ok, $E \vdash_{hm} T{:}\mathbf{Type}$, $E \vdash_{hm} M{:}S$. 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 E.5 (hashes have to be ok), for any hash h that is a subterm of T', we have $\vdash h$ ok by a (proper) subproof of Π. Thus, by induction, $\mathrm{fv}\, h = \varnothing$ and in particular $x \notin \mathrm{fv}\, h$. Given the syntax of types, the only place where T' might have a free expression variable is inside a hash, so $x \notin \mathrm{fv}\, T'$. Hence $\mathrm{fv}\, T' \subseteq \mathrm{dom}\, E$ as desired.

Cases (TK.var), (eT.var), (US.var): The variable (X, x or U respectively) that the similarly written metavariable instantiates to is obviously present in the environment.

Case (eT.col): The conclusion is $E \vdash_{hm} [e]^{T}_{hm'}{:}T$. 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 E.4 (colours have to be ok), we have $\vdash hm'$ ok by a proper subproof, so by induction $\mathrm{fv}\, hm' = \varnothing$ whence the desired result.

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

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

□

Definition E.10 (correctness judgement) A correctness judgement is a coloured judgement whose right-hand side is of one of the following forms:

$$\text{ok},\ K\ \text{ok},\ T{:}\mathbf{Type},\ S\ \text{ok}. \tag{1}$$

Note that a derivation of a correctness judgement may involve other sorts of judgements. For example, in order to derive $U{:}S \vdash_{hm} U.\text{TYPE}{:}\mathbf{Type}$, one has to use (TK.mod), with a premise of the form $U{:}S \vdash_{hm} U{:}S'$.

Definition E.11 (type world judgement) A type world judgement is a coloured judgement whose right-hand side is of one of the following forms:

$$\text{ok},\ K\ \text{ok},\ K <: K',\ K == K',\ T == T',\ T{:}K,\ S\ \text{ok},\ S == S',\ S <: S',\ U{:}S. \tag{2}$$

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.

E.3 Variables and colours

Definition E.12 (hashes in something) The hashes in a syntactic entity are the subterms that are $\mathbf{hash}(N, M{:}S)$ and that are not themselves subterms of a hash.

Definition E.13 (partial order on colours) We call *topped colours* the set formed by all colours (i.e. hm) plus a point $\top$ that is distinct from any colour.

We define a partial order on topped colours as follows: $hm \preccurlyeq hm'$ iff $hm = hm'$ or $hm = \bullet$ or $hm' = \top$.

Note that $\preccurlyeq$ defines a poset with all greatest lower bounds, with $\bullet$ as the minimum element and $\top$ as the maximum element.

Definition E.14 (pvu) Let Π be a proof of a coloured judgement $E \vdash_{hm} J$, and let ζ be any variable. The set of colours at which the variable ζ is used in the proof Π, written $\text{pvu}_\zeta(\Pi)$, 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 $\text{pvu}_\zeta(\Pi)$ 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, $\text{pvu}_\zeta(\Pi')$ where Π' is the subproof of Π that leads to said premise.

Note that with our current rules:

- Since (as we will prove) ok environments have no repetition in the domain, only one metavariable $\aleph$ is concerned. It may be either a variable metavariable ($\hat{\zeta}$) or an environment metavariable ($\hat{E}$).
- If $\zeta \notin \text{dom}\, E$ then there is no $\aleph$, thus $\text{pvu}_\zeta(\Pi) = \varnothing$.
- The first case only arises if $\aleph$ is a variable metavariable, and the rule is one of (*.var) introducing that variable.
- The second case never arises (the only rule with an explicit bracket is (eT.col), and it has nothing from the environment inside the bracket).

Definition E.15 (alternate, informal definition of pvu) Consider a proof Π of a coloured judgement of the form $E_0, \zeta{:}\tau, E \vdash_{hm} J$. The set of colours at which the variable ζ is pulled from the environment in Π, written $\text{pvu}_\zeta(\Pi)$, is the set of colours hm such that the judgement $E_0, \zeta{:}\tau, E' \vdash_{hm} \zeta{:}\tau$ 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 $\text{pvu}_\zeta(\Pi)$ to be the empty set.

Lemma E.16 (monotonicity of pvu) If Π' is a subproof of Π then $\text{pvu}_\zeta(\Pi') \subseteq \text{pvu}_\zeta(\Pi)$. Hence $\min\left(\text{pvu}_\zeta(\Pi)\right) \preccurlyeq \min\left(\text{pvu}_\zeta(\Pi')\right)$.

Proof. Trivial from the first definition. □

Definition E.17 (substitution) Some potentially interesting cases:

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

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

Lemma E.18 (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 E.9 (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 E.19 (computing the pvu of a type world judgement) Let Π be a proof of a type world judgement $E_0, \zeta{:}\tau, E \vdash_{hm} J$. Then $hm \preccurlyeq \min\left(\mathrm{pvu}_\zeta\,(\Pi)\right)$.

Proof. Induct on the structure of Π. We freely use Lemma E.6 (environments have to be ok) and Lemma E.8 (ok environments have no repetition in the domain).

Consider first the rules whose conclusion is a type world judgement but for which $J \neq \mathrm{ok}$. 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, $hm \preccurlyeq \min\left(\mathrm{pvu}_\zeta\,(\Pi')\right)$.

Case 2: Then $\mathrm{pvu}_\zeta\,(\Pi') = \varnothing$ whence $hm \preccurlyeq \top = \min\left(\mathrm{pvu}_\zeta\,(\Pi')\right)$.

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 $hm \preccurlyeq \min\left(\mathrm{pvu}_\zeta\,(\Pi)\right)$.

Let us turn to the (envok.*) rules. We distinguish as to whether ζ is the variable being added to the environment.

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

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

□

Note that this lemma does not hold for expression typing judgements. We suspect that it would hold for *shortest proofs*, for some notion of shortest proof. Here is a counter example, where the problem is the use of X in a "useless" fashion.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} \text{ok}}\,\text{(envok.X)}}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} ():\text{UNIT}} \quad \dfrac{\dfrac{\dfrac{\vdots}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} X{:}\mathbf{Eq}(\text{UNIT})}\,\text{(TK.var)}}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} X == \text{UNIT}}}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} \text{UNIT} == X}\,\text{(Teq.sym)}}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} ():X} \quad \dfrac{\dfrac{\vdots}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} X{:}\mathbf{Eq}(\text{UNIT})}\,\text{(TK.var)}}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} X == \text{UNIT}}}{X{:}\mathbf{Eq}(\text{UNIT}) \vdash_{hm} ():\text{UNIT}}$$

Lemma E.20 (connection between fv and fse) Let $\aleph$ be anything in the syntax. Then $\text{fv}\,\aleph \subseteq \text{fse}\,\aleph$. If $x \in \text{fse}\,\aleph$ then $x \in \text{fv}\,\aleph$. If $X \in \text{fse}\,\aleph$ then $X \in \text{fv}\,\aleph$. If $U \in \text{fse}\,\aleph$ or $U.\text{term} \in \text{fse}\,\aleph$ or $U.\text{TYPE} \in \text{fse}\,\aleph$ then $U \in \text{fv}\,\aleph$.

We may use this lemma implicitly.

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

Lemma E.21 (types do not contain free expression variables) If $E \vdash_{hm} T{:}\mathbf{Type}$ then $\text{fse}\,T$ does not contain any expression substitutable entity (i.e. $\ddot{x}$). Also, if $E \vdash_{hm} K\,\text{ok}$ (respectively $E \vdash_{hm} S\,\text{ok}$) then $\text{fse}\,K$ (respectively $\text{fse}\,S$) does not contain any expression substitutable entity.

Note that $\text{fse}\,\aleph$ not containing any expression substitutable entity implies that $\text{fv}\,\aleph$ 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, $U.\text{TYPE}$) or obvious by induction (constructed type). The only non-trivial case is a hash h. By Lemma E.5 (hashes have to be ok), $\vdash h\,\text{ok}$. By Lemma E.9 (free variables of a judgement come from the environment), $\text{fv}\,h = \varnothing$, whence by Lemma E.20 (connection between fv and fse) $\text{fse}\,h = \varnothing$.

If $E \vdash_{hm} \mathbf{Eq}(T)\,\text{ok}$, then by reversing (Kok.Eq) we get $E \vdash_{hm} T{:}\mathbf{Type}$, whence by the first part of this lemma $\text{fse}\,K = \text{fse}\,T$ has the desired property. The case $E \vdash_{hm} \mathbf{Type}\,\text{ok}$ is trivial ($\text{fse}\,\mathbf{Type} = \varnothing$).

If $E \vdash_{hm} [X{:}K, T]\,\text{ok}$, then by reversing (Sok) we get $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$. By the previous two paragraphs, neither $\text{fse}\,K$ nor $\text{fse}\,T$ contains any expression substitutable entity, so the same holds for $\text{fse}\,S = \text{fse}\,K \cup (\text{fse}\,T \setminus X)$. ☐

Lemma E.22 (environments do not contain free expression variables) If $E_0, E_1 \vdash_{hm} \text{ok}$ then $\text{fse}\,E_1$ does not contain any expression substitutable entity.

Note that in particular $\text{fv}\,E_1$ 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 reversing the appropriate (envok.*) rule, we get $E_0, E_1' \vdash_{hm} \tau\,\text{ok}$. By Lemma E.6 (environments have to be ok), we have $E_0, E_1' \vdash_{hm} \text{ok}$. By induction, we have $\ddot{x} \notin \text{fse}\,E_1'$. Also, by Lemma E.21 (types do not contain free expression variables), $\ddot{x} \notin \text{fse}\,\tau$ (whether τ is a type, kind or signature). Hence $\ddot{x} \notin \text{fse}\,E_1$. ☐

Lemma E.23 (expression substitution in environment) If $E_0, E_1 \vdash_{hm} \text{ok}$ and $\ddot{x}$ is an expression substitutable entity (i.e. an expression variable or $U.\text{term}$ for some U) then $\{\ddot{x} \leftarrow \eta\}E_1 = E_1$.

Proof. Trivial consequence of Lemma E.22 (environments do not contain free expression variables). ☐

F Type preservation by substitution

Lemma F.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 F.2 (colour stripping judgements) If $E \vdash_{hm} J$ and $\vdash hm'\,\text{ok}$ and $hm \preccurlyeq hm'$ then $E \vdash_{hm'} J$ for all coloured statements J other than "type of a machine".

Proof. Induct on the derivation of $E \vdash_{hm} J$. 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 F.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 $E \vdash_{hm'} J$. We list the remaining cases.

Case (Teq.hash): Then $hm \neq \bullet$, so $hm' = hm$.

Case (TK.hash): Trivial ($\vdash hm'$ ok is assumed in this lemma).

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

Case (eT.marred): Trivial.

Case (eT.col): Trivial.

□

Lemma F.3 (weakening) If $E, E', E'' \vdash_{hm}$ ok and $E, E'' \vdash_{hm} J$ then $E, E', E'' \vdash_{hm} J$.

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

Proof. We freely use Lemma E.3 (non-membership in domain is interpreted trivially).

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

Induct on the derivation of $E, E'' \vdash_{hm} J$.

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.
   (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 $E, E'' \vdash_{hm} J$ 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 $E, E'', E_i \vdash_{hm_i} J_i$, where E_i is of length at most one. By Lemma E.6 (environments have to be ok), $E, E'', E_i \vdash_{hm_i} \text{ok}$ by a subproof.

If E_i is empty, then we have $E, E', E'', E_i \vdash_{hm_i} \text{ok}$. 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.*) rule from $E, E'' \vdash_{hm_i} \tau\,\text{ok}$ and $\zeta \notin \mathrm{dom}\,(E, E'')$. By induction, we have $E, E', E'' \vdash_{hm_i} \tau\,\text{ok}$, whence by the same (envok.*) rule as previously: $E, E', E'', \zeta{:}\tau \vdash_{hm_i} \text{ok}$ (recalling that we performed alpha-conversion on the proof so that $\zeta \notin \mathrm{dom}\, E''$ whence $\zeta \notin \mathrm{dom}\,(E, E', E'')$). If furthermore $\zeta \in \mathrm{dom}\, E'$ and $\mathrm{pvu}_\zeta\,(\Pi) = \varnothing$, then the induction gives a proof Π'' of $E, E', E'', x{:}T \vdash_{hm_i} \text{ok}$ such that $\mathrm{pvu}_\zeta\,(\Pi'') = \varnothing$.

In any case, $E, E', E'', E_i \vdash_{hm_i} \text{ok}$, so we can apply induction, getting $E, E', E'', E_i \vdash_{hm_i} J_i$ as desired.

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

Case 2b: The premise is $\zeta' \notin \mathrm{dom}\,(E, E''')$ with ζ' in $\mathrm{dom}\,(E, E''', E'''')$. Hence $\zeta' \in \mathrm{dom}\, E''''$. Since $E, E', E''', E'''' \vdash_{hm} \text{ok}$, by Lemma E.8 (ok environments have no repetition in the domain), we have $\zeta \notin \mathrm{dom}\,(E, E', E''')$ as desired.

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

We have a derivation of all the premises in ω, so we get a proof Π' of its conclusion. Note further that if $\zeta \in \mathrm{dom}\, E'$ and $\mathrm{pvu}_\zeta\,(\Pi) = \varnothing$, we do get that $\mathrm{pvu}_\zeta\,(\Pi') = \varnothing$. (If ω is an instance of a (*.var) rule, then ζ is not the variable being introduced by $\zeta \in \mathrm{dom}\, E'$ and Lemma E.8 (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.*): 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 $E''', \zeta'{:}\tau, E'''', E', E'' \vdash_{hm} \text{ok}$. By (*.var) we get a proof Π' of $E''', \zeta'{:}\tau, E'''', E', E'' \vdash_{hm} \zeta'{:}\tau$ as desired.

If $\zeta \in \mathrm{dom}\, E'$ and $\mathrm{pvu}_\zeta\,(\Pi) = \varnothing$, then $\mathrm{pvu}_\zeta\,(\Pi') = \varnothing$, because $\zeta \neq \zeta'$ since $\zeta \in \mathrm{dom}\, E'$ and Lemma E.8 (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 $\mathbf{nil} \vdash_{hm} \text{ok}$, and we desire a proof of $E' \vdash_{hm} \text{ok}$, which holds by assumption: take $\Pi' = \Pi$. □

Lemma F.4 (merging environments) If $E, E' \vdash_{hm} \text{ok}$ and $E, E'' \vdash_{hm} \text{ok}$ and $\mathrm{dom}\, E' \cap \mathrm{dom}\, E'' = \varnothing$ then $E, E', E'' \vdash_{hm} \text{ok}$.

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

Proof. We freely use Lemma E.3 (non-membership in 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 $E, E'', \zeta{:}\tau \vdash_{hm} \text{ok}$ and $\mathrm{dom}\, E' \cap \mathrm{dom}\,(E'', \zeta{:}\tau) = \varnothing$.

By reversing the appropriate (envok.*) rule, we get $E, E'' \vdash_{hm} \tau\,\text{ok}$. By Lemma E.6 (environments have to be ok), we have $E, E'' \vdash_{hm} \text{ok}$. By induction, we get $E, E', E'' \vdash_{hm} \text{ok}$. By Lemma F.3 (weakening), we have $E, E', E'' \vdash_{hm} \tau\,\text{ok}$. Then by applying the appropriate (envok.*) rule we get $E, E', E'', \zeta{:}\tau \vdash_{hm} \text{ok}$ as desired.

Suppose furthermore that $\zeta \in \mathrm{dom}\, E'$ and $\mathrm{pvu}_\zeta\,(\Pi) = \varnothing$. Then the proof of $E, E', E'' \vdash_{hm} \text{ok}$ obtained above by induction and that of $E, E', E'' \vdash_{hm} \tau\,\text{ok}$ obtained by Lemma F.3 (weakening) have an empty pvu for ζ, so the proof Π' of $E, E', E'', \zeta{:}\tau$ that we build satisfies $\mathrm{pvu}_\zeta\,(\Pi') = \varnothing$. □

Lemma F.5 (combined weakening) If $E, E' \vdash_{hm} \text{ok}$ and $E, E'' \vdash_{hm} J$ and $\mathrm{dom}\, E' \cap \mathrm{dom}\, E'' = \varnothing$ then $E, E', E'' \vdash_{hm} J$.

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

Proof. Trivial combination of Lemma F.4 (merging environments) and Lemma F.3 (weakening). □

Lemma F.6 (kinds are smaller than Type) If $E \vdash_{hm} K\,\text{ok}$ then $E \vdash_{hm} K <: \mathbf{Type}$.

Proof. If $K = \mathbf{Type}$, then we get $E \vdash_{hm} \mathbf{Type} <: \mathbf{Type}$ by reversing (Kok.Type) and applying (Keq.Type) and (Ksub.refl). Otherwise there exists T such that $K = \mathbf{Eq}(T)$. Then by reversing (Kok.Eq) and applying (Ksub.Eq) we get $E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Type}$. □

Lemma F.7 (relating type-is-kind and subkinding) If $E \vdash_{hm} T{:}K$ then $E \vdash_{hm} \mathbf{Eq}(T) <: K$.

Proof. If $K = \mathbf{Type}$, then we get the desired result by (Ksub.Eq). Otherwise there exists T' such that $K = \mathbf{Eq}(T')$. Then by (Teq.Eq), (Keq.Eq) and (Ksub.refl) we get $E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Eq}(T')$ as desired. □

Lemma F.8 (components of modules are ok) If $E \vdash_{hm} [T, v^{hm}]{:}[X{:}K, T']$ then there exists T'' such that $E \vdash_{hm} T{:}K$ and $E, X{:}K \vdash_{hm} T'{:}\mathbf{Type}$ and $E, X{:}\mathbf{Eq}(T) \vdash_{hm} T'' == T'$ and $E \vdash_{hm} v^{hm}{:}T''$ and $E \vdash_{hm} K$ ok.

Proof. Reverse (MS.struct) to get the first four judgements. As for the last one, since $E, X{:}K \vdash_{hm} T'{:}\mathbf{Type}$, by Lemma E.6 (environments have to be ok) and reversing (envok.X), we get $E \vdash_{hm} K$ ok. □

Lemma F.9 (types are ok provided their hashes are) $E \vdash_{hm} T{:}\mathbf{Type}$ iff fv $T \subseteq \mathrm{dom}\, E$ and $E \vdash_{hm}$ ok and all the hashes in T are ok.

Proof. Suppose that fv $T \subseteq E$ and all the hashes in T are ok. We prove that $E \vdash_{hm} T{:}\mathbf{Type}$ by induction on the syntax of T.

Case $T = \text{UNIT}$ **or** $T = \text{STRING}$**:** Trivial by (TK.unit) or (TK.mar).

Case there exist $T_1, ..., T_j$ **such that** $T = T_1 * ... * T_j$**:** Note that fv $T_i \subseteq$ fv $T \subseteq \mathrm{dom}\, E$. By induction, $E \vdash_{hm} T_i$ for $1 \leqslant i \leqslant j$. By (TK.tuple), we have $E \vdash_{hm} T_1 * ... * T_j{:}\mathbf{Type}$.

Case there exist T_1, T_2 **such that** $T = T_1 \to T_2$**:** Note that fv $T_i \subseteq$ fv $T \subseteq \mathrm{dom}\, E$. By induction, $E \vdash_{hm} T_i$ for $1 \leqslant i \leqslant 2$. By (TK.fun), we have $E \vdash_{hm} T_1 \to T_2{:}\mathbf{Type}$.

Case T **is a hash** h**:** The hash h is ok by assumption, so we get $E \vdash_{hm} h{:}\mathbf{Type}$ by (TK.hash).

Case T **is a type variable** X**:** Since $\{X\} =$ 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 $E \vdash_{hm} X{:}K$. By Lemma E.7 (prefixes of ok environments are ok) and reversing (envok.X), we have $E_1 \vdash_{hm} K$ ok, whence by Lemma F.3 (weakening) $E \vdash_{hm} K$ ok. By Lemma F.6 (kinds are smaller than Type), we have $E \vdash_{hm} K <: \mathbf{Type}$, whence by (TK.sub) $E \vdash_{hm} X{:}\mathbf{Type}$ as desired.

Case there exists U **such that** $T = U.\text{TYPE}$**:** Since $\{U\} =$ fv $T \subseteq \mathrm{dom}\, E$, there exist E_1, K, T' and E_2 such that $E = E_1, U{:}[X{:}K, T'], E_2$. By (US.var) and (TK.mod), we get $E \vdash_{hm} U.\text{TYPE}{:}K$. By Lemma E.7 (prefixes of ok environments are ok) and reversing (envok.U), we have $E_1 \vdash_{hm} [X{:}K, T']$ ok, whence by Lemma F.3 (weakening) $E \vdash_{hm} [X{:}K, T']$ ok. By reversing (Sok), we have $E \vdash_{hm} K$ ok. By Lemma F.6 (kinds are smaller than Type), we have $E \vdash_{hm} K <: \mathbf{Type}$, whence by (TK.sub) $E \vdash_{hm} U.\text{TYPE}{:}\mathbf{Type}$ as desired.

Now suppose $E \vdash_{hm} T{:}\mathbf{Type}$. Then fv $T \subseteq \mathrm{dom}\, E$ by Lemma E.9 (free variables of a judgement come from the environment). Also all the hashes in T are ok by Lemma E.5 (hashes have to be ok). □

Lemma F.10 (colour change preserves type okedness) If $\mathbf{nil} \vdash_{hm_0} T{:}\mathbf{Type}$ and $\mathbf{nil} \vdash_{hm_1}$ ok then $\mathbf{nil} \vdash_{hm_1} T{:}\mathbf{Type}$.

Proof. Trivial application of Lemma F.9 (types are ok provided their hashes are). □

Definition F.11 (unresolved free variables of an environment) The unresolved free variables of an environment, written ufv E, are defined as follows:

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

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

Lemma F.12 (computing unresolved free variables) ufv $(E, E') =$ ufv $E \cup$ (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 F.13 (ok environments have no unresolved free variables) If $E \vdash_{hm}$ ok then $\mathrm{ufv}\,E = \varnothing$.

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

The remaining rules are (envok.*). If we write the conclusion as $E, \zeta{:}\tau \vdash_{hm}$ ok, one premise is $E \vdash_{hm} \tau\,\mathrm{ok}$. We have $\mathrm{ufv}\,E = \varnothing$ by induction. By Lemma E.9 (free variables of a judgement come from the environment), we have $\mathrm{fv}\,\tau \subseteq \mathrm{dom}\,E$. By Lemma F.12 (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 F.14 (type preservation by substitution) If $E_0, \zeta{:}\tau, E \vdash_{hm} J$ by a proof Π such that $hm' \preccurlyeq \min\left(\mathrm{pvu}_\zeta\,(\Pi)\right)$ and $E_0 \vdash_{hm'} \eta{:}\tau$ then $E_0, \sigma E \vdash_{hm} \sigma J$ where $\sigma = \{\zeta \leftarrow \eta\}$ and $\zeta{:}\tau$ is an expression or type binding.

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

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 $E_0, \zeta{:}\tau, E \vdash_{hm} J$ 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 $E_0, \zeta{:}\tau, E \vdash_{hm} J$ 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 E.18 (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 $\sigma E_0, \sigma E'' \vdash_{hm} \sigma J$, which is also $E_0, \sigma E'' \vdash_{hm} J$ as $\zeta \notin \mathrm{fv}\,E_0$ by Lemma E.6 (environments have to be ok), Lemma E.7 (prefixes of ok environments are ok) and Lemma F.13 (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 $E_0, \zeta{:}\tau, E'', E'_i \vdash_{hm_i} J_i$. Given Lemma E.16 (monotonicity of pvu), we can apply induction, getting $E_0, \sigma E'', \sigma E'_i \vdash_{hm_i} \sigma J_i$, 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 E.9 (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: $E_0, \sigma E'', \sigma E' \vdash_{hm} \sigma J$, 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.*): Then $E = \mathbf{nil}$ and $E_2 = \mathbf{nil}$. The proof obligation is $E_1 \vdash_{hm}$ ok, i.e. $E_0 \vdash_{hm}$ ok, which holds by Lemma E.7 (prefixes of ok environments are ok).

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

$$\frac{E_0, \zeta{:}\tau, E' \vdash_{hm} \text{ ok}}{E_0, \zeta{:}\tau, E' \vdash_{hm} \zeta{:}\tau}$$

and $\sigma = \{\zeta \leftarrow \eta\}$, and we have $E_0 \vdash_{hm'} \eta{:}\tau$ with $hm' \preccurlyeq \min\left(\mathrm{pvu}_\zeta\,(\Pi)\right) \preccurlyeq hm$. By induction (which we can apply thanks to Lemma E.16 (monotonicity of pvu)), we have $E_0, \sigma E' \vdash_{hm}$ ok.

Furthermore, since $\vdash hm$ ok by Lemma E.4 (colours have to be ok), $E_0 \vdash_{hm} \eta{:}\tau$ by Lemma F.2 (colour stripping judgements). By Lemma F.3 (weakening) we get $E_0, \sigma E' \vdash_{hm} \eta{:}\tau$ as desired.

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

Lemma F.15 (strengthening) If $E_0, \zeta{:}\tau, E \vdash_{hm} J$ and $\zeta \notin \mathrm{fv}\, E \cup \mathrm{fv}\, J$ and $\zeta{:}\tau$ is a type or expression variable binding then $E_0, E \vdash_{hm} J$.

Proof. By Lemma E.6 (environments have to be ok) and Lemma E.7 (prefixes of ok environments are ok), we have $E_0, \zeta{:}\tau \vdash_{hm}$ ok. Let us then reverse the rule (envok.*) that was applied to obtain this latter judgement:

Case $\zeta{:}\tau$ is $X{:}K$: Then we have $E_0 \vdash_{hm} K$ ok. If $K = \mathbf{Type}$, by (TK.unit), we have $E_0 \vdash_{hm} \mathrm{UNIT}{:}K$; let η be UNIT. Otherwise there exists a type η such that $K = \mathbf{Eq}(\eta)$; by reversing (Kok.Eq), we get $E_0 \vdash_{hm} \eta{:}\mathbf{Type}$. By Lemma F.10 (colour change preserves type okedness), we have $E_0 \vdash_{\bullet} \eta{:}\mathbf{Type}$. By (Teq.refl) and (TK.Eq), we get $E_0 \vdash_{hm} \eta{:}K$.

Case $\zeta{:}\tau$ is $x{:}T$: Then we have $E_0 \vdash_{hm} T{:}\mathbf{Type}$, whence by Lemma F.10 (colour change preserves type okedness): $E_0 \vdash_{\bullet} T{:}\mathbf{Type}$. Let η be $\mathbf{UnmarFailure}^T$. By (eT.Undynfailure), we have $E_0 \vdash_{\bullet} \eta{:}T$.

In any case, we have $E_0 \vdash_{\bullet} \eta{:}\tau$. By Lemma F.14 (type preservation by substitution), we have $E_0, \sigma E \vdash_{hm} \sigma J$ 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 $E_0, E \vdash_{hm} J$ as desired. □

This proof of strengthening calls the main substitution lemma. This has two inconveniences:

- It does not work for module variables. In fact, if we expanded the proof of the substitution lemma, we would see that the (envok.*) case dealing with ζ is never invoked, so it does not matter than the substitution lemma does not have such a case for module variables.

- It requires that kinds and types not be empty, which is true for a convoluted reason (all our kinds happen to be non-empty, and each of our types T contains the exception $\mathbf{UnmarFailure}^T$).

Definition F.16 (visible type-part of a module)

$$\mathrm{typepart}\,(N, [T, v^{hm}]{:}[X{:}K, T']) = \begin{cases} \mathbf{hash}(N, [T, v^{hm}]{:}[X{:}K, T']) & \text{if } K = \mathbf{Type} \\ T_0 & \text{if } K = \mathbf{Eq}(T_0) \text{ for some } T_0 \end{cases}$$

Definition F.17 (visible term-part of a module)

$$\operatorname{termpart}(N, [T, v^{hm}]{:}[X{:}K, T']) = \begin{cases} [v^{hm}]_h^{\{X \leftarrow h\}T'} & \text{if } K = \mathbf{Type} \\ v^{hm} & \text{if } K = \mathbf{Eq}(T_0) \text{ for some } T_0 \end{cases}$$

where $h = \mathbf{hash}(N, [T, v^{hm}]{:}[X{:}\mathbf{Type}, T'])$.

