Communauté
UNIVERSITÉ Grenoble Alpes

Thèse

pour obtenir le grade de

Docteur de l'Université Grenoble Alpes

spécialité : informatique
arrêté ministériel : 25 mai 2016

présentée par
Rodolphe Lepigre

dirigée par Christophe Raffalli
et codirigée par Pierre Hyvernat

préparée au sein du LAMA, Université Savoie Mont Blanc
et de l'école doctorale MSTII

Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes

Thèse soutenue publiquement le 18 juillet 2017
devant un jury composé de

Laurent Regnier, président du jury
Aix-Marseille Université

Thierry Coquand, rapporteur
Göteborgs Universitet, Suède

Alexandre Miquel, rapporteur
Universidad de la República, Uruguay

Andreas Abel, examinateur
Göteborgs Universitet, Suède

Frédéric Blanqui, examinateur
Inria (Deducteam)

Karim Nour, directeur de thèse (HDR)
Université Savoie Mont Blanc

Christophe Raffalli, directeur de thèse
Université Savoie Mont Blanc

Pierre Hyvernat, codirecteur de thèse
Université Savoie Mont Blanc

Semantics and Implementation of an Extension of ML for Proving Programs

Rodolphe Lepigre

Table of contents

Remerciements

Je tiens en premier lieu à remercier mes deux directeurs de thèse, Christophe Raffalli et Pierre Hyvernat, sans qui ce travail n'aurait jamais vu le jour. En particulier, merci à Christophe pour son grand enthousiasme et ses innombrables idées (bonnes, mauvaises, ou pour devenir riche), mais également pour nos trop nombreuses pauses café. Merci à Pierre pour ses relectures attentives, son soutien sans faille, ainsi que pour son immense culture des énigmes, casse-têtes et travaux de John Horton Conway (dont je n'ai malheureusement pas terminé la lecture...). Merci également à Karim Nour pour avoir accepté de nous prêter son habilitation, en attendant celle de Christophe qui est « presque terminée » depuis bien trop longtemps. Je souhaite également adresser mes plus sincères remerciements à Thierry Coquand et à Alexandre Miquel, pour m'avoir fait l'honneur de rapporter cette thèse, ainsi que pour leurs nombreuses remarques et suggestions. Merci également à Andreas Abel, à Frédéric Blanqui et à Laurent Regnier d'avoir accepté de compléter mon jury. En particulier, merci à Andreas et à Frédéric pour leurs nombreuses notes et corrections, et merci à Laurent d'avoir été un président du jury exemplaire.

Depuis le début de ma thèse, le LAMA m'a offert un environnement de travail dans lequel je me suis pleinement épanoui, aussi bien sur le plan scientifique que sur le plan personnel. Je suis donc infiniment reconnaissant envers tous les membres du laboratoire avec qui j'ai pu interagir durant mon court séjour. En particulier, j'ai eu le plaisir de travailler et de partager mes pauses déjeuner avec les membres de l'équipe LIMD, qui interviennent au département d'informatique (incluant bien entendu Christophe et Pierre). J'ai trouvé en ces personnes bien plus que de simples collègues, et je suis vraiment peiné que nos chemins doivent se séparer bientôt. Merci donc à Tom Hirschowitz pour son aide avec Coq, pour sa contribution au pavage de salles de bains en école primaire, mais aussi pour m'avoir fait découvrir son modèle de virilité. Merci à Jacques-Olivier Lachaud pour ses efforts de dérision envers mon travail (et surtout l'égalité à droite), pour nos nombreuses discussions sur la connerie humaine, et également pour son humour tranchant (en particulier lors des soutenances de projet des étudiants). Merci à Xavier Provençal pour son aide avec les enseignements, pour m'avoir encouragé à perdre mon temps à cliquer sur des cookies, mais aussi pour sa solidarité avec mon penchant pour les burgers et la nourriture forte en fromage (dont l'intersection est bien entendu non vide). Merci à Clovis Eberhart pour sa complicité dans la lecture des travaux douteux d'un certain Professeur des Universités CNU, pour ses tentatives d'explications de concepts catégoriques, mais également pour de nombreux fous rires provoqués par les perles de nos étudiants. Merci à Pierre Villemot pour son intrusion dans notre équipe, pour sa grande culture mathématique, et pour son enthousiasme vis-à-vis de la représentation des constructibles en Caml (qu'il faudra bien entreprendre un jour).

Je voudrais aussi remercier toutes les thésardes et tous les thésards avec qui j'ai eu l'occasion de partager un fond de couloir. En particulier, merci à Marion Foare qui m'a suivi (ou que j'ai suivie), depuis la licence de mathématiques jusqu'aux galères administratives de la thèse, en passant par les maths à modeler. Merci aussi à Lama Tarsissi pour nous avoir fait découvrir la cuisine libanaise, et pour ses très bonnes relations avec l'école doctorale. Merci également à Boulos El Hilany, Bilal Al Taki, Michel Raibaut et à tous les autres. J'en profite pour mentionner également quelques thésards de la génération précédente. Merci à Florian Hatat pour ses précieux conseils (notamment d'ordre administratifs), pour le champagne, ainsi que pour son aide avec les problèmes techniques de dernière minute. Merci à Pierre-Étienne Meunier de m'avoir permis de programmer (ma thèse en) Patoline, ce qui à pu me faire perdre un peu de temps, occasionnellement. Merci également à Thomas Seiller pour ses encouragements dans des moments de doute.

Enfin, je voudrais remercier mes parents et mes grand parents, qui ont toujours été là pour moi (et en tout cas jamais bien loin). Merci aussi à mon petit frère Charles, à mes cousins bourguignons Lucie et Clément ainsi qu'à leurs parents, à mes belles sœurs Elin et Mirain et à mes beaux parents, à feu Minette, aux abeilles et au reste de ma famille (en France comme au Pays de Galles). Pour finir, un grand merci à Branwen, qui accomplit l'exploit de me supporter au quotidien.

1 From Programming to Program Proving

[In 1949,] as soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realised that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Maurice Wilkes (1913-2010)

Since the apparition of the very first computers, every generation of programmers has been faced with the issue of code reliability. Statically typed languages such as Java, Haskell, OCaml, Rust or Scala have addressed the problem by running syntactic checks at compile time to detect incorrect programs. Their strongly typed discipline is especially useful when several incompatible data objects have to be manipulated together. For example, a program computing an integer addition on a boolean (or function) argument is immediately rejected. In recent years, the benefit of static typing has even begun to be recognised in the dynamically typed languages community. Static type checkers are now available for languages like Javascript [*Microsoft 2012*, *Facebook 2014*] or Python [*Lehtosalo 2014*].

In the last thirty years, significant progress has been made in the application of type theory to computer languages. The Curry-Howard correspondence, which links the type systems of functional programming languages to mathematical logic, has been explored in two main directions. On the one hand, proof assistants like Coq or Agda are based on very expressive logics. To prove their consistency, the underlying programming languages need to be restricted to contain only programs that can be proved terminating. As a result, they forbid the most general forms of recursion. On the other hand, functional programming languages like OCaml or Haskell are well-suited for programming, as they impose no restriction on recursion. However, they are based on inconsistent logics, which means that they cannot be used for proving mathematical formulas.

The aim of this work is to provide a uniform environment in which programs can be designed, specified and proved. The idea is to combine a full-fledged ML-like programming language with an enriched type system allowing the specification of computational behaviours. This language can thus be used as ML for type-safe general programming, and as a proof assistant for proving properties of ML programs. The uniformity of the framework implies that programs can be incrementally refined to obtain more guarantees. In particular, there is no syntactic distinction between programs and proofs in the system. This means that programming and proving features can be mixed when constructing proofs or programs. For instance, proofs can be composed with programs for them to transport properties (e.g., addition carrying its commutativity proof). In programs, proof mechanisms can be used to eliminate dead code (i.e., portions of a program that cannot be reached during its execution).

1.1 Writing functional programs

In this thesis, our first goal is to design a type system for a practical, functional programming language. Out of the many possible technical choices, we decided to consider a call-by-value language similar to OCaml or SML, as they have proved to be highly practical and efficient. Our language provides polymorphic variants [*Garrigue 1998*] and SML style records, which are convenient for encoding data types. As an example, the type of lists can be defined and used as follows.

```
type rec list⟨a⟩ = [Nil ; Cons of {hd : a ; tl : list}]

val rec exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    case l {
      Nil     → false
      Cons[c] → if pred c.hd { true } else { exists pred c.tl }
    }
  }

val rec fold_left : ∀a b, (a ⇒ b ⇒ a) ⇒ a ⇒ list⟨b⟩ ⇒ a =
  fun f acc l {
    case l {
      Nil     → acc
      Cons[c] → fold_left f (f acc c.hd) c.tl
    }
  }
```

The `exists` function takes as input a predicate and a list, and it returns a boolean indicating whether there is an element satisfying the predicate in the list. The `fold_left` function iterates a function on all the elements of a list, gathering the result in an accumulator which initial value is given. For example, the `exists` function can be implemented with `fold_left` as follows, but this version does not stop as soon as possible when an element satisfying the predicate is found.

```
val exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    let f = fun acc e { if pred e { true } else { acc } };
    fold_left f false l
  }
```

Note that both `exists` and `fold_left` are polymorphic, they can for instance be applied to lists containing elements of an arbitrary type. In our syntax, polymorphism is explicitly materialised using universally quantified type variables.

Polymorphism is an important feature as it allows for more generic programs. In general, ML-like languages only allow a limited form of polymorphism on let-bindings. In these systems, generalisation can only happen on expressions of the form "`let x = t in u`". As a consequence, the following function is rejected by the OCaml type checker.

```
let silly_ocaml : ('a → 'a) → unit → unit option =
  fun f u → f (Some (f u))
```

In our system, polymorphism is not limited: universal quantification is allowed anywhere in types. Our types thus contain the full power of System F [*Girard 1972*, *Reynolds 1974*]. In particular, the equivalent of `silly_ocaml` is accepted by our type-checker.

```
include lib.option
val silly : (∀a, a ⇒ a) ⇒ {} ⇒ option⟨{}⟩ =
  fun f u { f Some[f u] }
```

In fact, System F polymorphism is not the only form of quantification that is supported in our system. It also provides existential types, which are an essential first step towards the encoding of a module system supporting a notion of abstract interface. Moreover, our system is based on higher-order logic, which means that types are not the only objects that can be quantified over in types. In particular, we will see that quantifiers can range over terms in the next section.

The programming languages of the ML family generally include effectful operations, for example references (i.e., mutable variables). Our system is no exception as it provides

control operators. As first discovered by Timothy G. Griffin [*Griffin 1990*], control operators like Lisp's *call/cc* can be used to give a computational interpretation to classical logic. On the programming side, they can be seen as a form of exception mechanism. For example, the following definition of `exists`, using `fold_left`, stops as soon as possible when there is an element satisfying the predicate in the list.

```
val exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    save k {
      let f = fun acc e { if pred e { restore k true } else { acc } };
      fold_left f false l
    }
  }
```

Here, the continuation is saved in a variable `k` before calling the `fold_left` function, and it is restored with the value `true` if an element satisfying the predicate is found. In this case, the evaluation of `fold_left` is simply aborted.

Our control operators can also be used to define programs whose types correspond to logical formulas that are only valid in classical logic. For instance, the type of the following programs corresponds to Peirce's law, the principle of double negation elimination and the law of the excluded middle.

```
val peirce : ∀a b, ((a ⇒ b) ⇒ a) ⇒ a =
  fun x {
    save k { x (fun y { restore k y }) }
  }

// Usual definition of logical negation
type neg⟨a⟩ = a ⇒ ∀x, x

val dneg_elim : ∀a, neg⟨neg⟨a⟩⟩ ⇒ a =
  peirce

// Disjoint sum of two types (logical disjunction)
type either⟨a,b⟩ = [InL of a ; InR of b]

val excl_mid : ∀a, {} ⇒ either⟨a, neg⟨a⟩⟩ =
  fun _ {
    save k { InR[fun x { restore k InL[x] }] }
  }
```

Note that the definition of `excl_mid` contains a dummy function constructor. Its presence is required for a reason related to value restriction (see Section 1.3). It would not be necessary if `excl_mid` was not polymorphic in `a`. Moreover, note that `dneg_elim` can be defined to be exactly `peirce` thanks to subtyping (see Chapter 6).

From a computational point of view, manipulating continuations using control operators can be understood as cheating. For example `excl_mid` (or rather, `excl_mid {}`) saves the continuation and immediately returns a (possibly false) proof of `neg⟨a⟩`. Now, if this proof is ever applied to a proof of `a` (which would result in absurdity), the program backtracks and returns the given proof of `a`. This interpretation in terms of cheating has been well-known for a long time (see, for example, [*Wadler 2003, Section 4*]).

1.2 Proofs of ML programs

The system presented in this thesis is not only a programming language, but also a proof assistant focusing on program proving. Its proof mechanism relies on equality types of the form `t ≡ u`, where `t` and `u` are arbitrary (possibly untyped) terms of the language itself. Such an equality type is inhabited by `{}` (i.e., the record with no fields) if the denoted equivalence is true, and it is empty otherwise. Equivalences are managed using a partial decision procedure that is driven by the construction of programs. An equational context is maintained by the type checker to keep track of the equivalences that are assumed to be true during the construction of proofs. This context is extended whenever a new equation is learned (e.g., when a lemma is applied), and equations are proved by looking for contradictions (e.g., when two different variants are supposed equivalent).

To illustrate the proof mechanism, we will consider simple examples of proofs on unary natural number (a.k.a. Peano numbers). Their type is given below, together with the corresponding addition function defined using recursion on its first argument.

```
type rec nat = [Zero ; Succ of nat]

val rec add : nat ⇒ nat ⇒ nat =
  fun n m {
    case n { Zero → m | Succ[k] → Succ[add k m] }
  }
```

As a first example, we will show that `add Zero n ≡ n` for all `n`. To express this property we can use the type `∀n:ι, add Zero n ≡ n`, where `ι` can be thought of as the set of all the usual program values. This statement can then be proved as follows.

```
val add_z_n : ∀n:ι, add Zero n ≡ n = {}
```

Here, the proof is immediate (i.e., {}) as we have `add Zero n ≡ n` by definition of the `add` function. Note that this equivalence holds for all `n`, whether it corresponds to an element of `nat` or not. For instance, it can be used to show `add Zero true ≡ true`.

Let us now show that for every `n` we have `add n Zero ≡ n`. Although this property looks similar to `add_z_n`, the following proof is invalid.

```
// val add_n_z : ∀n:ι, add n Zero ≡ n = {}
```

Indeed, the equivalence `add n Zero ≡ n` does not hold when `n` is not a unary natural number. In this case, the computation of `add n Zero` will produce a runtime error. As a consequence, we need to rely on a form of quantification that only ranges over unary natural numbers. This can be achieved with the type `∀n∈nat, add n Zero ≡ n`, which corresponds to a (dependent) function taking as input a natural number `n` and returning a proof of `add n Zero ≡ n`. This property can then be proved using induction and case analysis as follows.

```
val rec add_n_z : ∀n∈nat, add n Zero ≡ n =
  fun n {
    case n {
      Zero    → {}
      Succ[k] → let ih = add_n_z k; {}
    }
  }
```

If `n` is `Zero`, then we need to show `add Zero Zero ≡ Zero`, which is immediate by definition of `add`. In the case where `n` is `Succ[k]` we need to show `add Succ[k] Zero ≡ Succ[k]`. By definition of `add`, this can be reduced to `Succ[add k Zero] ≡ Succ[k]`. We can then use the induction hypothesis (i.e., `add_n_z k`) to learn `add k Zero ≡ k`, with which we can conclude the proof.

It is important to note that, in our system, a program that is considered as a proof needs to go through a termination checker. Indeed, a looping program could be used to prove anything otherwise. For example, the following proof is rejected.

```
// val rec add_n_z_loop : ∀n∈nat, add n Zero ≡ n =
//   fun n { let ih = add_n_z_loop n; {} }
```

It is however easy to see that `add_z_n` and `add_n_z` are terminating, and hence valid. In the following, we will always assume that the programs used as proofs have been shown terminating.

There are two main ways of learning new equations in the system. On the one hand, when a term `t` is matched in a case analysis, a branch can only be reached when the corresponding pattern `C[x]` matches. In this case we can extend the equational context with the equivalence `t ≡ C[x]`. On the other hand, it is possible to invoke a lemma by calling the corresponding function. In particular, this must be done to use the induction hypothesis in proofs by induction like in `add_z_n` or the following lemma.

```
val rec add_n_s : ∀n m∈nat, add n Succ[m] ≡ Succ[add n m] =
  fun n m {
    case n {
      Zero    → {}
      Succ[k] → let ind_hyp = add_n_s k m; {}
    }
  }
```

In this case, the equation corresponding to the conclusion of the used lemma is directly added to the context. Of course, more complex results can be obtained by combining more lemmas. For example, the following proves the commutativity of addition using a proof by induction with `add_n_z` and `add_n_s`.

```
val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero    → let lem = add_n_z m; {}
      Succ[k] → let ih  = add_comm k m;
                let lem = add_n_s m k; {}
    }
  }
```

Many more examples of proofs and programs are provided in Chapter 7 (and even more with the implementation of the system). Each of them (including those in the current chapter) have been automatically checked upon the generation of this document. They are thus correct with respect to the implementation.

1.3 A brief history of value restriction

A soundness issue related to side-effects and call-by-value evaluation arose in the seventies with the advent of ML. The problem stems from a bad interaction between side-effects and Hindley-Milner polymorphism. It was first formulated in terms of references, as explained in [*Wright 1995, Section 2*]. To extend an ML-style language with references, the naive

approach consist in defining an abstract type `'a ref` and polymorphic procedures with the following signature (given in OCaml syntax).

```
type 'a ref
val ref  : 'a → 'a ref
val (:=) : 'a ref → 'a → unit
val (!)  : 'a ref → 'a
```

Here, the function `ref` takes as input a value of some type and creates a new reference cell containing an element of the corresponding type. The value of a reference can then be updated using the infix operator `(:=)` (to be pronounced "set"), and its value can be obtained using the prefix operator `(!)` (to be pronounced "get").

These immediate additions quickly lead to trouble when working with polymorphic references. The problem can be demonstrated by the following example, which is accepted by the naive extension of the type system.

```
let l = ref [] in
l := [true]; (List.hd !l) + 1
```

On the first line, variable `l` is given the polymorphic type `'a list ref`, which can be unified both with `bool list ref` and `int list ref` on the second line. This is an obvious violation of type safety, which is the very purpose of a type system.

To solve the problem, alternative type systems such as [*Tofte 1990*, *Damas 1982*, *Leroy 1991*, *Leroy 1993*] were designed. However, they all introduced a complexity that contrasted with the elegance and simplicity of ML systems (see [*Wright 1995*, *Section 2*] and [*Garrigue 2004*, *Section 2*] for a detailed account). A simple and elegant solution was finally found by Andrew Wright in the nineties. He suggested restricting generalisation (i.e., introduction of polymorphism) to syntactic values [*Wright 1994*, *Wright 1995*].

In ML, generalisation usually happens in expressions of the form "`let x = u in t`", called let-bindings. The type-checking of such an expression proceeds by inferring the type of the term `u`, which may contain unification variables. The type of `u` is then generalized by universally quantifying over these unification variables. Finally, the term `t` is type-checked under the assumption that `x` has the most general type of `u`. With value restriction, the generalisation of the type of `u` only happens if `u` is a syntactic value. Consequently, the example above is rejected since `ref []` is not a value, and hence its inferred type `'_a list ref` is only weakly polymorphic (i.e., it can only be unified with exactly one, yet unknown type). Thus, it cannot be unified with both `bool list ref` and `nat list ref`.

As mentioned in Section 1.1, the system presented in this thesis does not include references, but control structures. One way of extending ML with control structures is again to introduce an abstract type equipped with polymorphic operations.

```
type 'a cont
val callcc : ('a cont → 'a) → 'a
val throw  : 'a cont → 'a → 'b
```

The function `callcc` corresponds to the control operator *call/cc*, which was first introduced in the *Scheme* programming language. When called, this function saves the current continuation (i.e., the current state of the program's environment) and feeds it to the function it is given as an argument. The continuation can be restored in the body of this function using the `throw` function.

As for references, the addition of control structures breaks the type safety of ML in the presence of polymorphism. A complex counterexample was first discovered by Robert Harper and Mark Lillibridge [*Harper 1991*].

```
let c = callcc
  (fun k → ((fun x → x), (fun f → throw k (f, (fun _ → ((((((
in
print_string ((fst c) "Hello world!");
(snd c) (fun x → x+2)
```

Intuitively, the program first saves the continuation and builds a pair `c` containing two functions. The first one is simply the identity function. The second one takes a function `f` as argument and calls `throw` to restore the previously saved continuation. It then replaces `c` with a pair containing `f` and a constant function. Consequently, the first element of the pair `c` can be used as the (polymorphic) identity function as long as the second element of `c` has not been used. However, when the second element of `c` is called with a function `g`, then `g` becomes the first element of `c` and the computation restarts. This is problematic since the function `fun x → x+2` is then applied to a value of type `string`, which is thus fed to an integer addition.

During type-checking, the type that is inferred for the pair `c` (prior to generalisation) is `('_a → '_a) * (('_a → '_a) → unit)`. Thus, in absence of value restriction, the last two lines of the counterexample are type-checked under the assumption that `c` has the polymorphic type `('a → 'a) * (('a → 'a) → unit)`. In particular, the type `'a → 'a` can be unified with both `string → string` and `int → int`. As with references, the value restriction forbids such unifications.

Note that it is relatively easy to translate the counter example into our language. Indeed, terms of the form `callcc (fun k → t)` are translated to `save k → t` and terms of the form `throw k u` to `restore k u`. Moreover, as our system contains system F, value restriction needs to be stated differently. It appears on the typing rule for the introduction of the universal quantifier. In the system, value restriction corresponds to only applying this rule to terms that are values.

1.4 Dependent functions and relaxed restriction

One of the main features of our system is a dependent function type. It is essential for building proofs as it provides a form of typed quantification. However, combining call-by-value evaluation, side-effects and dependent functions is not straightforward. Indeed, if `t` is a dependent function of type `∀x∈a, b⟨x⟩` and if `u` has type `a`, then is it not always the case that `t u` evaluates to a value of type `b⟨u⟩`. As a consequence, we need to restrict the application of dependent functions to make sure that it is type safe. The simplest possible approach consists in only allowing syntactic values as arguments of dependent functions, which is another instance of the value restriction. It is however not satisfactory as it considerably weakens the expressiveness of dependent functions. For example, `add_n_z` cannot be used to prove `add (add Zero Zero) Zero ≡ add Zero Zero`. Indeed, the term `add Zero Zero` is not a value, which means that it cannot be used as argument of a dependent function. This problem arises very often as proofs rely heavily on dependent functions. As a consequence, the value restriction breaks the modularity of our proof system.

Surprisingly, our equality types provide a solution to the problem. Indeed, they allow us to identify terms having the same observable computational behaviour. We can then relax the restriction to terms that are equivalent to some value. In other words, we consider that a term `u` is a value if we can find a value `v` such that `u ≡ v`. This idea can be applied whenever value restriction was previously required. Moreover, the obtained system is (strictly) more expressive that the one with the syntactic restriction. Indeed, finding a value that is equivalent to a term that is already a value can always be achieved using reflexivity. Although this new idea seems simple, establishing the soundness of the obtained system is relatively subtle [*Lepigre 2016*].

In practice, using a term as a value is not always immediate. For example, the system is not able to directly prove that `add n m` is a value, provided that `n` and `m` are two natural numbers. It is however possible to establish this fact internally as follows.

```
val rec add_total : ∀n m∈nat, ∃v:ι, add n m ≡ v =
  fun n m {
    case n {
      Zero    → {}
      Succ[k] → let ih = add_total k m; {}
    }
  }
```

Here, `add_total` proves that for any values `n` and `m` in the type `nat`, there is some value `v` such that `add n m ≡ v`. Note that we did not specifically require `v` to be a natural number as this is usually not necessary in practice. Thanks to `add_total`, we can give a proof of the associativity of our addition function.

```
val rec add_asso : ∀n m p∈nat, add n (add m p) ≡ add (add n m) p =
  fun n m p {
    let tot_m_p = add_total m p;
    case n {
      Zero    → {}
      Succ[k] → let tot_k_m = add_total k m;
                let ih = add_asso k m p; {}
    }
  }
```

Note that the proof requires two calls to `add_total`. The first one is used in both the base case and the induction case. It is required so that the system can unfold the definition of `add n (add m p)` according to the head constructor of `n`. As we are in call-by-value, we can only reduce the definition of a function when it is applied to values. It is the case here as `n` is a variable and `add m p` is equivalent to some value, as witnessed by `tot_m_p`. The second call to `add_total` is required for a similar reason in the successor case.

1.5 Handling undecidability

Typing and subtyping are most likely to be undecidable in our system. Indeed, it contains Mitchell's variant of System F [*Mitchell 1991*] for which typing and subtyping are both known to be undecidable [*Tiuryn 1996*, *Tiuryn 2002*, *Wells 1994*, *Wells 1999*]. Moreover, as argued in [*Lepigre 2017*], we believe that there are no practical, complete semi-algorithms for extensions of System F like ours. Instead, we propose an incomplete semi-algorithm that may fail or even diverge on a typable program. In practice we almost never meet non termination, but even in such an eventuality, the user can always interrupt the program to obtain a relevant error message. This design choice is a very important distinguishing feature of the system. To our knowledge, such ideas have only be used (and implemented) in some unpublished work of Christophe Raffalli [*Raffalli 1998*, *Raffalli 1999*] and in [*Lepigre 2017*].

One of the most important ideas, that makes the system practical and possible to implement, is to only work with syntax-directed typing and subtyping rules. This means that only one of our typing rules can be applied for each different term constructor. Similarly, we have only two subtyping rules per type constructor: one where it appears on the left of the inclusion, and one where it appears on the right. As type-checking can only diverge in subtyping, an error message can be built using the last applied typing rule. Moreover, all the undecidability of the system is concentrated into the management of unification variables, termination checking and equivalence derivation.

As a proof of concept, we implemented our system in a prototype called PML2. The last version of its source code is available online (http://lepigre.fr/these/). Its implementation mostly follows the typing and subtyping rules of the system given in Chapter 6. Overall,

our system provides a similar user experience to statically typed functional languages like OCaml or Haskell. In such languages, type annotations are also required for advanced features like polymorphic recursion.

1.6 Related work and similar systems

To our knowledge, the combination of call-by-value evaluation, side-effects and dependent products has never been achieved before. At least not for a dependent product fully compatible with effects and call-by-value. For example, the Aura language [*Jia 2008*] forbids dependency on terms that are not values in dependent applications. Similarly, the F* language [*Swamy 2011*] relies on (partial) let-normal forms to enforce values in argument position. Daniel Licata and Robert Harper have defined a notion of positively dependent types [*Licata 2009*] which only allow dependency over strictly positive types. Finally, in languages like ATS [*Xi 2003*] and DML [*Xi 1999*] dependencies are limited to a specific index language.

The system that seems the most similar to ours is NuPrl [*Constable 1986*], although it is inconsistent with classical reasoning and not effectful. NuPrl accommodates an observational equivalence relation similar to ours (Howe's *squiggle* relation [*Howe 1989*]). It is partially reflected in the syntax of the system. Being based on a Kleene style realizability model, NuPrl can also be used to reason about untyped terms.

The central part of this paper consists in the construction of a classical realizability model in the style of Jean-Louis Krivine [*Krivine 2009*]. We rely on a call-by-value presentation which yields a model in three layers (values, terms and stacks). Such a technique has already been used to account for classical ML-like polymorphism in call-by-value in the work of Guillaume Munch-Maccagnoni [*Munch 2009*]. It is here extended to include dependent products. Note that our main result (Theorem 5.4.15) is unrelated to Lemma 9 in Munch-Maccagnoni's work [*Munch 2009*].

The most actively developed proof assistants following the Curry-Howard correspondence are Coq and Agda [*CoqTeam 2004*, *Norell 2008*]. The former is based on Coquand and Huet's calculus of constructions and the latter on Martin-Löf's dependent type theory [*Coquand 1988*, *Martin-Löf 1982*]. These two constructive theories provide dependent types, which allow the definition of very expressive specifications. Coq and Agda do not directly give a computational interpretation to classical logic. Classical reasoning can only be done through a negative translation or the definition of axioms such as the law of the excluded middle. In particular, these two languages are not effectful. However, they are logically consistent, which means that they only accept terminating programs. As termination checking is a difficult (and undecidable) problem, many terminating programs are rejected. Although this is not a problem for formalizing mathematics, this makes programming tedious. In our system, only proofs need to be shown terminating. Moreover, it is possible to reason about non-terminating and even untyped programs.

The TRELLYS project [*Casinghino 2014*] aims at providing a language in which a consistent core interacts with type-safe dependently-typed programming with general recursion. Although the language defined in [*Casinghino 2014*] is call-by-value and effectful, it suffers from value restriction like Aura [*Jia 2008*]. The value restriction does not appear explicitly but is encoded into a well-formedness judgement appearing as the premise of the typing rule for application. Apart from value restriction, the main difference between the language of the TRELLYS project and ours resides in the calculus itself. Their calculus is Church-style (or explicitly typed) while ours is Curry-style (or implicitly typed). In particular, their terms and types are defined simultaneously, while our type system is constructed on top of an untyped calculus.

Another similar system can be found in the work of Alexandre Miquel on the implicit calculus of inductive constructions [*Miquel 2001*], in which quantifiers are Curry-style. This system has been extended with classical logic at the level of propositions [*Miquel 2007*], but the considered language is call-by-name. As a consequence, it does not have to deal with the soundness issues that arise in call-by-value.

The PVS system [*Owre 1996*] is similar to ours as it is based on classical higher-order logic. However this tool does not seem to be a programming language, but rather a specification language coupled with proof checking and model checking utilities. It is nonetheless worth mentioning that the undecidability of PVS's type system is handled by generating proof obligations. The Why3 language [*Filliâtre 2013*] also relies on generated proof obligations but it embeds a programming language (called WhyML) corresponding to a very restricted subset of ML. Our system takes a completely different approach and relies on a non-backtracking type-checking algorithm. Although our system is likely to be undecidable, we argue as in [*Lepigre 2017*] that this seems not to be a problem in practice and allows for a simpler implementation of the type system.

Several systems have been proposed for proving ML programs. ProPre [*Manoury 1992*] relies on a notion of *algorithms*, corresponding to equational specifications of programs. It is used in conjunction with a type system based on intuitionistic logic. Although it is possible to use classical logic to prove that a program meets its specification, the underlying programming language is not effectful. Similarly, the PAF! system [*Baro 2003*] implements a logic supporting proofs of programs, but it is restricted to a purely functional subset of ML. Another approach for reasoning about purely functional ML programs is given in [*Régis-Gianas 2007*], where Hoare logic is used to specify program properties. Finally, it is also possible to reason about ML programs (including effectful ones) by compiling them down to higher-order formulas [*Charguéraud 2010*, *Charguéraud 2011*], which can then be manipulated using an external prover like Coq [*CoqTeam 2004*]. In this case, the user is required to master at least two languages, contrary to our system in which programming and proving take place in a uniform framework.

1.7 Thesis overview

The starting point of this thesis is an untyped, call-by-value language. It is defined in Chapter 2, following a gentle introduction to the λ-calculus and its evaluation in abstract machines. The formal definition of our language is itself based on an abstract environment machine, which allows us to account for computational effects easily. Another benefit of this presentation is that it provides a natural definition of contextual equivalence. It is given in Chapter 3, where a broader class of relations is studied.

A higher-order type system for our language, together with its semantics, is then defined in Chapter 4. Its most singular feature is an equality type over terms, that is interpreted using the untyped notion of equivalence described in Chapter 2. This enables the specification of program properties that can then be proved using equational reasoning. The adequacy of our type system with respect to its semantics is then proved using classical realizability techniques. As our language is call-by-value, the interpretation of types is spread among three sets related by orthogonality: a set of values, a set of evaluation contexts and a set of terms.

The type system defined in Chapter 4 provides a weak form of dependent function type, which can be used to preform typed quantification. However, value restriction is required on the arguments of dependent functions, which makes them practically useless. Chapter 5 provides a solution to this problem by proposing a relaxed restriction expressed using observational equivalence. The soundness of this new approach is established by constructing a novel (and somewhat surprising) realizability model. It relies on a new instruction, that internalises our notion of program equivalence into the reduction relation of our abstract machine. For the definition of reduction and equivalence not to be circular, we need to rely on a stratified construction of these relations.

In Chapter 6, a more practical approach is taken. The system is extended with a notion of subtyping, which yields a system that can be directly implemented with syntax-directed rules. While remaining compatible with the realizability model of Chapter 5, our notion of subtyping is able to handle all the connectives that do not have algorithmic contents. This means that quantifiers and equality types are only managed by subtyping. At the end of Chapter 6, we sketch the extension of the system with inductive and coinductive types, and with general recursion. To this aim, we rely on a recently submitted paper [*Lepigre 2017*].

Finally, Chapter 7 is dedicated to examples of programs and proofs and to discussions on the implementation of the system. The source code of the prototype is distributed with this document. The latest version of the prototype, this document and other attached files are available online (http://lepigre.fr/these/).

2 Untyped calculus and abstract machine

In this chapter, we introduce the programming language that will be considered throughout this thesis. Its operational semantics is expressed in terms of an abstract machine, which will allow us to account for computational effects.

2.1 The pure λ-calculus

In this thesis, we consider a programming language of the ML family, similar to OCaml or SML. Like every functional language, its syntax is based on the λ-calculus. Introduced by Alonzo Church in the Thirties, the λ-calculus [*Church 1941*] is a formalism for representing computable functions, and in particular recursive functions. As shown by Alan Turing, the λ-calculus is a *universal model of computation* [*Turing 1937*].

Definition 2.1.1. The terms of the λ-calculus (or λ-terms) are built from a countable alphabet of variables (or λ-variables) $\mathcal{V}_\iota = \{x, y, z...\}$. The set of all the λ-terms is denoted Λ and is defined as the language recognised by the following BNF grammar.

$$t, u ::= x \mid \lambda x.t \mid t\ u \qquad\qquad x \in \mathcal{V}_\iota$$

A term of the form $\lambda x.t$ is called an abstraction (or λ-abstractions) and a term of the form $t\ u$ is called an application.

Remark 2.1.2. Throughout this thesis, the definition of languages using BNF grammars will implicitly introduce a naming convention for meta-variables. For example, the above definition implies that the letters t and u (with a possible subscript) will always be used to denote elements of Λ.

Intuitively, a λ-abstraction $\lambda x.t$ forms a function by binding the variable x in the term t. This would be denoted $x \mapsto t$ in common mathematics. Similarly, a term of the form $t\ u$ denotes the application of (the function) t to (the argument) u. This would be denoted $t(u)$ in common mathematics.

Remark 2.1.3. As λ-terms have a tree-like structure, parentheses are sometimes required for disambiguation. For example, the term $\lambda x.t\ u$ can be read both as $(\lambda x.t)\ u$ and as $\lambda x.(t\ u)$. To lighten the notations we will consider application to be left-associative and to have higher precedence than abstraction. As a consequence, we will always read the term $\lambda x.t\ x\ u$ as $\lambda x.((t\ x)\ u)$.

Remark 2.1.4. The syntax of the λ-calculus only allows for one-place functions. To form a function of two arguments (or more) one must rely on Curryfication. Indeed, a function of two arguments can be seen as a function of one argument returning a function. Following this scheme, the multiple arguments of the function are given in turn, and not simultaneously. As an example, the function $(x, y) \mapsto x$ can be encoded as $\lambda x.\lambda y.x$.

Although this is not reflected explicitly in the syntax of λ-terms, a λ-variable may play two very different roles. It can be used either as a constant, like y in the constant function $\lambda x.y$, or as a reference to a binder, like x in the identity function $\lambda x.x$. Variable binding and the associated notions of free and bound variable are hence essential.

Definition 2.1.5. Given a term t, we denote by $\mathrm{FV}_\iota(t)$ the set of its free λ-variables and $\mathrm{BV}_\iota(t)$ the set of its bound λ-variables. These sets are defined inductively on the structure of the term t.

$$\begin{aligned} \mathrm{FV}_\iota(x) &= \{x\} & \mathrm{BV}_\iota(x) &= \emptyset \\ \mathrm{FV}_\iota(\lambda x.t) &= \mathrm{FV}_\iota(t) \setminus \{x\} & \mathrm{BV}_\iota(\lambda x.t) &= \mathrm{BV}_\iota(t) \cup \{x\} \\ \mathrm{FV}_\iota(t\ u) &= \mathrm{FV}_\iota(t) \cup \mathrm{FV}_\iota(u) & \mathrm{BV}_\iota(t\ u) &= \mathrm{BV}_\iota(t) \cup \mathrm{BV}_\iota(u) \end{aligned}$$

Remark 2.1.6. Nothing prevents a λ-variable to have both free and bound occurrences in a term. For example, in $t = \lambda x.y\ \lambda y.x\ y$ the first occurrence of y is free while its second occurrence is bound. We have $y \in \mathrm{FV}_\iota(t) = \{y\}$ and $y \in \mathrm{BV}_\iota(t) = \{x, y\}$.

When a λ-abstraction (i.e., a function) is applied to an argument, we obtain a term of the form $(\lambda x.t)\ u$, called a β-redex. The reduction of such β-redexes plays an essential role in computation. Intuitively, the reduction of the β-redex $(\lambda x.t)\ u$ will be performed by replacing every occurrence of the bound variable x by u in the term t. This operation, called substitution, is formally defined as follows.

Definition 2.1.7. Let $t \in \Lambda$ and $u \in \Lambda$ be two λ-terms, and $x \in \mathcal{V}_\iota$ be a λ-variable. We denote $t[x \leftarrow u]$ the term t in which every free occurrence of x has been replaced by u. This operation is defined inductively on the structure of t.

$$\begin{aligned} x[x \leftarrow u] &= u & (\lambda y.t)[x \leftarrow u] &= \lambda y.t[x \leftarrow u] \\ y[x \leftarrow u] &= y & (t_1\ t_2)[x \leftarrow u] &= t_1[x \leftarrow u]\ t_2[x \leftarrow u] \\ (\lambda x.t)[x \leftarrow u] &= \lambda x.t \end{aligned}$$

Substitution is a subtle notion, and care should be taken to avoid capture of variables. For example, let us consider the function $\lambda x.\lambda y.x$ which takes an argument x and returns a constant function with value x. If we apply this function to y, the expected result is a constant function with value y. However, if we blindly substitute x with y in $\lambda y.x$ we obtain the identity function $\lambda y.y$. Indeed, the free variable y has been captured and now references a binder that had (coincidentally) the same name.

To solve this problem, we need to make sure that whenever a substitution $t[x \leftarrow u]$ is performed, no free variable of u is bound in t (i.e., $\mathrm{FV}_\iota(u) \cap \mathrm{BV}_\iota(t) = \emptyset$). Although we cannot rename the free variables of u, it is possible to rename the bound variables of t. Indeed, changing the name of a bound variable has no effect on the computational behaviour of a term. Two terms that are equivalent up to the names of their bound variables are said to be α-equivalent.

Definition 2.1.8. The α-equivalence relation $(\equiv_\alpha) \subseteq \Lambda \times \Lambda$ is defined, like in [*Krivine 1990*], as the smallest relation such that:

- if $x \in \mathcal{V}_\iota$ then $x \equiv_\alpha x$,
- if $t_1 \equiv_\alpha t_2$ and $u_1 \equiv_\alpha u_2$ then $t_1\ u_1 \equiv_\alpha t_2\ u_2$,
- if $t_1[x_1 \leftarrow y] \not\equiv_\alpha t_2[x_2 \leftarrow y]$ for only finitely many $y \in \mathcal{V}_\iota$ then $\lambda x_1.t_1 \equiv_\alpha \lambda x_2.t_2$.

Lemma 2.1.9. Given a term $t \in \Lambda$ and a finite set of variables $V \subseteq \mathcal{V}_\iota$, it is always possible to find a term $t_0 \in \Lambda$ such that $t_0 \equiv_\alpha t$ and $\mathrm{BV}_\iota(t_0) \cap V = \emptyset$.

Proof. A full proof is available in [*Krivine 1990, Lemma 1.11*]. □

Definition 2.1.10. Let $t \in \Lambda$ and $u \in \Lambda$ be two λ-terms, and $x \in \mathcal{V}_\iota$ be a λ-variable. We denote $t[x := u]$ the capture-avoiding substitution of x by u in t. It is defined as $t_0[x \leftarrow u]$ where $t_0 \in \Lambda$ is a term such that $t_0 \equiv_\alpha t$ and $\mathrm{BV}_\iota(t_0) \cap \mathrm{FV}(u) = \emptyset$. Such a term exists according to Lemma 2.1.9.