Definition F.18 (set of equations used to type a module)

$$\operatorname{equations}(N, [T, v^{hm}]{:}[X{:}K, T']) = \begin{cases} \mathbf{hash}(N, [T, v^{hm}]{:}[X{:}K, T']) & \text{if } K = \mathbf{Type} \\ \bullet & \text{if } K = \mathbf{Eq}(T_0) \text{ for some } T_0 \end{cases}$$

Lemma F.19 (reflexivity of kind equivalence) If $E \vdash_{hm} K$ ok then $E \vdash_{hm} K == K$.

Proof. If $K = \mathbf{Type}$ then this lemma holds by (Keq.Type). Otherwise there exists T such that $K = \mathbf{Eq}(T)$. The judgement $E \vdash_{hm} K$ ok must have been derived by (Kok.Eq), from $E \vdash_{hm} T{:}\mathbf{Type}$. Then by (Teq.refl) followed by (Keq.Eq) we have $E \vdash_{hm} \mathbf{Eq}(T) == \mathbf{Eq}(T)$ as desired. ☐

Lemma F.20 (weakening kind to ok kind in the environment) If $E_0 \vdash_{hm} K <: K'$ and $E_0 \vdash_{hm} K$ ok and $E_0, X{:}K', E_1 \vdash_{hm} J$ and J is a type world judgement right-hand side then $E_0, X{:}K, E_1 \vdash_{hm} J$.

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

Proof. Since $E_0 \vdash_{hm} K$ ok, by (envok.X), $E_0, Y{:}K \vdash_{hm}$ ok where Y is fresh. By Lemma F.3 (weakening), from $E_0, Y{:}K \vdash_{hm}$ ok and $E_0 \vdash_{hm} K'$ ok, we get $E_0, Y{:}K \vdash_{hm} K'$ ok. By (envok.X), $E_0, Y{:}K, X{:}K' \vdash_{hm}$ ok.

By Lemma F.5 (combined weakening), $E_0, Y{:}K, X{:}K', E_1 \vdash_{hm} J$.

From $E_0, Y{:}K \vdash_{hm}$ ok, by (TK.var), we get $E_0, Y{:}K \vdash_{hm} Y{:}K$. By Lemma F.3 (weakening), we also get $E_0, Y{:}K \vdash_{hm} K <: K'$. By (TK.sub), we get $E_0, Y{:}K \vdash_{hm} Y{:}K'$. By Lemma F.14 (type preservation by substitution), using Lemma E.19 (computing the pvu of a type world judgement) $E_0, Y{:}K', \{X \leftarrow Y\}E_1 \vdash_{hm} \{X \leftarrow Y\}J$. By alpha-conversion, we have $E_0, X{:}K', E_1 \vdash_{hm} J$ as desired. ☐

Lemma F.21 (things have to be ok) If $E \vdash_{hm} T{:}K$ then $E \vdash_{hm} K$ ok.
If $E \vdash_{hm} T == T'$ then $E \vdash_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T'{:}\mathbf{Type}$.
If $E \vdash_{hm} K == K'$ or $E \vdash_{hm} K <: K'$ then $E \vdash_{hm} K$ ok and $E \vdash_{hm} K'$ ok.
If $E \vdash_{hm} S == S'$ or $E \vdash_{hm} S <: S'$ then $E \vdash_{hm} S$ ok and $E \vdash_{hm} S'$ ok.
If $E \vdash_{hm} e{:}T$ or $E \vdash_{hm} m{:}T$ then $E \vdash_{hm} T{:}\mathbf{Type}$.
If $E \vdash_{hm} M{:}S$ or $E \vdash_{hm} U{:}S$ then $E \vdash_{hm} S$ ok.

Proof. Induct on the size of the derivation of the hypothesis. Consider the last rule used in said derivation.

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

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

Case (TK.mod): The conclusion is $E \vdash_{hm} U.\mathrm{TYPE}{:}K$. The premise is $E \vdash_{hm} U{:}[X{:}K, T]$. By induction we have $E \vdash_{hm} [X{:}K, T]$ ok. This must have been obtained by applying (Sok), with the premise $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$. By Lemma E.6 (environments have to be ok), we have $E, X{:}K \vdash_{hm}$ ok that is smaller. Hence, by reversing (envok.X), we get a proof of $E \vdash_{hm} K$ ok.

Cases (TK.unit), (TK.fun), (TK.mar), (TK.tuple), (TK.hash): The conclusion is of the form $E \vdash_{hm} T{:}\mathbf{Type}$ for some T. By Lemma E.6 (environments have to be ok), $E \vdash_{hm}$ ok. By (Kok.Type), we have $E \vdash_{hm} \mathbf{Type}$ ok.

Case (TK.var): The conclusion is of the form $E, X{:}K, E' \vdash_{hm} X{:}K$. The premise is $E, X{:}K, E' \vdash_{hm}$ ok. By Lemma E.7 (prefixes of ok environments are ok) and reversing (envok.X), we get $E \vdash_{hm} K$ ok. By Lemma F.3 (weakening), we get $E, X{:}K, E' \vdash_{hm} K$ ok as desired.

Note that the proof may be bigger, since the part that proves K has to be duplicated...

Case (Teq.Eq): The conclusion is $E \vdash_{hm} T == T'$. The premise is $E \vdash_{hm} T{:}\mathbf{Eq}(T')$. By induction we have $E \vdash_{hm} \mathbf{Eq}(T')\,\mathrm{ok}$, which must have been derived by (Kok.Eq) from $E \vdash_{hm} T'{:}\mathbf{Type}$. Also, from this, we get $E \vdash_{hm} \mathbf{Eq}(T') <: \mathbf{Type}$ by (Ksub.Eq). We can apply (TK.sub) to get $E \vdash_{hm} T{:}\mathbf{Type}$.

Note that this may well be the shortest way to obtain $E \vdash_{hm} T{:}\mathbf{Type}$ (take $T = U.\mathrm{TYPE}$).

Case (Teq.hash): The conclusion is $E \vdash_{hm} hm == T$, and $hm = h = \mathbf{hash}(N, [T, v^{hm_1}]{:}S)$. The premise is $E \vdash_{hm}$ ok. By Lemma E.4 (colours have to be ok), we get $\vdash \mathbf{hash}(N, [T, v^{hm_1}]{:}S)\,\mathrm{ok}$. By reversing (hmok.hash) and (MS.struct), we get $\mathbf{nil} \vdash_{\bullet} T{:}\mathbf{Type}$. By Lemma F.3 (weakening), we get $E \vdash_{hm} T{:}\mathbf{Type}$. We also have $E \vdash_{h} h{:}\mathbf{Type}$ by (TK.hash).

Case (Teq.refl): Trivial.

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

Case (Teq.cong.fun): The conclusion is $E \vdash_{hm} T_0 \to T_1 == T'_0 \to T'_1$. By induction on the premises, we get $E \vdash_{hm} T_j{:}\mathbf{Type}$ and $E \vdash_{hm} T'_j{:}\mathbf{Type}$ for $j = 0, 1$. By (TK.fun), we get $E \vdash_{hm} T_0 \to T_1{:}\mathbf{Type}$ and $E \vdash_{hm} T'_0 \to T'_1{:}\mathbf{Type}$.

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

Case (Ksub.Eq): The conclusion is $E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Type}$. The premise is $E \vdash_{hm} T{:}\mathbf{Type}$. From this, by (Kok.Eq), we get $E \vdash_{hm} \mathbf{Eq}(T)\,\mathrm{ok}$. By Lemma E.6 (environments have to be ok), we have $E \vdash_{hm}$ ok. Thus, by (Kok.Type), we have $E \vdash_{hm} \mathbf{Type}\,\mathrm{ok}$.

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

Case (Keq.Type): The premise is $E \vdash_{hm}$ ok. By (Kok.Type), we get $E \vdash_{hm} \mathbf{Type}\,\mathrm{ok}$.

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

Case (Ssub.struct): The conclusion is $E \vdash_{hm} [X{:}K, T] <: [X{:}K', T']$. One premise is $E, X{:}K \vdash_{hm} T == T'$. By induction, we get $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T'{:}\mathbf{Type}$, whence the desired results by (Sok).

Case (Seq.struct): Similar to case (Ssub.struct).

Case (Ssub.refl): Trivial.

Case (Ssub.tran): Trivial.

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

Case (eT.eq): The conclusion is $E \vdash_{hm} e{:}T'$. One premise is $E \vdash_{hm} T == T'$. By induction we get $E \vdash_{hm} T'{:}\mathbf{Type}$.

Case (eT.mod): The conclusion is $E \vdash_{hm} U.\mathrm{term}{:}T$. One premise is $E \vdash_{hm} T{:}\mathbf{Type}$.

Case (eT.ap): The conclusion is $E \vdash_{hm} e\,e'{:}T'$. One premise is $E \vdash_{hm} e'{:}T \to T'$. By induction we get $E \vdash_{hm} T \to T'{:}\mathbf{Type}$. By Lemma F.9 (types are ok provided their hashes are) applied to $E \vdash_{hm} T \to T'$ then to $E \vdash_{hm} T'$, we get $E \vdash_{hm} T'{:}\mathbf{Type}$ as desired.

Case (eT.fun): The conclusion is $E \vdash_{hm} \lambda x{:}T.e{:}T \to T'$. The premise is $E, x{:}T \vdash_{hm} e{:}T'$. By induction, we get $E, x{:}T \vdash_{hm} T'{:}\mathbf{Type}$ by a proof Π. By Lemma E.21 (types do not contain free expression variables), $x \notin \mathrm{fv}\,T'$. By Lemma E.6 (environments have to be ok), we have $E \vdash_{hm}$ ok. By Lemma F.9 (types are ok provided their hashes are) applied once in each direction, we get first that the hashes in T' are ok and $\mathrm{fv}\,T' \subseteq \mathrm{dom}\,E \cup \{\mathrm{fv}\}$ (hence $\mathrm{fv}\,T' \subseteq \mathrm{dom}\,E$), then that $E \vdash_{hm} T'\,\mathrm{ok}$. Also, by Lemma E.6 (environments have to be ok), we get $E, x{:}T \vdash_{hm}$ ok, whence $E \vdash_{hm} T{:}\mathbf{Type}$ by reversing (envok.x). By (TK.fun), we get $E \vdash_{hm} T \to T'{:}\mathbf{Type}$.

Cases (eT.send), (eT.recv), (eT.mar), (eT.marred), (eT.unit): By Lemma E.6 (environments have to be ok), we have $E \vdash_{hm}$ ok. Then (TK.unit) or (TK.mar) gives the desired result.

Cases (eT.unmar), (eT.Undynfailure), (eT.col): Trivial (one of the premises is what we need to prove).

Case (eT.tuple): The conclusion is $E \vdash_{hm} (e_1, ..., e_j){:}T_1 * ... * T_j$. The premises are $E \vdash_{hm} e_i{:}T_i$ for $1 \leqslant i \leqslant j$. By induction, we have $E \vdash_{hm} T_i{:}\mathbf{Type}$ for all i, whence by (TK.tuple): $E \vdash_{hm} T_1 * ... * T_j{:}\mathbf{Type}$.

Case (eT.proj): Similar to (eT.ap).

Case (MS.struct): The conclusion is $E \vdash_{hm} M{:}[X{:}K, T]$. One premise is $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$, whence by (Sok): $E \vdash_{hm} [X{:}K, T]\,\mathrm{ok}$.

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

Case (US.sub): The conclusion is $E \vdash_{hm} U{:}S'$. One premise is $E \vdash_{hm} S <: S'$. By induction, we have $E \vdash_{hm} S'\,\mathrm{ok}$.

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

By Lemma E.6 (environments have to be ok) and reversing (envok.X), we have $E \vdash_{hm} K\,\mathrm{ok}$. By Lemma F.6 (kinds are smaller than Type), we get $E \vdash_{hm} K <: \mathbf{Type}$. By (TK.sub), given that $E \vdash_{hm} U.\mathrm{TYPE}{:}K$, we have $E \vdash_{hm} U.\mathrm{TYPE}{:}\mathbf{Type}$, whence by (Kok.Eq): $E \vdash_{hm} \mathbf{Eq}(U.\mathrm{TYPE})\,\mathrm{ok}$.

If $K = \mathbf{Type}$, by Lemma F.6 (kinds are smaller than Type), we get $E \vdash_{hm} \mathbf{Eq}(U.\mathrm{TYPE}) <: \mathbf{Type}$. Otherwise there exists T_1 such that $K = \mathbf{Eq}(T_1)$. From $E \vdash_{hm} U.\mathrm{TYPE}{:}\mathbf{Eq}(T_1)$, by (Teq.Eq), (Keq.Eq) and (Ksub.refl), we get $E \vdash_{hm} \mathbf{Eq}(U.\mathrm{TYPE}) <: \mathbf{Eq}(T_1)$. In either case we have $E \vdash_{hm} \mathbf{Eq}(U.\mathrm{TYPE}) <: K$.

By Lemma F.20 (weakening kind to ok kind in the environment), $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} T{:}\mathbf{Type}$. Hence by (Sok) we have $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T]\,\mathrm{ok}$.

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

□

Lemma F.22 (weakening kind in the environment) If $E_0 \vdash_{hm} K <: K'$ and $E_0, X{:}K', E_1 \vdash_{hm} J$ and J is a type world judgement right-hand side then $E_0, X{:}K, E_1 \vdash_{hm} J$.

Proof. By Lemma F.21 (things have to be ok), $E_0 \vdash_{hm} K\,\mathrm{ok}$. By Lemma F.20 (weakening kind to ok kind in the environment), we get the desired result. □

Lemma F.23 (type preservation by guarded expression variable substitution) If $E_0, x{:}T, E \vdash_{hm} J$ and $E_0 \vdash_{hm'} e{:}T$ and $E_0 \vdash_{\bullet}$ ok then $E_0, \sigma E \vdash_{hm} \sigma J$ where $\sigma = \{x \leftarrow [e]^T_{hm'}\}$.

Proof. $E_0 \vdash_{hm'} T{:}\mathbf{Type}$ by Lemma F.21 (things have to be ok). By Lemma F.9 (types are ok provided their hashes are) applied one in each direction, we get $E_0 \vdash_{\bullet} T{:}\mathbf{Type}$. Applying (eT.col) to this and $E_0 \vdash_{hm'} e{:}T$ yields $E_0 \vdash_{\bullet} [e]^T_{hm'}{:}T$. We can now apply Lemma F.14 (type preservation by substitution) to get the desired result. □

Lemma F.24 (type equivalence is a congruence) If $E \vdash_{hm} T' == T''$ and $E, X{:}\mathbf{Type} \vdash_{hm} T{:}\mathbf{Type}$ then $E \vdash_{hm} \{X \leftarrow T'\}T == \{X \leftarrow T''\}T$.

Proof. Induct on the structure of T.

Case $T = \mathrm{UNIT}$ or $T = \mathrm{STRING}$ or $T = U.\mathrm{TYPE}$ or $T = Y \neq X$ or $T = h_1$: Then $X \notin \mathrm{fv}\, T$ (if $T = h_1$, this is because $\mathrm{fv}\, T = \varnothing$ by Lemma E.5 (hashes have to be ok) and Lemma E.9 (free variables of a judgement come from the environment)). By Lemma F.9 (types are ok provided their hashes are), the hashes of T are ok and $\mathrm{fv}\, T \subseteq \mathrm{dom}\, E \cup \{X\}$. By Lemma F.9 (types are ok provided their hashes are) in the other direction, since $\mathrm{fv}\, T \subseteq E$, we have $E \vdash_{hm} T{:}\mathbf{Type}$. By (Teq.refl), we get $E \vdash_{hm} T == T$ which is the desired result.

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

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

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

□

Lemma F.25 (type substitution in equivalence) If $E \vdash_{hm} \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$ and $E = E_0, X{:}\mathbf{Eq}(T_0), E_1$ then $E \vdash_{hm} \mathrm{ok}\, T == T'$.

Proof. By (TK.var) and (Teq.Eq), we have $E \vdash_{hm} X == T_0$. Let Y be a fresh variable, i.e. $Y \notin \mathrm{dom}\, E$. By Lemma E.6 (environments have to be ok) and (Kok.Type) and (envok.X), we have $E, Y{:}\mathbf{Type} \vdash_{hm} \mathrm{ok}$. By Lemma F.9 (types are ok provided their hashes are) applied once in each direction, we get that $E, Y{:}\mathbf{Type} \vdash_{hm} \{X \leftarrow Y\}T{:}\mathbf{Type}$. Then, by Lemma F.24 (type equivalence is a congruence), we get $E \vdash_{hm} \{Y \leftarrow X\}\{X \leftarrow Y\}T == \{Y \leftarrow T_0\}\{X \leftarrow Y\}T$, i.e. $E \vdash_{hm} T == \{X \leftarrow T_0\}T$.

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

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

Lemma F.26 (reversing subsignaturing judgement) If $E \vdash_{hm} [X{:}K, T] <: [X{:}K', T']$ then $E \vdash_{hm} K <: K'$ and $E, X{:}K \vdash_{hm} T == T'$.

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

Case (Ssub.refl): The premise is $E \vdash_{hm} [X{:}K, T]\,\mathrm{ok}$, which must have been derived by (Sok) from $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$. By (Teq.refl), we have $E, X{:}K \vdash_{hm} T == T$. By Lemma E.6 (environments have to be ok), we have $E, X{:}K \vdash_{hm} \mathrm{ok}$. By reversing (envok.X), we get $E \vdash_{hm} K\,\mathrm{ok}$ whence $E \vdash_{hm} K <: K$ by Lemma F.19 (reflexivity of kind equivalence) and (Ksub.refl).

Case (Ssub.tran): The premises are $E \vdash_{hm} [X{:}K, T] <: [X{:}K'', T'']$ and $E \vdash_{hm} [X{:}K'', T''] <: [X{:}K', T']$. By induction twice, we have: $E \vdash_{hm} K <: K''$, $E, X{:}K \vdash_{hm} T == T''$, $E \vdash_{hm} K'' <: K'$ and $E, X{:}K'' \vdash_{hm} T'' == T'$. By (Ksub.tran), we have $E \vdash_{hm} K <: K'$. By Lemma F.22 (weakening kind in the environment), we get $E, X{:}K \vdash_{hm} T == T'$ as desired.

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

□

Lemma F.27 (reversing module value variable typing judgement) If $E \vdash_{hm} U{:}S$ then there exist E_0, E_1, K, T such that $E = E_0, U{:}[X{:}K, T], E_1$ and $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: S$.

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

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 $E \vdash_{hm} \mathrm{ok}$. By (US.self) and (TK.mod), we have $E \vdash_{hm} U.\mathrm{TYPE}{:}K$. By Lemma F.7 (relating type-is-kind and subkinding), we get $E \vdash_{hm} \mathbf{Eq}(U.\mathrm{TYPE}) <: K$. By Lemma F.21 (things have to be ok) and reversing (Sok), we get $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$. By Lemma F.9 (types are ok provided their hashes are) applied once in each direction, given that $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} \mathrm{ok}$ by Lemma F.21 (things have to be ok) and (envok.X), we get $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} T{:}\mathbf{Type}$. By (Ssub.struct), we get $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: S$ as desired.

Case (US.sub): There exists S' such that the premises are $E \vdash_{hm} U{:}S'$ and $E \vdash_{hm} S' <: S$. By induction, we have $E = E_0, U{:}[X{:}K, T], E_1$ and $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: S'$. By (Ssub.tran) we get $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: S$ as desired.

Case (US.self): There exist K' and T' such that $S = [\mathbf{Eq}(U.\mathrm{TYPE}), T']$ and the premise is $E \vdash_{hm} U{:}[X{:}K', T']$. By induction we have $E = E_0, U{:}[X{:}K, T], E_1$ and $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: [X{:}K', T']$. By Lemma F.26 (reversing subsignaturing judgement), we have $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} T == T'$. Given that we also have $E \vdash_{hm} \mathbf{Eq}(U.\mathrm{TYPE}) <: \mathbf{Eq}(U.\mathrm{TYPE})$ (by Lemma F.21 (things have to be ok), (Teq.refl), (Keq.Eq) and (Ksub.refl)), by (Ssub.struct), we get $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: S$ as desired.

□

Lemma F.28 (obtaining module value variable typing judgement) If $E_0, U{:}[X{:}K, T], E_1 \vdash_{hm} [X{:}\mathbf{Eq}(U.\mathrm{TYPE}), T] <: S$ then $E_0, U{:}[X{:}K, T], E_1 \vdash_{hm} U{:}S$.

Proof. Write $E = E_0, U{:}[X{:}K, T], E_1$. By Lemma E.6 (environments have to be ok), (US.var) and (US.self), we have $E \vdash_{hm} U{:}[X{:}\mathbf{Eq}(U.\text{TYPE}), T]$. By (US.sub), we get $E \vdash_{hm} U{:}S$ as desired. □

Lemma F.29 (type preservation by module substitution in coloured judgements) Suppose $E_0, U{:}[X{:}K, T], E \vdash_{hm} J$ and z and Z are fresh and $\sigma = \{U.\text{TYPE} \leftarrow Z, U.\text{term} \leftarrow z\}$.

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

Otherwise $E_0, Z{:}K, z{:}\{X \leftarrow Z\}T, \sigma E \vdash_{hm} \sigma J$

Proof. Write $S = [X{:}K, T]$. Without loss of generality, we assume that $X \notin \operatorname{dom} E$. We induct on the derivation Π of $E_0, U{:}S, E \vdash_{hm} J$.

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. Also the right-hand side of the premise is not $U{:}S'$ for any S'.
   (b) The premise is $\hat{\zeta} \notin \operatorname{dom} \hat{E}$ for some $\hat{\zeta}$.
   (c) The premise is a judgement with an empty environment and does not mention $\hat{E}$.

Suppose that $E_0, U{:}S, E \vdash_{hm} J$ was derived by an instance α of such a rule. Without loss of generality, U is not in a binding position (including the domain of an environment) anywhere in α except as the first binding in an instance of $\hat{E}$. We distinguish two possibilities, using 1:

General case: The instance α was obtained by instantiating the metavariable $\hat{E}$ with $E_0, U{:}S, E''$, where E'' is an environment. Hence E is of the form E'', E', where E' is an environment. Also, in this part of the proof, we assume that J is not of the form $U{:}S'$, and that the rule is not one of (TK.mod) or (eT.mod) with U in the conclusion.

Since we have a raw term rewriting system, we also get an instance β of the same rule by instantiating $\hat{E}$ by $E_0, Z{:}K, z{:}\{X \leftarrow Z\}T, E''$ and other metavariables as in α. Note that U does not appear in any binding position in β.

Note that 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 $\sigma(Z{:}K, z{:}\{X \leftarrow Z\}T, E'') = Z{:}K, z{:}\{X \leftarrow Z\}T, \sigma E''$.

Consider the premises in α, depending on which case of condition 2 holds:

Case 2a: The premise is of the form $E_0, U{:}S, E'', E'_i \vdash_{hm_i} J_i$. Furthermore J_i is not $U{:}S'$ for any S', because we have excluded the rules (TK.mod) and (eT.mod) in the problematic case. By induction, we get $E_0, Z{:}K, z{:}\{X \leftarrow Z\}T, \sigma E'', \sigma E'_i \vdash_{hm_i} \sigma J_i$, which is the corresponding premise in ω.

Case 2b: The premise is of the form $\zeta' \notin \operatorname{dom}(U{:}S, E'')$ for some ζ'. Given that $\operatorname{dom} \sigma E'' = \operatorname{dom} E'' \subseteq \operatorname{dom}(U{:}S, E'')$, we have $\zeta' \notin \operatorname{dom} \sigma E''$. Since Z and z are fresh, we have $\zeta' \notin \operatorname{dom}(Z{:}K, z{:}\{X \leftarrow Z\}T, \sigma E'')$.

Case 2c: The premise is a judgement AJ with an empty environment, so by Lemma E.9 (free variables of a judgement come from the environment) U is not free in AJ. Hence $\sigma\text{AJ} = \text{AJ}$. 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: $E_0, Z{:}K, z{:}\{X \leftarrow Z\}T, \sigma E'', \sigma E' \vdash_{hm} \sigma J$, which is what we set out to prove.

Case $\hat{E}$ is instantiated to a prefix of E_0 and J is not $U{:}S'$ for any S': Only the following cases of the following rules are concerned.

Case (envok.U): We have $E_0, U{:}S \vdash_{hm}$ ok, and one of the premises is $\mathbf{nil} \vdash_{hm} S$ ok. By reversing (Sok), we have $E_0, X{:}K \vdash_{hm} T{:}\mathbf{Type}$. By alpha-conversion, we get $E_0, Z{:}K \vdash_{hm} \{X \leftarrow Z\}T{:}\mathbf{Type}$. By (envok.x), we have $E_0, Z{:}K, z{:}\{X \leftarrow Z\}T \vdash_{hm}$ ok as desired.

Case (US.var): Impossible: J would be $U{:}S$.

Case (TK.mod) where $J = U.\text{TYPE}{:}K'$ for some K': Then there exists T' such that the premise is $E_0, U{:}S, E \vdash_{hm} U{:}[X{:}K', T']$. By induction, we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma[X{:}K', T']$. By Lemma F.26 (reversing subsignaturing judgement), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} \mathbf{Eq}(Z) <: \sigma K'$. By Lemma E.6 (environments have to be ok), (TK.var) and (TK.sub), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}\mathbf{Type}$. By (Teq.refl), and (TK.Eq), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}\mathbf{Eq}(Z)$. By (TK.sub), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}\sigma K'$.

Case (eT.mod) where $J = U.\text{term}{:}T'$ for some T': Then there exists K' such that the premises are $E_0, U{:}S, E \vdash_{hm} U{:}[X{:}K', T']$ and $E_0, U{:}S, E \vdash_{hm} T'{:}\mathbf{Type}$. By applying induction, we get that $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma[X{:}K', T']$. By Lemma F.26 (reversing subsignaturing judgement), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E, X{:}\mathbf{Eq}(Z) \vdash_{hm} T == \sigma T'$.

By Lemma E.6 (environments have to be ok), Lemma E.7 (prefixes of ok environments are ok) and (TK.var), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}K$. By Lemma F.21 (things have to be ok), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} K$ ok. By Lemma F.6 (kinds are smaller than Type) and (TK.sub), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}\mathbf{Type}$. By (Teq.refl) and (TK.Eq), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}\mathbf{Eq}(Z)$.

By Lemma F.14 (type preservation by substitution), given Lemma E.19 (computing the pvu of a type world judgement), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} \{X{\leftarrow}Z\}T == \{X{\leftarrow}Z\}\{U.\text{TYPE}{\leftarrow}Z, U.\text{term}{\leftarrow}z\}T'$. Since $E_0, U{:}S, E \vdash_{hm} T'{:}\mathbf{Type}$, by Lemma E.9 (free variables of a judgement come from the environment), we have $X \notin \text{fv}\, T'$, so we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} \{X{\leftarrow}Z\}T == \sigma T'$.

By Lemma E.6 (environments have to be ok) and (eT.var), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} z{:}\{X{\leftarrow}Z\}T$. By (eT.eq), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} z{:}\sigma T'$ as desired.

Case $J = U{:}S'$ for some S': We have $E_0, U{:}S, E \vdash_{hm} U{:}S'$.

Subcase (US.var): We have $E_0, U{:}S, E \vdash_{hm} U{:}S$ (thanks to Lemma E.8 (ok environments have no repetition in the domain)). The premise is $E_0, U{:}S, E \vdash_{hm}$ ok.

By induction, we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm}$ ok.

By (TK.var) $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}K$. By Lemma F.7 (relating type-is-kind and subkinding), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} \mathbf{Eq}(Z) <: K$.

By Lemma E.6 (environments have to be ok), Lemma E.7 (prefixes of ok environments are ok), reversing (envok.U) and Lemma F.21 (things have to be ok), we have $\mathbf{nil} \vdash_{hm} S$ ok. By reversing (Sok), we get $X{:}K \vdash_{hm} T{:}\mathbf{Type}$. By Lemma E.6 (environments have to be ok), we have $X{:}K \vdash_{hm}$ ok. By reversing (envok.X) and Lemma F.3 (weakening), we have $X{:}K \vdash_{hm} K$ ok, so by (envok.X), we get $E_0, Z{:}K, X{:}K \vdash_{hm}$ ok. By Lemma F.3 (weakening), we have $E_0, Z{:}K, X{:}K \vdash_{hm} T{:}\mathbf{Type}$.

By Lemma E.7 (prefixes of ok environments are ok), and by (TK.var) and Lemma F.7 (relating type-is-kind and subkinding) as before, we have $E_0, Z{:}K \vdash_{hm} \mathbf{Eq}(Z) <: K$. By Lemma F.22 (weakening kind in the environment), we have $E_0, Z{:}K, X{:}\mathbf{Eq}(Z) \vdash_{hm} T{:}\mathbf{Type}$.