2.2 Evaluation contexts and reduction

To define the most general notion of reduction over λ-terms, we need to be able to refer to any β-redex. To this aim, we introduce the notion of evaluation context. Intuitively, a context will consist in a term with a hole (i.e., a place-holder for a subterm) and it will allow us to focus on any particular subterm of a term.

Definition 2.2.11. The set of evaluation contexts $[\Lambda]$ is defined as the language recognised by the following BNF grammar.

$$E, F ::= [-] \mid \lambda x.E \mid E\ t \mid t\ E \qquad\qquad x \in \mathcal{V}_\iota, t \in \Lambda$$

Definition 2.2.12. Given a term $u \in \Lambda$ and an evaluation context $E \in [\Lambda]$, we denote $E[u]$ the term formed by putting u into the hole of the evaluation context E. It is defined by induction on the structure of E as follows.

$$\begin{array}{rclcrcl} [-][u] & = & u & \qquad & (E\ t)[u] & = & E[u]\ t \\ (\lambda x.E)[u] & = & \lambda x.E[u] & & (t\ E)[u] & = & t\ E[u] \end{array}$$

Remark 2.2.13. Note that free variables of a term u may be captured when forming $E[u]$. For example, if we take $u = x$ and $E = \lambda x.\lambda y.[-]$ then x is free in u, but it does not appear free in $E[u] = \lambda x.\lambda y.x$.

Definition 2.2.14. Given a set of evaluation contexts $C \subseteq [\Lambda]$, we denote $\mathcal{R}(C) \subseteq \Lambda \times \Lambda$ the β-reduction relation induced by C. It is defined as the smallest relation such that for every $E \in C$, for every terms $t \in \Lambda$ and $u \in \Lambda$, and for every variable $x \in \mathcal{V}_\iota$ we have the following.

$$(E[(\lambda x.t)\ u]\,,\, E[t[x := u]]) \in \mathcal{R}(C)$$

Definition 2.2.15. The general β-reduction $(\to_\beta) \subseteq \Lambda \times \Lambda$ is defined as $\mathcal{R}([\Lambda])$. We say that the term $t \in \Lambda$ is in β-normal-form if there is no $u \in \Lambda$ such that $t \to_\beta u$. We denote $(\to_\beta^*)$ the reflexive, transitive closure of $(\to_\beta)$.

The general β-reduction relation $(\to_\beta)$ is non-deterministic. Indeed, given a term t, there might be two (different) terms u_1 and u_2 such that $t \to_\beta u_1$ and $t \to_\beta u_2$. For example, $((\lambda x_1.x_1)\ \lambda x_2.x_2)\ ((\lambda x_3.x_3)\ \lambda x_4.x_4)$ can either reduce to $(\lambda x_2.x_2)\ ((\lambda x_3.x_3)\ \lambda x_4.x_4)$ or to $((\lambda x_1.x_1)\ \lambda x_2.x_2)\ (\lambda x_4.x_4)$. Indeed, we can focus on the β-redex $(\lambda x_1.x_1)\ \lambda x_2.x_2$ using the evaluation context $[-]\ ((\lambda x_3.x_3)\ \lambda x_4.x_4)$, or on the β-redex $(\lambda x_3.x_3)\ \lambda x_4.x_4$ using the evaluation context $((\lambda x_1.x_1)\ \lambda x_2.x_2)\ [-]$. Although it is non-deterministic, the general β-reduction relation $(\to_\beta)$ has the Church-Rosser property [*Church 1936*].

Theorem 2.2.16. Let $t \in \Lambda$ be a term. If there are $u_1 \in \Lambda$ and $u_2 \in \Lambda$ such that $t \to_\beta^* u_1$ and $t \to_\beta^* u_2$, then there must be $u \in \Lambda$ such that $u_1 \to_\beta^* u$ and $u_2 \to_\beta^* u$.

Proof. A full proof is available in [*Church 1936*] or [*Barendregt 1981*] for example. □

Intuitively, the Church-Rosser property enforces a weak form of determinism. Indeed, it implies that a program can only compute one particular result, even if it can be attained in several different ways.

In the following, we are going to consider an effectful language that does not have the Church-Rosser property. As a consequence, we will need to restrict ourselves to a deterministic subset of the general β-reduction relation. If we were to work with a completely non-deterministic reduction relation, it would be extremely difficult to reason about our language. Programs would not only compute different possible results, but also terminate in a non-deterministic way.

The choice of the order in which β-redexes are reduced is called an *evaluation strategy*. The two evaluation strategies that are the most widely used in practice are called *call-by-name* and *call-by-value*. They both reduce outermost β-redexes first, and do not reduce β-redexes that are contained in the body of a λ-abstraction. This means that the term $\lambda x.(\lambda y.y)\ x$ is considered to be in normal form and cannot be evaluated further. In call-by-name, terms that are in function position are reduced first, and the computation of their arguments is delayed to the time of their effective use. In call-by-value, both arguments and functions are evaluated before performing the β-reduction. One way to formalize these evaluation strategies is to restrict the notion of evaluation context, to only allow focusing on the β-redex that is going to be reduced next.

Definition 2.2.17. The set of call-by-name evaluation contexts $[N] \subseteq [\Lambda]$ is defined as the language recognised by the following BNF grammar.

$$E, F ::= [-] \mid E\ t \qquad\qquad t \in \Lambda$$

The call-by-name reduction relation $(\to_N) \subseteq \Lambda \times \Lambda$ is defined as $\mathcal{R}([N])$.

In call-by-value, both the function and its argument need to be evaluated before the application can be performed. Consequently, two different call-by-value strategies can be defined: left-to-right and right-to-left call-by-value evaluation. The former evaluates the terms that are in function position first and the latter evaluates the terms that are in argument position first. Although left-to-right call-by-value evaluation is most widely used, some practical languages like OCaml use right-to-left evaluation. In this thesis, we make the same choice and only consider right-to-left call-by-value evaluation.

Definition 2.2.18. A term t is said to be a value if it is either a λ-variable or a λ-abstraction. The set $\Lambda_\iota \subseteq \Lambda$ of all the values is generated by the following BNF grammar.

$$v, w ::= x \mid \lambda x.t \qquad\qquad x \in \mathcal{V}_\iota$$

Definition 2.2.19. The set of right-to-left call-by-value evaluation contexts $[V] \subseteq [\Lambda]$ is defined as the language recognised by the following BNF grammar.

$$E, F ::= [-] \mid E\ v \mid t\ E \qquad\qquad v \in \Lambda_\iota, t \in \Lambda$$

The right-to-left call-by-value reduction relation $(\to_V) \subseteq \Lambda \times \Lambda$ is defined as $\mathcal{R}([V])$.

Remark 2.2.20. Left-to-right call-by-value evaluation can be defined using the evaluation contexts generated by the following BNF grammar.

$$E, F ::= [-] \mid E\ t \mid v\ E \qquad\qquad t \in \Lambda, v \in \Lambda_\iota$$

A given term of the λ-calculus may reduce in very different ways depending on what evaluation strategy is chosen. For example, the evaluation of $(\lambda y.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$ stops in one step in call-by-name

$$(\lambda y.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \to_N z$$

and it goes into a loop in call-by-value.

$$(\lambda y.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \to_V (\lambda y.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

Remark 2.2.21. Our reduction relations can be alternatively defined using deduction rules. A deduction rule is formed using premisses $\{P_i\}_{1 \leqslant i \leqslant n}$ and a conclusion C separated by an horizontal bar.

$$\frac{P_1 \quad \dots \quad P_n}{C}$$

The meaning of such a rule is that the conclusion C can be deduced when all the premisses P_i are true. In particular, if there is no premise then the conclusion can be deduced immediately. Using this formalism, the call-by-name reduction corresponds to the smallest relation satisfying the following two rules.

$$\frac{}{(\lambda x.t)\ u \to_N t[x := u]} \qquad\qquad \frac{t_1 \to_N t_2}{t_1\ u \to_N t_2\ u}$$

Similarly, the right-to-left call-by-value reduction relation ($\to_V$) corresponds to the smallest relation satisfying the following three rules.

$$\frac{}{(\lambda x.t)\ v \to_V t[x := v]} \qquad \frac{u_1 \to_V u_2}{t\ u_1 \to_V t\ u_2} \qquad \frac{t_1 \to_V t_2}{t_1\ v \to_V t_2\ v}$$

In this thesis, λ-terms and programs in general will be evaluated in an abstract machine called a Krivine machine [*Krivine 2007*]. This machine will emulate the right-to-left evaluation relation ($\to_V$). It will provide us with a computational framework in which programs and their evaluation contexts can be manipulated easily.

2.3 Call-by-value Krivine machine

In the previous section, we introduced the syntax of the λ-calculus and the evaluation of λ-terms. We will now reformulate these definitions in terms of a call-by-value Krivine abstract machine [*Krivine 2007*]. Our presentation will differ from the original machine, which is call-by-name. Although call-by-value Krivine machines have rarely been published, they are well-known in the classical realizability and compiler communities.

The main idea behind the Krivine abstract machine is to think of a term $t_0 \in \Lambda$ as a pair $(t, E) \in \Lambda \times [V]$ such that $t_0 = E[t]$ (the term t is said to be in head position). Using this representation, β-reduction proceeds in two steps. First, the machine state (t, E) is transformed into a state of the form $((\lambda x.u)\ v, F)$, in such a way that $E[t] = F[(\lambda x.u)\ v]$. The β-redex can then be reduced to obtain the state $(u[x := v], F)$. This behaviour can be attained using the following reduction rules, which are obtained naturally from the definition of right-to-left call-by-value evaluation.

$$\begin{array}{rcll}
(t\ u, E) & \to & (u, E[t\ [-]]) & \text{when } u \notin \Lambda_\iota \\
(v, E[t\ [-]]) & \to & (t\ v, E) & \\
(t\ v, E) & \to & (t, E[[-]\ v]) & \text{when } t \notin \Lambda_\iota \\
(v, E[[-]\ w]) & \to & (v\ w, E) & \\
((\lambda x.t)\ v, E) & \to & (t[x := v], E) &
\end{array}$$

The four first rules are responsible for bringing the next β-redex (according to our reduction strategy) in head position, and the last rule performs the β-reduction. Note that the first four rules do not change the represented term, and only move arguments or functions between the term and the evaluation context. Our set of reduction rules can be simplified to the following, by composing the last two pairs of rules.

$$\begin{array}{rcl} (t\ u, E) & \to & (u, E[t\ [-]]) \\ (v, E[t\ [-]]) & \to & (t, E[[-]\ v]) \\ (\lambda x.t, E[[-]\ v]) & \to & (t[x := v], E) \end{array}$$

The first rule is used to focus on the argument of an application, to compute it first. When the argument has been evaluated to a value, the second rule can be used to swap the argument with the unevaluated function. The computation can then continue with the evaluation of the function, which should (hopefully) evaluate to a λ-abstraction. If this is the case, the third rule can be applied to actually perform the β-reduction.

The state of the abstract machine can be seen as a zipper [*Huet 1997*] on the tree structure of a term. Indeed, the term that is in head position is the subterm on which the machine is focusing. It is also worth noting that the machine manipulates evaluation contexts from the inside out, which results in a heavy syntax. However, it is possible to represent right-to-left call-by-value evaluation contexts using a stack of functions (i.e., terms) and argument (i.e., values). We will take this approach in the following.

Definition 2.3.22. Values, terms, stacks and processes are generated by the following BNF grammar. The names of the corresponding sets are displayed on the left.

$$\begin{array}{lll} (\Lambda_v) & v, w ::= x \mid \lambda x.t & x \in \mathcal{V}_v \\ (\Lambda) & t, u ::= v \mid t\ u & \\ (\Pi) & \pi, \xi ::= \varepsilon \mid v \cdot \pi \mid [t]\pi & \\ (\Lambda \times \Pi) & p, q ::= t * \pi & \end{array}$$

The syntactic distinction between terms and values is specific to the call-by-value presentation, they would be collapsed in call-by-name. Intuitively, a stack can be thought of as an evaluation context represented as a list of terms and values. The values can be seen as arguments to be fed to the term in the context, and the terms can be considered as functions to which the term in the context will be applied. The symbol ε is used to denote an empty stack. A process $t * \pi$ forms the state of our abstract machine, and its reduction will consist in the interaction between the term t and its evaluation context encoded into the stack π.

Since our calculus is call-by-value, only values are (and should be) substituted to λ-variables during evaluation. From now on, we will hence work with the following definition of substitution. In particular, a substitution of the form $t[x := u]$ will be forbidden if u is not a syntactic value.

Definition 2.3.23. Let $t \in \Lambda$ be a term, $x \in \mathcal{V}_v$ be a λ-variable and $v \in \Lambda_v$ be a value. We denote $t[x := v]$ the capture-avoiding substitution of x by v in t.

Definition 2.3.24. The reduction relation $(\succ) \subseteq (\Lambda\times\Pi)\times(\Lambda\times\Pi)$ is the smallest relation satisfying the following rules. We denote $(\succ^*)$ its reflexive and transitive closure.

$$\begin{array}{rcl} t\ u * \pi & \succ & u * [t]\pi \\ v * [t]\pi & \succ & t * v\,.\,\pi \\ \lambda x.t * v\,.\,\pi & \succ & t[x := v] * \pi \end{array}$$

Three reduction rules are used to handle call-by-value evaluation. When an application is encountered, the function is stored in a stack-frame in order to evaluate its argument first. Once the argument has been completely computed, a value faces the stack-frame containing the function. At this point the function can be evaluated and the value is stored in the stack, ready to be consumed by the function as soon as it evaluates to a λ-abstraction. A capture-avoiding substitution can then be performed to effectively apply the argument to the function. As an example, $(\lambda x.x\ y)\ \lambda z.z * \varepsilon$ reduces to $y * \varepsilon$ as follows and it cannot evaluate further.

$$\begin{array}{rcl} (\lambda x.x\ y)\ \lambda z.z * \varepsilon & \succ & \lambda z.z * [\lambda x.x\ y]\varepsilon \\ & \succ & \lambda x.x\ y * \lambda z.z\,.\,\varepsilon \\ & \succ & (\lambda z.z)\ y * \varepsilon \\ & \succ & y * [\lambda z.z]\varepsilon \\ & \succ & \lambda z.z * y\,.\,\varepsilon \\ & \succ & y * \varepsilon \end{array}$$

Remark 2.3.25. It is possible to prove that the abstract machine and its $(\succ)$ reduction relation indeed implement the $(\to_v)$ evaluation on λ-terms. Although this result is not required here, it has been formalised by the author in the *Coq* proof assistant [*CoqTeam 2004*]. The proof sketch is available online (http://lepigre.fr/these/cbvMachine.v).

Remark 2.3.26. A left-to-right call-by-value machine can be defined in a similar way by swapping the roles of terms and values in stacks (stack frames contain values and terms are pushed on the stack). The reduction relation is then the following.

$$\begin{array}{rcl} t\ u * \pi & \succ_{RL} & t * u\,.\,\pi \\ v * u\,.\,\pi & \succ_{RL} & u * [v]\pi \\ v * [\lambda x.t]\pi & \succ_{RL} & t[x := v] * \pi \end{array}$$

The state of our abstract machine contains two parts: a term being evaluated (i.e., the term in head position) and its evaluation context (i.e., the stack). As a consequence, it is possible to define reduction rules that manipulate the stack (i.e., the evaluation context) as a first class object. Such reduction rules produce computational effects.

2.4 Computational effects and λμ-calculus

In the programming languages community, computational effects (a.k.a. side-effects) refer to modifications made by a program to its environment as a byproduct of the computation of its result. For example, a program may generate computational effects by writing to a tape, or by modifying the value of a global memory cell. In our calculus, the environment of a program (i.e., a term) only consists of an evaluation context encoded as a stack. Computational effects can hence be produced if a program is able to modify the stack as a whole during its evaluation in the abstract machine.

We are now going to extend our calculus and our abstract machine with operations allowing the manipulation of the stack. More precisely, we will provide a way to save the stack (i.e., the evaluation context), so that it can be restored at a later stage. A natural way to extend our language is to use the syntax of (Philippe de Groote's variant of) Michel Parigot's λμ-calculus [*Parigot 1992, de Groote 1994*]. We hence introduce a new binder $\mu\alpha.t$ capturing the current stack in the μ-variable α. The stack can then be restored in t using the syntax $[\alpha]u$.

Definition 2.4.27. Let $\mathcal{V}_\sigma = \{\alpha, \beta, \gamma...\}$ be a countable set of μ-variables (or stack variables) disjoint from $\mathcal{V}_\iota$. Values, terms, stacks and processes are now generated by the following grammar. The names of the corresponding sets are displayed on the left.

$$
\begin{array}{lll}
(\Lambda_\iota) & v, w ::= x \mid \lambda x.t & x \in \mathcal{V}_\iota \\
(\Lambda) & t, u ::= v \mid t\ u \mid \mu\alpha.t \mid [\pi]t & \\
(\Pi) & \pi, \xi ::= \varepsilon \mid \alpha \mid v \cdot \pi \mid [t]\pi & \alpha \in \mathcal{V}_\sigma \\
(\Lambda \times \Pi) & p, q ::= t * \pi &
\end{array}
$$

Note that terms of the form $[\pi]t$ will only be available to the user if π is a stack variable. Allowing arbitrary stacks allows us to substitute μ-variables by stacks during computation. Like with λ-variable, we will need to be careful and avoid variable capture. However, we will not give the full details this time.

Definition 2.4.28. Given a value, term, stack or process ψ, we denote $FV_\iota(\psi)$ (resp. $BV_\iota(\psi)$) the set of its free (resp. bound) λ-variables and $FV_\sigma(\psi)$ (resp. $BV_\sigma(\psi)$) the set of its free (resp. bound) μ-variables. These sets are defined in a similar way to Definition 2.1.5.

Definition 2.4.29. Let $t \in \Lambda$ be a term, $\pi \in \Pi$ be a stack and $\alpha \in \mathcal{V}_\sigma$ be a μ-variable. We denote $t[\alpha := \pi]$ the (capture-avoiding) substitution of α by π in t.

Definition 2.4.30. The reduction relation $(>)$ is extended with two new reduction rules.

$$\begin{array}{rcl} t\ u * \pi & \succ & u * [t]\pi \\ v * [t]\pi & \succ & t * v \cdot \pi \\ \lambda x.t * v \cdot \pi & \succ & t[x := v] * \pi \\ \mu\alpha.t * \pi & \succ & t[\alpha := \pi] * \pi \\ [\xi]t * \pi & \succ & t * \xi \end{array}$$

When the abstract machine encounters a μ-abstraction $\mu\alpha.t$, the current stack π is substituted to the μ-variables α. Consequently, every subterm of the form $[\alpha]u$ in t becomes $[\pi]u$. When the machine then reaches a state of the form $[\xi]u * \pi$, the current stack π is erased, and computation resumes with the stored stack ξ. For example, if t and v are arbitrary terms and values, then the process $\lambda x.\mu\alpha.t\ [\alpha]x * v \cdot \varepsilon$ reduces as follows.

$$\begin{array}{rcl} \lambda x.\mu\alpha.t\ [\alpha]x * v \cdot \varepsilon & \succ & \mu\alpha.t[x := v]\ [\alpha]v * \varepsilon \\ & \succ & t[x := v]\ [\varepsilon]v * \varepsilon \\ & \succ & [\varepsilon]v * [t[x := v]]\varepsilon \\ & \succ & v * \varepsilon \end{array}$$

Note that when a stack is erased, arbitrary terms might be erased. In particular, we could have chosen $t = \Omega = (\lambda x.x\ x)\ \lambda x.x\ x$ in the previous example, although the reduction of this term does not terminate. Indeed, we have

$$\begin{array}{rcl} \Omega * \pi & \succ & \lambda x.x\ x * [\lambda x.x\ x]\pi \\ & \succ & \lambda x.x\ x * \lambda x.x\ x \cdot \pi \\ & \succ & \Omega * \pi \end{array}$$

for every possible stack π.

The abstract machine defined in this section can be used for evaluating terms of the $\lambda\mu$-calculus. Although this language is very elegant and concise, it is not suitable for practical programming. In the following section, our language will be extended with records (i.e., tuples with named fields) and variants (i.e., constructors and pattern-matching). We will thus obtain a simple language with a concise formal definition, but that will be closer to being a practical programming language.

2.5 Full syntax and operational semantics

In this section, we present the syntax and the reduction relation of the abstract machine that will be considered throughout this thesis. Although the following extends definitions given in the previous sections, we choose not to avoid repetitions so that this section remains completely self-contained.

In this thesis, we consider a language expressed in terms of a *Krivine Abstract Machine* [*Krivine 2007*]. Our machine has the peculiarity of being call-by-value, which requires a

syntax formed with four entities: values, terms, stacks and processes. Note that the distinction between terms and values is specific to our call-by-value presentation, they would be collapsed in call-by-name.

Definition 2.5.31. We require three disjoint, countable sets of variables: $\mathcal{V}_\iota = \{x, y, z ...\}$ for λ-variables, $\mathcal{V}_\sigma = \{\alpha, \beta, \gamma ...\}$ for μ-variables, $\mathcal{V}_\tau = \{a, b, c ...\}$ for term variables.

As usual, λ-variables and μ-variables will be bound in terms to respectively form functions and capture continuations. Term variables are intended to be substituted by (unevaluated) terms, and not only values. They will be bound in types to express properties ranging over the set of all terms (see Chapter 4).

Definition 2.5.32. Values, terms, stacks and processes are mutually inductively defined as the languages recognised by the following BNF grammar. The names of the corresponding sets are displayed on the left.

$$
\begin{array}{lll}
(\Lambda_\iota) & v, w & ::= x \mid \lambda x.t \mid C_k[v] \mid \{(l_i = v_i)_{i\in I}\} \mid \Box \\
(\Lambda) & t, u & ::= a \mid v \mid t\ u \mid \mu\alpha.t \mid [\pi]t \mid v.l_k \mid [v \mid (C_i[x_i] \to t_i)_{i\in I}] \mid Y_{t,v} \mid R_{v,t} \mid \delta_{v,w} \\
(\Pi) & \pi, \rho & ::= \varepsilon \mid \alpha \mid v \cdot \pi \mid [t]\pi \\
(\Lambda\times\Pi) & p, q & ::= t * \pi
\end{array}
$$

Terms and values form a variation of the $\lambda\mu$-calculus [*Parigot 1992*], enriched with ML-like constructs (i.e., records and variants). Values of the form $C_k[v]$ (where $k \in \mathbb{N}$) correspond to variants, or constructors. Note that they always have exactly one argument in our language. Case analysis on variants is performed using the syntax $[v \mid (C_i[x_i] \to t_i)_{i\in I}]$, in which the pattern $C_i[x_i]$ is mapped to the term t_i for all i in $I \subseteq_{\text{fin}} \mathbb{N}$. Similarly, values like $\{(l_i = v_i)_{i\in I}\}$ correspond to records, which are tuples with named fields. The projection operation $v.l_k$ can be used to access the value labelled l_k in a record v.

Remark 2.5.33. The syntax $[v \mid (C_i[x_i] \to t_i)_{i\in I}]$ for matchings and the syntax $\{(l_i = v_i)_{i\in I}\}$ for records are part of our meta-language. We only use them as shortcuts to designate arbitrary lists of patterns or record fields. In the actual syntax, the full list of patterns or fields always needs to be specified. For example, we would write $\{l_1 = v_1; l_2 = v_2\}$ for a record or $[v \mid C_1[x_1] \to t_1 \mid C_2[x_2] \to t_2]$ for a case analysis when $I = \{1, 2\}$.

Terms of the form $Y_{t,v}$ denote a fixpoint, which can be used for general recursion. They roughly corresponds to OCaml's "`let rec`" construct. The value $\Box$ and terms of form $R_{v,t}$ or $\delta_{v,w}$ are only included for technical purposes. In particular, they are not intended to be used for programming. The value $\Box$ will be used in the definition of our semantics (see Chapters 4 and 6). Terms of the form $R_{v,t}$ will help us to distinguish records from other

sorts of values in our definition of observational equivalence (see Chapter 3). Finally, terms of the form $\delta_{v,w}$ will be used to obtain an essential property of our realizability model (see Chapter 5, Theorem 5.4.15).

A stack can be either the empty stack ε, a stack variable α, a value pushed on top of a stack $v \cdot \pi$, or a stack frame containing a term on top of a stack $[t]\pi$. The need for two stack constructors is specific to the call-by-value presentation as a stack not only needs to store the arguments to functions, but also the functions themselves while their arguments are being computed. In call-by-name only the arguments are stored in the stack.

Remark 2.5.34. We enforce values in constructors, record fields, projections and case analysis. This makes the calculus simpler because only β-reduction needs to manipulate the stack. Syntactic sugar such as the following can be defined to hide these restrictions.

$$t.l_k := (\lambda x.x.l_k)\ t \qquad\qquad C_k[t] := (\lambda x.C_k[x])\ t$$

Note that the elimination of such syntactic sugar corresponds to a form of partial `let`-normalization [*Moggi 1989*] or A-normalization [*Flanagan 1993*]. The translation can hence be seen as a natural compilation step [*Tarditi 1996*, *Chlipala 2005*].

Definition 2.5.35. Given a value, term, stack or process ψ we denote $FV_\iota(\psi)$ (resp. $FV_\sigma(\psi)$, $FV_\tau(\psi)$) the set of free λ-variables (resp. free μ-variables, term variables) contained in ψ. We also denote $FV(\psi) = FV_\iota(\psi) \cup FV_\sigma(\psi) \cup FV_\tau(\psi)$ the set of all the free variables of ψ. We say that ψ is closed if $FV(\psi) = \emptyset$. We denote Λ^*_ι the set of all the closed values, Λ^* the set of all the closed terms and Π^* the set of all the closed stacks.

Definition 2.5.36. A *substitution* is a map ρ such that for all $x \in \mathcal{V}_\iota$ we have $\rho(x) \in \Lambda_\iota$, for all $\alpha \in \mathcal{V}_\sigma$ we have $\rho(\alpha) \in \Pi$ and for all $a \in \mathcal{V}_\tau$ we have $\rho(a) \in \Lambda$. For ρ to be a substitution, we also require that $\rho(\chi) \neq \chi$ for only finitely many $\chi \in \mathcal{V}_\iota \cup \mathcal{V}_\sigma \cup \mathcal{V}_\tau$. We denote $\mathcal{S}$ the set of all the substitutions and $\mathrm{dom}(\rho) = \{\chi \mid \rho(\chi) \neq \chi\}$ the domain of the substitution ρ. In particular, the substitution $\rho_{id} \in \mathcal{S}$ is called the *identity* substitution and is defined as $\rho_{id}(\chi) = \chi$ for all $\chi \in \mathcal{V}_\iota \cup \mathcal{V}_\sigma \cup \mathcal{V}_\tau$.

Definition 2.5.37. For every $\rho \in \mathcal{S}$ we denote $\rho[x := v]$ the substitution remapping variable $x \in \mathcal{V}_\iota$ to $v \in \Lambda_\iota$ in ρ. In particular, $(\rho[x := v])(x) = v$ and $(\rho[x := v])(\chi) = \rho(\chi)$ if $\chi \neq x$. Similarly, we denote $\rho[\alpha := \pi]$ the substitution remapping $\alpha \in \mathcal{V}_\sigma$ to $\pi \in \Pi$ in ρ and $\rho[a := t]$ the substitution remapping $a \in \mathcal{V}_\tau$ to $t \in \Lambda$ in ρ. In the case where $\rho = \rho_{id}$ we will write $[x := v]$, $[\alpha := \pi]$ and $[a := t]$.

Definition 2.5.38. Let $\rho \in \mathcal{S}$ be a substitution and ψ be a value, term, stack or process. We denote $\psi\rho$ the value, term, stack or process formed by simultaneously substituting (without capture) every variable $\chi \in \mathrm{FV}(\psi)$ with $\rho(\chi)$ in ψ.

Definition 2.5.39. Given $\rho_1, \rho_2 \in \mathcal{S}$ we denote $\rho_1 \circ \rho_2$ the substitution formed by composing ρ_1 and ρ_2. It is defined by taking $(\rho_1 \circ \rho_2)(\chi) = (\rho_2(\chi))\rho_1$ for all $\chi \in \mathrm{dom}(\rho_2)$ and it coincides with ρ_1 on every other variables. In particular, if ψ is a value, term, stack or process we will have $\psi(\rho_1 \circ \rho_2) = (\psi\rho_2)\rho_1$.

Processes form the internal state of our abstract machine. They are to be thought of as a term put in some evaluation context represented using a stack. Intuitively, the stack π in the process $t * \pi$ contains the arguments to be fed to t. Since we are in call-by-value the stack also handles the storing of functions while their arguments are being evaluated. This is why we need stack frames (i.e., stacks of the form $[t]\pi$). The operational semantics of our language is given by a relation $(\succ)$ over processes.

Definition 2.5.40. The relation $(\succ) \subseteq (\Lambda \times \Pi) \times (\Lambda \times \Pi)$ is defined as the smallest relation satisfying the following reduction rules.

$$
\begin{array}{rcll}
t\ u * \pi & \succ & u * [t]\pi & \\
v * [t]\pi & \succ & t * v \cdot \pi & \text{when } v \notin \mathcal{V}_\iota \cup \{\Box\} \\
\lambda x.t * v \cdot \pi & \succ & t[x := v] * \pi & \\
\mu\alpha.t * \pi & \succ & t[\alpha := \pi] * \pi & \\
[\xi]t * \pi & \succ & t * \xi & \\
\{(l_i = v_i)_{i\in I}\}.l_k * \pi & \succ & v_k * \pi & \text{when } k \in I \\
[C_k[v] \mid (C_i[x_i] \to t_i)_{i\in I}] * \pi & \succ & t_i[x_i := v] * \pi & \text{when } k \in I \\
Y_{t,v} * \pi & \succ & t\ (\lambda x.Y_{t,x})\ v * \pi & \\
R_{\{(l_i = v_i)_{i\in I}\},u} * \pi & \succ & u * \pi & \\
\Box * [t]\pi & \succ & \Box * \pi & \\
\Box * v \cdot \pi & \succ & \Box * \pi & \\
[\Box \mid (C_i[x_i] \to t_i)_{i\in I}] * \pi & \succ & \Box * \pi & \\
\Box.l_k * \pi & \succ & \Box * \pi &
\end{array}
$$

We will denote $(\succ^+)$ its transitive closure, $(\succ^*)$ its reflexive-transitive closure and $(\succ^k)$ its k-fold application.

The first three rules are those that handle β-reduction. When the abstract machine encounters an application, the function is stored in a stack-frame in order to evaluate its argument first. Once the argument has been completely computed, a value faces the stack-frame containing the function. At this point the function can be evaluated and the value is stored

in the stack ready to be consumed by the function as soon as it evaluates to a λ-abstraction. A capture-avoiding substitution can then be performed to effectively apply the argument to the function. The fourth and fifth rules rules handle the classical part of computation. When a μ-abstraction is reached, the current stack is captured and substituted for the corresponding μ-variable. Conversely, when a term of the form $[\xi]t$ is reached, the current stack is discarded and evaluation resumes with the process $t * \xi$. In addition to the reduction rules for β-reduction and stack manipulation, we provide reduction rules for handling record projection, case analysis and recursion using a fixpoint operator.

A rule is then provided for reducing processes of the form $R_{v,u} * \pi$ to $u * \pi$ when the value v is a record. Note that if v is not a record then no reduction rule apply on processes of the form $R_{v,u} * \pi$. These facts will be used in Chapter 3 to show that records, λ-abstractions and other forms of values have a different computational behaviour in our abstract machine. The last four rules are used to handle the special value □. Intuitively, □ may consume the surrounding part of its computational environment when it consists of a term in function position, a value in argument position, a case analysis or a projection. This will be discussed further when defining the semantical interpretation of our type system in Chapter 4 and Chapter 6.

Remark 2.5.41. Our reduction relation ($\succ$) does not provide any way of reducing processes of the form $\delta_{v,w} * \pi$. We will give a reduction rule for such processes in Chapter 5. However, they can be considered as constants for now.

Theorem 2.5.42. Let $\rho \in \mathcal{S}$ be a substitution and $p, q \in \Lambda \times \Pi$ be two processes. If $p \succ q$ (resp. $p \succ^* q$, $p \succ^+ q$, $p \succ^k q$) then $p\rho \succ q\rho$ (resp. $p\rho \succ^* q\rho$, $p\rho \succ^+ q\rho$, $p\rho \succ^k q\rho$).

Proof. Case analysis on the reduction rules, all rules being local. □

2.6 Classification of processes

We are now going to give the vocabulary that will be used to describe some specific classes of processes. In particular we need to identify processes that are to be considered as the evidence of a successful computation, and those that are to be recognised as the expression of a failure of the machine (i.e., a crash).

Definition 2.6.43. A process $p \in \Lambda \times \Pi$ is said to be:
- *final* if $p = v * \varepsilon$ for some value $v \in \Lambda_\iota$,
- *δ-like* if $p = \delta_{v,w} * \pi$ for some values $v, w \in \Lambda_\iota$ and a stack $\pi \in \Pi$,
- *blocked* if there is no $q \in \Lambda \times \Pi$ such that $p \succ q$,
- *stuck* if it is not final nor δ-like and if $p\rho$ is blocked for every substitution ρ,

- *non-terminating* if there is an infinite sequence of processes $(p_i)_{i\in\mathbb{N}}$ such that $p_0 = p$ and for all $i \in \mathbb{N}$ we have $p_i \succ p_{i+1}$.

When a process becomes stuck, non-terminating or δ-like during its reduction, it will remain so forever. In particular, no substitution will ever be able to turn it into a process that might lead to a successful end of computation (i.e., reduce to a final process).

Lemma 2.6.44. Let $p \in \Lambda\times\Pi$ be a process and $\rho \in \mathcal{S}$ be a substitution. If p is final (resp. δ-like, stuck, non-terminating), then so is $p\rho$.

Proof. If p is final then $p = v * \varepsilon$ for some $v \in \Lambda_\iota$. Since $(v * \varepsilon)\rho = v\rho * \varepsilon\rho = v\rho * \varepsilon$ the process $p\rho$ is also final. If p is δ-like then $p = \delta_{v,w} * \pi$ for some $v, w \in \Lambda_\iota$ and $\pi \in \Pi$. Since $(\delta_{v,w} * \pi)\rho = (\delta_{v,w})\rho * \pi\rho = \delta_{v\rho,w\rho} * \pi\rho$ the process $p\rho$ is also δ-like. If p is stuck, then we suppose that there is a substitution $\rho_0 \in \mathcal{S}$ such that $(p\rho)\rho_0$ is not blocked. This contradicts the fact that p is stuck since $p(\rho_0\circ\rho) = (p\rho)\rho_0$ is not blocked, and hence the process $p\rho$ is stuck. Finally, if p is non-terminating then we have a sequence $(p_i)_{i\in\mathbb{N}}$ such that $p_0 = p$ and $p_i \succ p_{i+1}$ for all $i \in \mathbb{N}$. To show that $p\rho$ is non-terminating we need to construct a sequence $(q_i)_{i\in\mathbb{N}}$ such that $q_0 = p\rho$ and $q_i \succ q_{i+1}$ for all $i \in \mathbb{N}$. We can take $q_i = p_i\rho$ for all $i \in \mathbb{N}$. Indeed, we have $q_0 = p\rho$ since $p_0 = p$ and for all $i \in \mathbb{N}$ we have $p_i\rho \succ p_{i+1}\rho$ by Lemma 2.5.42 as $p_i \succ p_{i+1}$. □

Lemma 2.6.45. A process is stuck if and only if it is of one of the following forms, where n, m, $k \in \mathbb{N}$ and I, J, $K \subseteq_{\text{fin}} \mathbb{N}$ such that $k \notin K$.

$$C_n[v].l_m * \pi \qquad (\lambda x.t).l_m * \pi \qquad C_n[v] * w\,.\,\pi \qquad \{(l_i = v_i)_{i\in I}\} * v\,.\,\pi$$
$$[\lambda x.t \mid (C_i[x_i] \to t_i)_{i\in I}] * \pi \qquad [\{(l_i = v_i)_{i\in I}\} \mid (C_j[x_j] \to t_j)_{j\in J}] * \pi$$
$$[C_k[v] \mid (C_i[x_i] \to t_i)_{i\in K}] * \pi \qquad \{(l_i = v_i)_{i\in K}\}.l_k * \pi$$
$$R_{\lambda x.t,u} * \pi \qquad R_{C_n[v],u} * \pi \qquad R_{\square,u} * \pi$$

Proof. Using a simple case analysis we first rule out the thirteen forms of processes that immediately reduce using ($\succ$). As stuck processes are neither final nor δ-like, we can again rule out two forms of processes. We are now left with eighteen forms of processes, among which seven are not stuck (see the proof of Lemma 2.6.46). It is easy to see that the eleven remaining forms of processes are stuck. Indeed, given their structure no reduction rule will ever apply to them, even after a substitution. □

The proof of Lemma 2.6.45 has been (partially) checked using OCaml's exhaustivity checker for patterns. Indeed, the abstract syntax tree corresponding to our language can be encoded into OCaml data types easily. It is then possible to use pattern matching to enumerate

possible forms of processes in such a way that it is neither redundant nor incomplete (i.e., that the OCaml compiler does not complain with a warning). The OCaml source file used for this purpose is available online (http://lepigre.fr/these/classification.ml).

Lemma 2.6.46. A blocked process $p \in \Lambda \times \Pi$ is either stuck, final, δ-like, or of one of the following seven forms.

$$x.l_k * \pi \qquad x * \nu \,.\, \pi \qquad [x \mid (C_i[x_i] \to t_i)_{i \in I}] * \pi$$
$$x * [t]\pi \qquad R_{x,u} * \pi \qquad a * \pi \qquad \nu * \alpha$$

Proof. As for Lemma 2.6.45, we can rule out the thirteen forms of processes that immediately reduce using $(\succ)$, final processes and δ-like processes. After ruling out the eleven forms of stuck processes of Lemma 2.6.45 we are left with seven forms of processes. It remains to show that they are not stuck by finding a substitution $\rho \in \mathcal{S}$ unlocking their reduction. For processes of the first four forms we can take $\rho = [x := \Box]$ since we have respectively $\Box.l_k * \pi\rho \succ \Box * \pi\rho$, $\Box * \nu\rho \,.\, \pi\rho \succ \Box * \pi\rho$, $[\Box \mid (C_i[x_i] \to t_i\rho)_{i \in I}] * \pi \succ \Box * \pi\rho$ and $\Box * [t\rho]\pi\rho \succ \Box * \pi\rho$. For a process of the form $R_{x,u} * \pi$ we can take $\rho = [x := \{\}]$ as $R_{\{\},u\rho} * \pi\rho \succ u\rho * \pi\rho$. For a process of the form $a * \pi$ we can take $\rho = [a := \{l_k = \{\}\}.l_k]$ as $\{l_k = \{\}\}.l_k * \pi\rho \succ \{\} * \pi\rho$. For a process of the form $\nu * \alpha$ we can take $\rho = [\alpha := [\{\}]\varepsilon]$ as we will have $\nu\rho * [\{\}]\varepsilon \succ \{\} * \nu\rho \,.\, \varepsilon$ if $\nu \neq \Box$ and $\nu\rho * [\{\}]\varepsilon \succ \Box * \varepsilon$ otherwise. □

3 Observational equivalence of programs

In this chapter, we introduce an equivalence relation over programs. Two programs will be considered equivalent if and only if they have the same observable behaviour in terms of computation. General equational properties will then be derived for any equivalence relation satisfying specific compatibility conditions. These properties will then be essential for the definition of our realizability semantics in the next chapters. Moreover, they will be used for implementing a partial decision procedure for program equivalence.

3.1 Equivalence relation and properties

We will now consider a notion of *observational equivalence* over terms. More precisely, we will say that two terms are equivalent if and only if they have the same computational behaviour in every evaluation context. Using the formalism of our abstract machine, it is very easy to quantify over every such context. Indeed, it only amounts to quantifying over every stack. In this thesis, the considered observable behaviour is successful termination (versus non-termination or runtime error). We will consider that a process terminates successfully if it eventually reduces to a final process.

Definition 3.1.1. Let $R \subseteq (\Lambda\times\Pi)\times(\Lambda\times\Pi)$ be a relation such that for every final process $p \in \Lambda\times\Pi$, there is no $q \in \Lambda\times\Pi$ such that $p\,R\,q$. We say that a process $p \in \Lambda\times\Pi$ converges for the relation R, and we write $p \Downarrow_R$, if there is a final process $q \in \Lambda\times\Pi$ such that $p\,R^*q$. If p does not converge we say that it diverges (for the relation R) and we write $p \Uparrow_R$.

Note that the previous definition is rather general, but we will only use it with relations extending $(>)$. Of course, such extensions should not allow final processes to be reduced. Indeed, this would go against the intuition that they are the evidence of a successfully terminated computation.

The idea now is to relate terms that form converging processes against exactly the same stacks. However, quantifying only over stacks leads to free variables being undistinguishable. To avoid this problem, we not only quantify over stacks, but also over substitutions. In this way, our equivalence relation will work with both closed and open terms.