By (TK.var), Lemma F.21 (things have to be ok), Lemma F.6 (kinds are smaller than Type) and (TK.sub), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} Z{:}\mathbf{Type}$. Thus, by (Kok.Eq) and (envok.X), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E, X{:}\mathbf{Eq}(Z) \vdash_{hm}$ ok.

By Lemma F.3 (weakening), we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E, X{:}\mathbf{Eq}(Z) \vdash_{hm} T{:}\mathbf{Type}$. By (Teq.refl), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E, X{:}\mathbf{Eq}(Z) \vdash_{hm} T == T$.

By (Ssub.struct), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: [X{:}K, T]$.

By Lemma E.9 (free variables of a judgement come from the environment), $U \notin \text{fv}\, S$. So we have $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma S$ as desired.

Subcase (US.sub): We have $E_0, U{:}S, E \vdash_{hm} U{:}S'$, and there exists S'' such that the premises are $E_0, U{:}S, E \vdash_{hm} U{:}S''$ and $E_0, U{:}S, E \vdash_{hm} S'' <: S'$. By induction over each premise, we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma S''$ and $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} \sigma S'' <: \sigma S'$. By (Ssub.tran), we get $E_0, Z{:}K, z{:}\{X{\leftarrow}Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma S'$ as desired.

Subcase (US.self): We have $E_0, U{:}S, E \vdash_{hm} U{:}[X{:}\mathbf{Eq}(U.\textsc{type}), T']$. There exists K' such that the premise is $E_0, U{:}S, E \vdash_{hm} U{:}[X{:}K', T']$. By induction, we get $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: \sigma[X{:}K', T']$. By Lemma F.26 (reversing subsignaturing judgement), we get $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \sigma E, X{:}\mathbf{Eq}(Z) \vdash_{hm} T == \sigma T'$.

By Lemma E.6 (environments have to be ok) and (TK.var), we have $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \sigma E \vdash_{hm} Z{:}K$. By Lemma F.21 (things have to be ok) and Lemma F.6 (kinds are smaller than Type), we have $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \sigma E \vdash_{hm} Z{:}\mathbf{Type}$. By (Teq.refl), (Keq.Eq) and (Ksub.refl), we have $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \sigma E \vdash_{hm} \mathbf{Eq}(Z) <: \mathbf{Eq}(Z)$.

By (Ssub.struct), we get $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \sigma E \vdash_{hm} [X{:}\mathbf{Eq}(Z), T] <: [X{:}\mathbf{Eq}(Z), \sigma T']$ as desired.

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

Lemma F.30 (type world judgements do not contain free expression variables) If $E \vdash_{hm} J$ is a derivable type world judgement then $\mathrm{fse}\, E \cup \mathrm{fse}\, J$ does not contain any free expression substitutable entity.

Proof. Given Lemma F.21 (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 E.21 (types do not contain free expression variables), no free substitutable entity in J is an expression substitutable entity. Also, by Lemma E.6 (environments have to be ok), $E \vdash_{hm}$ ok, so by Lemma E.22 (environments do not contain free expression variables), E does not have any free substitutable entity. ☐

Lemma F.31 (type preservation by module substitution in coloured judgements for type world judgements) Suppose $E_0, U{:}[X{:}K, T], E \vdash_{hm} J$ is a derivable type world judgement and J is not of the form $U{:}S'$ for any S' and Z is fresh and $\sigma = \{U.\textsc{type}\leftarrow Z\}$. Then $E_0, Z{:}K, \sigma E \vdash_{hm} \sigma J$.

Proof. By Lemma F.29 (type preservation by module substitution in coloured judgements), for Z and z fresh, we have $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, E' \vdash_{hm} J'$ where $E' = \{U.\textsc{type}\leftarrow Z, U.\mathrm{term}\leftarrow z\}E$ and $J' = \{U.\textsc{type}\leftarrow Z, U.\mathrm{term}\leftarrow z\}J$. Given the syntax of environments and type world judgements, z may appear free in E' or J' only inside a hash. Given Lemma E.5 (hashes have to be ok), any hash in E' or J' is ok, and by Lemma E.9 (free variables of a judgement come from the environment), none of these hashes has a free occurence of z. Therefore $z \notin \mathrm{fv}\, E'$ and $z \notin \mathrm{fv}\, J'$. Thus we can write the judgement in question as $E_0, Z{:}K, z{:}\{X\leftarrow Z\}T, \{U.\textsc{type}\leftarrow Z\}E \vdash_{hm} \{U.\textsc{type}\leftarrow Z\}J$. Given that $z \notin \mathrm{fv}\,\{U.\textsc{type}\leftarrow Z\}E \cup \mathrm{fv}\,\{U.\textsc{type}\leftarrow Z\}J$, by Lemma F.15 (strengthening), we get $E_0, Z{:}K, \{U.\textsc{type}\leftarrow Z\}E \vdash_{hm} \{U.\textsc{type}\leftarrow Z\}J$. ☐

Lemma F.32 (simplified module and type equality substitution for type world judgements) Suppose $U{:}[X{:}\mathbf{Eq}(T), T'], E \vdash_{hm} J$ and $\chi = U.\textsc{type}$, or $X{:}\mathbf{Eq}(T), E \vdash_{hm} J$ and $\chi = X$, where both are type world judgements and J is not of the form $U'{:}S'$. Then $\{\chi\leftarrow T\}E \vdash_{hm} \{\chi\leftarrow T\}J$.

Proof. We consider each case.

$U{:}[X{:}\mathbf{Eq}(T), T'], E \vdash_{hm} J$**:** By Lemma F.31 (type preservation by module substitution in coloured judgements for type world judgements), this case reduces to the other case.

$X{:}\mathbf{Eq}(T), E \vdash_{hm} J$ **by some proof** Π**:** By Lemma E.6 (environments have to be ok), $X{:}\mathbf{Eq}(T), E \vdash_{hm}$ ok. By Lemma E.7 (prefixes of ok environments are ok), $X{:}\mathbf{Eq}(T) \vdash_{hm}$ ok. By (TK.var) and (Teq.Eq), $\mathbf{nil} \vdash_{hm} T{:}\mathbf{Eq}(T)$. By Lemma E.19 (computing the pvu of a type world judgement) applied to Π, we have $hm \preccurlyeq \min(\mathrm{pvu}_X(\Pi))$. By Lemma F.14 (type preservation by substitution), $\{X\leftarrow T\}E \vdash_{hm} \{X\leftarrow T\}J$, as desired.

☐

Lemma F.33 (type preservation by fully carried out module substitution)
If $U{:}[X{:}\mathbf{Type}, T_1], E \vdash_{\bullet} J$ and J is not $U{:}S$ for any S and $\mathbf{nil} \vdash_{\bullet} [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T_1]$
then $\sigma E \vdash_{\bullet} \sigma J$, where $\sigma = \{U.\textsc{type}\leftarrow h, U.\mathrm{term}\leftarrow [v^\bullet]_h^{\{X\leftarrow h\}T_1}\}$, where $h = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T_1])$, for any N.

Proof. From $\mathbf{nil} \vdash_{\bullet} [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T_1]$, Lemma F.8 (components of modules are ok), there exists T_2 such that $X{:}\mathbf{Eq}(T_0) \vdash_{\bullet} T_2 == T_1$ and $X{:}\mathbf{Type} \vdash_{\bullet} T_1{:}\mathbf{Type}$ and $\mathbf{nil} \vdash_{\bullet} T_0{:}\mathbf{Type}$ and $\mathbf{nil} \vdash_{\bullet} v^\bullet{:}T_2$. Also, by (hmok.hash), we get $\vdash h$ ok.

From $\mathbf{nil} \vdash_{\bullet} v^\bullet{:}T_2$, by Lemma F.2 (colour stripping judgements), we get $\mathbf{nil} \vdash_h v^\bullet{:}T_2$. By Lemma E.6 (environments have to be ok), $X{:}\mathbf{Eq}(T_0) \vdash_{\bullet}$ ok, and note that any proof of this finishes with (envok.X) and therefore has an empty pvu for X. By Lemma F.2 (colour stripping judgements), we get $X{:}\mathbf{Eq}(T_0) \vdash_h \text{ ok}$, also by a proof with an empty pvu for X. From $\mathbf{nil} \vdash_h v^\bullet{:}T_2$, by Lemma F.3 (weakening), we get $X{:}\mathbf{Eq}(T_0) \vdash_h v^\bullet{:}T_2$ by a proof Π such that $\mathrm{pvu}_X(\Pi) = \varnothing$. From $X{:}\mathbf{Eq}(T_0) \vdash_\bullet T_2 == T_1$, by Lemma F.2 (colour stripping judgements), we get $X{:}\mathbf{Eq}(T_0) \vdash_h T_2 == T_1$. By Lemma E.19 (computing the pvu of a type world judgement), any proof Π' of this judgement is such that $\mathrm{pvu}_X(\Pi') \subseteq \{h\}$. We can combine Π and Π' by (eT.eq) to get a proof Π'' of $X{:}\mathbf{Eq}(T_0) \vdash_h v^\bullet{:}T_1$, such that $h \preccurlyeq \min(\mathrm{pvu}_X(\Pi''))$.

By (Teq.hash) and (TK.Eq), we have $\mathbf{nil} \vdash_h h{:}\mathbf{Eq}(T_0)$. By Lemma F.14 (type preservation by substitution), we get $\mathbf{nil} \vdash_h v^\bullet{:}\{X{\leftarrow}h\}T_1$.

Now, by applying Lemma F.29 (type preservation by module substitution in coloured judgements) to $U{:}[X{:}\mathbf{Type}, T_1], E \vdash_\bullet J$ and performing alpha-conversion, we have $X{:}\mathbf{Type}, x{:}T_1, \sigma_U E \vdash_\bullet \sigma_U J$, where $\sigma_U = \{U.\mathrm{TYPE}{\leftarrow}X, U.\mathrm{term}{\leftarrow}x\}$.

From the latter judgement and $\mathbf{nil} \vdash_\bullet h{:}\mathbf{Type}$, apply Lemma F.14 (type preservation by substitution). We get $x{:}\{X{\leftarrow}h\}T_1, \{X{\leftarrow}h\}\sigma_U E \vdash_\bullet \{X{\leftarrow}h\}\sigma_U J$.

Finally, we can apply Lemma F.23 (type preservation by guarded expression variable substitution) to the latter judgement combined with $\mathbf{nil} \vdash_h v^\bullet{:}\{X{\leftarrow}h\}T_1$. We obtain $\{x{\leftarrow}[v^\bullet]_h^{\{X\leftarrow h\}T_1}\}\{X{\leftarrow}h\}\sigma_U E \vdash_\bullet \{x{\leftarrow}[v^\bullet]_h^{\{X\leftarrow h\}T_1}\}\{X{\leftarrow}h\}\sigma_U J$, i.e. $\sigma E \vdash_\bullet \sigma J$, as desired. □

G Type decomposition and type preservation for reduction

Lemma G.1 (shortening typing proof) If $E \vdash_{hm} e{:}T$ then there exists T' such that $E \vdash_{hm} e{:}T'$ by a subproof that does not have (eT.eq) as the last rule used and $E \vdash_{hm} T' == T$.

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

If the proof $E \vdash_{hm} e{:}T$ does not have an instance of (eT.eq) as the last step, then we have the desired result, given that $E \vdash_{hm} T{:}\mathbf{Type}$ by Lemma F.21 (things have to be ok), whereupon we can apply (Teq.refl).

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

Lemma G.2 (reversing typing proof through a context) If $\mathbf{nil} \vdash_{hm'} CC_{hm}^{hm'}.e{:}T'$ then there exists T such that $\mathbf{nil} \vdash_{hm} e{:}T$. If furthermore $\mathbf{nil} \vdash_{hm} e_1{:}T$ then $\mathbf{nil} \vdash_{hm'} CC_{hm}^{hm'}.e_1{:}T'$.

Proof. Induct on the structure of $CC_{hm}^{hm'}$. By Lemma G.1 (shortening typing proof), there exists T_0 such that $\mathbf{nil} \vdash_{hm'} T_0 == T'$, and $\mathbf{nil} \vdash_{hm'} CC_{hm}^{hm'}.e{:}T_0$ by a proof Π that does not end with (eT.eq).

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

Case $CC_{hm}^{hm'} = C_{hm_1}^{hm'}.CC_{hm}^{hm_1}$: Then Π ends with an application of (eT.tuple), (eT.proj), (eT.ap), (eT.mar), (eT.marred), (eT.unmar), (eT.send) or (eT.col) (depending on $C_{hm_1}^{hm'}$). In any case, one premise is $\mathbf{nil} \vdash_{hm_1} CC_{hm}^{hm_1}.e{:}T_1$ for some T_1. By induction we get $\mathbf{nil} \vdash_{hm} e{:}T$ for some T.

If furthermore $\mathbf{nil} \vdash_{hm} e_1{:}T$, then we have $\mathbf{nil} \vdash_{hm_1} CC_{hm}^{hm_1}.e_1{:}T_1$ by induction. Each of the (eT.*) rules considered above is linear with respect to the expression metavariable instantiated by $CC_{hm}^{hm_1}.e$, with exactly one occurence above the line and one below. Instantiating this metavariable by $CC_{hm}^{hm_1}.e_1$ yields another instance of the rule. By replacing in Π the derivation leading to $\mathbf{nil} \vdash_{hm_1} CC_{hm}^{hm_1}.e{:}T_1$ by that leading to $\mathbf{nil} \vdash_{hm_1} CC_{hm}^{hm_1}.e_1{:}T_1$, we get a derivation of $\mathbf{nil} \vdash_{hm'} CC_{hm}^{hm'}.e_1{:}T_0$. By (eT.eq), since $\mathbf{nil} \vdash_{hm'} T_0 == T'$, we have $\mathbf{nil} \vdash_{hm'} CC_{hm}^{hm'}.e_1{:}T'$ as desired.

□

Lemma G.3 (transitivity of kind equivalence) If $E \vdash_{hm} K == K'$ and $E \vdash_{hm} K' == K''$ then $E \vdash_{hm} K == K''$.

Proof. Both hypotheses have to be derived by the same rule.

Case (Keq.Type): Trivial.

Case (Keq.Eq): Trivial by (Teq.tran) and (Keq.Eq).

□

Lemma G.4 (discreteness of subkinding below Type) If $E \vdash_{hm} K <: \mathbf{Eq}(T')$ then $E \vdash_{hm} K == \mathbf{Eq}(T')$ by a subproof.

Proof. Induct on the derivation of $E \vdash_{hm} K <: \mathbf{Eq}(T')$. If the last rule in the proof is (Ksub.tran), then the result holds by induction and Lemma G.3 (transitivity of kind equivalence). Otherwise the last rule is (Ksub.refl) and the premise is the desired result. □

Definition G.5 (bare bones environment) A bare bones environment is one that contains only bindings of the form $X{:}\mathbf{Type}$.

Definition G.6 (purely abstract environment) A purely abstract environment is one that contains only bindings of the form $U{:}[X{:}\mathbf{Type}, T]$ or $X{:}\mathbf{Type}$.

Lemma G.7 (signature rewriting in a type world judgement) If $E_0, U{:}[X{:}K, T], E_1 \vdash_{hm} J$ is a derivable type world judgement and J is not of the form $U{:}S'$ for any S' and $E_0 \vdash_{hm} [X{:}K, T']$ ok then $E_0, U{:}[X{:}K, T'], E_1 \vdash_{hm} J$.

Proof. By (envok.U), $E_0, U'{:}[X{:}K, T'] \vdash_{hm}$ ok where U' is fresh. By Lemma F.5 (combined weakening), $E_0, U'{:}[X{:}K, T'], U{:}[X{:}K, T], E_1 \vdash_{hm} J$. By Lemma F.31 (type preservation by module substitution in coloured judgements for type world judgements), for Z fresh, we have $E_0, U'{:}[X{:}K, T'], Z{:}K, \{U.\textsc{type}{\leftarrow}Z\}E_1 \vdash_{hm} \{U.\textsc{type}{\leftarrow}Z\}J$.

By (US.var) and (TK.mod), from $E_0, U'{:}[X{:}K, T'] \vdash_{hm}$ ok, we get $E_0, U'{:}[X{:}K, T'] \vdash_{hm} U'.\textsc{type}{:}K$. By Lemma E.19 (computing the pvu of a type world judgement) and Lemma F.14 (type preservation by substitution), we get $E_0, U'{:}[X{:}K, T'], \{Z{\leftarrow}U'.\textsc{type}\}\{U.\textsc{type}{\leftarrow}Z\}E_1 \vdash_{hm} \{Z{\leftarrow}U'.\textsc{type}\}\{U.\textsc{type}{\leftarrow}Z\}J$. Since Z is fresh, we have $E_0, U'{:}[X{:}K, T'], \{U.\textsc{type}{\leftarrow}U'.\textsc{type}\}E_1 \vdash_{hm} \{U.\textsc{type}{\leftarrow}U'.\textsc{type}\}J$.

By Lemma F.30 (type world judgements do not contain free expression variables), $U.\mathrm{term} \notin \mathrm{fse}\, E_1 \cup \mathrm{fse}\, J$. Furthermore, J is not of the form $U{:}S'$ for any S', so U only appears in J as $U.\textsc{type}$. Hence we have $E_0, U'{:}[X{:}K, T'], \{U{\leftarrow}U'\}E_1 \vdash_{hm} \{U{\leftarrow}U'\}J$. By alpha-conversion, we get $E_0, U{:}[X{:}K, T'], E_1 \vdash_{hm} J$ as desired. □

Lemma G.8 (type substitution in a purely abstract environment) If $E_0, E_1 \vdash_{hm} J$ is a derivable type world judgement and J is not of the form $U{:}S$ for any U, S and $E_0, E_2 \vdash_{hm}$ ok and E_1 and E_2 are purely abstract and E_1 and E_2 bind the same variables in the same order then $E_0, E_2 \vdash_{hm} J$.

Proof. Induct on the length of E_1.

Case $E_1 = \mathbf{nil}$: Trivial.

Case $E_1 = X{:}\mathbf{Type}, E_1'$ and $E_2 = X{:}\mathbf{Type}, E_2'$: Trivial by induction (incorporating $X{:}\mathbf{Type}$ at the tail of E_0).

Case $E_1 = U{:}[X{:}\mathbf{Type}, T_1], E_1'$ and $E_2 = U{:}[X{:}\mathbf{Type}, T_2], E_2'$: By Lemma E.6 (environments have to be ok) and Lemma E.7 (prefixes of ok environments are ok) and reversing (envok.U), we get $E_0 \vdash_{hm} [X{:}\mathbf{Type}, T_2]$ ok. Then, by Lemma G.7 (signature rewriting in a type world judgement), we have $E_0, U{:}[X{:}\mathbf{Type}, T_2], E_1' \vdash_{hm} J$. By induction (incorporating $U{:}[X{:}\mathbf{Type}, T_2]$ at the tail of E_0), we get $E_0, U{:}[X{:}\mathbf{Type}, T_2], E_2' \vdash_{hm} J$ as desired.

□

Lemma G.9 (equality kinding in an uncontributing environment) If $E \vdash_{hm} T{:}\mathbf{Eq}(T')$ and E is a bare bones environment then $E \vdash_{hm} T == T'$ 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 $T = T' = U.\textsc{type}$, with a proof whose last steps are (US.var), (US.self) and lastly (TK.mod). To prove that $E \vdash_{hm} U.\textsc{type} == U.\textsc{type}$, 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 $\mathbf{nil} \vdash_{hm} T{:}K$ and $\mathbf{nil} \vdash_{hm} K <: \mathbf{Eq}(T')$. By Lemma G.4 (discreteness of subkinding below Type), $\mathbf{nil} \vdash_{hm} K == \mathbf{Eq}(T')$ by a subproof, whence by reversing (Keq.Eq) there exists T'' such that $K = \mathbf{Eq}(T'')$ and $\mathbf{nil} \vdash_{hm} T'' == T'$, the latter being derived by a proper subproof of the original proof. By induction on $\mathbf{nil} \vdash_{hm} T{:}\mathbf{Eq}(T'')$, we get $\mathbf{nil} \vdash_{hm} T == T''$ by a smaller proof. By (Teq.tran), we get $\mathbf{nil} \vdash_{hm} T == T'$, 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 the kind $\mathbf{Type}$.

Case (TK.mod): Impossible by Lemma E.9 (free variables of a judgement come from the environment), since E contains no module variable binding.

□

Lemma G.10 (equivalence of small types in an uncontributing environment) If $E \vdash_h T_0 == T_1$ and E is a bare bones environment and $h = \mathbf{hash}(N, [T, v]{:}S)$ and T is not a subterm of T_0 or T is not a subterm of T_1 then $T_0 = T_1$.

Proof. Induct on the derivation of $E \vdash_h T_0 == T_1$.

Case (Teq.Eq): Then by Lemma G.9 (equality kinding in an uncontributing environment), $E \vdash_h T_0 == T_1$ by a strictly smaller proof, so we get the desired result by induction.

Case (Teq.hash): We have $h = T_0$ and $T = T_1$. T is a subterm of h, so T is a subterm of both T_0 and T_1, which is impossible.

Case (Teq.refl): Trivial.

Case (Teq.sym): Trivial by induction.

Case (Teq.tran): There exists T_2 such that the premises are $E \vdash_h T_0 == T_2$ and $E \vdash_h T_2 == T_1$. Apply induction to the premise containing T_i, T_i being the one of T_0 and T_1 that does not contain T: we get $T_2 = T_i$. Then apply induction to the other premise: we get $T_{1-i} = T_2$, hence $T_0 = T_1$ as desired.

Case (Teq.cong.fun): Then there are T_0', T_0'', T_1' and T_1'' such that $T_i = T_i' \rightarrow T_i''$ for $i = 0, 1$. The premises are $E \vdash_h T_0' == T_1'$ and $E \vdash_h T_0'' == T_1''$. Suppose that T is not a subterm of T_i ($i = 0$ or $i = 1$). Then T is not a subterm of T_i' nor of T_i'', so by induction we get that $T_0' = T_1'$ and $T_0'' = T_1''$, hence $T_0 = T_1$ as desired.

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

□

Lemma G.11 (type decomposition) Let E be a purely abstract environment.

1. If $E \vdash_{hm} T_0 == TC(T_1, ..., T_j)$ or $E \vdash_{hm} TC(T_1, ..., T_j) == T_0$ then there exist $T_1', ..., T_j'$ such that $E \vdash_{hm} T_i' == T_i$ for $1 \leqslant i \leqslant j$ and either $T_0 = TC(T_1', ..., T_j')$ or there exist v and T'' and N such that $T_0 = hm = \mathbf{hash}(N, [TC(T_1', ..., T_j'), v]{:}[X{:}\mathbf{Type}, T''])$.
2. If $E \vdash_{hm} T_0 == h_1$ or $E \vdash_{hm} h_1 == T_0$ then one of the following cases holds:
   (a) There are T' and v and N such that $hm = h_1 = \mathbf{hash}(N, [T_0, v]{:}[X{:}\mathbf{Type}, T'])$.
   (b) There are T' and v and N such that $T_0 = hm = \mathbf{hash}(N, [h_1, v]{:}[X{:}\mathbf{Type}, T'])$.
   (c) $T_0 = h_1$.

Proof. Let us first prove the case when E is a bare bones environment. We induct on the size of the derivation of the hypothesis. We write the hypothesis as $E \vdash_{hm} T_0 == T$ or $E \vdash_{hm} T == T_0$.

Consider the last step of the proof of the hypothesis.

Case (Teq.Eq): By Lemma G.9 (equality kinding in an uncontributing environment), $E \vdash_{hm} T_0 == T$ (or the converse) by a strictly smaller proof, so we can apply induction.

Case (Teq.hash): One of the following cases holds:

Case $E \vdash_{hm} T_0 == TC(T_1, ..., T_j)$ **and** $T_0 = hm$**:** We almost have the second alternative of the conclusion of the lemma (with $T'_i = T_i$); all that remains to be proved is $E \vdash_{hm} T_i == T_i$. Write $hm = \mathbf{hash}(N, [TC(T_1, ..., T_j), v]{:}S)$.

The premise of (Teq.hash) is $E \vdash_{hm}$ ok. By Lemma E.4 (colours have to be ok) and reversing (hmok.hash) and Lemma F.8 (components of modules are ok), we get $E \vdash_{\bullet} TC(T_1, ..., T_j){:}\mathbf{Type}$. By Lemma F.9 (types are ok provided their hashes are) applied to $TC(T_1, ..., T_j)$ then to each T_i, we get that $E \vdash_{\bullet} T_i{:}\mathbf{Type}$, whence $E \vdash_{hm} T_i == T_i$ by Lemma F.2 (colour stripping judgements) and (Teq.refl). This completes the proof of this subcase.

Case $E \vdash_{hm} TC(T_1, ..., T_j) == T_0$ **and** $TC(T_1, ..., T_j) = hm$**:** Contradictory.

Case $E \vdash_{hm} T_0 == h_1$ **and** $T_0 = hm$**:** Also $hm = \mathbf{hash}(N, [h_1, v]{:}S)$ for some v and S and N. We have case 2b of the conclusion of the lemma.

Case $E \vdash_{hm} h_1 == T_0$ **and** $h_1 = hm$**:** Also $hm = \mathbf{hash}(N, [T_0, v]{:}S)$ for some v and S and N. We have case 2a of the conclusion of the lemma.

Case (Teq.refl): Then $T = T_0$. This completes the proof for statements 2 and 1 if $j = 0$. If $j \neq 0$, take $T'_i = T_i$. By Lemma F.21 (things have to be ok), $E \vdash_{hm} T_0{:}\mathbf{Type}$, i.e. $E \vdash_{hm} TC(T'_1, ..., T'_j){:}\mathbf{Type}$. By Lemma F.9 (types are ok provided their hashes are) applied to T_0 and then to each T'_i, we get $E \vdash_{hm} T'_i{:}\mathbf{Type}$, whence $E \vdash_{hm} T'_i == T'_i$ by (Teq.refl).

Case (Teq.sym): Trivial by induction.

Case (Teq.tran): There exists T'' such that the premises are $E \vdash_{hm} T_0 == T''$ and $E \vdash_{hm} T'' == T$, or they are $E \vdash_{hm} T'' == T_0$ and $E \vdash_{hm} T == T''$. We call Π' the subproof leading to the premise referring to T_0.

We apply induction to the subproof leading to $E \vdash_{hm} T'' == T$ or its converse. One of the following cases holds:

Case same constructor in 1: There exist $T''_1, ..., T''_j$ such that $E \vdash_{hm} T''_i == T_i$ (or the converse) for $1 \leqslant i \leqslant j$ and $T'' = TC(T''_1, ..., T''_j)$. Apply induction to Π'. There exist $T'_1, ..., T'_j$ such that $E \vdash_{hm} T''_i == T'_i$ (or the converse) for $1 \leqslant i \leqslant j$; by (Teq.tran), we get $E \vdash_{hm} T'_i == T_i$ (or the converse) for $1 \leqslant i \leqslant j$; and one of the cases of the lemma holds.

Case use of hash in 1: There exist $T'_1, ..., T'_j$, v, T''' and N such that $E \vdash_{hm} T'_i == T_i$ (or the converse) for $1 \leqslant i \leqslant j$ and $T'' = hm = \mathbf{hash}(N, [TC(T'_1, ..., T'_j), v]{:}[X{:}\mathbf{Type}, T'''])$. We have $E \vdash_{hm} T_0 == hm$ (or the converse). By induction, one of the following cases holds:

Case $hm = \mathbf{hash}(N, [T_0, v']{:}[X{:}\mathbf{Type}, T''''])$**:** Then $\mathbf{hash}(N, [TC(T'_1, ..., T'_j), v]{:}[X{:}\mathbf{Type}, T''']) = hm = \mathbf{hash}(N, [T_0, v']{:}[X{:}\mathbf{Type}, T''''])$, therefore $T_0 = TC(T'_1, ..., T'_j)$, and we have the desired result.

Case $T_0 = hm$**:** Then we have the desired result.

Case use of hash (2a) in 2: Then $hm = h_1$ and they are implemented by T''. Write $hm = h_1 = \mathbf{hash}(N_1, [T'', v]{:}[X{:}\mathbf{Type}, T'''])$. One of the following cases holds, We do case analysis on the structure of T''.