Definition 3.1.2. The relation $(\equiv_>) \subseteq \Lambda \times \Lambda$ is defined as follows.

$$(\equiv_>) = \left\{(t, u) \mid \forall \pi \in \Pi, \forall \rho \in \mathcal{S}, t\rho * \pi \Downarrow_> \Leftrightarrow u\rho * \pi \Downarrow_>\right\}$$

Lemma 3.1.3. $(\equiv_>)$ is an equivalence relation.

Proof. Immediate. □

We will now show that our equivalence relation $(\equiv_>)$ is well-behaved with respect to substitutions. First, we are going to check that arbitrary substitutions preserve equivalence. This property is summarised in the following theorem.

Theorem 3.1.4. Let $t, u \in \Lambda$ be two terms and $\rho \in \mathcal{S}$ be a substitution. If we have $t \equiv_> u$ then $t\rho \equiv_> u\rho$.

Proof. Let us take $\rho_0 \in \mathcal{S}$ and $\pi_0 \in \Pi$ and prove $(t\rho)\rho_0 * \pi_0 \Downarrow_> \Leftrightarrow (u\rho)\rho_0 * \pi_0 \Downarrow_>$, which can be rewritten as $t(\rho_0 \circ \rho) * \pi_0 \Downarrow_> \Leftrightarrow u(\rho_0 \circ \rho) * \pi_0 \Downarrow_>$. We can thus conclude using the definition of $t \equiv_> u$ with the substitution $\rho_0 \circ \rho$ and the stack π_0. □

Another essential property of our equivalence relation is extensionality. In other words, it is possible to replace equals by equals at any place in terms without changing their observed behaviour. This property is expressed in the following two theorems.

Theorem 3.1.5. Let $v_1, v_2 \in \Lambda_\iota$ be values, $t \in \Lambda$ be a term and $x \in \mathcal{V}_\iota$ be a λ-variable. If $v_1 \equiv_> v_2$ then $t[x := v_1] \equiv_> t[x := v_2]$.

Proof. We are going to prove the contrapositive so we suppose $t[x := v_1] \not\equiv_> t[x := v_2]$ and we show $v_1 \not\equiv_> v_2$. Let us first assume that neither v_1 nor v_2 is equal to $\Box$ or to a λ-variable. By definition, we know that there is a stack π and a substitution ρ such that we have $(t[x := v_1])\rho * \pi \Downarrow_>$ and $(t[x := v_2])\rho * \pi \Uparrow_>$ (up to symmetry). As x is bound we can rename it so that $(t[x := v_1])\rho = t\rho[x := v_1\rho]$ and $(t[x := v_2])\rho = t\rho[x := v_2\rho]$. To finish the proof, we need to find a stack π_0 and a substitution ρ_0 such that $v_1\rho_0 * \pi_0 \Downarrow_>$ and $v_2\rho_0 * \pi_0 \Uparrow_>$ (up to symmetry). We can take $\pi_0 = [\lambda x.t\rho]\pi$ and $\rho_0 = \rho$ since by definition we have $v_1\rho * [\lambda x.t\rho]\pi \succ^* t\rho[x := v_1\rho] * \pi \Downarrow_>$ and $v_2\rho * [\lambda x.t\rho]\pi \succ^* t\rho[x := v_2\rho] * \pi \Uparrow_>$. Note that

here, it is essential that $v_1\rho$ and $v_2\rho$ are not equal to $\Box$ or to some λ-variable as otherwise the first reduction steps could not be taken.

It remains to show that $v_1 \not\equiv_> v_2$ in the cases where v_1, v_2 or both are equal to $\Box$ or a λ-variable. First, we can assume that $v_1 \neq v_2$ as otherwise we would immediately get a contradiction with $t[x := v_1] \not\equiv_> t[x := v_2]$ by reflexivity of $(\equiv_>)$. As a consequence, we cannot have $v_1 = v_2 = \Box$ or $v_1 = v_2 = x \in \mathcal{V}_\iota$. Now, if $v_1 = \Box$ and $v_2 \notin \mathcal{V}_\iota$ then we can distinguish them using the stack $\pi = [\{\}]\varepsilon$. Indeed, we have $\Box * [\{\}]\varepsilon \succ \Box * \varepsilon \Downarrow_>$ and $v_2 * [\{\}]\varepsilon \succ \{\} * v_2 \cdot \varepsilon \Uparrow_>$ since we cannot have $v_2 = \Box$. In a symmetric way, we can also distinguish v_1 and v_2 when $v_2 = \Box$ and $v_1 \notin \mathcal{V}_\iota$. Now, if we have $v_1 = \Box$ and $v_2 = x \in \mathcal{V}_\iota$ then we can tell them apart using the substitution $\rho = [x := \{\}]$ and the stack $\pi = [\{\}]\varepsilon$. Indeed, we have $v_1\rho * \pi = \Box * [\{\}]\varepsilon \succ \Box * \varepsilon \Downarrow_>$ and $v_2\rho * \pi = \{\} * [\{\}]\varepsilon \succ \{\} * [\{\}]\varepsilon \Uparrow_>$. The same goes if $v_1 = x \in \mathcal{V}_\iota$ and $v_2 = \Box$. Now, if $v_1 = x_1 \in \mathcal{V}_\iota$ and $v_2 = x_2 \in \mathcal{V}_\iota$ then they can be easily distinguished with Theorem 3.1.4 and a substitution replacing x_1 and x_2 with two non-equivalent values. If we have $v_1 = x \in \mathcal{V}_\iota$ and v_2 is neither a variable nor $\Box$ then we can use the substitution $\rho = [x := \Box]$ and the stack $\pi = [\{\}]\varepsilon$ to tell v_1 and v_2 apart. Indeed, in this case we have $v_1\rho * \pi = \Box * [\{\}]\varepsilon \succ \Box * \varepsilon \Downarrow_>$ and $v_2\rho * \pi = v_2\rho * [\{\}]\varepsilon \succ \{\} * v_2\rho \cdot \varepsilon \Uparrow_>$. A symmetric reasoning can be used in the case where v_1 is not $\Box$ nor a variable and $v_2 = x \in \mathcal{V}_\iota$. □

Lemma 3.1.6. Let $p \in \Lambda\times\Pi$ be a process, $a \in \mathcal{V}_\tau$ be a term variable and t be a term such that $p[a := t] \Downarrow_>$. Either there is a value $v \in \Lambda_\iota$ such that $p \succ^* v * \varepsilon$ or there is a stack $\pi \in \Pi$ such that $p \succ^* a * \pi$.

Proof. There must be a blocked process q such that $p \succ^* q$. If it were not the case p would be non-terminating, and hence $p[a := t]$ would also be non-terminating according to Lemma 2.6.44. This would hence contradict the hypothesis that $p[a := t] \Downarrow_>$. Since $p \succ^* q$ we obtain $p[a := t] \succ^* q[a := t]$ by Theorem 2.5.42 and hence $q[a := t] \Downarrow_>$. We can then proceed by case analysis according to Lemma 2.6.46 as q is blocked. If q is not of the form $v * \varepsilon$ nor of the form $a * \pi$ then $q[a := t]$ is a blocked process that is not final. Consequently, we obtain a contradiction with $q[a := t] \Downarrow_>$. □

Theorem 3.1.7. Let u_1, u_2 and $t \in \Lambda$ be three terms and $a \in \mathcal{V}_\tau$ be a term variable. If $u_1 \equiv_> u_2$ then $t[a := u_1] \equiv_> t[a := u_2]$.

Proof. Let us suppose $u_1 \equiv_> u_2$ and show $t[a := u_1] \equiv_> t[a := u_2]$. We take a stack π, a substitution ρ and we show $(t[a := u_1])\rho * \pi \Downarrow_> \Leftrightarrow (t[a := u_2])\rho * \pi \Downarrow_>$. As we are free to rename a we may assume $(t[a := u_1])\rho = t\rho[a := u_1\rho]$, $(t[a := u_2])\rho = t\rho[a := u_2\rho]$ and $a \notin FV(\pi)$. Consequently our goal is now $t\rho[a := u_1\rho] * \pi \Downarrow_> \Leftrightarrow t\rho[a := u_2\rho] * \pi \Downarrow_>$. By symmetry we can suppose that $t\rho[a := u_1\rho] * \pi \Downarrow_>$ and show $t\rho[a := u_2\rho] * \pi \Downarrow_>$. Let us now

consider the reduction of the process $t\rho * \pi$. According to Lemma 3.1.6 there are two possibilities.

- If $t\rho * \pi \succ^* \nu * \varepsilon$ for some value $\nu \in \Lambda_\iota$ then $(t\rho * \pi)[a := u_2] \succ^* (\nu * \varepsilon)[a := u_2]$ by Theorem 2.5.42. This rewrites to $t\rho[a := u_2] * \pi \succ^* \nu[a := u_2] * \varepsilon$ as $a \notin \mathrm{FV}(\pi)$, and hence we obtain $t\rho[a := u_2] * \pi \Downarrow_{\succ}$.
- If $t\rho * \pi \succ^* a * \pi_0$ for some stack $\pi_0 \in \Pi$ then $(t\rho * \pi)[a := u_1] \succ^* (a * \pi_0)[a := u_1]$ and $(t\rho * \pi)[a := u_2] \succ^* (a * \pi_0)[a := u_2]$ by Theorem 2.5.42. We thus assume that $(a * \pi_0)[a := u_1] \Downarrow_{\succ}$ and show $(a * \pi_0)[a := u_2] \Downarrow_{\succ}$. We will build a sequence of stacks $(\pi_i)_{i \leqslant n}$ starting with π_0 and such that $(a * \pi_i)[a := u_1] \succ^+ (a * \pi_{i+1})[a := u_1]$ for all $i < n$. Note that the sequence has to be finite since otherwise there would be an infinite sequence of reductions from $(a * \pi_0)[a := u_1]$. To define π_{i+1} we consider the process $u_1 * \pi_i$. We have $(a * \pi_0)[a := u_1] \succ^* (a * \pi_i)[a := u_1] = (u_1 * \pi_i)[a := u_1]$ by transitivity and hence $(u_1 * \pi_i)[a := u_1] \Downarrow_{\succ}$. According to Lemma 3.1.6 there are two possibilities for the reduction of $u_1 * \pi_i$. Either $u_1 * \pi_i \succ^* \nu * \varepsilon$ for some value ν and the sequence ends with $n = i$, or $u_1 * \pi_i \succ^* a * \xi$ for some stack ξ and in this case we take $\pi_{i+1} = \xi$.

  To end the proof, we will now show $(a * \pi_i)[a := u_2] \Downarrow_{\succ}$ for all $i \leqslant n$. For $i = 0$ this will give us $(a * \pi_0)[a := u_2] \Downarrow_{\succ}$ which is the expected result. For $i = n$ we know that $u_1 * \pi_n \succ^* \nu * \varepsilon \Downarrow_{\succ}$, and hence $u_2 * \pi_n \Downarrow_{\succ}$ since $u_1 \equiv_{\succ} u_2$. As a consequence, we obtain that $(a * \pi_n)[a := u_2] = (u_2 * \pi_n)[a := u_2] \Downarrow_{\succ}$ by Lemma 2.6.44. Let us now suppose that we have $(a * \pi_{i+1})[a := u_2] = u_2 * \pi_{i+1}[a := u_2] \Downarrow_{\succ}$ for some $i < n$ and show that $(a * \pi_i)[a := u_2] = u_2 * \pi_i[a := u_2] \Downarrow_{\succ}$. Since we know $u_1 \equiv_{\succ} u_2$ we can deduce that $u_1 * \pi_{i+1}[a := u_2] \Downarrow_{\succ}$. Moreover, since $u_1 * \pi_i \succ^* a * \pi_{i+1}$ we may use Theorem 2.5.42 to obtain $u_1 * \pi_i[a := u_2] = (u_1 * \pi_i)[a := u_2] \succ^* (a * \pi_{i+1})[a := u_2] = u_2 * \pi_{i+1}[a := u_2]$. As a consequence, we have $u_1 * \pi_i[a := u_2] \Downarrow_{\succ}$ from which we obtain $u_2 * \pi_i[a := u_2] \Downarrow_{\succ}$ since $u_1 \equiv_{\succ} u_2$. □

3.2 Compatible equivalence relations

The purpose of the theorems given in the previous section was to show that $(\equiv_{\succ})$ is a form of congruence. This property will be essential to the construction of our realizability model and semantics in the following chapter.

Definition 3.2.8. An equivalence relation $(\equiv) \subseteq \Lambda \times \Lambda$ is said to be a congruence if it satisfies the following conditions.

- Given $t, u \in \Lambda$, if $t \equiv u$ then for all $\rho \in \mathcal{S}$ we have $t\rho \equiv u\rho$.
- Given $\nu_1, \nu_2 \in \Lambda_\iota$, $t \in \Lambda$ and $x \in \mathcal{V}_\iota$, if $\nu_1 \equiv \nu_2$ then $t[x := \nu_1] \equiv t[x := \nu_2]$.
- Given $u_1, u_2, t \in \Lambda$ and $a \in \mathcal{V}_\tau$, if $u_1 \equiv u_2$ then $t[a := u_1] \equiv t[a := u_2]$.

Theorem 3.2.9. The relation $(\equiv_{\succ})$ is a congruence.

Proof. Combination of Theorem 3.1.4, Theorem 3.1.5 and Theorem 3.1.7.

□

The precise definition of our equivalence relation will not play a direct role in our construction. In fact, we will be able to build several different models using several different equivalence relations. To use such an equivalence, we will only have to make sure that it is a congruence and that it is compatible with $(\equiv_>)$ in some sense. Of course, this property will hold for $(\equiv_>)$ itself, but we will eventually need to use a different, compatible equivalence relation starting from Chapter 5.

Definition 3.2.10. An equivalence relation $(\equiv) \subseteq \Lambda\times\Lambda$ is said to be compatible with $(\equiv_>)$, or (simply) compatible, if it satisfies the following conditions.

- For all terms $t, u \in \Lambda$ such that $t \equiv u$ we have $t \equiv_> u$ (i.e., $(\equiv) \subseteq (\equiv_>)$).
- Given $t, u \in \Lambda$, if for every stack $\pi \in \Pi$ there is a process $p \in \Lambda\times\Pi$ such that both $t * \pi \succ^* p$ and $u * \pi \succ^* p$, then $t \equiv u$.
- Let t_1, t_2 be arbitrary terms such that $x \in FV_\iota(t_1) \cap FV_\iota(t_2)$. If there is a closed term $u \in \Lambda^*$ such that $t_1[x := v] * [u]\pi \succ^* v * \pi$ and $t_2[x := v] * [u]\pi \succ^* v * \pi$ for all $v \in \Lambda_\iota$ and $\pi \in \Pi$, then $t_1[x := v_1] \equiv t_2[x := v_2]$ implies $v_1 \equiv v_2$ for all $v_1, v_2 \in \Lambda_\iota$.

Note that the second and third conditions require the equivalence to include the $(\succ)$ reduction relation in some sense. Of course, this is only an intuition as the equivalence relation ranges over terms while the reduction relation ranges over processes.

Remark 3.2.11. In the third condition of Definition 3.2.10, the role of the term u is to extract (or project out) the value v from $t_1[x := v]$ and $t_2[x := v]$, along the relation $(\succ)$.

Theorem 3.2.12. The relation $(\equiv_>)$ is compatible.

Proof. The first condition is immediate. To prove the second condition let us suppose that for all $\pi \in \Pi$ we have a $p \in \Lambda\times\Pi$ such that $t * \pi \succ^* p$ and $u * \pi \succ^* p$. Let us take $\pi_0 \in \Pi$, $\rho_0 \in \mathcal{S}$ and show that we have $t\rho_0 * \pi_0 \Downarrow_> \Leftrightarrow u\rho_0 * \pi_0 \Downarrow_>$. We now consider a renaming substitution σ mapping every variable of $FV(\pi_0)$ to a distinct fresh variable. Note that σ has an inverse σ^{-1} mapping $\sigma(\chi)$ to χ for all $\chi \in dom(\tau)$. We thus obtain $t\rho_0 * \pi_0 = (t\rho_0 * \pi_0\sigma)\sigma^{-1} = (t * \pi_0\sigma)(\rho_0 \circ \sigma^{-1})$ and $u\rho_0 * \pi_0 = (u * \pi_0\sigma)(\rho_0 \circ \sigma^{-1})$. Our hypothesis then gives us a common reduct p_0 of $t * \pi_0\sigma$ and $u * \pi_0\sigma$. We can thus use Lemma 2.5.42 to conclude, since it gives us $t\rho_0 * \pi_0 = (t * \pi_0\sigma)(\rho_0 \circ \sigma^{-1}) \succ^* p_0(\rho_0 \circ \sigma^{-1})$ and $u\rho_0 * \pi_0 = (u * \pi_0\sigma)(\rho_0 \circ \sigma^{-1}) \succ^* p_0(\rho_0 \circ \sigma^{-1})$.

Let us now prove the third condition. We take $t_1, t_2 \in \Lambda$ and we suppose that we have $u \in \Lambda^*$ such that $t_1[x := v] * [u]\pi \succ^* v * \pi$ and $t_2[x := v] * [u]\pi \succ^* v * \pi$ for all $v \in \Lambda_\iota$ and $\pi \in \Pi$. Let us take $v_1, v_2 \in \Lambda_\iota$ such that $v_1 \not\equiv_> v_2$ and show that $t_1[x := v_1] \not\equiv_> t_2[x := v_2]$. By

definition we know that there is $\pi_0 \in \Pi$ and $\rho_0 \in \mathcal{S}$ such that $\nu_1\rho_0 * \pi_0 \Downarrow_>$ and $\nu_2\rho_0 * \pi_0 \Uparrow_>$ (up to symmetry). We need to find $\pi_1 \in \Pi$ and $\rho_1 \in \mathcal{S}$ such that $(t_1[x := \nu_1])\rho_1 * \pi_1 \Downarrow_>$ and $(t_2[x := \nu_2])\rho_1 * \pi_1 \Uparrow_>$. We consider a renaming substitution σ mapping the free variable of π_0 to distinct fresh variables, and we denote σ^{-1} its inverse. We will now show that $\pi_1 = [u]\pi_0\sigma$ and $\rho_1 = \rho_0 \circ \sigma^{-1}$ are suitable. According to our main hypothesis, we have $t_1[x := \nu_1] * \pi_1 \succ^* \nu_1 * \pi_0\sigma$ and $t_2[x := \nu_2] * \pi_1 \succ^* \nu_2 * \pi_0\sigma$. We can thus use Lemma 2.5.42 to conclude with $(t_1[x := \nu_1])\rho_0 * [u]\pi_0 = (t_1[x := \nu_1] * \pi_1)\rho_1 \succ^* (\nu_1 * \pi_0\sigma)\rho_1 = \nu_1\rho_0 * \pi_0 \Downarrow_>$, and similarly $(t_2[x := \nu_2])\rho_0 * \pi_0 \succ^* \nu_2\rho_0 * \pi_0 \Uparrow_>$. □

From now on and until the end of the current chapter we will consider properties of compatible equivalences in general. In particular, we will use the symbol $(\equiv)$ to denote an arbitrary compatible equivalence. Moreover, we will always assume that $(\equiv)$ is a congruence. Although this property is rarely required in this chapter, it will be absolutely necessary for using an equivalence relation in the definition of our semantics.

3.3 Equivalences from reduction

The main aim of a compatible equivalence relation $(\equiv)$ is to identify terms with the same computational behaviour. In particular, taking reduction steps does not fundamentally change the observable computational behaviour of a term. We can thus derive several primitive equivalences using the second property of compatible equivalences (Definition 3.2.10), that is recalled below as a lemma.

Lemma 3.3.13. Let $t, u \in \Lambda$ be two terms. If for all stacks $\pi \in \Pi$ there is a process $p \in \Lambda\times\Pi$ such that $t * \pi \succ^* p$ and $u * \pi \succ^* p$ then $t \equiv u$.

Proof. By definition of a compatible equivalence. □

Lemma 3.3.14. Let $t, u \in \Lambda$ be two terms. If we have $t * \pi \succ^* u * \pi$ for every stack $\pi \in \Pi$ then $t \equiv u$.

Proof. Direct consequence of Lemma 3.3.13 using $p = u * \pi$. □

Based on the reduction rules of our abstract machine, we can derive six immediate equivalences. They correspond to record projection, case analysis, unfolding of the fixpoint combinator, elimination of special terms of the form $R_{\nu,u}$, and the erasure of a projection or a case analysis by the special value $\square$. The corresponding reduction rules do not involve an interaction with the stack. In particular, they do not "observe" the stack and they leave it unchanged.

Theorem 3.3.15. Let $I \subseteq_{\text{fin}} \mathbb{N}$ be a finite set of indices such that $v_i \in \Lambda_\iota$, $x_i \in \mathcal{V}_\iota$ and $t_i \in \Lambda$ for all $i \in I$. Let $v \in \Lambda_\iota$ be a value, $x \in \mathcal{V}_\iota$ be a λ-variable and $t, u \in \Lambda$ be terms. The following equivalences hold.

$$\{(l_i = v_i)_{i\in I}\}.l_k \equiv v_k \quad \text{if} \quad k \in I \qquad\qquad Y_{t,v} \equiv t\ (\lambda x.Y_{t,x})\ v$$

$$[C_k[v] \mid (C_i[x_i] \to t_i)_{i\in I}] \equiv t_k[x_k := v] \quad \text{if} \quad k \in I \qquad\qquad R_{\{(l_i=v_i)_{i\in I}\},u} \equiv u$$

$$\square.l_k \equiv \square \qquad\qquad [\square \mid (C_i[x_i] \to t_i)_{i\in I}] \equiv \square$$

Proof. For every equivalence $t_1 \equiv t_2$ that we need to prove, the definition of $(\succ)$ gives us $t_1 * \pi \succ t_2 * \pi$ for all $\pi \in \Pi$. As a consequence, we can use Lemma 3.3.14. □

To go a little bit further, we can look at the first three rules of $(\succ)$, which are used to handle β-reduction (see Definition 2.5.40). In particular, these rules can be composed immediately when a λ-abstraction is applied to a value. This corresponds exactly to a call-by-value β-reduction step. Similar reasoning can be used to obtain two other equivalences involving the special value $\square$.

Theorem 3.3.16. For every $x \in \mathcal{V}_\iota$, $t \in \Lambda$ and $v \in \Lambda_\iota$ such that $v \neq \square$ and $v \notin \mathcal{V}_\iota$ we have $(\lambda x.t)\ v \equiv t[x := v]$.

Proof. For all $\pi \in \Pi$ we have $(\lambda x.t)\ v * \pi \succ v * [\lambda x.t]\pi \succ \lambda x.t * v\,.\,\pi \succ t[x := v] * \pi$, we can thus conclude using Lemma 3.3.14. Note that the second reduction step can only be taken because v is not equal to $\square$ or to a λ-variable. □

Theorem 3.3.17. The equivalence $\square\ v \equiv \square$ holds for every value $v \in \Lambda_\iota$ that is not a λ-variable. Similarly, the equivalence $t\ \square \equiv \square$ holds for every term $t \in \Lambda$.

Proof. If $v \neq \square$ then for all $\pi \in \Pi$ we have $\square\ v * \pi \succ v * [\square]\pi \succ \square * v\,.\,\pi \succ \square * \pi$ since v is not a λ-variable. If $v = \square$ then for all $\pi \in \Pi$ we have $\square\ \square * \pi \succ \square * [\square]\pi \succ \square * \pi$. In both cases Lemma 3.3.14 gives us $\square\ v \equiv \square$. For all $\pi \in \Pi$ we have $t\ \square * \pi \succ \square * [t]\pi \succ \square * \pi$ and thus we can also use Lemma 3.3.14 to obtain $t\ \square \equiv \square$. □

The reduction rule for processes of the form $\mu\alpha.t * \pi$ is inherently non-local. Indeed, the bound μ-variable can be substituted anywhere in t and any reduction rule may apply afterwards (depending on the form of t and π). However, we can still derive equivalences corresponding to the usual reduction rules of the $\lambda\mu$-calculus [*Parigot 1992*]. In particular, we can obtain two equivalences corresponding to call-by-value *structural reductions*.

Theorem 3.3.18. For every $t, u \in \Lambda$ and $\alpha, \beta \in \mathcal{V}_\sigma$ such that $\beta \notin \mathrm{FV}(t)$ and $\beta \notin \mathrm{FV}(u)$ we have $t\ (\mu\alpha.u) \equiv \mu\beta.t\ u[\alpha := [t]\beta]$.

Proof. Since for every stack π we have $t\ (\mu\alpha.u) * \pi \succ \mu\alpha.u * [t]\pi \succ u[\alpha := [t]\pi] * [t]\pi$ and $\mu\beta.t\ u[\alpha := [t]\beta] * \pi \succ t\ u[\alpha := [t]\pi] * \pi \succ u[\alpha := [t]\pi] * [t]\pi$ we can conclude using Lemma 3.3.13. □

Theorem 3.3.19. For every $t \in \Lambda$, $\alpha, \beta \in \mathcal{V}_\sigma$ and $\nu \in \Lambda_\iota$ such that $\beta \notin \mathrm{FV}(t) \cup \mathrm{FV}(\nu)$ and $\nu \notin \mathcal{V}_\iota \cup \{\Box\}$ we have $(\mu\alpha.t)\ \nu \equiv \mu\beta.t[\alpha := \nu \cdot \beta]\ \nu$.

Proof. Since we have $(\mu\alpha.t)\ \nu * \pi \succ \nu * [\mu\alpha.t]\pi \succ \mu\alpha.t * \nu \cdot \pi \succ t[\alpha := \nu \cdot \pi] * \nu \cdot \pi$ and $\mu\beta.t[\alpha := \nu \cdot \beta]\ \nu * \pi \succ t[\alpha := \nu \cdot \pi]\ \nu * \pi \succ \nu * [t[\alpha := \nu \cdot \pi]]\pi \succ t[\alpha := \nu \cdot \pi] * \nu \cdot \pi$ for all stack π, we can conclude using Lemma 3.3.13. □

Similarly, the following theorem provides an equivalence corresponding to *renaming*. Note that our version of renaming is more general than the one found in [*Parigot 1992*] as our formalism includes stacks. Indeed, only μ-variables can be used in named terms in the original version of the $\lambda\mu$-calculus.

Theorem 3.3.20. For every $\xi \in \Pi$, $\beta \in \mathcal{V}_\sigma$ and $t \in \Lambda$ we have $[\xi]\mu\beta.t \equiv [\xi]t[\beta := \xi]$.

Proof. As $[\xi]\mu\beta.t * \pi \succ \mu\beta.t * \xi \succ t[\beta := \xi] * \xi$ and $[\xi]t[\beta := \xi] * \pi \succ t[\beta := \xi] * \xi$ we can conclude using Lemma 3.3.13. □

As a named term has the effect of erasing the whole stack, terms that are applied (as functions) to a named term can always be removed. Similarly, values used as arguments of a named term can be removed as they will never be considered. The following two theorems will hence allow us to discard unnecessary subterms as early as possible, when attempting to prove an equivalence.

Theorem 3.3.21. For every $t \in \Lambda$, $\xi \in \Pi$ and $\nu \in \Lambda_\iota$ such that $\nu \notin \mathcal{V}_\iota \cup \{\Box\}$ the equivalence $([\xi]t)\ \nu \equiv [\xi]t$ holds.

Proof. Since for every stack π we have $([\xi]t)\ \nu * \pi \succ \nu * [[\xi]t]\pi \succ [\xi]t * \nu \cdot \pi \succ t * \xi$ and $[\xi]t * \pi \succ t * \xi$ we can conclude using Lemma 3.3.13. □

Theorem 3.3.22. For every $t, u \in \Lambda$ and $\xi \in \Pi$ we have $t\ ([\xi]u) \equiv [\xi]u$.

Proof. Since we have $t\ ([\xi]u) * \pi \succ [\xi]u * [t]\pi \succ u * \xi$ and $[\xi]u * \pi \succ u * \xi$ for every stack π, we can conclude using Lemma 3.3.13. □

We can also remark that using two consecutive μ-abstractions leads to saving the same stack twice. We can thus obtain the same computational behaviour by saving the

stack only once. Similarly, using two named terms in a row is the same as using only the later one.

Theorem 3.3.23. For every $\alpha, \beta \in \mathcal{V}_\sigma$ and $t \in \Lambda$ we have $\mu\alpha.\mu\beta.t \equiv \mu\beta.t[\alpha := \beta]$.

Proof. We have $\mu\alpha.\mu\beta.t * \pi \succ \mu\beta.t[\alpha := \pi] * \pi \succ (t[\alpha := \pi])[\beta := \pi] * \pi$ and we also have $\mu\beta.t[\alpha := \beta] * \pi \succ (t[\alpha := \beta])[\beta := \pi] * \pi$. To be able to conclude using Lemma 3.3.13 we need to show that $(t[\alpha := \pi])[\beta := \pi] = (t[\alpha := \beta])[\beta := \pi]$. This is immediate as we may assume $\alpha \notin \mathrm{FV}_\sigma(\pi)$ and $\beta \notin \mathrm{FV}_\sigma(\pi)$ up to renaming. □

Theorem 3.3.24. For every $\xi_1, \xi_2 \in \Pi$ and $t \in \Lambda$ we have $[\xi_1][\xi_2]t \equiv [\xi_2]t$.

Proof. As we have $[\xi_1][\xi_2]t * \pi \succ [\xi_2]t * \xi_1 \succ t * \xi_2$ and $[\xi_2]t * \pi \succ t * \xi_2$ for all π we can use Lemma 3.3.13. □

Finally, we provide two last equivalences allowing the simplification of terms involving μ-abstractions. For instance, if the variable bound by a μ-abstraction does not occur in its body, then it can be removed. Similarly, it is not useful to restore a stack right after it has been saved.

Theorem 3.3.25. For every $\alpha \in \mathcal{V}_\sigma$ and $t \in \Lambda$ such that $\alpha \notin \mathrm{FV}_\sigma(t)$ we have $\mu\alpha.t \equiv t$.

Proof. As we have $\mu\alpha.t * \pi \succ t[\alpha := \pi] * \pi = t * \pi$ we can use Lemma 3.3.14. □

Theorem 3.3.26. For every $\alpha \in \mathcal{V}_\sigma$ and $t \in \Lambda$ we have $\mu\alpha.[\alpha]t \equiv \mu\alpha.t$.

Proof. As $\mu\alpha.[\alpha]t * \pi \succ [\pi]t[\alpha := \pi] * \pi = t[\alpha := \pi] * \pi$ and $\mu\alpha.t * \pi \succ t[\alpha := \pi] * \pi$ for all π we can use Lemma 3.3.13. □

Note that we can compose the previous two theorems to obtain $\mu\alpha.[\alpha]t \equiv t$ in the case where $\alpha \notin \mathrm{FV}_\sigma(t)$. This can be seen as a form of η-equivalence for μ-abstraction, as remarked by Michel Parigot in [*Parigot 1992*].

3.4 Inequivalences from counter-examples

Using the theorems of the previous section, it is possible to derive equivalences in a direct way. However, we will sometimes need to reason in an indirect way by exhibiting a contradiction. We will hence provide several means of identifying inequivalences. The idea here is to rely on the first property of compatible equivalences (Definition 3.2.10), which is recalled in the following lemma.

Lemma 3.4.27. For all terms $t, u \in \Lambda$ such that $t \equiv u$ we have $t \equiv_> u$. This exactly means that we have $(\equiv) \subseteq (\equiv_>)$.

Proof. By definition of a compatible equivalence. □

We will now state two lemmas that will be convenient for proving inequivalences using the definition of $(\succ)$. The former will in fact be exactly equivalent to Lemma 3.4.27 (through its contrapositive), and the latter will directly follow.

Definition 3.4.28. Given t and $u \in \Lambda$, the negations of $t \equiv_> u$ and $t \equiv u$ are respectively denoted $t \not\equiv_> u$ and $t \not\equiv u$.

Lemma 3.4.29. Let $t, u \in \Lambda$ be two terms. If there is a stack $\pi \in \Pi$ and a substitution $\rho \in \mathcal{S}$ such that $t\rho * \pi \Downarrow_> p$ and $u\rho * \pi \Uparrow_> p$ then $t \not\equiv u$.

Proof. By definition of $(\equiv_>)$ we have $t \not\equiv_> u$. We can thus conclude that $t \not\equiv u$ using (the contrapositive of) Lemma 3.4.27. □

Lemma 3.4.30. Let $t, u \in \Lambda$ be two terms. If there is a stack $\pi \in \Pi$ such that we have $t * \pi \Downarrow_>$ and $u * \pi \Uparrow_>$ then $t \not\equiv u$.

Proof. Immediate consequence of Lemma 3.4.29 using π and $\rho = \rho_{id}$. □

We will now start by showing that records with different fields cannot be equivalent. Similarly, we will show that variants with different constructors are never equivalent. In both cases, it is not difficult to find a stack distinguishing the two values.

Theorem 3.4.31. Let $m, n \in \mathbb{N}$ be two natural numbers and $v, w \in \Lambda_\iota$ be two values. If we have $m \neq n$ then $C_m[v] \not\equiv C_n[w]$.

Proof. We can apply Lemma 3.4.30 with $\pi = [\lambda y.[y \mid C_m[z] \to \{\}]]\varepsilon$. Indeed, $C_m[v] * \pi \Downarrow_>$ since $C_m[v] * \pi \succ \lambda y.[y \mid C_m[z] \to \{\}] * C_m[v] \cdot \varepsilon \succ [C_m[v] \mid C_m[z] \to \{\}] * \varepsilon \succ \{\} * \varepsilon$. Moreover, we have $C_n[w] * \pi \Uparrow_>$ as $C_n[v] * \pi \succ \lambda y.[y \mid C_m[z] \to \{\}] * C_n[w] \cdot \varepsilon \succ [C_n[w] \mid C_m[z] \to \{\}] * \varepsilon$ and $[C_n[w] \mid C_m[z] \to \{\}] * \varepsilon$ is stuck since $m \neq n$. □

Theorem 3.4.32. Let I_1, I_2 be two sets of indices such that for all $i \in I_1$ we have $v_i \in \Lambda_\iota$ and for all $i \in I_2$ we have $w_i \in \Lambda_\iota$. If $I_1 \neq I_2$ then $\{(l_i = v_i)_{i \in I_1}\} \not\equiv \{(l_i = w_i)_{i \in I_2}\}$.

Proof. Since $I_1 \neq I_2$ there must be k such that $k \in I_1$ and $k \notin I_2$ (up to symmetry). We can hence apply Lemma 3.4.30 with the stack $\pi = [\lambda x.x.l_k]\varepsilon$. Indeed, $\{(l_i = v_i)_{i \in I_1}\} * \pi \Downarrow_>$ since

$\{(l_i = v_i)_{i\in I_1}\} * \pi \succ \lambda x.x.l_k * \{(l_i = v_i)_{i\in I_1}\} . \varepsilon \succ \{(l_i = v_i)_{i\in I_1}\}.l_k * \varepsilon \succ v_k * \varepsilon$. Moreover, we have $\{(l_i = w_i)_{i\in I_2}\} * \pi \succ \lambda x.x.l_k * \{(l_i = w_i)_{i\in I_2}\} . \varepsilon \succ \{(l_i = w_i)_{i\in I_2}\}.l_k * \varepsilon$, and as $k \notin I_2$ we know that $\{(l_i = w_i)_{i\in I_2}\}.l_k * \varepsilon$ is stuck. □

Similarly, we can show that a record and a variant can never be equivalent. Indeed, pattern-matching on a record will lead to a stuck state.

Theorem 3.4.33. Let $m \in \mathbb{N}$ be a natural number, $v \in \Lambda_\iota$ be a value and I be a finite set of indices such that $v_i \in \Lambda_\iota$ for all $i \in I$. We have $C_m[v] \not\equiv \{(l_i = v_i)_{i\in I}\}$.

Proof. We can apply Lemma 3.4.30 using the stack $\pi = [\lambda y.[y \mid C_m[z] \to \{\}]]\varepsilon$. Indeed, we have $C_m[v] * \pi \Downarrow_\succ$ like in the proof of Theorem 3.4.31. Moreover, $\{(l_i = v_i)_{i\in I}\} * \pi \Uparrow_\succ$ since $\{(l_i = v_i)_{i\in I}\} * \pi \succ \lambda y.[y \mid C_m[z] \to \{\}] * \{(l_i = v_i)_{i\in I}\} . \varepsilon \succ [\{(l_i = v_i)_{i\in I}\} \mid C_m[z] \to \{\}] * \varepsilon$ and the process $[\{(l_i = v_i)_{i\in I}\} \mid C_m[z] \to \{\}] * \varepsilon$ is stuck. □

Theorem 3.4.34. Let $m \in \mathbb{N}$ be a natural number, $v \in \Lambda_\iota$ be a value, $x \in \mathcal{V}_\iota$ be a λ-variable and $t \in \Lambda$ be a term. We have $C_m[v] \not\equiv \lambda x.t$.

Proof. As for Theorem 3.4.33 we can apply Lemma 3.4.30 using $\pi = [\lambda y.[y \mid C_m[z] \to \{\}]]\varepsilon$. Indeed, $\lambda x.t * \pi \Uparrow_\succ$ since $\lambda x.t * \pi \succ \lambda y.[y \mid C_m[z] \to z] * \lambda x.t . \varepsilon \succ [\lambda x.t \mid C_m[z] \to z] * \varepsilon$ and the process $[\lambda x.t \mid C_m[z] \to z] * \varepsilon$ is stuck □

We will now show that a record $\{(l_i = v_i)_{i\in I}\}$ cannot be equivalent to a λ-abstraction $\lambda x.t$. The most natural approach consists in using record projection to obtain a stuck state on the λ-abstraction. In other words, we can use a stack of the form $[\lambda y.y.l_k]\varepsilon$ provided that $k \in I$. However, this technique does not work with the empty record $\{\}$ (i.e., when $I = \emptyset$). In this case, a possible solution is to use a stack of the form $v . \pi$ where v is a value such that $t[x := v] * \pi \Downarrow_\succ$ (obviously, $\{(l_i = v_i)_{i\in I}\} * v . \pi$ is stuck). However, there is no guarantee that such a value v exists. If there is none, then it seem to be impossible to distinguish $\{\}$ from $\lambda x.t$ without relying on a specific term constructor like $R_{v,u}$. Terms of this form were added to the calculus for this very purpose.

Theorem 3.4.35. Let $x \in \mathcal{V}_\iota$ be a λ-variable, $t \in \Lambda$ be a term and $I \subseteq_{\text{fin}} \mathbb{N}$ be a finite set of indices such that $v_i \in \Lambda_\iota$ for all $i \in I$. We have $\lambda x.t \not\equiv \{(l_i = v_i)_{i\in I}\}$.

Proof. We can use Lemma 3.4.30 with $\pi = [\lambda y.R_{y,\{\}}]\varepsilon$. Indeed, we have $\{(l_i = v_i)_{i\in I}\} * \pi \Downarrow_\succ$ as $\{(l_i = v_i)_{i\in I}\} * \pi \succ \lambda y.R_{y,\{\}} * \{(l_i = v_i)_{i\in I}\} . \varepsilon \succ R_{\{(l_i = v_i)_{i\in I}\},\{\}} * \varepsilon \succ \{\} * \varepsilon$ and $\lambda x.t * \pi \Uparrow_\succ$ as $\lambda x.t * \pi \succ \lambda y.R_{y,\{\}} * \lambda x.t . \varepsilon \succ R_{\lambda x.t,\{\}} * \varepsilon$ and $R_{\lambda x.t,\{\}} * \varepsilon$ is stuck. □

To conclude this section, we consider two more ways of deriving an inequivalence when working with records and variants. Provided that two values stored in a given record field or in a constructor are not equivalent, it is possible to derive that the two records or the two variants are not equivalent. We will here need to rely on the third property of compatible equivalences (Definition 3.2.10), that is recalled below as a lemma.

Lemma 3.4.36. Let t_1, t_2 be two arbitrary terms with $x \in \mathrm{FV}_\iota(t_1) \cap \mathrm{FV}_\iota(t_2)$. If there is a closed term $u \in \Lambda^\star$ such that $t_1[x := v] * [u]\pi \succ^* v * \pi$ and $t_2[x := v] * [u]\pi \succ^* v * \pi$ for all $v \in \Lambda_\iota$ and $\pi \in \Pi$, then $t_1[x := v_1] \equiv t_2[x := v_2]$ implies $v_1 \equiv v_2$ for all $v_1, v_2 \in \Lambda_\iota$.

Proof. By definition of a compatible equivalence. □

Theorem 3.4.37. Let $m \in \mathbb{N}$ be a natural number and $v_1, v_2 \in \Lambda_\iota$ be two values. If $v_1 \not\equiv v_2$ then $C_m[v_1] \not\equiv C_m[v_2]$.