Case $T'' = h_2$ **for some** h_2**:** Apply induction to Π'. One of the following cases holds:

Case $hm = h_2 = \mathbf{hash}(N_2, [T_0, v']{:}[X{:}\mathbf{Type}, \_])$**:** Since $hm = \mathbf{hash}(N_1, [T'', v]{:}[X{:}\mathbf{Type}, T'''])$, we have $T_0 = T'' = h_2$ which is impossible.

Case $T_0 = hm = \mathbf{hash}(N_2, [h_2, v']{:}[X{:}\mathbf{Type}, \_])$**:** We have $T_0 = h_1$, i.e. we have alternative 2c of the conclusion of the lemma.

Case $T_0 = h_2$**:** Then $T'' = T_0$, and we have alternative 2a of the conclusion of the lemma.

Case T'' **is of the form** $TC''(T''_1, ..., T''_{j''})$**:** Apply induction to Π' (using part 1 of the lemma). There exist $T'_1, ..., T'_{j''}$ such that $E \vdash_{hm} T'_i == T''_i$ for $1 \leqslant i \leqslant j''$ and one of the following cases holds:

Case $T_0 = TC''(T'_1, ..., T'_{j''})$**:** For any i, T''_i is a proper subterm of T'' which is a subterm of hm, therefore hm is not a subterm of T''_i, hence by Lemma G.10 (equivalence of small types in an uncontributing environment) $T'_i = T''_i$. Thus $T_0 = T''$, and we have the first alternative of the conclusion of the lemma.

Case use of hash: Then $T_0 = hm = h_1$ and we have the second alternative of the conclusion of the lemma.

Case $T'' = hm = \mathbf{hash}(N, [h_1, v]{:}S)$ **(2b) in 2:** Apply induction to Π'. One of the following cases holds:

Case $hm = \mathbf{hash}(N, [T_0, v']{:}S')$ **(2a):** We have $T_0 = h_1$, i.e. alternative 2c of the conclusion of the lemma.

Case $T_0 = hm = \mathbf{hash}(N, [hm, v']{:}S')$ **(2b):** Impossible.

Case $T_0 = hm$ **(2c):** Then $T_0 = hm = \mathbf{hash}(N, [h_1, v]{:}S)$, i.e. we have alternative 2b of the lemma.

Case $T'' = h_1$ **(2c) in 2:** Apply induction to Π'.

Case (Teq.cong.fun): We treat the case $E \vdash_{hm} T_0 == T$, the converse is similar. We must be in part 1 of the lemma, with $TC = {}_{-1}{\rightarrow}{}_{-2}$, and there exist T'_0 and T''_0 such that the conclusion of the rule is $E \vdash_{hm} T'_1 {\rightarrow} T'_2 == T_1 {\rightarrow} T_2$ and the premises are $E \vdash_{hm} T'_i == T_i$ for $i = 1, 2$. This proves the first alternative of the conclusion of the lemma.

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

We now address the possibility of E containing module bindings. We give the proof for the case when $E \vdash_{hm} T_0 == T$; a symmetric proof applies to the symmetric case. We induct on the number of module bindings in E. The base case (no module binding) has already been proved. Consider now the case $E = E_0, U{:}[X{:}\mathbf{Type}, T'], E_1$ where E_1 contains only type variable bindings.

Let Z and z be fresh variables. By applying Lemma F.29 (type preservation by module substitution in coloured judgements), given that U is not free in E_1, we get $E_0, Z{:}\mathbf{Type}, z{:}\{X{\leftarrow}Z\}T', E_1 \vdash_{hm} \{U.\mathrm{TYPE}{\leftarrow}Z, U.\mathrm{term}{\leftarrow}z\}T_0 == \{U.\mathrm{TYPE}{\leftarrow}Z, U.\mathrm{term}{\leftarrow}z\}T$. By Lemma E.21 (types do not contain free expression variables), $z \notin \mathrm{fv}\ T_0 \cup \mathrm{fv}\ T$. Hence, by Lemma F.15 (strengthening), $E_0, Z{:}\mathbf{Type}, E_1 \vdash_{hm} \{U.\mathrm{TYPE}{\leftarrow}Z\}T_0 == \{U.\mathrm{TYPE}{\leftarrow}Z\}T$. Apply induction to that last judgement.

Case proving 1: We get $T'_1, ..., T'_j$ such that $E_0, Z{:}\mathbf{Type}, E_1 \vdash_{hm} T'_i == \{U.\mathrm{TYPE}{\leftarrow}Z\}T_i$ for all i. By Lemma E.6 (environments have to be ok) and Lemma E.7 (prefixes of ok environments are ok), we have $E_0, U{:}[X{:}\mathbf{Type}, T'] \vdash_{hm}$ ok. By Lemma F.5 (combined weakening), given that $U \notin \mathrm{fv}\, E_0 \cup \mathrm{fv}\, E_1$ by Lemma E.8 (ok environments have no repetition in the domain), we have $E_0, U{:}[X{:}\mathbf{Type}, T'], Z{:}\mathbf{Type}, E_1 \vdash_{hm} T'_i == T_i$. Let Π be a proof of this last judgement. By Lemma E.19 (computing the pvu of a type world judgement), we have $hm \preccurlyeq \min\left(\mathrm{pvu}_\zeta(\Pi)\right)$. Since $E_0, U{:}[X{:}\mathbf{Type}, T'] \vdash_{hm}$ ok, by (TK.mod), we have $E_0, U{:}[X{:}\mathbf{Type}, T'] \vdash_{hm} U.\mathrm{TYPE}{:}\mathbf{Type}$. Then by Lemma F.14 (type preservation by substitution), we have $E \vdash_{hm} \{Z{\leftarrow}U.\mathrm{TYPE}\}T'_i == T_i$ (note that $\{Z{\leftarrow}U.\mathrm{TYPE}\}\{U.\mathrm{TYPE}{\leftarrow}Z\}T_i = T_i$ since $Z \notin \mathrm{fv}\ T_i$).

Furthermore, by the same induction, we get one of the following properties: $\{Z{\leftarrow}U.\mathrm{TYPE}\}T_0 = TC(T'_1, ..., T'_j)$ or $\{Z{\leftarrow}U.\mathrm{TYPE}\}T_0 = hm = \mathbf{hash}(N, [TC(T'_1, ..., T'_j), v]{:}[X{:}\mathbf{Type}, T''])$. Since Z was chosen fresh, $\{U.\mathrm{TYPE}{\leftarrow}Z\}$ is injective. Therefore we have $T_0 = TC(\{Z{\leftarrow}U.\mathrm{TYPE}\}T'_1, ..., \{Z{\leftarrow}U.\mathrm{TYPE}\}T'_j)$ or $T_0 = hm = \mathbf{hash}(N, [TC(T'_1, ..., T'_j), v]{:}[X{:}\mathbf{Type}, T''])$ (using Lemma E.4 (colours have to be ok) and Lemma E.9 (free variables of a judgement come from the environment)).

Case proving 2: Here $T = h_1$. One of the following properties holds:
$hm = \{U.\mathrm{TYPE}{\leftarrow}Z\}h_1 = \mathbf{hash}(N, [\{U.\mathrm{TYPE}{\leftarrow}Z\}T_0, v]{:}[X{:}\mathbf{Type}, T'])$
or $\{U.\mathrm{TYPE}{\leftarrow}Z\}T_0 = hm = \mathbf{hash}(N, [\{U.\mathrm{TYPE}{\leftarrow}Z\}h_1, v]{:}[X{:}\mathbf{Type}, T'])$
or $\{U.\mathrm{TYPE}{\leftarrow}Z\}T_0 = \{U.\mathrm{TYPE}{\leftarrow}Z\}h_1$.
In any case, given Lemma E.5 (hashes have to be ok) and Lemma E.9 (free variables of a judgement come from the environment), $\mathrm{fv}\, h_1 = \varnothing$ and $\mathrm{fv}\ T_0 = \varnothing$ (since $\mathrm{fv}\,\{U.\mathrm{TYPE}{\leftarrow}Z\}T_0 = \varnothing$). So in the properties above, $\{U.\mathrm{TYPE}{\leftarrow}Z\}$ is the identity, and we have the desired disjunction.

□

Lemma G.12 (decomposition of type equivalence) If $\mathbf{nil} \vdash_{hm} TC(T_1, ..., T_j) == TC(T_1, ..., T'_j)$ then $\mathbf{nil} \vdash_{hm} T_i == T'_i$ for $1 \leqslant i \leqslant j$.

Proof. Trivial consequence of part 1 of Lemma G.11 (type decomposition). □

Lemma G.13 (structural dependence of values on their types) Suppose $\mathbf{nil} \vdash_{hm} v^{hm}{:}T_0$ and $\mathbf{nil} \vdash_{hm} T_0 == TC(T_1, ..., T_j)$. Consider the possible forms of TC.

1. If $TC = \_{1} \to \_{2}$, i.e. we have $\mathbf{nil} \vdash_{hm} T_0 == T_1 \to T_2$, then there exists e and T_1' such that $v^{hm} = \lambda x{:}T_1'.e$ and $\mathbf{nil} \vdash_{hm} T_1' == T_1$.

2. If $TC = \_{1} * ... * \_{j}$, i.e. we have $\mathbf{nil} \vdash_{hm} T_0 == T_1 * ... * T_j$ then there exists v_1^{hm}, ..., v_j^{hm} such that $v^{hm} = (v_1^{hm}, ..., v_j^{hm})$.

3. If $TC = ()$, i.e. we have $\mathbf{nil} \vdash_{hm} T_0 == \text{UNIT}$ then $v^{hm} = ()$.

4. If $TC = \text{STRING}$, i.e. we have $\mathbf{nil} \vdash_{hm} T_0 == \text{STRING}$ then there exist $v^\bullet$ and T such that $\mathbf{nil} \vdash_\bullet T{:}\mathbf{Type}$ and $\mathbf{nil} \vdash_\bullet v^\bullet{:}T$ and $v^{hm} = \mathbf{marshalled}\,(v^\bullet{:}T)$.

Proof. Induct on the size of the derivation of $\mathbf{nil} \vdash_{hm} v^{hm}{:}T_0$.

Consider the last step of the derivation of $\mathbf{nil} \vdash_{hm} v^{hm}{:}T_0$. The rules not mentionned here cannot have been used because v^{hm} is a value.

Case (eT.eq): There is T_0' such that $\mathbf{nil} \vdash_{hm} T_0' == T_0$, and $\mathbf{nil} \vdash_{hm} v^{hm}{:}T_0'$ by a smaller proof compared with $\mathbf{nil} \vdash_{hm} v^{hm}{:}T_0$. Using (Teq.tran), we get $\mathbf{nil} \vdash_{hm} T_0' == TC(T_1, ..., T_j)$. By induction we get the desired result.

Case (eT.fun): There exist e, T_1' and T_2' such that $v^{hm} = \lambda x{:}T_1'.e$ and $T_0 = T_1' \to T_2'$. By Lemma G.11 (type decomposition), we are proving case 1, and we have $\mathbf{nil} \vdash_{hm} T_1 == T_1'$.

Case (eT.unit): Then $v^{hm} = ()$ and $T_0 = \text{UNIT}$. By Lemma G.11 (type decomposition), we are proving case 3.

Case (eT.tuple): There exist $T_1', ..., T_j'$, $v_1^{hm}, ..., v_j^{hm}$ such that $v^{hm} = (v_1^{hm}, ..., v_j^{hm})$ and $T_0 = T_1' * ... * T_2'$. By Lemma G.11 (type decomposition), we are proving case 2.

Case (eT.marred): There exist $v^\bullet$ and T such that $v^{hm} = (\mathbf{marshalled}\, v^\bullet{:}T)$ and $T_0 = \text{STRING}$. By Lemma G.11 (type decomposition), we are proving case 4. One premise of (eT.marred) is $\mathbf{nil} \vdash_\bullet v^\bullet{:}T$. By Lemma F.21 (things have to be ok), we have $\mathbf{nil} \vdash_\bullet T{:}\mathbf{Type}$.

Case (eT.col): Given that v^{hm} is a value that is a bracket expression, T_0 can be but a hash h_0. By Lemma G.11 (type decomposition), this can only happen if there are $T_1', ..., T_j'$ and v^{hm} such that $hm = h$ and $T_0 = h$, with $h = \mathbf{hash}(N, [TC(T_1', ..., T_j'), v^{hm}]{:}[X{:}\mathbf{Type}, X])$. However $T_0 = hm$ is impossible as per the definition of values.

□

Lemma G.14 (triviality of type equivalence in a trivial environment) If $\mathbf{nil} \vdash_\bullet T == T'$ then $T = T'$.

Proof. Induct on the structure of T.

Case T is of the form $TC(T_1, ..., T_j)$: By Lemma G.11 (type decomposition) and the trivial colour of the hypothesis, there exist $T_1', ..., T_j'$ such that $\mathbf{nil} \vdash_\bullet T_i == T_i'$ 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 $\mathbf{nil} \vdash_\bullet T_i == T_i'$ for $1 \leqslant i \leqslant j$ we have $T_i = T_i'$ hence $T = T'$ as desired.

Case T is of the form h: By Lemma G.11 (type decomposition) and the trivial colour of the hypothesis, $T = T'$.

Case T is of the form $U.\text{TYPE}$ or X: Impossible by Lemma E.9 (free variables of a judgement come from the environment)

□

Theorem G.15 (type preservation for expression reduction) If $\mathbf{nil} \vdash_{hm} e{:}T$ and $e \longrightarrow_{hm} e'$ then $\mathbf{nil} \vdash_{hm} e'{:}T$.

Proof. Note that by Lemma G.1 (shortening typing proof), there exists T' such that $\mathbf{nil} \vdash_{hm} T' == T$ and $\mathbf{nil} \vdash_{hm} e{:}T'$ 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 F.21 (things have to be ok), we have $\mathbf{nil} \vdash_{hm} T'{:}\mathbf{Type}$, and by Lemma E.6 (environments have to be ok), we have $\mathbf{nil} \vdash_{hm}$ ok.

We induct on the derivation of the reduction.

Case (ered.proj): There exist $v_1^{hm}, ..., v_j^{hm}$ and i such that $e = \mathbf{proj}_i\,(v_1^{hm}, ..., v_j^{hm})$. Then $\mathbf{nil} \vdash_{hm} e{:}T'$ must have been derived by (eT.proj), and the ith premise is $\mathbf{nil} \vdash_{hm} v_i^{hm}{:}T'$, i.e. $\mathbf{nil} \vdash_{hm} e'{:}T'$.

Case (ered.ap): There exist e_1, v_2^{hm} and T_2 such that $e = (\lambda x{:}T_2.e_1)\, v_2^{hm}$ and $e' = \{x \leftarrow [v_2^{hm}]_{hm}^{T_2}\}e_1$. $\mathbf{nil} \vdash_{hm} e{:}T'$ must have been derived by (eT.ap), with the premises $\mathbf{nil} \vdash_{hm} v_2^{hm}{:}T_2$ and $\mathbf{nil} \vdash_{hm} (\lambda x{:}T_2.e_1){:}T_2 \to T'$.

By Lemma G.1 (shortening typing proof), there exists T'' such that $\mathbf{nil} \vdash_{hm} (\lambda x{:}T_2.e_1){:}T''$ by a proof that does not end in (eT.eq), and $\mathbf{nil} \vdash_{hm} T'' == T_2 \to T'$. By reversing (eT.fun), there exists T_1 such that $T'' = T_2 \to T_1$ and $x{:}T_2 \vdash_{hm} e_1{:}T_1$. By Lemma G.12 (decomposition of type equivalence), we have $\mathbf{nil} \vdash_{hm} T_1 == T'$.

By Lemma F.23 (type preservation by guarded expression variable substitution), we have $\mathbf{nil} \vdash_{hm} \{x \leftarrow [v_2^{hm}]_{hm}^{T_2}\}e_1{:}T_1$. By (eT.eq), we get $\mathbf{nil} \vdash_{hm} e'{:}T'$.

Case (ered.mar): There exist v_1^{hm} and T_1 such that $e = \mathbf{mar}\,(v_1^{hm}{:}T_1)$, and $e' = \mathbf{marshalled}\,([v_1^{hm}]_{hm}^{T_1}{:}T_1)$. $\mathbf{nil} \vdash_{hm} e{:}T'$ must have been derived by (eT.mar), with the premise $\mathbf{nil} \vdash_{hm} v_1^{hm}{:}T_1$; also $T' = \text{STRING}$.

By Lemma F.21 (things have to be ok) and Lemma F.10 (colour change preserves type okedness), $\mathbf{nil} \vdash_{\bullet} T_1{:}\mathbf{Type}$. By (eT.col), we get $\mathbf{nil} \vdash_{\bullet} [v_1^{hm}]_{hm}^{T_1}{:}T_1$. By Lemma E.6 (environments have to be ok), we have $\mathbf{nil} \vdash_{hm}$ ok. By (eT.marred), we get $\mathbf{nil} \vdash_{hm} e'{:}\text{STRING}$.

Case (ered.unmar): There exist v_1^{hm}, T_1 and T_2 such that $e = \mathbf{unmar}\,(\mathbf{marshalled}\,(v_1^{hm}{:}T_1)){:}T_2$. By reversing (eT.unmar), we get $\mathbf{nil} \vdash_{hm} T_2{:}\mathbf{Type}$ and $\mathbf{nil} \vdash_{hm} \mathbf{marshalled}\,(v_1^{hm}{:}T_1){:}\text{STRING}$; also $T' = T_2$. There are two possible outcomes.

Case e' is $\mathbf{UnmarFailure}^{T'}$: By Lemma F.21 (things have to be ok), $\mathbf{nil} \vdash_{hm} T'{:}\mathbf{Type}$, hence $\mathbf{nil} \vdash_{hm} \mathbf{UnmarFailure}^{T'}{:}T'$ by (eT.Undynfailure).

Case e' is the value v_1^{hm} and $T_1 = T'$: By Lemma G.1 (shortening typing proof) and reversing (eT.marred), we get $\mathbf{nil} \vdash_{\bullet} v_1^{hm}{:}T_1$. By Lemma F.2 (colour stripping judgements), this implies that $\mathbf{nil} \vdash_{hm} v_1^{hm}{:}T_1$, i.e. $\mathbf{nil} \vdash_{hm} v_1^{hm}{:}T'$.

Cases (ered.col.*): There exist hm' and e_0 such that $e = [e_0]_{hm'}^{T'}$. By reversing (eT.col), we get $\mathbf{nil} \vdash_{hm'} e_0{:}T'$. Note than $\vdash hm$ ok and $\vdash hm'$ ok by Lemma E.4 (colours have to be ok).

Case (ered.col.unit): Then $T' = \text{UNIT}$ and $e_0 = ()$ and $e' = ()$. By (eT.unit), $\mathbf{nil} \vdash_{hm} e'{:}\text{UNIT}$.

Case (ered.col.tuple): Then there exist $T_1, ..., T_j$, $e_1, ..., e_j$ such that $T' = T_1 * ... * T_j$ and $e_0 = (e_1, ..., e_j)$ and $e' = ([e_1]_{hm'}^{T_1}, ..., [e_j]_{hm'}^{T_j})$. Hence $T' = T$. By Lemma G.1 (shortening typing proof) and reversing (eT.tuple), there exist $T_1', ..., T_j'$ such that $\mathbf{nil} \vdash_{hm'} T_1' * ... * T_j' == T'$ and $\mathbf{nil} \vdash_{hm'} e_i{:}T_i'$ for $1 \leqslant i \leqslant j$. By Lemma G.12 (decomposition of type equivalence), we have $\mathbf{nil} \vdash_{hm'} T_i' == T_i$ for every i. Hence, by applying (eT.eq) j times, we get $\mathbf{nil} \vdash_{hm'} e_i{:}T_i$ for every i. By Lemma F.9 (types are ok provided their hashes are) applied to T' then to each T_i, we get $\mathbf{nil} \vdash_{hm} T_i{:}\mathbf{Type}$ for every i, whence by (eT.col) and (eT.tuple): $\mathbf{nil} \vdash_{hm} e'{:}T'$.

Case (ered.col.fun): Then there exist T_0, T_1, T_2 and e_1 such that $T' = T_2 \to T_1$ and $e_0 = \lambda x{:}T_0.e_1$ and $e' = \lambda x{:}T_2.[\{x \leftarrow [x]_{hm'}^{T_0}\}e_1]_{hm'}^{T_1}$. By Lemma G.1 (shortening typing proof) and reversing (eT.fun), there exists T_3 such that $x{:}T_0 \vdash_{hm'} e_1{:}T_3$ and $\mathbf{nil} \vdash_{hm'} T_0 \to T_3 == T'$. By Lemma G.12 (decomposition of type equivalence), we have $\mathbf{nil} \vdash_{hm'} T_2 == T_0$ (possibly using (Teq.sym)) and $\mathbf{nil} \vdash_{hm'} T_3 == T_1$.

By Lemma F.21 (things have to be ok), $\mathbf{nil} \vdash_{hm'} T_2{:}\mathbf{Type}$. By Lemma F.10 (colour change preserves type okedness), we also have $\mathbf{nil} \vdash_{\bullet} T_2{:}\mathbf{Type}$ and $\mathbf{nil} \vdash_{hm'} T_2{:}\mathbf{Type}$. By (envok.x), we get $y{:}T_2 \vdash_{hm'}$ ok and $y{:}T_2 \vdash_{\bullet}$ ok and $y{:}T_2 \vdash_{hm'}$ ok. By (eT.var), we get $y{:}T_2 \vdash_{hm'} y{:}T_2$. By (eT.eq), we get $y{:}T_2 \vdash_{hm'} y{:}T_0$.

By Lemma F.21 (things have to be ok), we get $\mathbf{nil} \vdash_{hm'} T_1{:}\mathbf{Type}$. By Lemma F.10 (colour change preserves type okedness), we have $\mathbf{nil} \vdash_{hm} T_1{:}\mathbf{Type}$. By Lemma F.3 (weakening), we get $y{:}T_2 \vdash_{hm} T_1{:}\mathbf{Type}$.

We have $y{:}T_2 \vdash_{\bullet}$ ok and $y{:}T_2 \vdash_{hm'} y{:}T_0$ and $y{:}T_2, x{:}T_0 \vdash_{hm'} e_1{:}T_3$. By Lemma F.23 (type preservation by guarded expression variable substitution), we get $y{:}T_2 \vdash_{hm'} \{x \leftarrow [y]_{hm'}^{T_0}\}e_1{:}T_3$.

$$\dfrac{\dfrac{y{:}T_2 \vdash_{hm} T_1{:}\mathbf{Type} \qquad \dfrac{y{:}T_2 \vdash_{hm'} T_3 == T_1 \quad y{:}T_2 \vdash_{hm'} \{x \leftarrow [y]_{hm'}^{T_0}\}e_1{:}T_3}{y{:}T_2 \vdash_{hm'} \{x \leftarrow [y]_{hm'}^{T_0}\}e_1{:}T_1}\,(\text{eT.eq})}{y{:}T_2 \vdash_{hm} [\{x \leftarrow [y]_{hm'}^{T_0}\}e_1]_{hm'}^{T_1}{:}T_1}\,(\text{eT.col})}{\mathbf{nil} \vdash_{hm} \lambda y{:}T_2.[\{x \leftarrow [y]_{hm'}^{T_0}\}e_1]_{hm'}^{T_1}{:}T_2 \to T_1}\,(\text{eT.fun})$$

By alpha-conversion, we get $\mathbf{nil} \vdash_{hm} e'\colon T_2 \to T_1$.

Case (ered.col.marred): Then there exist $v_1^\bullet$ and T_1 such that $T' = \text{STRING}$ and $e_0 = \mathbf{marshalled}\,(v_1^\bullet\colon T_1) = e'$. Since $\mathbf{nil} \vdash_{hm'} e_0\colon T'$, by Lemma G.1 (shortening typing proof) and reversing (eT.marred), we have $\mathbf{nil} \vdash_\bullet v_1^\bullet\colon T_1$. Since $\mathbf{nil} \vdash_{hm}$ ok, by (eT.marred), we have $\mathbf{nil} \vdash_{hm} e'\colon\text{STRING}$.

Case (ered.col.col): Then there exist h_0, h_1 and e_1 such that $T' = h_0$ and $hm' = h_1$ and $e_0 = [e_1]^{h_0}_{h_0}$. By Lemma G.1 (shortening typing proof) and reversing (eT.col), we get $\mathbf{nil} \vdash_{h_0} e_1\colon h_0$. By Lemma E.5 (hashes have to be ok), we have $\vdash h_0$ ok, whence $\mathbf{nil} \vdash_{hm} h_0\colon\mathbf{Type}$ by (TK.hash). Then, by (eT.col), we get $\mathbf{nil} \vdash_{hm} [e_1]^{h_0}_{h_0}\colon T'$.

Case (ered.col.le): Then $hm' \preccurlyeq hm$ and $e' = e_0$. Since $\mathbf{nil} \vdash_{hm'} e_0\colon T'$, i.e. $\mathbf{nil} \vdash_{hm'} e'\colon T'$, by Lemma F.2 (colour stripping judgements), we have $\mathbf{nil} \vdash_{hm'} e'\colon T$.

Case (ered.cong): There exist hm_0, $C^{hm}_{hm_0}$, e_0 and e_0' such that $e = C^{hm}_{hm_0}.e_0$ and $e' = C^{hm}_{hm_0}.e_0'$ and $e_0 \longrightarrow_{hm_0} e_0'$ by a proper subproof. Recall that we have $\mathbf{nil} \vdash_{hm} e\colon T'$. By the first part of Lemma G.2 (reversing typing proof through a context), there exists T_0 such that $\mathbf{nil} \vdash_{hm_0} e_0\colon T_0$. By induction, we get $\mathbf{nil} \vdash_{hm_0} e_0'\colon T_0$. By the second part of Lemma G.2 (reversing typing proof through a context), we get $\mathbf{nil} \vdash_{hm} e'\colon T'$.

In any case, we have $\mathbf{nil} \vdash_{hm} e'\colon T'$, whence by (eT.eq), $\mathbf{nil} \vdash_{hm} e'\colon T$. □

Lemma G.16 (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 G.17 (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 G.15 (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 G.16 (type preservation for network structural congruence), we get $\vdash n_0$ ok. By induction we get $\vdash n_1$ ok. By Lemma G.16 (type preservation for network structural congruence), we get $\vdash n'$ ok.

Case (nred.comm): There exist $CC^\bullet_{hm}$, v^{hm}, $CC^\bullet_{hm'}$ such that $n = CC^\bullet_{hm}.!\,v^{hm} \mid CC^\bullet_{hm'}.?$ and $n' = CC^\bullet_{hm}.() \mid CC^\bullet_{hm'}.v^{hm}$. By reversing (nok.par), then (nok.expr) twice, we get $\mathbf{nil} \vdash_\bullet CC^\bullet_{hm}.!\,v^{hm}\colon\text{UNIT}$ and $\mathbf{nil} \vdash_\bullet CC^\bullet_{hm'}.?\colon\text{UNIT}$.

From $\mathbf{nil} \vdash_\bullet CC^\bullet_{hm}.!\,v^{hm}$, by Lemma G.2 (reversing typing proof through a context), we get $\mathbf{nil} \vdash_{hm} !\,v^{hm}\colon T'$ for some T'. By Lemma G.1 (shortening typing proof) and reversing (eT.send), we get $\mathbf{nil} \vdash_{hm} v^{hm}\colon\mathbf{mar}$, and we see that $\mathbf{nil} \vdash_{hm} !\,v^{hm}\colon\text{UNIT}$ and $\mathbf{nil} \vdash_{hm} \text{UNIT} == T'$. By Lemma E.6 (environments have to be ok), (eT.unit) and (eT.eq), we get $\mathbf{nil} \vdash_{hm} ()\colon T'$, so by the second part of Lemma G.2 (reversing typing proof through a context), we get $\mathbf{nil} \vdash_\bullet CC^\bullet_{hm}.()\colon\text{UNIT}$.

By Lemma G.13 (structural dependence of values on their types), there exists $v^\bullet$ and T_0 such that $v^{hm} = \mathbf{marshalled}\,(v^\bullet\colon T_0)$ and $\mathbf{nil} \vdash_\bullet T_0\colon\mathbf{Type}$ and $\mathbf{nil} \vdash_\bullet v^\bullet\colon T_0$.

From $\mathbf{nil} \vdash_\bullet CC^\bullet_{hm'}.?\colon\text{UNIT}$, by Lemma G.2 (reversing typing proof through a context), we get $\mathbf{nil} \vdash_{hm'} ?\colon T''$ for some T''. By Lemma G.1 (shortening typing proof) and reversing (eT.recv), we get $\mathbf{nil} \vdash_{hm'}$ ok and $\mathbf{nil} \vdash_{hm'}$

STRING $==T''$. By (eT.marred), we get $\mathbf{nil} \vdash_{hm'}$ **marshalled** $(v^\bullet\!:T_0)$:STRING, so by the second part of Lemma G.2 (reversing typing proof through a context), we get $\mathbf{nil} \vdash_\bullet CC^\bullet_{hm'}.v^{hm}$:UNIT.

By (mT.expr) twice, then (nok.par), we get $\vdash CC^\bullet_{hm}.() \mid CC^\bullet_{hm'}.v^{hm}$ ok as desired.

□

Theorem G.18 (type preservation for machine reduction) If $\mathbf{nil} \vdash_\bullet m\!:T$ and $m \longrightarrow_{\mathsf{c}} m'$ then $\mathbf{nil} \vdash_\bullet m'\!:T$.

Proof. Consider each rule.

Case (mred.Type): We have that m is of the form $\mathbf{module}\, N_U = [T_0, v^\bullet]\!:\![X\!:\!\mathbf{Type}, T_1]$ **in** m_0 and m' is of the form σm_0 where $\sigma = \{U.\text{TYPE}\leftarrow h, U.\text{term}\leftarrow [v^\bullet]_h^{\{X\leftarrow h\}T_1}\}$ and $h = \mathbf{hash}(N, [T_0, v^\bullet]\!:\![X\!:\!\mathbf{Type}, T_1])$. The judgement $\mathbf{nil} \vdash_\bullet m\!:T$ can only be derived by (mT.let) with the following premises: $\mathbf{nil} \vdash_\bullet T\!:\!\mathbf{Type}$ and $\mathbf{nil} \vdash_\bullet [T_0, v^\bullet]\!:\![X\!:\!\mathbf{Type}, T_1]$ and $U\!:\![X\!:\!\mathbf{Type}, T_1] \vdash_\bullet m_0\!:T$.

By Lemma F.33 (type preservation by fully carried out module substitution), we have $\mathbf{nil} \vdash_\bullet \sigma(m_0\!:T)$, as desired.

Case (mred.Eq): We have that m is of the form **Let** $U = [T_0, v^\bullet]\!:\![X\!:\!\mathbf{Eq}(T_3), T_1]$ **in** m_0 and m' is of the form σm_0 where $\sigma = \{U.\text{TYPE}\leftarrow T_3, U.\text{term}\leftarrow v^\bullet\}$. The judgement $\mathbf{nil} \vdash_\bullet m\!:T$ can only be derived by (mT.let) with the following premises: $\mathbf{nil} \vdash_\bullet T\!:\!\mathbf{Type}$ and $\mathbf{nil} \vdash_\bullet [T_0, v^\bullet]\!:\![X\!:\!\mathbf{Eq}(T_3), T_1]$ and $U\!:\![X\!:\!\mathbf{Eq}(T_3), T_1] \vdash_\bullet m_0\!:T$.

From $\mathbf{nil} \vdash_\bullet [T_0, v^\bullet]\!:\![X\!:\!\mathbf{Eq}(T_3), T_1]$, by Lemma F.8 (components of modules are ok), there exists T_2 such that $X\!:\!\mathbf{Eq}(T_0) \vdash_\bullet T_2 == T_1$ and $X\!:\!\mathbf{Eq}(T_3) \vdash_\bullet T_1\!:\!\mathbf{Type}$ and $\mathbf{nil} \vdash_\bullet T_0\!:\!\mathbf{Eq}(T_3)$ and $\mathbf{nil} \vdash_\bullet v^\bullet\!:T_2$.

By Lemma F.29 (type preservation by module substitution in coloured judgements) and alpha-conversion we have $X\!:\!\mathbf{Eq}(T_3), x\!:T_1 \vdash_\bullet \sigma_U(m_0\!:T)$, where $\sigma_U = \{U.\text{TYPE}\leftarrow X, U.\text{term}\leftarrow x\}$.

By Lemma E.6 (environments have to be ok), $X\!:\!\mathbf{Eq}(T_3) \vdash_\bullet$ ok, whence by reversing (envok.X) $\mathbf{nil} \vdash_\bullet T_3\!:\!\mathbf{Type}$, so $\mathbf{nil} \vdash_\bullet T_3\!:\!\mathbf{Eq}(T_3)$ by (Teq.refl) and (TK.Eq). Therefore, by Lemma F.14 (type preservation by substitution), we have $x\!:\!\{X\leftarrow T_3\}T_1 \vdash_\bullet \{U.\text{TYPE}\leftarrow T_3, U.\text{term}\leftarrow x\}(m_0\!:T)$.

From $\mathbf{nil} \vdash_\bullet v^\bullet\!:T_2$ (given that $X\!:\!\mathbf{Eq}(T_0) \vdash_\bullet$ ok by Lemma E.6 (environments have to be ok)) by Lemma F.3 (weakening), we get $X\!:\!\mathbf{Eq}(T_0) \vdash_\bullet v^\bullet\!:T_2$. By (eT.eq), we get $X\!:\!\mathbf{Eq}(T_0) \vdash_\bullet v^\bullet\!:T_1$.

From $\mathbf{nil} \vdash_\bullet T_0\!:\!\mathbf{Eq}(T_3)$, by (Teq.Eq), (Teq.sym) and (TK.Eq), we get $\mathbf{nil} \vdash_\bullet T_3\!:\!\mathbf{Eq}(T_0)$. By Lemma F.14 (type preservation by substitution), we get $\mathbf{nil} \vdash_\bullet v^\bullet\!:\!\{X\leftarrow T_3\}T_1$. By Lemma F.14 (type preservation by substitution), we get $\mathbf{nil} \vdash_\bullet \{U.\text{TYPE}\leftarrow T_3, U.\text{term}\leftarrow v^\bullet\}(m_0\!:T)$ as desired.

□

H Progress

H.1 Classical progress theorems

Definition H.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 H.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 H.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 H.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 H.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 H.6 (progress of expressions) If $\mathbf{nil} \vdash_{hm} e{:}T$ then one of the following cases holds:

- e is legitimately stuck in hm.
- e can reduce, i.e. there exists e' such that $e \longrightarrow_{hm} e'$.

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 E.9 (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 $\mathbf{nil} \vdash_{hm} e_0{:}T_1{\rightarrow}T$ and $\mathbf{nil} \vdash_{hm} e_1{:}T_1$. Apply the inductive hypothesis to e_0.

Case e_0 can reduce: there exists e_0' such that $e_0 \longrightarrow_{hm} e_0'$. By (ered.cong), $e \longrightarrow_{hm} e_0'\, e_1$.

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' such that $e_1 \longrightarrow_{hm} e_1'$. By (ered.cong), $e \longrightarrow_{hm} e_0\, e_1'$.

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: By Lemma G.13 (structural dependence of values on their types), there exists e_2 such that $e_0 = \lambda x{:}T_1.e_2$. Then $e = (\lambda x{:}T_1.e_2)\, e_1$. By (ered.ap), $e \longrightarrow_{hm} \{x{\leftarrow}[e_1]_{hm}^{T_1}\}e_2$.

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 $T = \textsc{string}$. If e_0 is dormant or reduces then the same holds for e by Lemma H.4 (reduction in context) and Lemma H.3 (dormancy in context). Otherwise, by the inductive hypothesis on e_0, e_0 is a hm-value, so by (ered.mar), $e \longrightarrow_{hm} \mathbf{marshalled}\,([e_0]_{hm}^{T}{:}T)$.

Case (eT.marred): There exist an e_0 and a T_0 such that $e = \mathbf{marshalled}\,(e_0{:}T_0)$, and $T = \textsc{string}$. If e_0 is dormant or reduces then the same holds for e by Lemma H.4 (reduction in context) and Lemma H.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): There is an e_0 such that $e = (\mathbf{unmar}\; e_0{:}T)$. If e_0 is dormant or reduces then the same holds for e by Lemma H.4 (reduction in context) and Lemma H.3 (dormancy in context). Otherwise, by the inductive hypothesis on e_0, e_0 is a value. Its type is $\textsc{string}$, so by Lemma G.13 (structural dependence of values on their types), there is a $v^\bullet$ and a T_0 such that $e_0 = \mathbf{marshalled}\,(v^\bullet{:}T_0)$. Then e reduces by (ered.unmar).

Case (eT.Undynfailure): e is dormant.

Case (eT.unit): e is a value.

Case (eT.tuple): There are $e_1, ..., e_j$ such that $e = (e_1, ..., e_j)$. Let i be the smallest index k such that e_1 through e_{k-1} are values. If $i = j + 1$ then e is a value. Otherwise, apply the inductive hypothesis to e_i. Since e_i is not a value, it is dormant or reduces, and in either case, the same holds for e by Lemma H.3 (dormancy in context) and Lemma H.4 (reduction in context), as $(e_1, ..., e_{i-1}, \_, e_{i+1}, ..., e_j)$ is an evaluation context.

Case (eT.proj): There is an e' such that $e = \mathbf{proj}_i\ e'$, and there are $T_1, ..., T_j$ such that $\mathbf{nil} \vdash_{hm} e' : T_1 * ... * T_j$. If e' is dormant or reduces, then the same holds for e since $\mathbf{proj}_i\ \_$ is an evaluation context. Otherwise, apply the inductive hypothesis to e': it is a value. By Lemma G.13 (structural dependence of values on their types), there are $v_1^{hm}, ..., v_j^{hm}$ such that $e' = (v_1^{hm}, ..., v_j^{hm})$. Then $e \longrightarrow_{hm} v_i^{hm}$ by (ered.proj).

Case (eT.col): There is an e_0 and an hm_0 such that $e = [e_0]_{hm_0}^{T}$. 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 G.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), $e \longrightarrow_{hm} [e_0']_{hm_0}^{T}$.

Case e_0 is an hm_0-value that is not a bracket expression: Then, by reversing the appropriate rule amongst (eT.unit), (eT.tuple), (eT.fun) or (eT.marred), we have that T' is some constructed type $TC(T_1', ..., T_j')$. Since $\mathbf{nil} \vdash_{hm_0} T' == T$, by Lemma G.11 (type decomposition), one of the following cases holds:

Case T is a constructed type: There exist $T_1, ..., T_j$ such that $\mathbf{nil} \vdash_{hm} T_i' == T_i$ for all i and $T = TC(T_1, ..., T_j)$. Then e reduces by the appropriate rule amongst (ered.col.unit), (ered.col.tuple), (ered.col.fun) or (ered.col.marred).

Case $T = hm_0$: If $hm_0 \preccurlyeq hm$ then $e = [e_0]_{hm_0}^{hm_0}$ reduces by (ered.col.le). Otherwise e is a value.

Case $e_0 = [v^{h_1}]_{h_1}^{h_1}$ for some h_1 and $hm_0 = \bullet$: We have $e = [[v^{h_1}]_{h_1}^{h_1}]_{\bullet}^{T}$. By Lemma G.1 (shortening typing proof) and reversing (eT.col), we have $\mathbf{nil} \vdash_{\bullet} h_1 == T'$. By (Teq.tran), we have $\mathbf{nil} \vdash_{\bullet} h_1 == T$. By Lemma G.14 (triviality of type equivalence in a trivial environment), $h_1 = T$. Then $e \longrightarrow_{hm} e_0$ by (ered.col.le).

Case $e_0 = [v^{h_1}]_{h_1}^{h_1}$ for some h_1 and $hm_0 \neq \bullet$: We have $\mathbf{nil} \vdash_{hm_0} [v^{h_1}]_{h_1}^{h_1} : T$. Then $\mathbf{nil} \vdash_{hm_0} [v^{h_1}]_{h_1}^{h_1} : T'$ must have been obtained by an application of (eT.col), so $T' = h_1$. Since $\mathbf{nil} \vdash_{hm_0} h_1 == T$, by Lemma G.11 (type decomposition) and some more pattern matching, one of the following cases holds:

Case $T = h_1 \neq hm_0$ and $hm_0 \neq hm$: Then $e = [[v^{h_1}]_{h_1}^{h_1}]_{hm_0}^{h_1}$. By (ered.col.col), $e \longrightarrow_{hm} [v^{h_1}]_{h_1}^{h_1}$ (recall that $hm_0 \neq \bullet$).

Case $T = h_1 \neq hm_0$ and $hm_0 = hm$: Then $e = [[v^{h_1}]_{h_1}^{h_1}]_{hm}^{h_1}$. By (ered.col.le), $e \longrightarrow_{hm} e_0$.

Case $hm_0 = h_1$: Then $e = [[v^{h_1}]_{h_1}^{h_1}]_{h_1}^{T}$. By (ered.col.le), $[v^{h_1}]_{h_1}^{h_1} \longrightarrow_{h_1} v^{h_1}$. Hence, by (ered.cong), $e \longrightarrow_{hm} [v^{h_1}]_{h_1}^{T}$.

□

Corollary H.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

$$
\begin{array}{lll}
n_{()} & ::= \mathbf{0} & \text{null} \\
& \quad n_{()} \mid n_{()} & \text{parallel composition} \\
& \quad () & \text{unit} \\
\\
n_{\text{fail}} & ::= \mathbf{0} & \text{null} \\
& \quad n_{\text{fail}} \mid n_{\text{fail}} & \text{parallel composition} \\
& \quad CC^{\bullet}_{hm}.\mathbf{UnmarFailure}^{T} & \text{dead} \\
\\
n_{?} & ::= n_{?} \mid n_{?} & \text{parallel composition} \\
& \quad CC^{\bullet}_{hm}.? & \text{waiting to input} \\
\\
n_{!} & ::= n_{!} \mid n_{!} & \text{parallel composition} \\
& \quad CC^{\bullet}_{hm}.!\, v & \text{waiting to output}
\end{array}
$$

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 H.6 (progress of expressions), one of the following cases holds:

Case n is a $\bullet$-value: By (nok.expr), the value n has type UNIT. By Lemma G.13 (structural dependence of values on their types), n is an $n_{()}$.

Case n is dead: n is an n_{fail}.

Case n is waiting for input: n is an $n_{?}$.

Case n is waiting for output: n is an $n_{!}$.

Case n reduces: n reduces.

□

Theorem H.8 (progress of machines) If $\mathbf{nil} \vdash_{\bullet} m{:}T$ 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.let): It is obvious that m reduces, by either (mred.Type) or (mred.Eq).

□

H.2 Determinism of reduction

Theorem H.9 (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 $m = \mathbf{module}\, N_U = M{:}[X{:}\mathbf{Type}, T]\ \mathbf{in}\ m'$: The only applicable rule is (mred.Type).

Case $m = \mathbf{module}\, N_U\ = M{:}[X{:}\mathbf{Eq}(T'),\, T]\ \mathbf{in}\ m'$**:** The only applicable rule is (mred.Eq).

□

Lemma H.10 (values do not reduce) If $e \longrightarrow_{hm} e'$ 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 $v^{hm} = (\lambda x{:}T.e)$**:** 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 $v^{hm} = [v^{h_1}]_{h_1}^{h_1}$ **where** $h_1 \neq hm$**:** The rule (ered.col.le) requires $h_1 \preccurlyeq hm$, which given the definition of $\preccurlyeq$ implies that $h_1 = hm$, a contradiction. The rule (ered.col.col) does not apply as it requires the type and colour annotation on the bracket to be different. The other (ered.col.*) rules do not apply as they require the type annotation on the bracket not to be a hash. If (ered.cong) applies, then it is with a context of the form $[\_]_{h_1}^{h_1}$. But then $v^{h_1} \longrightarrow_{h_1}$, which is impossible by induction.

□

Theorem H.11 (determinism of expression reduction) Reduction of expressions and machines is deterministic, i.e. if $e \longrightarrow_{hm} e'$ and $e \longrightarrow_{hm} e''$ then $e' = e''$ and both reductions use the same rule on the same redex.

Proof. Induct on the structure of e.

Cases $e = x$**,** $e = U.\mathrm{term}$**,** $e = \mathbf{UnmarFailure}^T$**:** 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 H.10 (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 H.10 (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 $e = \mathbf{proj}_i\ e_0$**:** If e_0 is a hm-value, given Lemma H.10 (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 $e = e_1\ e_2$**:** If e_1 and e_2 are both hm-values, given Lemma H.10 (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 H.10 (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 $e = [v^{hm_1}]^{T}_{hm_1}$ **where** T **is not a hash:** The rules (ered.col.col) and (ered.col.le) do not apply since T is not a hash. Also (ered.cong) does not apply since v^{hm_1} is a value and we have Lemma H.10 (values do not reduce). The only rules that may apply are the rules to push brackets in ((ered.col.unit), (ered.col.tuple), (ered.col.fun), (ered.col.marred)) which are mutually exclusive.

Case $e = [v^{hm_1}]^{h_2}_{hm_1}$**:** By Lemma H.10 (values do not reduce), no reduction can be derived from (ered.cong) since v^{hm_1} is a value. The only potentially applicable rules are (ered.col.col) and (ered.col.le). If $h_2 = hm$ then (ered.col.col) does not apply. Otherwise we do not have $h_2 \preccurlyeq hm$ so (ered.col.le) does not apply. In both cases, only one reduction is possible.

Case $e = [e_1]^{T}_{hm_1}$ **where** e_1 **is not a value:** Then the only possibility of reduction is (ered.cong). Note that (ered.col.col) is out since its side conditions would force e_1 to be a value. By induction there is at most one way to reduce e_1, so there is at most one way to reduce e.

□

Discussion H.12 (strength of determinism) Recall the rule to remove a bracket around another bracket:

$$[[v^{h_0}]^{h_0}_{h_1}]^{h_0}_{h_0} \longrightarrow_{hm} [v^{h_0}]^{h_0}_{h_0} \qquad \text{if } h_0 \neq h_1 \text{ and } h_1 \neq hm \qquad \text{(ered.col.col)}$$

These side conditions do not in fact affect the reductions possible: we have $[[v^{h_0}]^{h_0}_{h_0}]^{h_0}_{h_0} \longrightarrow_{hm} [v^{h_0}]^{h_0}_{h_0}$ by (ered.col.le) and (ered.cong), and $[[v^{h_0}]^{h_0}_{h_0}]^{h_0}_{h_1} \longrightarrow_{h_1} [v^{h_0}]^{h_0}_{h_0}$ by (ered.col.le). We include the side conditions to make our determinism result, Theorem H.11 (determinism of expression reduction), stronger. Without these side conditions, we could state determinism in the following way: if $e \longrightarrow_{hm} e_1$ and $e \longrightarrow_{hm} e_2$ then $e_1 = e_2$.

Of course, network reduction is not deterministic, as befits a concurrent system.

I Compilation

I.1 Decidability of type checking

Definition I.1 (revelation of the implementation of a hash) Let $h = \mathbf{hash}(N, [T_0, v]{:}[X{:}\mathbf{Type}, T_1])$. The type $\mathrm{reveal}_h\, T$ is obtained by replacing any subterm of T that is equal to h and not itself inside a hash by the implementation type T_0. Thus:

- $\mathrm{reveal}_h \mathrm{UNIT} = \mathrm{UNIT}$
- $\mathrm{reveal}_h (T_1 * \dots * T_j) = (\mathrm{reveal}_h\, T_1) * \dots * (\mathrm{reveal}_h\, T_j)$
- $\mathrm{reveal}_h (T_1 \to T_2) = (\mathrm{reveal}_h\, T_1) \to (\mathrm{reveal}_h\, T_2)$
- $\mathrm{reveal}_h X = X$
- $\mathrm{reveal}_h \mathrm{STRING} = \mathrm{STRING}$
- $\mathrm{reveal}_h\, U.\mathrm{TYPE} = U.\mathrm{TYPE}$
- $\mathrm{reveal}_h\, h = T_0$
- $\mathrm{reveal}_h\, h' = h'$ if $h' \neq h$

Also let $\mathrm{reveal}_\bullet\, T = T$.

Definition I.2 (partial type substitution associated to an environment)

$$\begin{aligned}
\mathrm{partenvsub}_{\mathbf{nil}} &= \mathrm{id} \\
\mathrm{partenvsub}_{E, x:T} &= \mathrm{partenvsub}_E \\
\mathrm{partenvsub}_{E, X:\mathbf{Type}} &= \mathrm{partenvsub}_E \\
\mathrm{partenvsub}_{E, X:\mathbf{Eq}(T)} &= \mathrm{partenvsub}_E\{X \leftarrow T\} \\
\mathrm{partenvsub}_{E, U:[X:\mathbf{Type}, T']} &= \mathrm{partenvsub}_E \\
\mathrm{partenvsub}_{E, U:[X:\mathbf{Eq}(T), T']} &= \mathrm{partenvsub}_E\{U.\mathrm{TYPE} \leftarrow T\}
\end{aligned}$$

Recall Definition G.6 (purely abstract environment).

Lemma I.3 (a purely abstract suffix does not change the substitution) If E_1 is a purely abstract environment then $\mathrm{partenvsub}_{E_0,E_1} = \mathrm{partenvsub}_{E_0}$.

Proof. Trivial from Definition I.2 (partial type substitution associated to an environment). □

Lemma I.4 (stability of types through revelation) If $E \vdash_{hm} T{:}\mathbf{Type}$ then $E \vdash_{hm} \mathrm{reveal}_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T == \mathrm{reveal}_{hm} T$ and $E \vdash_{hm} \mathrm{partenvsub}_E T{:}\mathbf{Type}$ and $E \vdash_{hm} T == \mathrm{partenvsub}_E T$.

A trivial consequence that we also use is that if $E \vdash_{hm} T{:}\mathbf{Type}$ then $E \vdash_{hm} \mathrm{reveal}_{hm}\mathrm{partenvsub}_E T{:}\mathbf{Type}$ and $E \vdash_{hm} T == \mathrm{reveal}_{hm}\mathrm{partenvsub}_E T$. Also, we freely use (Teq.sym) on the conclusion of this lemma.

Proof. Let $h = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T_1])$. The type $\mathrm{reveal}_{hm} T$ can be seen as T with T_0 for hm. By similar reasoning to the proof of Lemma F.24 (type equivalence is a congruence), starting from $E \vdash_{hm} h == T_0$ obtained through (Teq.hash), we get $E \vdash_{hm} T == \mathrm{reveal}_{hm} T$.

By applying Lemma E.6 (environments have to be ok), Lemma E.7 (prefixes of ok environments are ok) and Lemma F.24 (type equivalence is a congruence) to each concrete binding in E, and using various rules, we get that $E \vdash_{hm} T == \mathrm{partenvsub}_E T$. By Lemma F.21 (things have to be ok), we have $E \vdash_{hm} \mathrm{partenvsub}_E T{:}\mathbf{Type}$. □

Lemma I.5 (distinction of fresh type variables) If E is purely abstract and $E \vdash_{hm} X == T$ or $E \vdash_{hm} T == X$ or $E \vdash_{hm} X{:}\mathbf{Eq}(T)$ or $E \vdash_{hm} T{:}\mathbf{Eq}(X)$ or $E \vdash_{hm} [Y{:}\mathbf{Eq}(T), T'] <: [Y{:}\mathbf{Eq}(X), T'']$ or $E \vdash_{hm} [Y{:}\mathbf{Eq}(X), T''] <: [Y{:}\mathbf{Eq}(T), T']$ then $T = X$. Also, if E is purely abstract, then $E \vdash_{hm} U{:}[Y{:}\mathbf{Eq}(X), T']$ cannot be derived.

Proof. Induct on the derivation of the assumed judgement.

Case (TK.sub), $E \vdash_{hm} X{:}\mathbf{Eq}(T)$: There exists K such that the premises are $E \vdash_{hm} X{:}K$ and $E \vdash_{hm} K <: \mathbf{Eq}(T)$. By Lemma G.4 (discreteness of subkinding below Type) and reversing (Keq.Eq), there exists T_1 such that $K = \mathbf{Eq}(T_1)$ and $E \vdash_{hm} T_1 == T$ by a subproof. By induction on the subproof of $E \vdash_{hm} X{:}\mathbf{Eq}(T_1)$, we get $T_1 = X$. Since $E \vdash_{hm} X == T$ by a subproof, by induction, we get $T = X$.

Case (TK.sub), $E \vdash_{hm} T{:}\mathbf{Eq}(X)$: There exists K such that the premises are $E \vdash_{hm} T{:}K$ and $E \vdash_{hm} K <: \mathbf{Eq}(X)$. By Lemma G.4 (discreteness of subkinding below Type) and reversing (Keq.Eq), there exists T_1 such that $K = \mathbf{Eq}(T_1)$ and $E \vdash_{hm} T_1 == X$ by a subproof. By induction, we get $T_1 = X$. Since $E \vdash_{hm} T{:}\mathbf{Eq}(X)$ by a subproof, by induction, we get $T = X$.

Case (TK.Eq): Trivial by induction.

Case (TK.mod): There exists U and T' such that the rule derives $E \vdash_{hm} U.\mathrm{TYPE}{:}\mathbf{Eq}(X)$ from $E \vdash_{hm} U{:}[Y{:}\mathbf{Eq}(X), T']$. By induction this is impossible.

Case (TK.var): Impossible since E is purely abstract.

Case (Teq.Eq): Trivial by induction.

Case (Teq.hash): Impossible by Lemma E.4 (colours have to be ok) and Lemma E.9 (free variables of a judgement come from the environment) applied to $\vdash hm$ ok.

Case (Teq.refl): Then $T = X$ as desired.

Case (Teq.sym): Trivial by induction.

Case (Teq.tran): Then by induction the "middle type" is X, whence by induction again $T = X$.

Case (Ssub.struct): We have either $E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Eq}(X)$ or $E \vdash_{hm} \mathbf{Eq}(X) <: \mathbf{Eq}(T)$ by a subproof. By Lemma G.4 (discreteness of subkinding below Type) and reversing (Keq.Eq), either $E \vdash_{hm} T == X$ or $E \vdash_{hm} X == T$ by a subproof. By induction we get $T = X$ as desired.

Case (Ssub.refl): Trivial.

Case (Ssub.tran): Similar to (Teq.tran) above.

Case (US.var): Impossible since E is purely abstract.

Case (US.sub): The conclusion is $E \vdash_{hm} U{:}[Y{:}\mathbf{Eq}(X), T']$. The premises are $E \vdash_{hm} U{:}[Y{:}K, T'']$ and $E \vdash_{hm} [Y{:}K, T''] <: [Y{:}\mathbf{Eq}(X), T']$. By Lemma F.26 (reversing subsignaturing judgement), $E \vdash_{hm} K <: \mathbf{Eq}(X)$ by a subproof. By Lemma G.4 (discreteness of subkinding below Type) and reversing (Keq.Eq), there exists T_1 such that $K = \mathbf{Eq}(T_1)$ and $E \vdash_{hm} T_1 == X$ by a subproof. By induction we get $T_1 = X$. The other premise of the bottommost rule is therefore $E \vdash_{hm} U{:}[Y{:}\mathbf{Eq}(X), T'']$, which is impossible by induction.

Case (US.self): Impossible.

□

Lemma I.6 (open interpretation of type equivalence) $E \vdash_{hm} T == T'$ iff $E \vdash_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T'{:}\mathbf{Type}$ and $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T'$.

Proof. Assume $E \vdash_{hm} T == T'$. By Lemma F.21 (things have to be ok), we have $E \vdash_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T'{:}\mathbf{Type}$. We will now prove the last statement by induction on the lexically ordered pair (m, n) where m is the length of E and n is the number of module bindings in E. Let E_1 be the longest bare bones suffix of E (see Definition G.5 (bare bones environment)).

Case $E = E_1$**:** Then $\mathrm{partenvsub}_E = \mathrm{id}$, so the proof obligation is $\mathrm{reveal}_{hm}\, T = \mathrm{reveal}_{hm}\, T'$. Induct on the structure of T.

Case $T = TC(T_1, ..., T_j)$**:** By Lemma G.11 (type decomposition), there exist $T'_1, ..., T'_j$ such that $E \vdash_{hm} T'_i == T_i$ for each i. Then, by induction, for each i, we have $\mathrm{reveal}_{hm}\, T'_i = \mathrm{reveal}_{hm}\, T_i$. Furthermore, one of the following cases holds:

Case $T' = TC(T'_1, ..., T'_j)$**:** Then we have $\mathrm{reveal}_{hm}\, T' = \mathrm{reveal}_{hm}\, T$.

Case $T' = hm = \mathbf{hash}(N, [TC(T'_1, ..., T'_j), v]{:}[X{:}\mathbf{Type}, T''])$**:** Then $\mathrm{reveal}_{hm}\, T' = TC(T'_1, ..., T'_j) = \mathrm{reveal}_{hm}(TC(T'_1, ..., T'_j)) = T$ (note that for all i, $\mathrm{reveal}_{hm}\, T'_i = T'_i$ since hm is not a subterm of T'_i).