Proof. Let us take $u = \lambda y.[y \mid C_m[z] \to z]$ and $t_1 = t_2 = C_m[x]$. For all $v \in \Lambda_\iota$ and $\pi \in \Pi$ we have $C_m[v] * [u]\pi \succ u * C_m[v] \,.\, \pi \succ [C_m[v] \mid C_m[z] \to z] * \pi \succ v * \pi$. As a consequence we can apply Lemma 3.4.36 with v_1 and v_2 to obtain that $C_m[v_1] \equiv C_m[v_2]$ implies $v_1 \equiv v_2$. We can thus conclude as this is the contrapositive of what we want to show. □

Theorem 3.4.38. Let I be a finite set of indices such that for all $i \in I$ we have $v_i, w_i \in \Lambda_\iota$. If there is $k \in I$ such that $v_k \not\equiv w_k$ then $\{(l_i = v_i)_{i\in I}\} \not\equiv \{(l_i = w_i)_{i\in I}\}$.

Proof. We will show the contrapositive, so we suppose $\{(l_i = v_i)_{i\in I}\} \equiv \{(l_i = w_i)_{i\in I}\}$. We then take $k \in I$ and prove that $v_k \equiv w_k$. Let us define the term t_1 to be the record $\{(l_i = v_i)_{i\in I}\}$ in which the value v_k has been replaced by the variable x. Similarly, we define t_2 to be the record $\{(l_i = w_i)_{i\in I}\}$ in which the value w_k has been replaced by x. We can then conclude with Lemma 3.4.36 using t_1, t_2 and $u = \lambda y.y.l_k$. □

3.5 Canonical values

The idea now is to characterise the equivalence classes of the different forms of values. The results presented here will be required in Chapter 4 to show that the semantics of our types is well-formed in some sense. We will first start by showing that $\Box$ is only equivalent to itself among all values. Similarly, it is possible to show that λ-variables are only equivalent to themselves.

Theorem 3.5.39. Let $v \in \Lambda_\iota$ be a value. We have $\Box \equiv v$ if and only if $v = \Box$.

Proof. If $v = \square$ then we immediately have $\square \equiv v$ by reflexivity. It remains to show that $\square \not\equiv v$ for every value $v \neq \square$. In the case where $v \notin \mathcal{V}_\iota$ we can use Lemma 3.4.30 with the stack $[\{\}]\varepsilon$ as we have $\square * [\{\}]\varepsilon \succ \square * \varepsilon \Downarrow_>$ and $v * [\{\}]\varepsilon \succ \{\} * v \,.\, \varepsilon \Uparrow_>$. If $v = x \in \mathcal{V}_\iota$ then we can use Lemma 3.4.29 with $\rho = [x := \{\}]$ and $\pi = [\{\}]\varepsilon$ since $v\rho * \pi = \{\} * [\{\}]\varepsilon \succ \{\} * \{\} \,.\, \varepsilon \Uparrow_>$ and $\square * [\{\}]\varepsilon \succ \square * \varepsilon \Downarrow_>$ as above. □

Theorem 3.5.40. Let $x \in \mathcal{V}_\iota$ be a λ-variable and $v \in \Lambda_\iota$ be a value. The equivalence $x \equiv v$ holds if and only if $v = x$.

Proof. If $v = x$ then we have $v \equiv x$ by reflexivity. It remains to show that $x \not\equiv v$ for every value $v \neq x$. In the case where $v = \square$ we can conclude immediately using Theorem 3.5.39. If $v = y \in \mathcal{V}_\iota$ then we can use Lemma 3.4.29 with $\rho = [x := \{\}][y := \square]$ and $\pi = [\{\}]\varepsilon$. Indeed, we have $x\rho * \pi = \{\} * [\{\}]\varepsilon \succ \{\} * \{\} \,.\, \varepsilon \Uparrow_>$ and $v\rho * \pi = \square * [\{\}]\varepsilon \succ \square * \varepsilon \Downarrow_>$. Finally, if $v \neq \square$ and $v \notin \mathcal{V}_\iota$ then we can use Lemma 3.4.29 with $\rho = [x := \square]$ and $\pi = [\{\}]\varepsilon$. Indeed, $x\rho * \pi = \square * [\{\}]\varepsilon \succ \square * \varepsilon \Downarrow_>$ and $v\rho * \pi = v\rho * [\{\}]\varepsilon \succ \{\} * [v\rho]\varepsilon \Uparrow_>$. □

We will now characterise the values that are equivalent to a given variant, and those that are equivalent to a given record. In both cases, the equivalent values have the same structure and equivalent subvalues.

Theorem 3.5.41. Let $m \in \mathbb{N}$ be a natural number and $v, w_0 \in \Lambda_\iota$ be values. The equivalence $C_m[v] \equiv w_0$ holds if and only if $w_0 = C_m[w]$ for some $w \in \Lambda_\iota$ such that $w \equiv v$.

Proof. Let us first assume $w_0 = C_m[w]$ for some $w \in \Lambda_\iota$ such that $w \equiv v$. Provided that $(\equiv)$ is a congruence, we can use its extensionality property with the term $t = C_m[x]$ to obtain $C_m[v] \equiv w_0 = C_m[w]$.

Let us now suppose that $C_m[v] \equiv w_0$ and show that w_0 is of the form $C_m[w]$ for some $w \in \Lambda_\iota$ such that $w \equiv v$. We reason by case on the possible forms of the value w_0. Using Theorems 3.5.40, 3.5.39, 3.4.34, 3.4.33 and 3.4.31 we obtain $w_0 = C_m[w]$ for some $w \in \Lambda_\iota$. Now, if $w \not\equiv v$ then we immediatly obtain that $C_m[v] \not\equiv C_m[w]$ using Theorem 3.4.37. As a consequence, it must be that $v \equiv w$. □

Theorem 3.5.42. Let I be a finite set of indices such that $v_i \in \Lambda_\iota$ for all $i \in I$, and let $w \in \Lambda_\iota$ be a value. The equivalence $\{(l_i = v_i)_{i \in I}\} \equiv w$ holds if and only if we have $w = \{(l_i = w_i)_{i \in I}\}$ for some values $w_i \in \Lambda_\iota$ such that $w_i \equiv v_i$ for all $i \in I$.

Proof. Let us first assume that $w = \{(l_i = w_i)_{i \in I}\}$ with $v_i \equiv w_i$ for all $i \in I$ and show $\{(l_i = v_i)_{i \in I}\} \equiv \{(l_i = w_i)_{i \in I}\}$. Up to renaming, we may assume $I = \{i \in \mathbb{N} \mid 1 \leqslant i \leqslant n\}$ with $n = |I|$. For all $0 \leqslant k \leqslant n$ we define $R_k = \{(l_i = r_i)_{i \in I}\}$ where $r_i = w_i$ if $i < k$ and $r_i = v_i$ otherwise. We have $R_0 = \{(l_i = v_i)_{i \in I}\}$ and $R_n = \{(l_i = w_i)_{i \in I}\}$, so we need to show

that $R_0 \equiv R_n$. We are going to prove that $R_0 \equiv R_k$ for all $0 \leqslant k \leqslant n$ by induction on k. When $k = 0$ this is immediate by reflexivity. Let us now suppose that $R_0 \equiv R_k$ for some $0 \leqslant k < n$ and show that $R_0 \equiv R_{k+1}$. By transitivity, it is enough to show $R_k \equiv R_{k+1}$ which follows from the extensionality of $(\equiv)$ since we assumed $v_k \equiv w_k$ (and $(\equiv)$ is a congruence).

Let us now suppose $\{(l_i = v_i)_{i \in I}\} \equiv w$ and show that w is of the form $\{(l_i = w_i)_{i \in I}\}$ with $w_i \equiv v_i$ for all $i \in I$. Using Theorems 3.5.40, 3.5.39, 3.4.35, 3.4.33 and 3.4.32 we obtain $w = \{(l_i = w_i)_{i \in I}\}$ with $w_i \in \Lambda_\iota$ for all $i \in I$. Now, if $v_k \not\equiv w_k$ for some $k \in I$ then we immediately get $\{(l_i = v_i)_{i \in I}\} \not\equiv \{(l_i = w_i)_{i \in I}\}$ using Theorem 3.4.38. As a consequence, it must be that $v_k \equiv w_k$ for all $k \in I$. □

To conclude this chapter, we provide a last theorem establishing that λ-abstractions can only be equivalent to λ-abstractions.

Theorem 3.5.43. Let $x \in \mathcal{V}_\iota$ be a λ-variable, $t \in \Lambda$ be a term and $v \in \Lambda_\iota$ be a value. If the equivalence $\lambda x.t \equiv v$ holds, then there must be $y \in \mathcal{V}_\iota$ and $u \in \Lambda$ such that $v = \lambda y.u$.

Proof. We reason by case on the possible forms of v. Using Theorems 3.5.40, 3.5.39, 3.4.35 and 3.4.34 we obtain that v cannot be $\square$, a λ-variable, a record nor a variant. The only remaining possibility is that $v = \lambda y.u$ for some $y \in \mathcal{V}_\iota$ and $u \in \Lambda$. □

4 Types and realizability semantics

In this chapter, we present a new type system whose distinguishing feature is an embedded notion of program equivalence. It enables the specification of equational properties over programs, which can then be proved using equational reasoning. Our types are interpreted using standard classical realizability techniques, which allows for a semantical justification of our typing rules.

4.1 Observational equivalence type

One of the main goals of this thesis is to build a type system that can be used to reason about programs. To achieve this goal, we need to be able to specify program behaviours using types. We hence introduce equality types of the form $t \equiv u$, where t and u are (possibly untyped) terms. Note that here, the equivalence symbol is part of the syntax and does not refer to a specific equivalence relation. An equivalence relation ($\equiv$) will however be used for the semantical interpretation of equality types. We will require that it is a congruence (Definition 3.2.8) and compatible with ($\equiv_>$) (Definition 3.2.10), but it will remain otherwise unspecified. In other words, it will be considered as a parameter of our type system and semantics.

Intuitively, an equality type $t \equiv u$ will be interpreted as $\top$ (i.e., logical truth or the biggest type) if the equivalence $t \equiv u$ holds, and as $\bot$ (i.e., logical absurdity or the smallest type) otherwise. For example, the type $(\lambda x.x)\ \lambda x.x \equiv \lambda x.x$ will be equivalent to $\top$ as we have $(\lambda x.x)\ \lambda x.x \equiv \lambda x.x$ according to Theorem 3.3.16. However, the type $\lambda x.x \equiv \{\}$ will be equivalent to $\bot$ as $\lambda x.x \not\equiv \{\}$ according to Theorem 3.4.35. Of course, a compatible equivalence relation is likely to be undecidable. Indeed, such equivalence relations are similar to ($\equiv_>$), which is itself undecidable.

Theorem 4.1.1. Given $t, u \in \Lambda$ it is undecidable whether $t \equiv_> u$ or $t \not\equiv_> u$.

Proof. We are going to encode the halting problem H_t using equivalence. Given a closed term $t \in \Lambda^*$, H_t can be stated as $t * \varepsilon \Downarrow_>$ in our system. Let us consider a closed term $\Omega \in \Lambda^*$ such that $\Omega * \pi \Uparrow_>$ for all $\pi \in \Pi$. For example, we can take $\Omega = (\lambda x.x\ x)\ \lambda x.x\ x$ as $\Omega * \pi$ is non-terminating for all $\pi \in \Pi$ since $\Omega * \pi \succ^3 \Omega * \pi$. We will now show that H_t is equivalent to $[\varepsilon]t \not\equiv_> \Omega$. Let us suppose H_t and show $[\varepsilon]t \not\equiv_> \Omega$. We need to find a stack $\pi \in \Pi$ such that $[\varepsilon]t * \pi \Downarrow_>$ and $\Omega * \pi \Uparrow_>$. This is in fact true for all π as $\Omega * \pi \Uparrow_>$ by hypothesis and $[\varepsilon]t * \pi \succ t * \varepsilon \Downarrow_>$ since we supposed H_t. We now suppose $[\varepsilon]t \not\equiv_> \Omega$ and show H_t. By definition, there must be a stack $\pi_0 \in \Pi$ such that $[\varepsilon]t * \pi_0 \Downarrow_>$ since $\Omega * \pi \Uparrow_>$ for all $\pi \in \Pi$. Since $[\varepsilon]t * \pi_0 \succ t * \varepsilon$, we immediately obtain H_t. □

As a consequence, we cannot hope to decide, in general, whether an equivalence holds or not. We will hence need to rely on a partial decision procedure that will only approximate our observational equivalence relation.

In the system, proving a program equivalence amounts to showing that the corresponding equality type is inhabited. However, an equivalence type may also be used as an assumption. For example, it is possible to define a function whose input type is an equivalence. As a consequence, we need a form of context to store the set of program equivalences that are assumed to be true during a proof. This context will also be extended automatically during the construction of a proof (e.g., when entering branches of a case analysis).

Definition 4.1.2. An equational context $\Xi \subseteq \mathcal{P}(\Lambda \times \Lambda)$ is a finite set of pairs of terms denoting hypothetical equivalences. For convenience, equational contexts we will represented using lists generated using the following BNF grammar.

$$\Xi ::= \emptyset \mid \Xi, t \equiv u \qquad\qquad t, u \in \Lambda$$

During the construction of a proof, the equational context grows with new hypotheses and equivalences need to be proved eventually. As mentioned previously, we rely on a partial decision procedure that is supposed correct, but remains otherwise unspecified. Such a decision procedure has been successfully implemented to work with our prototype language (http://lepigre.fr/these/). Although we cannot hope for completeness, we will argue that it is a good enough approximation of our equivalent relation in practice.

Definition 4.1.3. Given an equational context Ξ and a substitution ρ, we say that ρ *realizes* Ξ and we write $\rho \Vdash \Xi$ if for every $(t, u) \in \Xi$ we have $t\rho \equiv u\rho$.

Definition 4.1.4. Let Ξ be an equational context and $t \in \Lambda$ and $u \in \Lambda$ be two terms. We write $\Xi \vdash t \equiv u$ if our (yet unspecified) decision procedure is able to show that for every substitution ρ such that $\rho \Vdash \Xi$ we have $t\rho \equiv u\rho$.

4.2 Quantification and membership type

Although equality types can be used to derive simple equational properties, their use is rather limited without a form of quantification. Indeed, they only allow the derivation of static equivalences like $(\lambda x.x)\ \{\} \equiv \{\}$, but they cannot be used to show more general properties like "$(\lambda x.x)\ v \equiv v$ for every value v". Terms can contain free variables of several sorts: λ-variables (i.e., value variables), term variables and μ-variables (i.e., stack variables). As open terms may appear in types, and in particular in equality types, it is natural to allow universal and existential quantification over all three sorts of variables. This will enable the specification of properties such as $\forall y.(\lambda x.x)\ y \equiv y$ or $\forall x.\forall y.x \equiv y \Rightarrow C_k[x] \equiv C_k[y]$ inside the system.

Remark 4.2.5. The forms of quantification described here range over all closed values, terms or stacks regardless of their types.

Remark 4.2.6. It is not clear whether quantification over stacks has a practical use. We only include it as it fits well in the framework at no extra cost.

Quantifying over all the closed values or terms is not always enough. Indeed, we often need to quantify over the values or the terms of a given type only. For example if we quantify over a λ-variable that is used in a case analysis, then a runtime error will be produced for values that do not correspond to matched constructors. This would not happen when quantifying over value of the appropriate sum type only.

To achieve typed quantification we introduce a membership type constructor $t \in A$ where t is a term and A is a type. The elements of $t \in A$ are those of A that are equivalent to t. In particular, $t \in A$ is empty if t does not have type A. Intuitively, $t \in A$ can be read as the proposition "t has type A", but we will see later that a more appropriate interpretation would be "t realizes A". Using membership, we can use the well-known relativised quantification scheme to obtain a dependent function type.

$$\Pi_{x \in A} B \coloneqq \forall x.(x \in A \Rightarrow B)$$

The dependent function type exactly correspond to typed quantification as its elements can only be applied to values of type A. Other values are simply filtered out. Note that we can define a dependent pair type using existential quantification as follows.

$$\Sigma_{x \in A} B \coloneqq \exists x.\{l_1 : x \in A\,;\, l_2 : B\}$$

We can also define the same kind of type constructors by quantifying over term variables in the exact same way.

$$\Pi_{a\in A}B \coloneqq \forall a.(a\in A \Rightarrow B) \qquad\qquad \Sigma_{a\in A}B \coloneqq \exists a.\{l_1 : a\in A\,; l_2 : B\}$$

Remark 4.2.7. Note that these encodings only make sense (i.e., correspond to dependent types) if the variable that is quantified over does not appear in A.

4.3 Sorts and higher-order types

Our type system allows universal and existential quantification over several sorts of objects. There are first-order quantifiers ranging over values, terms and stacks, as shown in the previous section. And the system also provides second order quantification (i.e., quantification over types), which corresponds to System F polymorphism and type abstraction. All of these different forms of quantifiers are handled uniformly in the syntax and in the semantics thanks to a higher-order formulation.

The higher-order features of the system allow us to define (and quantify over) types with parameters of any sort. For example, we can define a type parametrised by another type and a term. This leads to a system in which it is syntactically correct to use a (not fully applied) parametric type, or even a term, as a proposition. This does not make sense, and hence we must make sure that this does not happen. The usual approach to tackle this problem is to assign a form of type (called sort) to the types themselves. This will give use the guarantee that our types are "well-formed".

Definition 4.3.8. We denote $\mathcal{S}_0 = \{o\,, \iota, \tau, \sigma\}$ our set of atomic sorts. It contains the sort of propositions o, the sort of values ι, the sort of terms τ and the sort of stacks σ.

Definition 4.3.9. The set of all sorts $\mathcal{S}$ is generated from the set of atomic sorts $\mathcal{S}_0$ using the following BNF grammar.

$$s\,, r ::= \kappa \mid s \to r \qquad\qquad \kappa \in \mathcal{S}_0$$

The language of sorts only contains constants and an arrow constructor for functions. Our "sort system" will in fact be very similar to the simply typed λ-calculus. In particular, the syntax of our types will contain a constructor for building functions (i.e., λ-abstraction) and a corresponding constructor for application.

Definition 4.3.10. We require a countable set $\mathcal{V}_o = \{X\,, Y, Z...\}$ of propositional variables that does not intersect with $\mathcal{V}_\iota$, $\mathcal{V}_\tau$ and $\mathcal{V}_\sigma$.

Definition 4.3.11. We require a countable set of variables $\mathcal{V} = \{\chi, \xi, \varphi...\}$ containing the sets $\mathcal{V}_\iota, \mathcal{V}_\tau, \mathcal{V}_\sigma$ and $\mathcal{V}_o$. Given a variable $\chi \in \mathcal{V}$, we will sometimes write χ^s to mean that χ is to be considered as a variable of sort s.

Definition 4.3.12. The set of types (or formulas) $\mathcal{F}$ is built from $\mathcal{V}$, Λ_ι, Λ and Π using the following BNF grammar.

$$(\mathcal{F}) \quad A, B ::= \nu \mid t \mid \pi \mid \chi^s \mid (\chi^s \mapsto A) \mid A(B) \mid A \Rightarrow B \mid t \in A \mid A \upharpoonright t \equiv u \mid \{(l_i : A_i)_{i\in I}\} \mid [(C_i : A_i)_{i\in I}] \mid \forall \chi^s.A \mid \exists \chi^s.A$$

Our syntax contain values, terms and stacks, as they will correspond to types of sort ι, τ and σ. We then have variables of all sorts, abstraction to build types with an arrow sort and application. All the remaining constructors are used to build propositions (i.e., actual formulas) which will be given sort o. Note that the quantifiers may range over types of any sort. They are hence very general.

Remark 4.3.13. The BNF grammar of Definition 4.3.12 is ambiguous. For example, values are given twice as they are contained in terms. Some variables of $\mathcal{V}$ are also given more than once as $\mathcal{V}_\iota, \mathcal{V}_\tau$ and $\mathcal{V}_\sigma$ are contained in values, terms and stacks respectively.

To track the sort of variables in types we need to introduce a form of context. It can then be used to define our notion of well-formed type using a deduction rule system. We will then only consider types that can be shown well-formed in this system.

Definition 4.3.14. A *sorting context* is a finite map Σ over $\mathcal{V}$ such that for all $\chi \in \mathrm{dom}(\Sigma)$ we have $\Sigma(\chi) \in \mathcal{S}$. For convenience, we will represent sorting contexts using comma-separated lists of sort assignments generated by the following BNF grammar.

$$\Sigma ::= \emptyset \mid \Sigma, \chi : s \qquad\qquad \chi \in \mathcal{V}, s \in \mathcal{S}$$

Definition 4.3.15. A sorting judgment is a triple of a sorting context Σ, a type A and a sort s denoted $\Sigma \vdash A : s$. We say that the sorting judgment $\Sigma \vdash A : s$ is valid if and only if it can be derived using the deduction rules of Figure 4.1.