Case $T = X$**:** By Lemma I.5 (distinction of fresh type variables), we have $T' = X = T$.

Case $T = U.\mathrm{TYPE}$**:** Impossible by Lemma E.9 (free variables of a judgement come from the environment).

Case $T = h_1$**:** By Lemma G.11 (type decomposition), one of the following cases holds:

Case $hm = h_1 = \mathbf{hash}(N, [T', v]{:}[X{:}\mathbf{Type}, T''])$**:** Then $\mathrm{reveal}_{hm} h_1 = T' = \mathrm{reveal}_{hm}\, T'$ (note that hm is not a subterm of T').

Case $T' = hm = \mathbf{hash}(N, [h_1, v]{:}[X{:}\mathbf{Type}, T''])$**:** Then $\mathrm{reveal}_{hm} h_1 = h_1 = \mathrm{reveal}_{hm}\, T'$.

Case $T' = h_1$**:** Trivial.

Case $E = E_0, x{:}T_1, E_1$**:** By Lemma E.6 (environments have to be ok) and Lemma E.22 (environments do not contain free expression variables), $x \notin \mathrm{fv}\, E_1$. By Lemma E.21 (types do not contain free expression variables), $x \notin \mathrm{fv}\, T \cup \mathrm{fv}\, T'$. Then, by Lemma F.15 (strengthening), we get $E_0, E_1 \vdash_{hm} T == T'$. We can apply induction to E_0, E_1, getting $\mathrm{reveal}_{hm}\mathrm{partenvsub}_{E_0,E_1}\, T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_{E_0,E_1}\, T'$. Since $\mathrm{partenvsub}_{E_0,E_1} = \mathrm{partenvsub}_E$, this is the desired result.

Case $E = E_0, X{:}\mathbf{Eq}(T_1), E_1$**:** By Lemma E.19 (computing the pvu of a type world judgement), $hm \preccurlyeq \min(\mathrm{pvu}_X\,(E_1))$. By Lemma E.7 (prefixes of ok environments are ok) and reversing (envok.X), we have $E_0 \vdash_{hm} T_1{:}\mathbf{Type}$, whence by (Teq.refl) and (TK.Eq), $E_0 \vdash_{hm} T_1{:}\mathbf{Eq}(T_1)$. We can use Lemma F.14 (type preservation by substitution) to $\{X \leftarrow T_1\}$, getting $E' \vdash_{hm} \{X \leftarrow T_1\} T == \{X \leftarrow T_1\} T'$ where $E' = E_0, \{X \leftarrow T_1\} E_1$.

By induction, we get that $\mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\{X \leftarrow T_1\} T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\{X \leftarrow T_1\} T'$. By Definition I.2 (partial type substitution associated to an environment), $\mathrm{partenvsub}_{E_0,X{:}\mathbf{Eq}(T_1)} = \mathrm{partenvsub}_{E_0}\{X \leftarrow T_1\}$. By Lemma I.3 (a purely abstract suffix does not change the substitution), $\mathrm{partenvsub}_E = \mathrm{partenvsub}_{E_0}\{X \leftarrow T_1\}$. So we have $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T'$, as desired.

Case $E = E_0, U{:}[X{:}K, T_1], E_1$**:** Let Z and z be fresh. By Lemma F.29 (type preservation by module substitution in coloured judgements), we get $E_0, Z{:}\mathbf{Eq}(T_1), z{:}\{X \leftarrow Z\} T_2, \{U.\mathrm{TYPE} \leftarrow Z, U.\mathrm{term} \leftarrow z\} E_1 \vdash_{hm} \{U.\mathrm{TYPE} \leftarrow Z, U.\mathrm{term} \leftarrow z\} T == \{U.\mathrm{TYPE} \leftarrow Z, U.\mathrm{term} \leftarrow z\} T'$. Since E_1 is purely abstract, we have $\{U.\mathrm{TYPE} \leftarrow Z, U.\mathrm{term} \leftarrow z\} E_1 = E_1$. By Lemma E.21 (types do not contain free expression variables), $z \notin \mathrm{fv}\,\{U.\mathrm{TYPE} \leftarrow Z\} T$, whence $\{U.\mathrm{TYPE} \leftarrow Z, U.\mathrm{term} \leftarrow z\} T = \{U.\mathrm{TYPE} \leftarrow Z\} T$, and the same holds for T'. By Lemma F.15 (strengthening), we get $E' \vdash_{hm} \sigma T == \sigma T'$ where $E' = E_0, Z{:}K, E_1$ and $\sigma = \{U.\mathrm{TYPE} \leftarrow Z\}$. Now we do case analysis on the structure of K.

Case $K = \mathbf{Type}$**:** By induction on the last judgement above, we have $\mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\sigma T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\sigma T'$. Now $\mathrm{partenvsub}_{E'}\sigma = \mathrm{partenvsub}_{E}$, so we have the desired result.

Case $K = \mathbf{Eq}(T_0)$**:** Apply the case $E = E_0, Z{:}\mathbf{Eq}(T_0), E_1$ above to $E' \vdash_{hm} \sigma T == \sigma T'$. We get $\mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\sigma T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\sigma T'$. Now given Definition I.2 (partial type substitution associated to an environment) and Lemma I.3 (a purely abstract suffix does not change the substitution), we in fact have $\mathrm{partenvsub}_{E} = \mathrm{partenvsub}_{E}\sigma$. So we have the desired result.

We turn to the other half of the proof. Assume that $E \vdash_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T'{:}\mathbf{Type}$ and $\mathrm{reveal}_{hm}\mathrm{partenvsub}_{E} T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_{E} T'$. We induct on the number of bindings of E that are not of the form $U{:}[X{:}\mathbf{Type}, \_]$ nor $X{:}\mathbf{Type}$. We write E_1 for the longest purely abstract suffix of E.

Case E **is purely abstract:** Induct on the structure of T. Note that $\mathrm{partenvsub}_{E} = \mathrm{id}$.

Case $T = TC(T_1, ..., T_j)$**:** By definition of reveal_{hm}, one of the following cases holds:

Case $T' = \text{UNIT}$ **or** $T' = \mathbf{mar}$**:** Then $T = T'$, so $E \vdash_{hm} T == T'$ by (Teq.refl).

Case $T' = TC(T'_1, ..., T'_j)$ **with** $j \neq 0$**:** Still by the definition of reveal_{hm}, we have $\mathrm{reveal}_{hm} T_i = \mathrm{reveal}_{hm} T'_i$ for all i. Then by induction we have $E \vdash_{hm} T_i == T'_i$. By the appropriate rule amongst (Teq.cong.fun) and (Teq.cong.tuple), we have $E \vdash_{hm} T == T'$.

Case $T' = hm = \mathbf{hash}(N, [T, v]{:}[X{:}\mathbf{Type}, T''])$**:** We have $E \vdash_{hm} \mathrm{ok}$ by Lemma E.6 (environments have to be ok). Then, by (Teq.hash), we get $E \vdash_{hm} T' == T$, whence $E \vdash_{hm} T == T'$ by (Teq.sym).

Case $T = X$ **or** $T = U.\text{TYPE}$**:** Then $\mathrm{reveal}_{hm} T = T = T' = \mathrm{reveal}_{hm} T'$. By (Teq.refl), we have $E \vdash_{hm} T == T'$.

Case $T = hm$**:** By definition of reveal_{hm}, one of the following cases holds:

Case $T' = hm$**:** Then $E \vdash_{hm} T == T'$ by (Teq.refl).

Case $hm = \mathbf{hash}(N, [T', v]{:}[X{:}\mathbf{Type}, T''])$**:** We have $E \vdash_{hm} \mathrm{ok}$ by Lemma E.6 (environments have to be ok). Then, by (Teq.hash), we get $E \vdash_{hm} T == T'$.

Case T **is a hash but** $T \neq hm$**:** Then $\mathrm{reveal}_{hm} T = T$. By definition of reveal_{hm}, one of the following cases holds:

Case $T' = hm = \mathbf{hash}(N, [T, v]{:}[X{:}\mathbf{Type}, T''])$**:** We have $E \vdash_{hm} \mathrm{ok}$ by Lemma E.6 (environments have to be ok). Then, by (Teq.hash), we get $E \vdash_{hm} T' == T$, whence $E \vdash_{hm} T == T'$ by (Teq.sym).

Case $T' = T$**:** Then $E \vdash_{hm} T == T'$ by (Teq.refl).

Case $E = E_0, x{:}T_0, E_1$**:** By Lemma E.21 (types do not contain free expression variables), Lemma E.22 (environments do not contain free expression variables) and Lemma F.15 (strengthening), we have $E_0, E_1 \vdash_{hm} T{:}\mathbf{Type}$ and $E_0, E_1 \vdash_{hm} T'{:}\mathbf{Type}$. Since furthermore $\mathrm{partenvsub}_{E} = \mathrm{partenvsub}_{E_0,E_1}$, by induction, we get $E_0, E_1 \vdash_{hm} T == T'$. By Lemma E.6 (environments have to be ok) and Lemma E.7 (prefixes of ok environments are ok), we have $E_0, x{:}T_0 \vdash_{hm} \mathrm{ok}$, so $E \vdash_{hm} T == T'$ by Lemma F.5 (combined weakening) (note that $x \notin \mathrm{fv}\, E_1$ by Lemma E.8 (ok environments have no repetition in the domain)).

Case $E = E_0, X{:}\mathbf{Eq}(T_0), E_1$**:** Then $\mathrm{partenvsub}_{E} = \mathrm{partenvsub}_{E_0}\{X \leftarrow T_0\}$. By Lemma E.6 (environments have to be ok), Lemma E.7 (prefixes of ok environments are ok) and reversing (envok.X) and (Kok.Eq), we have $E_0 \vdash_{hm} T_0{:}\mathbf{Type}$. By Lemma E.19 (computing the pvu of a type world judgement), any proof Π of $E \vdash_{hm} T{:}\mathbf{Type}$ satisfies $hm \preccurlyeq \min(\mathrm{pvu}_X(\Pi))$. By Lemma F.14 (type preservation by substitution), we get $E_0, \{X \leftarrow T_0\}E_1 \vdash_{hm} \{X \leftarrow T_0\}T{:}\mathbf{Type}$. Similarily, we have $E_0, \{X \leftarrow T_0\}E_1 \vdash_{hm} \{X \leftarrow T_0\}T'{:}\mathbf{Type}$.

By induction, we get $E_0, \{X \leftarrow T_0\}E_1 \vdash_{hm} \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$.

Since $E_0, X{:}\mathbf{Eq}(T_0) \vdash_{hm} \mathrm{ok}$ and $X \notin \mathrm{dom}\, E_1$ by Lemma E.8 (ok environments have no repetition in the domain), by Lemma F.5 (combined weakening), we have $E_0, X{:}\mathbf{Eq}(T_0), \{X \leftarrow T_0\}E_1 \vdash_{hm} \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$.

By Lemma G.8 (type substitution in a purely abstract environment), given the latest judgement and $E_0, X{:}\mathbf{Eq}(T_0), E_1 \vdash_{hm} \mathrm{ok}$, we get $E_0, X{:}\mathbf{Eq}(T_0), E_1 \vdash_{hm} \{X \leftarrow T_0\}T == \{X \leftarrow T_0\}T'$.

By Lemma F.25 (type substitution in equivalence), we get $E \vdash_{hm} T == T'$ as desired.

Case $E = E_0, U{:}[X{:}\mathbf{Eq}(T_0), T_1], E_1$**:** Then $\mathrm{partenvsub}_E = \mathrm{partenvsub}_{E_0}\{U.\textsc{type}{\leftarrow}T_0\}$. Let Z be a fresh variable, and let $E' = E_0, Z{:}\mathbf{Eq}(T_0), \{U.\textsc{type}{\leftarrow}Z\}E_1$. By the definition of partenvsub, we have $\mathrm{partenvsub}_{E'} = \mathrm{partenvsub}_{E_0}\{Z{\leftarrow}T_0\}$. Since Z is fresh, we have $\mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\{U.\textsc{type}{\leftarrow}Z\}T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_{E'}\{U.\textsc{type}{\leftarrow}Z\}T'$.

By Lemma F.31 (type preservation by module substitution in coloured judgements for type world judgements), since $E \vdash_{hm} T{:}\mathbf{Type}$, we get $E' \vdash_{hm} \{U.\textsc{type}{\leftarrow}Z\}T{:}\mathbf{Type}$. Similarly we have $E' \vdash_{hm} \{U.\textsc{type}{\leftarrow}Z\}T'{:}\mathbf{Type}$.

By the previous case, $E' \vdash_{hm} \{U.\textsc{type}{\leftarrow}Z\}T == \{U.\textsc{type}{\leftarrow}Z\}T'$.

By Lemma E.6 (environments have to be ok) and Lemma E.7 (prefixes of ok environments are ok), we have $E_0, U{:}[X{:}\mathbf{Eq}(T_0), T_1] \vdash_{hm} \mathrm{ok}$. Using Lemma E.8 (ok environments have no repetition in the domain), we get $U \notin \{Z\} \cup \mathrm{dom}(\{U.\textsc{type}{\leftarrow}Z\}E_1)$, so by Lemma F.5 (combined weakening), we have $E_0, U{:}[X{:}\mathbf{Eq}(T_0), T_1], Z{:}\mathbf{Eq}(T_0), \{U.\textsc{type}{\leftarrow}Z\}E_1 \vdash_{hm} \{U.\textsc{type}{\leftarrow}Z\}T == \{U.\textsc{type}{\leftarrow}Z\}T'$.

By (US.var) and (TK.mod), we have $E_0, U{:}[X{:}\mathbf{Eq}(T_0), T_1] \vdash_{hm} U.\textsc{type}{:}\mathbf{Eq}(T_0)$. By Lemma E.19 (computing the pvu of a type world judgement), we can apply Lemma F.14 (type preservation by substitution). We get $E_0, U{:}[X{:}\mathbf{Eq}(T_0), T_1], \{Z{\leftarrow}U.\textsc{type}\}\{U.\textsc{type}{\leftarrow}Z\}E_1 \vdash_{hm} \{Z{\leftarrow}U.\textsc{type}\}\{U.\textsc{type}{\leftarrow}Z\}T == \{Z{\leftarrow}U.\textsc{type}\}\{U.\textsc{type}{\leftarrow}Z\}T'$. Since Z was fresh, we have $E \vdash_{hm} T == T'$, as desired.

□

Theorem I.7 (semantic interpretation of type world judgements) Let $E \vdash_{hm} J$ be any type world judgement. It is derivable iff all of the following conditions are met:

1. E has no repetition in the domain and unresolved free variables.
2. All the free variables of J are in $\mathrm{dom}\,E$.
3. All the hashes in $E \vdash_{hm} J$ are correct.
4. The following extra condition is met, depending on the form of J:

   Case $E \vdash_{hm} \mathrm{ok}$ **or** $E \vdash_{hm} K\,\mathrm{ok}$ **or** $E \vdash_{hm} T{:}\mathbf{Type}$ **or** $T \vdash_{hm} S\,\mathrm{ok}$**:** no extra condition.

   Case $E \vdash_{hm} T == T'$ **or** $E \vdash_{hm} T{:}\mathbf{Eq}(T')$**:** $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E T = \mathrm{reveal}_{hm}\mathrm{partenvsub}_E T'$.

   Case $E \vdash_{hm} K == K'$**:** either $K = K' = \mathbf{Type}$, or there exist T and T' such that $K = \mathbf{Eq}(T)$ and $K' = \mathbf{Eq}(T')$ and $E \vdash_{hm} T == T'$.

   Case $E \vdash_{hm} K <: K'$**:** either $K' = \mathbf{Type}$, or there exist T and T' such that $K = \mathbf{Eq}(T)$ and $K' = \mathbf{Eq}(T')$ and $E \vdash_{hm} T == T'$.

   Case $E \vdash_{hm} [X{:}K, T] <: [X{:}K', T']$**:** $E \vdash_{hm} K <: K'$ and $E, X{:}K \vdash_{hm} T == T'$.

   Case $E \vdash_{hm} U{:}[X{:}K, T]$**:** E contains a binding of the form $U{:}[X{:}K_1, T_2]$ for some K_1 and T_2, and $E \vdash_{hm} [X{:}\mathbf{Eq}(U.\textsc{type}), T_2] <: [X{:}K, T]$.

Proof. If $E \vdash_{hm} J$, then E has no repetition in the domain and no unresolved free variables by Lemma E.6 (environments have to be ok) and Lemma E.8 (ok environments have no repetition in the domain) and Lemma F.13 (ok environments have no unresolved free variables); all the hashes in J are correct by Lemma E.5 (hashes have to be ok), and all the free variables of J are in $\mathrm{dom}\,E$ by Lemma E.9 (free variables of a judgement come from the environment). We now assume that conditions 1–3 are met, and prove that $E \vdash_{hm} J$ is derivable iff condition 4 is met.

Cases $E \vdash_{hm} \mathrm{ok}$**,** $E \vdash_{hm} K\,\mathrm{ok}$**,** $E \vdash_{hm} T{:}\mathbf{Type}$**,** $E \vdash_{hm} [X{:}K, T]\,\mathrm{ok}$**:** Since there is no extra condition, we just have to prove that the judgement is derivable. We induct on the length of E.

Case nil $\vdash_{hm} \mathrm{ok}$**:** By condition 3 (if hm is a hash) or (hmok.zero) (if $hm = \bullet$), we can derive $\vdash hm\,\mathrm{ok}$, whence we apply (envok.nil).

Case $E, \zeta{:}\tau \vdash_{hm} \mathrm{ok}$**:** Given the (envok.*) rules, this judgement is derivable iff the judgement $E \vdash_{hm} \zeta{:}\tau$ is. By induction, these judgements are derivable.

Case $E \vdash_{hm} T{:}\mathbf{Type}$**:** By the previous cases, $E \vdash_{hm} \mathrm{ok}$ is derivable. By Lemma F.9 (types are ok provided their hashes are), $E \vdash_{hm} T{:}\mathbf{Type}$ is derivable.

Case $E \vdash_{hm} \mathbf{Type}\,\mathrm{ok}$**:** By the previous cases, $E \vdash_{hm}\ \mathrm{ok}$ is derivable. Then we can derive $E \vdash_{hm} \mathbf{Type}\,\mathrm{ok}$ by (Kok.Type).

Case $E \vdash_{hm} \mathbf{Eq}(T)\,\mathrm{ok}$**:** By the previous cases, $E \vdash_{hm}\ \mathrm{ok}$ is derivable. By Lemma F.9 (types are ok provided their hashes are), $E \vdash_{hm} T{:}\mathbf{Type}$ is derivable. Then we can derive $E \vdash_{hm} \mathbf{Eq}(T)\,\mathrm{ok}$ by (Kok.Eq).

Case $E \vdash_{hm} [X{:}K,\, T]\,\mathrm{ok}$**:** By the previous cases, $E \vdash_{hm} K\,\mathrm{ok}$ is derivable. By alpha-conversion, we can choose X not in $\mathrm{dom}\, E$. By (envok.X), we get a proof of $E, X{:}K \vdash_{hm}\ \mathrm{ok}$. By Lemma F.9 (types are ok provided their hashes are), since $\mathrm{fv}\, T \subseteq \mathrm{fv}\,[X{:}K,\, T] \cup \{X\} \subseteq \mathrm{dom}\, E \cup \{X\}$, $E, X{:}K \vdash_{hm} T{:}\mathbf{Type}$ is derivable. Then, by (Sok), we get $E \vdash_{hm} [X{:}K,\, T]\,\mathrm{ok}$.

Case $E \vdash_{hm} T == T'$**:** Note that by Lemma F.9 (types are ok provided their hashes are), we have $E \vdash_{hm} T{:}\mathbf{Type}$ and $E \vdash_{hm} T'{:}\mathbf{Type}$. Apply Lemma I.6 (open interpretation of type equivalence).

Case $E \vdash_{hm} T{:}\mathbf{Eq}(T')$**:** Given (TK.Eq) and (Teq.Eq), this judgement is equivalent to $E \vdash_{hm} T == T'$. Apply the previous case.

Case $E \vdash_{hm} K == K'$**:** Obvious from (Keq.*).

Case $E \vdash_{hm} K <: K'$**: Subcase** $K = K' = \mathbf{Type}$**:** Trivial.

Subcase $K = \mathbf{Eq}(T)$ **and** $K' = \mathbf{Type}$**:** By Lemma F.9 (types are ok provided their hashes are) and (Kok.Eq), $E \vdash_{hm} \mathbf{Eq}(T)\,\mathrm{ok}$ is derivable. By Lemma E.6 (environments have to be ok) and (Kok.Type), we have $E \vdash_{hm} \mathbf{Type}$. By (Ksub.Eq), we have $E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Type}$.

Subcase $K = \mathbf{Type}$ **and** $K' = \mathbf{Eq}(T')$**:** By Lemma G.4 (discreteness of subkinding below Type) and the (Keq.*) rules, this is impossible.

Subcase $K = \mathbf{Eq}(T)$ **and** $K' = \mathbf{Eq}(T')$**:** If $E \vdash_{hm} T == T'$ then $E \vdash_{hm} \mathbf{Eq}(T) <: \mathbf{Eq}(T')$ by (Keq.Eq) and (Ksub.refl). The converse follows from Lemma G.4 (discreteness of subkinding below Type).

Case $E \vdash_{hm} [X{:}K,\, T] <: [X{:}K,\, T']$**:** Apply Lemma F.26 (reversing subsignaturing judgement). Conversely, apply (Ssub.struct).

Case $E \vdash_{hm} U{:}[X{:}K,\, T]$**:** Apply Lemma F.27 (reversing module value variable typing judgement) and Lemma F.28 (obtaining module value variable typing judgement).

□

Lemma I.8 (uniqueness of expression typing) If $E \vdash_{hm} e{:}T$ then $E \vdash_{hm} e{:}T'$ iff $E \vdash_{hm} T == T'$.

Proof. If $E \vdash_{hm} e{:}T$ and $E \vdash_{hm} T == T'$ then $E \vdash_{hm} e{:}T'$ by (eT.eq). Now we prove the converse by induction on the structure of e. The crux of the matter is that (1) apart from (eT.eq), the expression typing rules are syntax-directed and (2) type equivalence is a congruence.

We have $E \vdash_{hm} e{:}T$ and $E \vdash_{hm} e{:}T'$. By Lemma G.1 (shortening typing proof), there exist T_1 and T_1' such that $E \vdash_{hm} e{:}T_1$ and $E \vdash_{hm} T_1'$ by proofs Π and Π' that do not end with (eT.eq), and $E \vdash_{hm} T == T_1$ and $E \vdash_{hm} T' == T_1'$. We now prove that $E \vdash_{hm} T_1 == T_1'$; by (Teq.tran) and (Teq.sym) this gives $E \vdash_{hm} T == T'$ as desired. Note that it is enough to prove that $T_1 = T_1'$. We do case analysis on the structure of e; in each case Π and Π' have to end with the same rule.

Case $e = x$ **(eT.var):** Then E contains a binding for x; by Lemma E.6 (environments have to be ok) and Lemma E.8 (ok environments have no repetition in the domain), this binding is unique, and the type attributed to x is $T_1 = T_1'$.

Case $e = U.\mathrm{term}$ **(eT.mod):** Then there exist K and K' such that $E \vdash_{hm} U{:}[X{:}K,\, T_1]$ and $E \vdash_{hm} U{:}[X{:}K,\, T_1']$. By Theorem I.7 (semantic interpretation of type world judgements) and Lemma F.26 (reversing subsignaturing judgement), we see that there exists T_2 such that $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} T_2 == T_1$ and $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} T_2 == T_1'$. By (Teq.tran) and (Teq.sym), we get $E, X{:}\mathbf{Eq}(U.\mathrm{TYPE}) \vdash_{hm} T_1 == T_1'$. Since $E \vdash_{hm} T_1{:}\mathbf{Type}$ and $E \vdash_{hm} T_1'{:}\mathbf{Type}$, by Lemma E.9 (free variables of a judgement come from the environment), $X \notin \mathrm{fv}\, T_1 \cup \mathrm{fv}\, T_1'$. Hence, by Lemma F.15 (strengthening), we get $E \vdash_{hm} T_1 == T_1'$ as desired.

Case $e = e_1\, e_2$ **(eT.ap):** Then there exist T_2 and T_2' such that $E \vdash_{hm} e_1{:}T_2 \rightarrow T_1$ and $E \vdash_{hm} e_1'{:}T_2' \rightarrow T_1'$. By induction, we get $E \vdash_{hm} T_2 \rightarrow T_1 == T_2' \rightarrow T_1'$. By Lemma I.6 (open interpretation of type equivalence) applied once in each direction, we get $E \vdash_{hm} T_1 == T_1'$ as desired.

Case $e = \lambda x{:}T_0.e_0$ **(eT.fun):** Then there exist T_2 and T_2' such that $T_1 = T_0 \rightarrow T_2$ and $T_1 = T_0 \rightarrow T_2'$ and $E, x{:}T_0 \vdash_{hm} e_0{:}T_2$ and $E, x{:}T_0 \vdash_{hm} e_0{:}T_2'$. By induction we get $E, x{:}T_0 \vdash_{hm} T_2 == T_2'$. By Lemma E.21 (types do not contain free expression variables) and Lemma F.15 (strengthening), we get $E \vdash_{hm} T_2 == T_2'$. By (Teq.refl) and (Teq.cong.fun), we get $E \vdash_{hm} T_1 == T_1'$ as desired.

Cases $e = !\, e_0$, $e = ()$ **(eT.send, eT.unit):** Then $T_1 = \text{UNIT} = T_1'$.

Cases $e = ?$, $e = \mathbf{mar}\,(e_0{:}T_0)$, $e = \mathbf{marshalled}\,(e_0{:}T_0)$ **(eT.recv, eT.mar, eT.marred):** Then $T_1 = \text{STRING} = T_1'$.

Cases $e = \mathbf{unmar}\; e_0{:}T_0$, $e = \mathbf{UnmarFailure}^{T_0}$, $e = [e_0]^{T_0}_{hm_0}$ **(eT.unmar, eT.Undynfailure, eT.col):** Then $T_1 = T_0 = T_1'$.

Case $e = (e_1, ..., e_j)$ **(eT.tuple):** Similar to the (eT.fun) case, using (Teq.cong.tuple).

Case $e = \mathbf{proj}_i\; e_0$ **(eT.proj):** Similar to the (eT.ap) case.

□

Definition I.9 (reconstructed type of an expression) We define the reconstructed type of an expression e in an environment E and a colour hm, written $\text{basictype}^E_{hm}\; e$, recursively as follows:

$$\begin{array}{rcll}
\text{basictype}^E_{hm}\,(x) & = & \sigma^E_{hm}\, T & \text{where } E = ..., x{:}T, ... \\
\text{basictype}^E_{hm}\,(U.\text{term}) & = & \sigma^E_{hm}\{X \leftarrow U.\text{TYPE}\}\, T & \text{where } E = ..., U{:}[X{:}K, T], ... \\
\text{basictype}^E_{hm}\,(\lambda x{:}T_0.e_0) & = & (\sigma^E_{hm}\, T_0) \rightarrow (\text{basictype}^{E,x:T_0}_{hm}\; e_0) & \\
\text{basictype}^E_{hm}\,(e_1\; e_2) & = & T_1 & \text{where basictype}^E_{hm}\; e_1 = T_2 \rightarrow T_1 \\
\text{basictype}^E_{hm}\,((e_1, ..., e_j)) & = & \text{basictype}^E_{hm}\; e_1 * ... * \text{basictype}^E_{hm}\; e_j & \\
\text{basictype}^E_{hm}\,(\mathbf{proj}_i\; e_0) & = & T_i & \text{where basictype}^E_{hm}\; e_0 = T_1 * ... * T_j \\
\text{basictype}^E_{hm}\,(\mathbf{mar}\,(e_0{:}T_0)) & = & \text{STRING} & \\
\text{basictype}^E_{hm}\,(\mathbf{marshalled}\,(e_0{:}T_0)) & = & \text{STRING} & \\
\text{basictype}^E_{hm}\,(\mathbf{unmar}\; e_0{:}T_0) & = & \sigma^E_{hm}\, T_0 & \\
\text{basictype}^E_{hm}\,(!\, e_0) & = & \text{UNIT} & \\
\text{basictype}^E_{hm}\,(?) & = & \text{STRING} & \\
\text{basictype}^E_{hm}\,([e_0]^{T_0}_{hm_0}) & = & \sigma^E_{hm}\, T_0 & \\
\text{basictype}^E_{hm}\,(\mathbf{UnmarFailure}^{T_0}) & = & \sigma^E_{hm}\, T_0 &
\end{array}$$

where $\sigma^E_{hm} = \text{reveal}_{hm}\text{partenvsub}_E$.

Note that this definition basictype^E_{hm} is partial: it fails if a subexpression does not have a type of the required form (cases $e_1\; e_2$, $(e_1, ..., e_j)$) or if a variable is not bound by the environment (cases x, $U.\text{term}$).

Lemma I.10 (simplicity of the reconstructed type) If $\text{basictype}^E_{hm}\; e$ is well-defined, it contains no hm, nor does it contain any variable that is bound concretely in E.

Proof. Trivial by induction. □

Lemma I.11 (assurance of correctness of the reconstructed type) If $E \vdash_{hm} e{:}T$ then $\text{basictype}^E_{hm}\; e$ is well-defined and $E \vdash_{hm} e{:}\text{basictype}^E_{hm}\; e$.

Proof. Induct on the derivation of $E \vdash_{hm} e{:}T$. By Lemma G.1 (shortening typing proof), there exists T' such that $E \vdash_{hm} e{:}T'$ by a subproof Π that does not end in (eT.eq), and $E \vdash_{hm} T' == T$. Note that by Lemma I.6 (open interpretation of type equivalence), $\text{reveal}_{hm}\text{partenvsub}_E\, T = \text{reveal}_{hm}\text{partenvsub}_E\, T'$. We discriminate on the last rule of Π.

Case (eT.var): We have $E = E_0, x{:}T', E_1$ and $e = x$. By Lemma I.4 (stability of types through revelation), $E \vdash_{hm} T' == \text{reveal}_{hm}\text{partenvsub}_E\, T'$, so by (eT.eq) we get $E \vdash_{hm} x{:}\text{reveal}_{hm}\text{partenvsub}_E\, T'$ as desired.

Case (eT.mod): The premises are $E \vdash_{hm} U{:}[X{:}K, T]$ and $E \vdash_{hm} T{:}\mathbf{Type}$. By Theorem I.7 (semantic interpretation of type world judgements) and Lemma F.26 (reversing subsignaturing judgement), there exist E_0, E_1, K' and T' such that $E = E_0, U{:}[X{:}K', T'], E_1$ and $E, X{:}\mathbf{Eq}(U.\text{TYPE}) \vdash_{hm} T' == T$. By Lemma E.9 (free variables of a judgement come from the environment), we have $X \notin \text{fv}\; T$. By Lemma E.19 (computing the pvu of a type world judgement) and Lemma F.14 (type preservation by substitution), we have $E \vdash_{hm} \{X \leftarrow U.\text{TYPE}\}T' ==$

T. By Lemma I.4 (stability of types through revelation) and (Teq.tran) and (Teq.sym), we have $E \vdash_{hm} T == \mathrm{reveal}_{hm}\mathrm{partenvsub}_E\{X \leftarrow U.\mathrm{TYPE}\}T'$ as desired.

Case (eT.fun): There exist T_1 and T_2 such that $e = \lambda x{:}T_1.e_1$ and $T' = T_1 \rightarrow T_2$, and the premise is $E, x{:}T_1 \vdash_{hm} e_1{:}T_2$. By induction, $E, x{:}T_1 \vdash_{hm} e_1{:}\mathrm{basictype}_{hm}^{E,x:T_1}\, e_1$. Thus, by (eT.fun), we get $E \vdash_{hm} \lambda x{:}T_1.e_1{:}T_1 \rightarrow \mathrm{basictype}_{hm}^{E,x:T_1}\, e_1$. By Lemma I.4 (stability of types through revelation), we have $E \vdash_{hm} \mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T_1 == T_1$, whence the desired result by (Teq.cong.fun).

Case (eT.ap): There exist e_1, e_2 and T_2 such that $e = e_1\, e_2$ the premises are $E \vdash_{hm} e_1{:}T_2 \rightarrow T'$ and $E \vdash_{hm} e_2{:}T_2$. By induction, we have $E \vdash_{hm} e_1{:}\mathrm{basictype}_E^{hm}\, e_1$. By Lemma I.8 (uniqueness of expression typing), we have $E \vdash_{hm} \mathrm{basictype}_E^{hm}\, e_1 == T_2 \rightarrow T'$. By Lemma I.10 (simplicity of the reconstructed type), $\mathrm{partenvsub}_{hm}(\mathrm{basictype}_E^{hm}\, e_1) = \mathrm{basictype}_E^{hm}\, e_1$. By using Lemma F.14 (type preservation by substitution) and Lemma F.33 (type preservation by fully carried out module substitution) once for each concrete binding in E, we get that $E' \vdash_{\mathrm{basictype}_E^{hm}\, e_1} == (\mathrm{partenvsub}_E\, T_2) \rightarrow (\mathrm{partenvsub}_E\, T')$ with E' fully abstract. By Lemma G.11 (type decomposition), and given that $\mathrm{basictype}_E^{hm}\, e_1 \neq hm$ by Lemma I.10 (simplicity of the reconstructed type), $\mathrm{basictype}_E^{hm}\, e_1$ must have the form $T_2' \rightarrow T_1'$, with $E' \vdash_{hm} T_1' == T_1$. By Lemma I.6 (open interpretation of type equivalence) applied once in each direction, we have $E \vdash_{hm} T_1' == T'$. Thus $E \vdash_{hm} (\mathrm{basictype}_{hm}^E\, e) == T'$, whence the desired result.

Case (eT.tuple): Similar to the lambda-abstraction case.

Case (eT.proj): Similar to the application case.

Cases (eT.mar), (eT.marred), (eT.send), (eT.recv): Trivial.

Case (eT.unmar), (eT.col), (eT.Undynfailure): Trivial.

□

Lemma I.12 (validity of the reconstructed type) If $\mathrm{basictype}_{hm}^E\, e$ is well-defined and $E \vdash_{hm}$ ok then $E \vdash_{hm} \mathrm{basictype}_{hm}^E\, e{:}\mathbf{Type}$ and $E \vdash_{hm} e{:}\mathrm{basictype}_{hm}^E\, e$.

Proof. Induct on the structure of e. The rules to define $\mathrm{basictype}_{hm}^E$ were chosen on purpose. Some steps apply $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E$, which produces an equivalent type by an easy application of Lemma I.6 (open interpretation of type equivalence). □

Corollary I.13 (deciding expression typing through type reconstruction) $E \vdash_{hm} e{:}T$ is derivable iff $\mathrm{basictype}_{hm}^E\, e$ is well-defined and $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T = \mathrm{basictype}_{hm}^E\, e$ and $E \vdash_{hm} T{:}\mathbf{Type}$.

Proof. If $E \vdash_{hm} e{:}T$ is derivable then, by Lemma I.11 (assurance of correctness of the reconstructed type), $\mathrm{basictype}_{hm}^E\, e$ is well-defined and $E \vdash_{hm} e{:}\mathrm{basictype}_{hm}^E\, e$. By Lemma I.8 (uniqueness of expression typing), $E \vdash_{hm} T == \mathrm{basictype}_{hm}^E\, e$. By Lemma I.10 (simplicity of the reconstructed type) and Lemma I.6 (open interpretation of type equivalence), $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T = \mathrm{basictype}_{hm}^E\, e$. Finally, $E \vdash_{hm} T{:}\mathbf{Type}$ by Lemma E.6 (environments have to be ok).

Conversely, assume $\mathrm{basictype}_{hm}^E\, e$ is well-defined and $\mathrm{reveal}_{hm}\mathrm{partenvsub}_E\, T = \mathrm{basictype}_{hm}^E\, e$ and $E \vdash_{hm} T{:}\mathbf{Type}$. By Lemma E.6 (environments have to be ok) and Lemma I.12 (validity of the reconstructed type), we have $E \vdash_{hm} e{:}\mathrm{basictype}_{hm}^E\, e$. By Lemma I.10 (simplicity of the reconstructed type) and Lemma I.6 (open interpretation of type equivalence), $E \vdash_{hm} (\mathrm{basictype}_{hm}^E\, e) == T$. Thus, by (eT.eq), we have $E \vdash_{hm} e{:}T$. □

Theorem I.14 (decidability of type checking) There is algorithm that decides whether any given judgement $\zeta \notin \mathrm{dom}\, E$, $\vdash hm$ ok, $E \vdash_{hm} J$ or $\vdash n$ ok is derivable.

Note that given Lemma I.12 (validity of the reconstructed type), type checking for expressions does not in fact need to be given a type, it can infer it. As in the simply-typed lambda-calculus, of course, we need to annotate function arguments; and we also have explicit type annotations on the dynamic typing constructs and coloured brackets.

Proof. We explain how to reduce the decidability of a judgement to the decidability of a number of strictly smaller judgements, the size of a judgement being here the lexically ordered pair (a, b) where a is the number of expression syntax nodes (including those inside hashes) and b is the total size of the judgement.

By Lemma E.3 (non-membership in domain is interpreted trivially), we get a decision procedure for checking derivability of $\zeta \notin \mathrm{dom}\, E$ judgements, provided we have a decision procedure for checking correctness of embedded hashes.

For type world judgements, Theorem I.7 (semantic interpretation of type world judgements) easily translates into an algorithm.

Given the rules (hmok.*), (MS.struct), (mT.*), and (nok.*), decidability of derivability for judgements of the form $\vdash hm\, \mathrm{ok}$, $E \vdash_{hm} M{:}S$, $E \vdash_{hm} m{:}T$, and $\vdash n\, \mathrm{ok}$ follows from decidability of smaler type world and expression typing judgements.

For expression typing, Corollary I.13 (deciding expression typing through type reconstruction) gives a decision procedure: to decide whether $E \vdash_{hm} e{:}T$ is derivable, construct $\mathrm{basictype}^{E}_{hm}\, e$ (if this fails, then e has no type), and check some smaller type world jdugements. $\square$

Definition I.15 (user source program) User source programs are programs that contain no hashes or coloured brackets.

Discussion I.16 (decidability of type checking for user programs) User source programs can be type checked without any computation on hashes. Following the decision procedure given in the previous proofs, type checking does not introduce extra hashes, hence hash-free programs can be checked without ever encountering hashes.

I.2 Bracket elimination

Definition I.17 (bracket elimination subsystem) The bracket elimination subsystem is a reduction relation $\underset{\mathrm{be}}{\longrightarrow}_{hm}$ on expressions consisting of (ered.cong) and the (ered.col.*) rules.

Lemma I.18 (determinism of bracket elimination) Bracket elimination is deterministic.

Proof. Trivial consequence of Theorem H.11 (determinism of expression reduction). $\square$

Lemma I.19 (termination of bracket elimination) Bracket elimination is strongly normalising.

Proof. Define the bracket elimination weight of an expression $w_{\mathrm{be}}(e)$ structurally on the expression.

$$\begin{aligned}
w_{\mathrm{be}}(()) &= 1 \\
w_{\mathrm{be}}((e_1, ..., e_j)) &= w_{\mathrm{be}}(e_1) + \cdots + w_{\mathrm{be}}(e_j) + 1 \\
w_{\mathrm{be}}(\mathbf{proj}_i\, e) &= w_{\mathrm{be}}(e) + 1 \\
w_{\mathrm{be}}(x) &= 1 \\
w_{\mathrm{be}}(\lambda x{:}T.e) &= 1 \\
w_{\mathrm{be}}(e_1\, e_2) &= w_{\mathrm{be}}(e_1) + w_{\mathrm{be}}(e_2) + 1 \\
w_{\mathrm{be}}(\mathbf{mar}\,(e{:}T)) &= w_{\mathrm{be}}(e) + 1 \\
w_{\mathrm{be}}(\mathbf{marshalled}\,(e{:}T)) &= w_{\mathrm{be}}(e) + 1 \\
w_{\mathrm{be}}(\mathbf{unmar}\, e{:}T) &= w_{\mathrm{be}}(e) + 1 \\
w_{\mathrm{be}}(!\, e) &= w_{\mathrm{be}}(e) + 1 \\
w_{\mathrm{be}}(?) &= 1 \\
w_{\mathrm{be}}(U.\mathrm{term}) &= 1 \\
w_{\mathrm{be}}([e]^{T}_{hm}) &= 2\, w_{\mathrm{be}}(e) \\
w_{\mathrm{be}}(\mathbf{UnmarFailure}^{T}) &= 1
\end{aligned}$$

It is obvious that $w_{\mathrm{be}}(e)$ is always a positive integer.

We prove that if $e \underset{\mathrm{be}}{\longrightarrow}_{hm} e'$ then $w_{\mathrm{be}}(e) > w_{\mathrm{be}}(e')$. We induct on the derivation of the reduction.

Case (ered.col.unit): Here $e = [()]^{\mathrm{UNIT}}_{hm'}$ and $e' = ()$. We have $w_{\mathrm{be}}(e) = 2 > 1 = w_{\mathrm{be}}(e')$.

Case (ered.col.tuple): Here $e = [(v_1^{hm'}, ..., v_j^{hm'})]^{T_1 * ... * T_j}_{hm'}$ and $e' = ([v_1^{hm'}]^{T_1}_{hm'}, ..., [v_j^{hm'}]^{T_j}_{hm'})$. We have $w_{\mathrm{be}}(e) = 2(1 + \sum_i w_{\mathrm{be}}(v_i^{hm'})) > \sum_i 2 w_{\mathrm{be}}(v_i^{hm'}) = w_{\mathrm{be}}(e')$.

Case (ered.col.fun): Here $e = [\lambda x{:}T.e_0]^{T' \to T''}_{hm'}$ and $e' = \lambda x{:}T'.[\{x \leftarrow [x]^{T'}_{hm}\} e_0]^{T''}_{hm'}$. We have $w_{\mathrm{be}}(e) = 2 > 1 = w_{\mathrm{be}}(e')$

Case (ered.col.marred): Here $e = [\mathbf{marshalled}\,(e_0{:}T)]_{hm'}^{\text{STRING}}$ and $e' = \mathbf{marshalled}\,(e_0{:}T)$. We have $w_{\text{be}}(e) = 2(1 + w_{\text{be}}(e_0)) > 1 + w_{\text{be}}(e_0) = w_{\text{be}}(e')$.

Case (ered.col.col): Here $e = [[v^{h_0}]_{h_0}^{h_0}]_{h_1}^{h_0}$ and $e' = [v^{h_0}]_{v^{h_0}}^{v^{h_0}}$. We have $w_{\text{be}}(e) = 2w_{\text{be}}(e') > w_{\text{be}}(e')$.

Case (ered.col.le): Here $e = [v^{hm'}]_{hm'}^{h''}$ and $e' = v^{hm'}$. We have $w_{\text{be}}(e) = 2w_{\text{be}}(e') > w_{\text{be}}(e')$.

Case (ered.cong): Here $e = C_{hm_0}^{hm}\,e_0$ and $e' = C_{hm_0}^{hm}\,e_0'$ and $e_0 \underset{\text{be}}{\longrightarrow}{}_{hm_0}\,e_0'$. By induction, $w_{\text{be}}(e_0) > w_{\text{be}}(e_0')$. If $C_{hm_0}^{hm} = [\_]_{hm_0}^{T_0}$, then $w_{\text{be}}(e) = 2w_{\text{be}}(e_0) > 2w_{\text{be}}(e_0') = w_{\text{be}}(e')$. Otherwise, there exists an integer k such that $w_{\text{be}}(e) = k + w_{\text{be}}(e_0) > k + w_{\text{be}}(e_0') = w_{\text{be}}(e')$.

Since the weight of an expression is a positive integer that decreases at each step of reduction, bracket elimination is strongly normalising (bracket elimination of e terminates in at most $w_{\text{be}}(e)$ steps). ☐

Definition I.20 (bracket erasure) If $\aleph$ is any syntactic entity, $\text{erase\_brackets}(\aleph)$ is $\aleph$ with all brackets outside hashes erased. In particular, $\text{erase\_brackets}(h) = h$, and $\text{erase\_brackets}([e]_{hm}^{T}) = \text{erase\_brackets}(e)$.

Definition I.21 (colourless proof) A colourless proof is one that does not use (hmok.hash) nor (eT.col).

A colourless expression, type, etc., is one whose validity can be derived by a colourless proof. It is left as an exercise to the reader to prove that colourless types are those that contain no hashes and colourless expressions are those that contain no brackets and whose type annotations only use colourless types.

Definition I.22 (bracketless expressions) A bracketless expression is one that does not have any coloured brackets in it. Ditto for values, contexts, and networks.

Definition I.23 (bracketless reductions) Define $e \underset{\text{nb}}{\longrightarrow} e'$ on bracketless expressions as given by the (ered.*) rules other than (ered.col.*), with the right-hand sides modified to not introduce brackets, namely:

- (ered.ap) becomes $(\lambda x{:}T.e)v \underset{\text{nb}}{\longrightarrow} \{x{\leftarrow}v\}e$;
- (ered.mar) becomes $\mathbf{mar}\,(e{:}T) \underset{\text{nb}}{\longrightarrow} \mathbf{marshalled}\,(e{:}T)$.

Also define $n \underset{\text{nb}}{\longrightarrow} n'$ for networks in the obvious way.

Lemma I.24 (progress and determinism of bracketless expression reduction) If $\grave{e}$ is a bracketless expression then exactly one of the following cases holds:

- $\grave{e}$ is a bracketless value;
- $\grave{e}$ is dormant, i.e. is $\mathbf{UnmarFailure}^{T}$ or a communication in a bracketless evalution context;
- $\grave{e}$ reduces by $\underset{\text{nb}}{\longrightarrow}$; moreover there is exactly one rule with one redex applicable.

Proof. Use Theorem H.6 (progress of expressions) for the "at least one" part. Similar to the proof of Theorem H.11 (determinism of expression reduction) for the "at most one" part. ☐

Theorem I.25 (bracket erasure preserves expression reduction outcomes) If $\mathbf{nil} \vdash_{hm} e{:}T$ and $e \longrightarrow_{hm} e'$ then $\text{erase\_brackets}(e) \underset{\text{nb}}{\longrightarrow}{}^{?}\text{erase\_brackets}(e')$.

Proof. (Sketch.) Induct on the derivation of the reduction. If $e \longrightarrow_{hm} e'$ by one of the bracket pushing rules, then $\text{erase\_brackets}(e) = \text{erase\_brackets}(e')$ (check each case). If $e \longrightarrow_{hm} e'$ by another rule, then $\text{erase\_brackets}(e) \underset{\text{nb}}{\longrightarrow} \text{erase\_brackets}(e')$ (check each case, noticing that if v^{hm_1} is a value then $\text{erase\_brackets}(v^{hm_1})$ is a bracketless value). Finally, the case of (ered.cong) follows by induction. ☐

Theorem I.26 (bracket erasure preserves reduction outcomes) If $\vdash n\,\text{ok}$ and $n \longrightarrow n'$ then $\text{erase\_brackets}(n) \underset{\text{nb}}{\longrightarrow}{}^{?}\text{erase\_brackets}(n')$.

Proof. Follows from Theorem I.25 (bracket erasure preserves expression reduction outcomes). ☐

Theorem I.27 (bracket erasure does not add expression reduction outcomes) If $\mathbf{nil} \vdash_{hm} e{:}T$ and $\mathrm{erase\_brackets}(e) \underset{\mathrm{nb}}{\longrightarrow} \grave{e}$ then there exists e' such that $\mathrm{erase\_brackets}(e') = \grave{e}$ and $e \longrightarrow^{+}_{hm} e'$

Proof. By Theorem H.6 (progress of expressions) and Lemma I.19 (termination of bracket elimination), there exists a sequence of expressions $e = e_0, e_1, ..., e_j$ such that $e_{i-1} \underset{\mathrm{be}}{\longrightarrow}{}_{hm} e_i$ for $1 \leqslant i \leqslant j$ and e_j does not reduce by any bracket pushing rule. By definition, $\mathrm{erase\_brackets}(e) = \mathrm{erase\_brackets}(e_j)$. Thus by hypothesis, $\mathrm{erase\_brackets}(e_j) \underset{\mathrm{nb}}{\longrightarrow} \grave{e}$. By Theorem H.6 (progress of expressions) one of the following cases holds:

Case e_j is an hm-value: Then $\mathrm{erase\_brackets}(e_j)$ is a bracketless value, so by Lemma I.24 (progress and determinism of bracketless expression reduction) does not reduce by $\underset{\mathrm{nb}}{\longrightarrow}$, a contradiction.

Case e_j is dormant: Then $\mathrm{erase\_brackets}(e_j)$ is a bracketless dormant expression, so by Lemma I.24 (progress and determinism of bracketless expression reduction) does not reduce by $\underset{\mathrm{nb}}{\longrightarrow}$, a contradiction.

Case e_j reduces: Therefore there exists e' such that $e_j \longrightarrow_{hm} e'$, a non bracket pushing reduction. Hence $\mathrm{erase\_brackets}(e_j) \underset{\mathrm{nb}}{\longrightarrow} \mathrm{erase\_brackets}(e')$. By Lemma I.24 (progress and determinism of bracketless expression reduction), $\mathrm{erase\_brackets}(e') = \grave{e}$.

□

Theorem I.28 (bracket erasure does not add reduction outcomes) If $\vdash n$ ok and $\mathrm{erase\_brackets}(n) \underset{\mathrm{nb}}{\longrightarrow} n_0$ then there exists n' such that $\mathrm{erase\_brackets}(n') = n_0$ and $n \longrightarrow^{+} n'$

This means we can erase all brackets once all the modules have been reduced away. Clearly $\underset{\mathrm{nb}}{\longrightarrow}$ is not type preserving but it represents a possible implementation strategy.

Proof. Follows from Theorem I.27 (bracket erasure does not add expression reduction outcomes). □

We might hope that: If $\vdash n$ ok then $n \longrightarrow^{*} n'$ iff $\mathrm{erase\_brackets}(n) \underset{\mathrm{nb}}{\longrightarrow}{}^{*}\mathrm{erase\_brackets}(n')$.

This is not true! Counterexample: $\mathrm{erase\_brackets}(3) \underset{\mathrm{nb}}{\longrightarrow}{}^{=}\mathrm{erase\_brackets}([3]_h^{\mathrm{INT}})$ but not $3 \longrightarrow^{*}_{\bullet} [3]_h^{\mathrm{INT}}$

Definition I.29 (type erasure) We define $\mathrm{erase}(e)$ to be e with all type annotations and brackets erased except that the type annotations on **mar**, **marshalled**, and **unmar** are left unchanged. Ditto for networks. We define $\underset{\mathrm{erase}}{\longrightarrow}$ to be like $\longrightarrow_{hm}$ by taking the erase-image of the left- and right-hand sides of each rule (and removing rules that would become $e \underset{\mathrm{erase}}{\longrightarrow} e$). Ditto for networks.

Note that erased terms do not form a subsyntax of our usual syntax since e.g. $\mathrm{erase}(\lambda x{:}T.e) = (\lambda x.e)$. We will not make explicit any formal reasoning about erased terms so do not construct this new syntax explicitly.

Note that the only difference between $\mathrm{erase}(e)$ and $\mathrm{erase\_brackets}(e)$ is that the former removes type annotations from lambda-expressions. Moreover, $\underset{\mathrm{erase}}{\longrightarrow}$ is the same as $\underset{\mathrm{nb}}{\longrightarrow}$ except that the type annotations in the lambda-expressions are erased.

Theorem I.30 (type erasure) Assume $\vdash n$ ok. We have that $n \longrightarrow n'$ implies $\mathrm{erase}(n) \underset{\mathrm{erase}}{\longrightarrow}{}^{?}\mathrm{erase}(n')$. Conversely, $\mathrm{erase}(n) \underset{\mathrm{erase}}{\longrightarrow} n_0$ implies that there exists n' such that $\mathrm{erase}(n') = n_0$ and $n \longrightarrow^{+} n'$.

Proof. By Theorem I.26 (bracket erasure preserves reduction outcomes) and Theorem I.28 (bracket erasure does not add reduction outcomes), the theorem holds for erase_brackets in place of erase. Note that the type annotation T in $\lambda x{:}T.e$ never influences $\underset{\mathrm{nb}}{\longrightarrow}$ reduction, as desired. □

J Coincidence of undyn-time type checking and static type checking

We prove a theorem relating the undyn-tyime type checking in a single machine to static typechecking.

Definition J.1 (multiple-let context)

$CL ::=$		multiple-let context
	$\_$	identity
	$\mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ CL$	module declaration

Definition J.2 (correct multiple-let context) A multiple-let context CL is correct in E if the following (recursive) properties hold:

Case $CL = \_$**:** We require $E \vdash_\bullet$ ok.

Case $CL = \mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ CL'$**:** We require $E \vdash_\bullet M{:}S$ and CL' is correct in $E, U{:}S$.

We say CL is correct if CL is correct in $\mathbf{nil}$.

Lemma J.3 (correct multiple-let contexts yield ok environments) If CL is correct in E then $E, \mathrm{envofcl}\, CL \vdash_\bullet$ ok.

Proof. Induct on the length of CL.

Case $CL = \_$**:** By the definition of correctness, $E \vdash_\bullet$ ok, as desired.

Case $CL = \mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ CL'$**:** Then $\mathrm{envofcl}\, CL\ =\ U{:}S, \mathrm{envofcl}\, CL'$. By definition of correctness, $E, U{:}S \vdash_\bullet$ ok and CL' is correct in $E, U{:}S$. By induction, $E, U{:}S, \mathrm{envofcl}\, CL' \vdash_\bullet$ ok, i.e. $E, \mathrm{envofcl}\, CL \vdash_\bullet$ ok as desired.

□

Lemma J.4 (machine judgement characterisation of correct multiple-let contexts) CL is a correct multiple-let context iff $\mathbf{nil} \vdash_\bullet CL()$:UNIT.

Proof. We prove that CL is correct in E iff $E \vdash_\bullet CL()$:UNIT. We induct on the length of CL.

Case $CL = \_$**:** If CL is correct in E, then $E \vdash_\bullet$ ok whence by (eT.unit) $E \vdash_\bullet ()$:UNIT. Conversely, assume $E \vdash_\bullet ()$:UNIT: by Lemma E.6 (environments have to be ok), we get $E \vdash_\bullet$ ok, i.e. CL is correct in E.

Case $CL = \mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ CL'$**, forward direction:** We have that $E \vdash_\bullet M{:}S$ and CL' is correct in $E, U{:}S$. By induction, $E, U{:}S \vdash_\bullet CL'()$:UNIT. By (mT.let), we get $E \vdash_\bullet \mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ CL'()$:UNIT, as desired.

Case $CL = \mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ CL'$**, backward direction:** We have $E \vdash_\bullet CL()$:UNIT. By reversing (mT.let), we get $E \vdash_\bullet M{:}S$ and $E, U{:}S \vdash_\bullet CL'()$:UNIT. By induction, CL' is correct in $E, U{:}S$. By definition of correctness, CL is correct in E.

□

Lemma J.5 (peeling outer let from a multiple-let context preserves correctness) Let CL be a multiple-let context.

1. If $CL\ =\ \mathbf{module}\, N_U\ =\ [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \ \mathbf{in}\ \ CL'$ is correct then $\sigma CL'$ is correct, where $\sigma\ =\ \{U.\mathrm{TYPE}{\leftarrow}T'', U.\mathrm{term}{\leftarrow}v^\bullet\}$.
2. If $CL\ =\ \mathbf{module}\, N_U\ =\ [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']\ \ \mathbf{in}\ \ CL'$ is correct then $\sigma CL'$ is correct where $\sigma\ =\ \{U.\mathrm{TYPE}{\leftarrow}h, U.\mathrm{term}{\leftarrow}[v^\bullet]_h^{\{X\leftarrow h\}T'}\}$ and $h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}\mathbf{Type}, T'])$.

Proof. By Lemma J.4 (machine judgement characterisation of correct multiple-let contexts), the hypotheses are equivalent to $\mathbf{nil} \vdash_\bullet CL()$:UNIT and the conclusions are equivalent to $\mathbf{nil} \vdash_\bullet \sigma CL'()$:UNIT. By the definition of machine reduction $CL()\ \longrightarrow_\bullet\ \sigma CL'()$ in both cases. By Theorem G.18 (type preservation for machine reduction), we get $\mathbf{nil} \vdash_\bullet \sigma CL'()$:UNIT, as desired. □

Now we define a metafunction hashlistofcl which maps a multiple-let context to a list of the corresponding hashes:

Definition J.6 (hash list of a multiple-let context)

$$\text{hashlistofcl}\,\_ = ()$$
$$\text{hashlistofcl}\,(\mathbf{module}\,N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \mathbf{in}\ CL) = \text{hashlistofcl}\,(\{U.\textsc{type}\leftarrow T'', U.\text{term}\leftarrow v^\bullet\}CL)$$
$$\text{hashlistofcl}\,(\mathbf{module}\,N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']\ \mathbf{in}\ CL) = h, \text{hashlistofcl}\,(\sigma CL)$$

where $h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}\mathbf{Type}, T'])$ and $\sigma = \{U.\textsc{type}\leftarrow h, U.\text{term}\leftarrow [v^\bullet]_h^{\{X\leftarrow h\}T'}\}$

Definition J.7 (no shadowing in a multiple-let context) There is no shadowing in the multiple-let context CL iff hashlistofcl CL contains no repetitions.

Definition J.8 (environment corresponding to a multiple-let context)

$$\text{envofcl}\,\_ = \mathbf{nil}$$
$$\text{envofcl}\,(\mathbf{module}\,N_U\ = M{:}S\ \mathbf{in}\ CL) = U{:}S, \text{envofcl}\,CL$$

Definition J.9 (substitution corresponding to a multiple-let context)

$$\text{subofcl}\,\_ = \text{id}$$
$$\text{subofcl}\,(\mathbf{module}\,N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \mathbf{in}\ CL) = (\text{subofcl}\,\sigma CL)\sigma$$
$$\text{where } \sigma = \{U.\textsc{type}\leftarrow T'', U.\text{term}\leftarrow v^\bullet\}$$
$$\text{subofcl}\,(\mathbf{module}\,N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']\ \mathbf{in}\ CL) = (\text{subofcl}\,\sigma CL)\sigma$$
$$\text{where } \sigma = \{U.\textsc{type}\leftarrow h, U.\text{term}\leftarrow [v^\bullet]_h^{\{X\leftarrow h\}T'}\}$$
$$\text{and } h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}\mathbf{Type}, T'])$$

Lemma J.10 (substituting through the partial type substitution associated to an environment) Let μ be any substitution of the form $\{\chi_1\leftarrow\eta_1, ..., \chi_j\leftarrow\eta_j\}$ where $(D\cup R)\cap\text{dom}\,E = \varnothing$ where $D = \text{fv}\,\chi_1\cup...\cup\text{fv}\,\chi_j$ and $R = \text{fv}\,\eta_1\cup...\cup\text{fv}\,\eta_j$. Then $\mu\text{partenvsub}_E = \text{partenvsub}_{\mu E}\mu$.

Proof. Induct on the length of E. The only non-trivial cases are as follows:

Case $E = E', X{:}\mathbf{Eq}(T)$**:** Then $\mu\text{partenvsub}_E = \mu\text{partenvsub}_{E'}\{X\leftarrow T\}$. Since $\text{dom}\,E' \subseteq \text{dom}\,E$, we can apply induction to get that the last composition is equal to $\text{partenvsub}_{\mu E'}\mu\{X\leftarrow T\}$. Since $X \notin D \cup R$, we can further rewrite the composition as $\text{partenvsub}_{\mu E'}\{X\leftarrow\mu T\}\mu = \text{partenvsub}_{\mu(E', X:\mathbf{Eq}(T))}\mu$, as desired.

Case $E = E', U{:}[X{:}\mathbf{Eq}(T), T']$**:** similar to the previous.

□

Note that in the previous result we refer to fv χ_0, not just χ_0, since χ_0 could be of the form $U.\textsc{type}$ or $U.\text{term}$.

Lemma J.11 (partial type substitution associated to an environment via left folding) Let partenvsub_E be as in Definition I.2 (partial type substitution associated to an environment). Then

$$\text{partenvsub}_{\mathbf{nil}} = \text{id}$$
$$\text{partenvsub}_{x:T,E} = \text{partenvsub}_E$$
$$\text{partenvsub}_{X:\mathbf{Type},E} = \text{partenvsub}_E$$
$$\text{partenvsub}_{X:\mathbf{Eq}(T),E} = \{X\leftarrow T\}\text{partenvsub}_E$$
$$\text{partenvsub}_{U:[X:\mathbf{Type},T'],E} = \text{partenvsub}_E$$
$$\text{partenvsub}_{U:[X:\mathbf{Eq}(T),T'],E} = \{U.\textsc{type}\leftarrow T\}\text{partenvsub}_E$$

Proof. Follows from the associativity of substitution composition. □

Definition J.12 (partially ok environment) An environment E is partially ok if it has no repetition in the domain and all decompositions $E = E_0, E_1$ have the property ufv $E_0 \cap \text{dom}\,E_1 = \varnothing$.

Lemma J.13 (ok environments are partially ok) If $E \vdash_{hm}$ ok then E is partially ok.

Proof. If $E \vdash_{hm}$ ok then E has no repetition in the domain by Lemma E.8 (ok environments have no repetition in the domain). By Lemma E.7 (prefixes of ok environments are ok) and Lemma F.13 (ok environments have no unresolved free variables), ufv $E_0 = \varnothing$, whence ufv $E_0 \cap \operatorname{dom} E_1 = \varnothing$. □

Lemma J.14 (partial type substitution associated to an environment, alternative characterisation) Let E be a partially ok environment. Let $\mathrm{partenvsub}_E$ be as in Definition I.2 (partial type substitution associated to an environment). Let

$$\begin{aligned}
\sigma'_{\mathbf{nil}} &= \mathrm{id} \\
\sigma'_{x:T,E} &= \sigma'_E \\
\sigma'_{X:\mathbf{Type},E} &= \sigma'_E \\
\sigma'_{X:\mathbf{Eq}(T),E} &= \sigma'_{\{X\leftarrow T\}E}\{X\leftarrow T\} \\
\sigma'_{U:[X:\mathbf{Type},T'],E} &= \sigma'_E \\
\sigma'_{U:[X:\mathbf{Eq}(T),T'],E} &= \sigma'_{\{U.\textsc{type}\leftarrow T\}E}\{U.\textsc{type}\leftarrow T\}
\end{aligned}$$

Then $\mathrm{partenvsub}_E = \sigma'_E$.

Proof. Induct on the length of E. By Lemma J.11 (partial type substitution associated to an environment via left folding) most cases are trivial, except for the following:

Case $E = X{:}\mathbf{Eq}(T), E'$**:** By Lemma J.11 (partial type substitution associated to an environment via left folding), $\mathrm{partenvsub}_E = \{X\leftarrow T\}\mathrm{partenvsub}_{E'}$. Since E is partially ok, $(\{X\} \cup \mathrm{fv}\ T) \cap \operatorname{dom} E' = \varnothing$. Thus by Lemma J.10 (substituting through the partial type substitution associated to an environment), we have that the composition is equal to $\mathrm{partenvsub}_{\{X\leftarrow T\}E'}\{X\leftarrow T\}$. Since $\{X\leftarrow T\}E'$ is also partially ok, by induction the composition is equal to $\sigma'_{\{X\leftarrow T\}E'}\{X\leftarrow T\} = \sigma'_{X:\mathbf{Eq}(T),E'}$, as desired.

Case $E = E', U{:}[X{:}\mathbf{Eq}(T), T']$**:** similar.

□

Lemma J.15 (correspondence between open and hashed interpretation of type equivalence) Let CL be a multiple-let context. Suppose

1. CL has no shadowing.
2. CL is correct.
3. None of the hashes in CL is in $\mathrm{hashlistofcl}\ CL$.
4. For all $i = 0, 1$ we have that none of the hashes in T_i is in $\mathrm{hashlistofcl}\ CL$.
5. For all $i = 0, 1$ we have that $\mathrm{envofcl}\ CL \vdash_{\bullet} T_i{:}\mathbf{Type}$.

Let $\mathrm{partenvsub}_E$ be the substitution from Definition I.2 (partial type substitution associated to an environment). Then $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL} T_0 = \mathrm{partenvsub}_{\mathrm{envofcl}\ CL} T_1$ iff $\mathrm{subofcl}\ CL\ T_0 = \mathrm{subofcl}\ CL\ T_1$.

Proof. We freely use Lemma J.14 (partial type substitution associated to an environment, alternative characterisation) with Lemma J.13 (ok environments are partially ok) to reason about $\mathrm{partenvsub}$.

We will prove some useful properties, then consider each of the mutual implications.

Prelude **Prelude of the prelude** Suppose that $CL = \mathbf{module}\ N_U = M{:}S\ \mathbf{in}\ CL'$. Write $M{:}S = [T, v^\bullet]{:}[X{:}K, T']$. Let $\sigma = \{U.\textsc{type}\leftarrow \mathrm{typepart}\,(N, M{:}S), U.\mathrm{term}\leftarrow \mathrm{termpart}\,(N, M{:}S)\}$ and $CL'' = \sigma\, CL'$.

By hypothesis 2 and Definition J.2 (correct multiple-let context), $\mathbf{nil} \vdash_{\bullet} M{:}S$. By Lemma E.9 (free variables of a judgement come from the environment), $U \notin \mathrm{fv}\ M \cup \mathrm{fv}\ S$. Whether S is concrete or abstract, this implies that $U \notin \mathrm{fv}\,\mathrm{typepart}\,(N, M{:}S)$ and $U \notin \mathrm{fv}\,\mathrm{termpart}\,(N, U{:}S)$. Hence

$$\begin{aligned}
&\{U.\mathrm{term}\leftarrow \mathrm{termpart}\,(N, M{:}S)\}\{U.\textsc{type}\leftarrow \mathrm{typepart}\,(N, M{:}S)\} \\
&\quad = \{U.\textsc{type}\leftarrow \mathrm{typepart}\,(N, M{:}S), U.\mathrm{term}\leftarrow \mathrm{termpart}\,(N, M{:}S)\} \\
&\quad = \{U.\textsc{type}\leftarrow \mathrm{typepart}\,(N, M{:}S)\}\{U.\mathrm{term}\leftarrow \mathrm{termpart}\,(N, M{:}S)\}.
\end{aligned}$$

We will use this fact freely throughout the proof.

By Lemma E.21 (types do not contain free expression variables), $\{U.\mathrm{term}\leftarrow\mathrm{termpart}\,(N, M{:}S)\}\,T_i = T_i$, whence $\sigma\,T_i = \{U.\textsc{type}\leftarrow\mathrm{typepart}\,(N, M{:}S)\}\,T_i$.

By Lemma E.23 (expression substitution in environment) applied to $\mathrm{envofcl}\,CL = (U{:}S, \mathrm{envofcl}\,CL')$ (which is correct by hypothesis 2 and Lemma J.3 (correct multiple-let contexts yield ok environments)), $\{U.\mathrm{term}\leftarrow\mathrm{termpart}\,(N, M{:}S)\}\mathrm{envofcl}\,CL' = \mathrm{envofcl}\,CL'$, whence $\mathrm{envofcl}\,CL'' = \sigma\mathrm{envofcl}\,CL' = \{U.\textsc{type}\leftarrow\mathrm{typepart}\,(N, M{:}S)\}\mathrm{envofcl}\,CL'$.

We now show that the numbered hypotheses hold for CL'' in place of CL and $\sigma\,T_i$ in place of T_i.

Prelude: concrete case: $K = \mathbf{Eq}(T'')$ 1. By definition, $\mathrm{hashlistofcl}\,CL'' = \mathrm{hashlistofcl}\,CL$, therefore, CL'' contains no shadowing since CL contains no shadowing (hypothesis 1).

2. Since CL is correct by hypothesis 2, by Lemma J.5 (peeling outer let from a multiple-let context preserves correctness), CL'' is correct.
3. Each hash in CL'' in an element of the union of the hashes in T'' and $v^\bullet$ and CL', which is disjoint from $\mathrm{hashlistofcl}\,CL = \mathrm{hashlistofcl}\,CL''$ by hypothesis 3.
4. Each hash in $\sigma\,T_i = \{U.\textsc{type}\leftarrow T''\}\,T_i$ is an element of the union of the hashes in T'' and T_i, which is disjoint from $\mathrm{hashlistofcl}\,CL = \mathrm{hashlistofcl}\,CL''$ by hypotheses 3 (for T'') and 4.
5. By hypothesis 5, for $i = 0, 1$ we have $U{:}[X{:}\mathbf{Eq}(T''), T'], \mathrm{envofcl}\,CL' \vdash_\bullet T_i{:}\mathbf{Type}$. By Lemma F.32 (simplified module and type equality substitution for type world judgements), $\{U.\textsc{type}\leftarrow T''\}(\mathrm{envofcl}\,CL') \vdash_\bullet \{U.\textsc{type}\leftarrow T''\}\,T_i{:}\mathbf{Type}$, i.e. $\mathrm{envofcl}\,CL'' \vdash_\bullet \sigma\,T_i{:}\mathbf{Type}$.

Prelude: abstract case: $K = \mathbf{Type}$ Let $h = \mathbf{hash}(N, M{:}[X{:}\mathbf{Type}, T'])$.

1. By definition, $(h, \mathrm{hashlistofcl}\,CL'') = \mathrm{hashlistofcl}\,CL$, therefore, CL'' contains no shadowing since CL contains no shadowing (hypothesis 1).
(†) Also, since CL contains no shadowing, h is not in $\mathrm{hashlistofcl}\,CL''$ (used below).
2. Since CL is correct by hypothesis 2, by Lemma J.5 (peeling outer let from a multiple-let context preserves correctness), CL'' is correct.
3. Each hash in CL'' in an element of the union of $\{h\}$ and the hashes in T'' and $v^\bullet$ and CL'. By (†), h does not appear in $\mathrm{hashlistofcl}\,CL''$. By hypothesis 3, the hashes in T'' and $v^\bullet$ and CL' do not appear in $\mathrm{hashlistofcl}\,CL$, hence do not appear in $\mathrm{hashlistofcl}\,CL''$.
4. Each hash in $\sigma\,T_i$ is an element of the union of $\{h\}$ and the hashes in T'' and $v^\bullet$ and T_i, which is disjoint from $\mathrm{hashlistofcl}\,CL''$, by (†) (for h) and hypothesis 3 (for T'' and $v^\bullet$) and hypothesis 4 (for T_i).
5. By hypothesis 5, for $i = 0, 1$ we have $\mathrm{envofcl}\,CL \vdash_\bullet T_i{:}\mathbf{Type}$, i.e. $U{:}[X{:}\mathbf{Type}, T'], \mathrm{envofcl}\,CL' \vdash_\bullet T_i{:}\mathbf{Type}$. Given $\mathbf{nil} \vdash_\bullet [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']$, we can apply Lemma F.33 (type preservation by fully carried out module substitution) to get $\sigma\mathrm{envofcl}\,CL' \vdash_\bullet \sigma\,T_i{:}\mathbf{Type}$, i.e. $\mathrm{envofcl}\,CL'' \vdash_\bullet \sigma\,T_i{:}\mathbf{Type}$.

We also prove that $\mathrm{partenvsub}_{\mathrm{envofcl}\,CL''}\sigma = \sigma\mathrm{partenvsub}_{\mathrm{envofcl}\,CL'}$.

Let $D = \{U\}$ be the variables in the domain of σ. Let $R = \mathrm{fv}\,h \cup \mathrm{fv}\,([v^\bullet]_h^{\{X\leftarrow h\}\,T'})$ be the free variables in the range of σ. By standard reasoning about free variables and substitutions, $R = \mathrm{fv}\,h \cup \mathrm{fv}\,v^\bullet \cup (\mathrm{fv}\,T' \setminus \{X\})$.

By hypothesis 2 and Definition J.2 (correct multiple-let context), $\mathbf{nil} \vdash_\bullet [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']$. By Lemma E.9 (free variables of a judgement come from the environment) and the definition of free variables, $\mathrm{fv}\,v^\bullet = \varnothing$ and $\mathrm{fv}\,T' \subseteq \{X\}$. Also, given $\mathbf{nil} \vdash_\bullet [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']$, we have $\vdash h\,\mathrm{ok}$ by (hmok.hash), whence by Lemma E.9 (free variables of a judgement come from the environment) $\mathrm{fv}\,h = \varnothing$.

By Lemma J.3 (correct multiple-let contexts yield ok environments), $\mathrm{envofcl}\,CL \vdash_\bullet \mathrm{ok}$. By Lemma E.8 (ok environments have no repetition in the domain), $U \notin \mathrm{dom}\,CL'$.

In summary, $(D \cup R) \cap \mathrm{dom}\,(\mathrm{envofcl}\,CL') = \varnothing$. By Lemma J.10 (substituting through the partial type substitution associated to an environment), $\mathrm{partenvsub}_{\mathrm{envofcl}\,CL''}\sigma = \mathrm{partenvsub}_{\sigma\mathrm{envofcl}\,CL'}\sigma = \sigma\mathrm{partenvsub}_{\mathrm{envofcl}\,CL'}$.

Direction $\mathrm{partenvsub}_{\mathrm{envofcl}\,CL}\,T_0 = \mathrm{partenvsub}_{\mathrm{envofcl}\,CL}\,T_1$ **implies** $\mathrm{subofcl}\,CL\,T_0 = \mathrm{subofcl}\,CL\,T_1$**:** Induct on the length of CL.

Case $CL = \_$**:** By hypothesis, $T_0 = T_1$, as desired.

Concrete case: $CL = (\mathbf{module}\, N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \mathbf{in}\ CL')$**:** We can write the hypothesis as $\mathrm{partenvsub}_{\{U.\text{TYPE}\leftarrow T''\}(\mathrm{envofcl}\ CL')}\{U.\text{TYPE}\leftarrow T''\}\,T_i$ is constant w.r.t. $i \in \{0,1\}$). By the prelude to the prelude, $\mathrm{envofcl}\, CL'' = \{U.\text{TYPE}\leftarrow T''\}(\mathrm{envofcl}\, CL')$ and $\sigma\, T_i = \{U.\text{TYPE}\leftarrow T''\}\,T_i$, so $\mathrm{partenvsub}_{(\mathrm{envofcl}\ CL'')}\sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$).

By the prelude (concrete case), we can apply induction. We get that $\mathrm{subofcl}\ CL''\ \sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$). By Definition J.9 (substitution corresponding to a multiple-let context), we have $\mathrm{subofcl}\ CL\ T_i$ is constant w.r.t. $i \in \{0,1\}$) as desired.

Abstract case: $CL = (\mathbf{module}\, N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']\ \mathbf{in}\ CL')$**:** We can write the hypothesis as $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL'}\,T_i$ is constant w.r.t. $i \in \{0,1\}$). Thus, $\sigma\mathrm{partenvsub}_{\mathrm{envofcl}\ CL'}\,T_i$ is constant w.r.t. $i \in \{0,1\}$), where $\sigma = \{U.\text{TYPE}\leftarrow h, U.\mathrm{term}\leftarrow [v^\bullet]_h^{\{X\leftarrow h\}T'}\}$ and $h = \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}\mathbf{Type}, T'])$.

Let $CL'' = \sigma\, CL'$. By the prelude (abstract case), $\sigma\mathrm{partenvsub}_{\mathrm{envofcl}\ CL'} = \mathrm{partenvsub}_{\mathrm{envofcl}\ CL''}\sigma$. Therefore $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL''}\sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$).

Given the prelude (abstract case), we can apply induction. We get that $\mathrm{subofcl}\,(\mathrm{envofcl}\,(CL''))\,\sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$). Hence by definition, $\mathrm{subofcl}\,(\mathrm{envofcl}\ CL)\ T_i$ is constant w.r.t. $i \in \{0,1\}$), as desired.

Direction $\mathrm{subofcl}\ CL\ T_0 = \mathrm{subofcl}\ CL\ T_1$ **implies** $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL}\,T_0 = \mathrm{partenvsub}_{\mathrm{envofcl}\ CL}\,T_1$**:** Induct on the length of CL.

Case $CL = \_$**:** By hypothesis, $T_0 = T_1$, as desired.

Concrete case: $CL = (\mathbf{module}\, N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \mathbf{in}\ CL')$**:** Recall that $CL'' = \sigma\, CL'$ and $\sigma = \{U.\text{TYPE}\leftarrow T'', U.\mathrm{term}\leftarrow v^\bullet\}$. We have $\mathrm{subofcl}\, CL = \mathrm{subofcl}\, CL''\ \sigma$. Our hypothesis is therefore that $\mathrm{subofcl}\ CL''\ \sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$).

By the prelude (concrete case), we can apply induction. We get $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL''}\sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$). By the prelude to the prelude, $\mathrm{partenvsub}_{\{U.\text{TYPE}\leftarrow T''\}\mathrm{envofcl}\ CL'}\{U.\text{TYPE}\leftarrow T''\}\,T_i$ is constant w.r.t. $i \in \{0,1\}$).

By Lemma J.14 (partial type substitution associated to an environment, alternative characterisation), $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL} = \mathrm{partenvsub}_{\{U.\text{TYPE}\leftarrow T''\}\mathrm{envofcl}\ CL'}\{U.\text{TYPE}\leftarrow T''\}$.
Thus $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL}\,T_i$ is constant w.r.t. $i \in \{0,1\}$) as desired.

Abstract case: $CL = (\mathbf{module}\, N_U\ = [T, v^\bullet]{:}[X{:}\mathbf{Type}, T']\ \mathbf{in}\ CL')$**:** By hypothesis, $\mathrm{subofcl}\ CL''\ \sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$). By the prelude (abstract case), we can apply induction, getting $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL''}\sigma\, T_i$ is constant w.r.t. $i \in \{0,1\}$).

By the prelude to the prelude, $\mathrm{partenvsub}_{\{U.\text{TYPE}\leftarrow T''\}\mathrm{envofcl}\ CL'}\{U.\text{TYPE}\leftarrow T''\}\,T_i$ is constant w.r.t. $i \in \{0,1\}$).

By Lemma J.14 (partial type substitution associated to an environment, alternative characterisation), $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL} = \mathrm{partenvsub}_{\{U.\text{TYPE}\leftarrow T''\}\mathrm{envofcl}\ CL'}\{U.\text{TYPE}\leftarrow T''\}$.
Thus $\mathrm{partenvsub}_{\mathrm{envofcl}\ CL}\,T_i$ is constant w.r.t. $i \in \{0,1\}$) as desired.

□

Theorem J.16 (coincidence between undyn-time and static type checking) Let CL be a multiple-let context. Suppose

- CL has no shadowing.
- CL is correct.
- None of the hashes in CL is in $\mathrm{hashlistofcl}\ CL$.
- For all $i = 0, 1$ we have $\mathrm{envofcl}\ CL \vdash_\bullet T_i{:}\mathbf{Type}$ and that none of the hashes in T_i is in $\mathrm{hashlistofcl}\ CL$.

Then $\mathrm{envofcl}\ CL \vdash_\bullet T_0 == T_1$ iff $\mathrm{subofcl}\ CL\ T_0 = \mathrm{subofcl}\ CL\ T_1$.

In other words, dynamic type checking corresponds exactly to static type checking when two conditions hold: no two module have the same hash (i.e. are of identical structure after their ancestors have been substituted in); the hashes in the modules and in the types being compared are different from the hashes that will be generated by reduction (such as is the case with a user program that contains no hashes at all).

Proof. Immediate from Lemma J.15 (correspondence between open and hashed interpretation of type equivalence) and Lemma I.6 (open interpretation of type equivalence). $\square$

Definition J.17 (list of external names in a multiple-let context) Define the list of external names in a multiple-let context by structural induction:

$$\begin{aligned} \mathrm{enlistofcl}\,\_ &= () \\ \mathrm{enlistofcl}\,(\mathbf{module}\,N_U \;= M{:}S\ \mathbf{in}\ CL) &= (N, \mathrm{enlistofcl}\,CL) \end{aligned}$$

Definition J.18 (external name shadowing) Let CL be a multiple-let context. We say that CL has external name shadowing iff $\mathrm{enlistofcl}\,CL$ contains duplicates.

Lemma J.19 (non-shadowing through external names) If CL has no external name shadowing then CL has no shadowing.

Proof. Trivial by induction on the length of CL. $\square$

Lemma J.20 (generalised machine judgement characterisation of correct multiple-let contexts) If $\mathbf{nil} \vdash_0 CL.e{:}\mathrm{UNIT}$ then CL is correct.

Proof. By reversing (mT.let) once for each binding in CL, substituting $\mathrm{envofcl}\,CL \vdash_0 (){:}\mathrm{UNIT}$ for $\mathrm{envofcl}\,CL \vdash_0 e{:}\mathrm{UNIT}$ and reapplying (mT.let)'s again, we get $\mathbf{nil} \vdash_0 CL.(){:}\mathrm{UNIT}$. By Lemma J.4 (machine judgement characterisation of correct multiple-let contexts), CL is correct. $\square$

Corollary J.21 (coincidence between undyn-time and static type checking with non-repeated external names) Suppose that $\mathbf{nil} \vdash_0 CL.e{:}\mathrm{UNIT}$, that CL contains no hashes, and that CL has no external name shadowing. Assume that T_0 and T_1 contain no hashes and $\mathrm{envofcl}\,CL \vdash_\bullet T_i{:}\mathbf{Type}$ for $i = 0, 1$. Then $\mathrm{envofcl}\,CL \vdash_\bullet T_0 == T_1$ iff $\mathrm{subofcl}\,CL\;T_0 = \mathrm{subofcl}\,CL\;T_1$.

Proof. By Lemma J.19 (non-shadowing through external names), CL has no shadowing. By Lemma J.20 (generalised machine judgement characterisation of correct multiple-let contexts), CL is correct. Thus we can apply Theorem J.16 (coincidence between undyn-time and static type checking). $\square$

K Table of theorems and definitions

References

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM TOPLAS*, 13(2):237–268, 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *J. Functional Programming*, 5(1):111–130, 1995.

[Ali03] The Alice Project. Alice manual: Pickling. `http://www.ps.uni-sb.de/alice/manual/pickling.html`, 2003.

[B+94] Andrei Z. Broder et al. Fingerprint.i3. `http://research.compaq.com/SRC/m3sources/html/fingerprint/src/Fingerprint.i3.html`, 1994.

[BHS+03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proc. ICFP 2003*, 2003. Full version available as UCAM-CL-TR-568. `http://www.cl.cam.ac.uk/~pes20/`.

[BNOW95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software – Practice & Experience*, 25(S4):87–130, 1995. Available in slightly different form as SRC-115 revised.

[CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, DEC SRC, March 10 1990.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proc. 30th POPL, New Orleans*, pages 236–249, 2003.

[Dug02] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *ACM TOPLAS*, 24(6):711–804, 2002.

[FW00] Jun Furuse and Pierre Weis. Entrées/sorties de valeurs en Caml. In *J. Francophones des Langages Applicatifs*, 2000.

[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.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd POPL*, pages 130–141, 1995.

[HWC00] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In *Proc. 3rd Workshop on Types in Compilation*, pages 147–176, 2000.

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

[Kna95] F. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, December 1995.

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

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd POPL*, pages 142–153, 1995.

[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`.

[Mac84] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symp. LISP and Func. Prog.*, pages 198–207, 1984.

[Mic01] Microsoft Corporation. .NET Framework developer's guide: Serializing objects. `http://msdn.microsoft.com/library/en-us/cpguide/html/cpovrserializingobjects.asp`, 2001.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, July 1988.

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

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

[PS00] Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism. `http://web.yl.is.s.u-tokyo.ac.jp/~sumii/pub/`, July 16 2000. Substantially revised version to appear in J. Comp. Security.

[Rob96] M. J. B. Robshaw. On recent results for MD2, MD4 and MD5. *RSA Laboratories' Bulletin*, (4), November 12 1996.

[Ros02] Andreas Rossberg. Dynamic opacity for abstract types. Technical report, Programming Systems Lab, Universität des Saarlandes, 2002. `http://www.ps.uni-sb.de/Papers/abstracts/opaque.html`.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, pages 236–247, 2001.

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

[Sun02] Sun Microsystems. Java object serialization specification 1.4.4. `http://java.sun.com/j2se/1.4.1/docs/guide/serialization/`, 2002.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, pages 278–298, 1996.

[Wei00] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proc. ICFP, Montreal*, pages 58–67, 2000.

[Wei02] Stephanie Weirich. Higher-order intensional type analysis. In *Proc. 11th ESOP, LNCS 2305, Grenoble, France*, 2002.

[ZGM99] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proc. ICFP, Paris*, pages 197–207, Sep 1999.

main body: `$Revision: 1.214 $ $Date: 2004/02/19 13:39:11 $`
appendices: `$Revision: 1.238 $ $Date: 2004/02/19 13:39:11 $`

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399