The main role of the sorting rules is to keep track of the sort of the free variables. The rules on the first five lines of Figure 4.1 simply traverse the structure of values, terms and stacks to save the sort of the free variables in the context. Note that the first rule on the sixth line is also used for value, term and stack variables.

$$\frac{\Sigma, x : \iota \vdash t : \tau}{\Sigma \vdash \lambda x.t : \iota} \qquad \frac{\Sigma \vdash \nu : \iota}{\Sigma \vdash C[\nu] : \iota} \qquad \frac{\{\Sigma \vdash \nu_i : \iota\}_{i \in I}}{\Sigma \vdash \{(l_i = \nu_i)_{i \in I}\} : \iota}$$

$$\frac{}{\Sigma \vdash \Box : \iota} \qquad \frac{\Sigma \vdash \nu : \iota}{\Sigma \vdash \nu : \tau} \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma \vdash u : \tau}{\Sigma \vdash t\, u : \tau}$$

$$\frac{\Sigma, \alpha : \sigma \vdash t : \tau}{\Sigma \vdash \mu\alpha.t : \tau} \qquad \frac{\Sigma \vdash \pi : \sigma \quad \Sigma \vdash t : \tau}{\Sigma \vdash [\pi]t : \tau} \qquad \frac{\Sigma \vdash \nu : \iota}{\Sigma \vdash \nu.l : \tau}$$

$$\frac{\Sigma \vdash \nu : \iota \quad \{\Sigma, x_i : \iota \vdash t_i : \tau\}_{i \in I}}{\Sigma \vdash [\nu \,|\, (C_i[x_i] \to t_i)_{i \in I}] : \tau} \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma \vdash \nu : \iota}{\Sigma \vdash Y_{t,\nu} : \tau}$$

$$\frac{\Sigma \vdash \nu : \iota \quad \Sigma \vdash t : \tau}{\Sigma \vdash R_{\nu,t} : \tau} \qquad \frac{\Sigma \vdash \nu : \iota \quad \Sigma \vdash w : \iota}{\Sigma \vdash \delta_{\nu,w} : \tau} \qquad \frac{}{\Sigma \vdash \varepsilon : \sigma}$$

$$\frac{\Sigma \vdash \nu : \iota \quad \Sigma \vdash \pi : \sigma}{\Sigma \vdash \nu\,.\,\pi : \sigma} \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma \vdash \pi : \sigma}{\Sigma \vdash [t]\pi : \sigma} \qquad \frac{}{\Sigma, \chi : s \vdash \chi : s}$$

$$\frac{\Sigma, \chi : s \vdash A : r}{\Sigma \vdash (\chi^s \mapsto A) : s \to r} \qquad \frac{\Sigma \vdash A : s \to r \quad \Sigma \vdash B : s}{\Sigma \vdash A(B) : r} \qquad \frac{\Sigma \vdash A : o \quad \Sigma \vdash B : o}{\Sigma \vdash A \Rightarrow B : o}$$

$$\frac{\{\Sigma \vdash A_i : o\}_{i \in I}}{\Sigma \vdash \{(l_i : A_i)_{i \in I}\} : o} \qquad \frac{\{\Sigma \vdash A_i : o\}_{i \in I}}{\Sigma \vdash [(C_i : A_i)_{i \in I}] : o} \qquad \frac{\Sigma, \chi : s \vdash A : o}{\Sigma \vdash \forall \chi^s.A : o}$$

$$\frac{\Sigma, \chi : s \vdash A : o}{\Sigma \vdash \exists \chi^s.A : o} \qquad \frac{\Sigma \vdash A : o \quad \Sigma \vdash t : \tau \quad \Sigma \vdash u : \tau}{\Sigma \vdash A \restriction t \equiv u : o} \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma \vdash A : o}{\Sigma \vdash t \in A : o}$$

Figure 4.1 – Sorting rules.

As an example $\lambda x.x \Rightarrow \{\}$ is not well-formed as the type at the left of the arrow does not have sort o (it has sort ι or τ). However, $(X \mapsto [C_0 : \{\} \,|\, C_1 : X])([C_2 : \{\} \,|\, C_3 : \{\}])$ is a well-formed type of sort o, as shown by the following proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X : o \vdash \{\} : o} \quad \dfrac{}{X : o \vdash X : o}}{X : o \vdash [C_0 : \{\} \,|\, C_1 : X] : o}}{\vdash (X \mapsto [C_0 : \{\} \,|\, C_1 : X]) : o \to o} \quad \dfrac{\dfrac{}{\vdash \{\} : o} \quad \dfrac{}{\vdash \{\} : o}}{\vdash ([C_2 : \{\} \,|\, C_3 : \{\}]) : o}}{\vdash (X \mapsto [C_0 : \{\} \,|\, C_1 : X])([C_2 : \{\} \,|\, C_3 : \{\}]) : o}$$

As types contain the λ-calculus, we need to consider their reduction and normalisation. First we define the notion of β-reduction in a type.

Definition 4.3.16. We call a *redex* a type of the form $(\chi^s \mapsto A)(B)$, and we say that a type is in normal form if it does not contain any redex. The reduction relation over types is the smallest relation $(\hookrightarrow) \subseteq \mathcal{T} \times \mathcal{T}$ such that:

- for every redex $(\chi^s \mapsto A)(B)$ we have $(\chi^s \mapsto A)(B) \hookrightarrow A[\chi \coloneqq B]$ and
- $(\hookrightarrow)$ is contextually closed (i.e., reduction considers all the redexes in a type).

We denote $(\hookrightarrow^*)$ its reflexive and transitive closure.

Remark 4.3.17. The $(\hookrightarrow)$ relation should not be confused with the call-by-value reduction relation of our abstract machine. In particular, it cannot be used to evaluate terms.

Theorem 4.3.18. Let Σ be a sorting context, $A \in \mathcal{T}$ be a type and $s \in \mathcal{S}$ be a sort. If the sorting judgment $\Sigma \vdash A : s$ is valid, then there is a unique type $B \in \mathcal{T}$ such that B is in normal form, $A \hookrightarrow^* B$ and the sorting judgment $\Sigma \vdash B : s$ is valid.

Proof. Our language of types can be seen as an instance of the simply typed λ-calculus extended with countably many constants. For instance, a type of the form $\forall\chi^s.A$ can be encoded using a constant $\forall_s$ of sort $(s \to o) \to o$. Note that an infinite number of constants is required to encode product types, sum types, records and pattern-matchings as they can be indexed by any finite subset of $\mathbb{N}$. As a consequence, the theorem follows from well-known properties of the simply typed λ-calculus with the subtyping relation induced by the axiom $\iota \leqslant \tau$ on base types (see, e.g., [*Mitchell 1996*]). □

In the following sections, we will always consider well-formed types. As a consequence, we may also assume that a type is in normal form as we can always normalise well-formed types according to the previous theorem.

4.4 Typing judgments for values and terms

As our language is call-by-value and has operations generating side-effects, we need to be careful to achieve type-safety. In particular, some of our typing rules will not apply to terms, but only to values (as mentioned in Chapter 1). Here, value restriction will be encoded using two forms of judgments: usual typing judgments ranging over terms (including values), and a restricted form of judgments ranging over values only.

To be able to assign types to terms containing free variables we will need our typing judgments to carry a form of context. It will provide us with a way of assuming a type for each of the free variables of the typed term. They will include λ-variable and μ-variable but no term variables.

$$\frac{}{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash_{\overline{\text{val}}} x : A}\,\text{Ax} \qquad \frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash t : B}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} \lambda x.t : A \Rightarrow B}\,\Rightarrow_i$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A}{\Sigma \mid \Gamma ; \Xi \vdash v : A}\,\uparrow \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \Rightarrow B \quad \Sigma \mid \Gamma ; \Xi \vdash u : A}{\Sigma \mid \Gamma ; \Xi \vdash t\, u : B}\,\Rightarrow_e$$

$$\frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp} ; \Xi \vdash t : A}{\Sigma \mid \Gamma ; \Xi \vdash \mu\alpha.t : A}\,\mu \qquad \frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp} ; \Xi \vdash t : A}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp} ; \Xi \vdash [\alpha]t : B}\,[-]$$

$$\frac{\Sigma, \chi : s \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : \forall \chi^s.A}\,\forall_i \qquad \frac{\Sigma, x : \iota, \chi : s \mid \Gamma, x : A ; \Xi \vdash t : C}{\Sigma, x : \iota \mid \Gamma, x : \exists \chi^s.A ; \Xi \vdash t : C}\,\exists_e$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : \forall \chi^s.A \quad \Sigma \vdash B : s}{\Sigma \mid \Gamma ; \Xi \vdash t : A[\chi := B]}\,\forall_e \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A[\chi := B] \quad \Sigma \vdash B : s}{\Sigma \mid \Gamma ; \Xi \vdash t : \exists \chi^s.A}\,\exists_i$$

$$\frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi, x \equiv t \vdash u : C}{\Sigma, x : \iota \mid \Gamma, x : t \in A ; \Xi \vdash u : C}\,\in_e \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \quad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma ; \Xi \vdash t : A \restriction u_1 \equiv u_2}\,\restriction_i$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : v \in A}\,\in_i \qquad \frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi, u_1 \equiv u_2 \vdash t : C}{\Sigma, x : \iota \mid \Gamma, x : A \restriction u_1 \equiv u_2 ; \Xi \vdash t : C}\,\restriction_e$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : \{(l_i : A_i)_{i \in I}\} \quad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash v.l_k : A_k}\,\times_e \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A_k \quad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} C_k[v] : [(C_i : A_i)_{i \in I}]}\,+_i$$

$$\frac{[\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v_i : A_i]_{i \in I}}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} \{(l_i = v_i)_{i \in I}\} : \{(l_i : A_i)_{i \in I}\}}\,\times_i$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : [(C_i : A_i)_{i \in I}] \quad [\Sigma, x_i : \iota \mid \Gamma, x_i : A_i ; \Xi, v \equiv C_i[x_i] \vdash t_i : B]_{i \in I}}{\Sigma \mid \Gamma ; \Xi \vdash [v \mid (C_i[x_i] \to t_i)_{i \in I}] : B}\,+_e$$

$$\frac{\Sigma \mid \Gamma[x := w_1] ; \Xi \vdash_{\overline{\text{val}}} v[x := w_1] : A[x := w_1] \quad \Xi \vdash w_1 \equiv w_2}{\Sigma \mid \Gamma[x := w_2] ; \Xi \vdash_{\overline{\text{val}}} v[x := w_2] : A[x := w_2]}\,\equiv_{\iota,\iota}$$

$$\frac{\Sigma \mid \Gamma[x := w_1] ; \Xi \vdash t[x := w_1] : A[x := w_1] \quad \Xi \vdash w_1 \equiv w_2}{\Sigma \mid \Gamma[x := w_2] ; \Xi \vdash t[x := w_2] : A[x := w_2]}\,\equiv_{\tau,\iota}$$

$$\frac{\Sigma \mid \Gamma[a := u_1] ; \Xi \vdash_{\overline{\text{val}}} v[a := u_1] : A[a := u_1] \quad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma[a := u_2] ; \Xi \vdash_{\overline{\text{val}}} v[a := u_2] : A[a := u_2]}\,\equiv_{\iota,\tau}$$

$$\frac{\Sigma \mid \Gamma[a := u_1] ; \Xi \vdash t[a := u_1] : A[a := u_1] \quad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma[a := u_2] ; \Xi \vdash t[a := u_2] : A[a := u_2]}\,\equiv_{\tau,\tau}$$

Figure 4.2 – Typing rules.

Definition 4.4.19. A *typing context* is a finite map Γ over $\mathcal{V}_\iota \cup \mathcal{V}_\sigma$ such that $\Gamma(x) \in \mathcal{F}$ for all $x \in \mathrm{dom}(\Gamma) \cap \mathcal{V}_\iota$ and $\Gamma(\alpha) \in \mathcal{F}$ for all $\alpha \in \mathrm{dom}(\Gamma) \cap \mathcal{V}_\sigma$. For convenience, we will represent typing contexts using comma-separated lists of type assignments generated by the following BNF grammar. Note that variables can only be mapped once in a context, the order in which they appear is thus irrelevant.

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, \alpha : A^\perp \qquad x \in \mathcal{V}_\iota, \alpha \in \mathcal{V}_\tau, A \in \mathcal{F}$$

The full context of typing judgments will be built using a typing context and an equational context. As these contexts may contain types and hence free variables, we need to define a notion of well-formedness for these two forms of contexts.

Definition 4.4.20. Given a sorting context Σ, we say that a typing context Γ is well-formed and write $\Sigma \vdash \Gamma$ if for all $x \in \mathrm{dom}(\Gamma) \cap \mathcal{V}_\iota$ we have $\Sigma \vdash \Gamma(x) : o$ and $\Sigma \vdash x : \iota$, and for all $\alpha \in \mathrm{dom}(\Gamma) \cap \mathcal{V}_\sigma$ we have $\Sigma \vdash \Gamma(\alpha) : o$ and $\Sigma \vdash \alpha : \sigma$.

Definition 4.4.21. Given a sorting context Σ, we say that an equational context Ξ is well-formed and write $\Sigma \vdash \Xi$ if for all $(t, u) \in \Sigma$ we have $\Sigma \vdash t : \tau$ and $\Sigma \vdash u : \tau$.

Using our three forms of contexts, we can now define our actual judgments. Again, a notion of well-formedness need to be considered as our judgments contain objects of different sorts, which all need to be well-formed.

Definition 4.4.22. A value judgement is a tuple of a typing context Γ, an equational context Ξ, a value $v \in \Lambda_\iota$ and a type $A \in \mathcal{F}$ that is denoted $\Gamma ; \Xi \vdash_{\mathrm{val}} v : A$. We say that such a judgment is well-formed under the sorting context Σ, and we write $\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} v : A$, if and only if we have $\Sigma \vdash \Gamma$, $\Sigma \vdash \Xi$, $\Sigma \vdash v : \iota$ and $\Sigma \vdash A : o$.

Definition 4.4.23. A term judgement is a tuple of a typing context Γ, an equational context Ξ, a term $t \in \Lambda$ and a type $A \in \mathcal{F}$ that is denoted $\Gamma ; \Xi \vdash t : A$. We say that such a judgment is well-formed under the sorting context Σ, and we write $\Sigma \mid \Gamma ; \Xi \vdash t : A$, if and only if we have $\Sigma \vdash \Gamma$, $\Sigma \vdash \Xi$, $\Sigma \vdash t : \tau$ and $\Sigma \vdash A : o$.

We can now give the typing rules of our system, which will involve both value and term judgments. Moreover, rules having a premise of the form $\Xi \vdash t \equiv u$ can only be applied if our (yet unspecified) equivalence decision procedure is able to show that t is equivalent to u in the equational context Ξ (see the first section of this chapter).

Definition 4.4.24. A value or term judgment is said to be valid if it can be derived using the typing rules of Figure 4.2.

The typing rules of our system, and hence our typing derivations, only involve well-formed judgments. In particular, a typing rule can only be applied if all the involved judgments are well-formed.

Remark 4.4.25. Note that the $\forall_i$ rule cannot apply if χ appears in the contexts Γ and Ξ or in t. This would prevent the conclusion judgment to be well-formed.

As our system is call-by-value and has effects, the $\forall_i$ rule needs value restriction to remain sound. In our formalism, this means that the $\forall_i$ rule only applies to value judgments. The $\in_i$ rule requires value restriction as well, as otherwise the system cannot be proved sound. As the $\in_i$ rule will be involved in the derivation of the typing rule for the elimination of the dependent function type at the beginning of Chapter 5, the latter will also need value restriction.

4.5 Call-by-value realizability semantics

The abstract machine presented in Chapter 2 is part of a *classical realizability* machinery that will be built upon here. We aim at obtaining a semantical interpretation of our higher-order type system. In particular, a proposition (i.e., a type of sort 0) will be interpreted by three sets: a set of values, a set of stacks and a set of terms. As always in classical realizability, the model is parametrised by a pole, which serves as an exchange point between the world of programs and the world of execution contexts, encoded as stacks.

Definition 4.5.26. Given a reduction relation $R \subseteq (\Lambda \times \Pi) \times (\Lambda \times \Pi)$, a *pole* is a set of processes $\bot\!\!\!\bot \subseteq \Lambda \times \Pi$ which is R-*saturated* (i.e., closed under backward reduction). More formally, if we have $q \in \bot\!\!\!\bot$ and $p \, R \, q$ then $p \in \bot\!\!\!\bot$.

It is important to note that in the remaining of this chapter, we will consider, if not otherwise specified, a fixed (but arbitrary) pole $\bot\!\!\!\bot$.

The notion of *orthogonality* is central in Krivine's classical realizability. In this framework a type is interpreted (or realized) by programs computing corresponding values. This interpretation is spread in a three-layered construction, even though it is fully determined by the first layer and the choice of the pole. The first layer consists of a set of values that we will call the *raw semantics*. It gathers all the syntactic values that should be considered as having the corresponding type. As an example, if we were to consider the type of natural numbers, its raw semantics would be the set $\{\hat{n} \mid n \in \mathbb{N}\}$ where $\hat{n}$ is some encoding of n. The second layer, called *falsity value*, is a set containing every closed stack that is a candidate for building a valid process using any value from the raw semantics. The notion of validity depends on the choice of the pole. The third layer, called *truth value* is a set of closed terms that is built by iterating the process once more. The formalism for the two levels of

orthogonality is given in the following definitions. Recall that Λ^*_ι, Λ^* and Π^* denote the set of closed values, closed terms and closed stacks respectively.

Definition 4.5.27. For every set $\Phi \subseteq \Lambda^*_\iota$ we define a set $\Phi^\perp \subseteq \Pi^*$ as follows.

$$\Phi^\perp = \{\pi \in \Pi^* \mid \forall\, v \in \Phi,\, v * \pi \in \perp\!\!\!\perp\}$$

Definition 4.5.28. For every set $\Psi \subseteq \Pi^*$ we define a set $\Psi^\perp \subseteq \Lambda^*$ as follows.

$$\Psi^\perp = \{t \in \Lambda^* \mid \forall\, \pi \in \Psi,\, t * \pi \in \perp\!\!\!\perp\}$$

We now give several general properties of orthogonality, which hold in every call-by-value classical realizability model. They will be useful when proving the soundness of our type system.

Lemma 4.5.29. If $\Phi \subseteq \Lambda^*_\iota$ is a set of values, then $\Phi \subseteq \Phi^{\perp\perp}$.

Proof. We take $v \in \Phi$ and show $v \in \Phi^{\perp\perp}$. By definition we need to show $v * \pi \in \perp\!\!\!\perp$ for all stacks $\pi \in \Phi^\perp$. This is immediate by definition of $\Phi^\perp$. □

Lemma 4.5.30. Let $\Phi, \Psi \subseteq \Lambda^*_\iota$ be sets of closed values. If $\Phi \subseteq \Psi$ then $\Psi^\perp \subseteq \Phi^\perp$.

Proof. Let us suppose that $\Phi \subseteq \Psi$, take $\pi \in \Psi^\perp$ and show that $\pi \in \Phi^\perp$. By definition, we know that for all $v \in \Psi$ we have $v * \pi \in \perp\!\!\!\perp$. Since $\Phi \subseteq \Psi$, this is also true for all $v \in \Phi$, and hence $\pi \in \Phi^\perp$. □

Lemma 4.5.31. Let $\Phi, \Psi \subseteq \Lambda^*_\iota$ be sets of closed values. If $\Psi^\perp \subseteq \Phi^\perp$ then $\Phi^{\perp\perp} \subseteq \Psi^{\perp\perp}$.

Proof. Let us suppose that $\Psi^\perp \subseteq \Phi^\perp$, take $t \in \Phi^{\perp\perp}$ and show that $t \in \Psi^{\perp\perp}$. By definition, we know that for all $\pi \in \Phi^\perp$ we have $t * \pi \in \perp\!\!\!\perp$. Since $\Psi^\perp \subseteq \Phi^\perp$, this is also true for all $\pi \in \Psi^\perp$, and hence $t \in \Psi^{\perp\perp}$. □

Lemma 4.5.32. Let $\Phi, \Psi \subseteq \Lambda^*_\iota$ be sets of closed values. If $\Phi \subseteq \Psi$ then $\Phi^{\perp\perp} \subseteq \Psi^{\perp\perp}$.

Proof. Let us suppose that $\Phi \subseteq \Psi$ and apply Lemma 4.5.30 to obtain $\Psi^\perp \in \Phi^\perp$. We can then conclude using Lemma 4.5.31. □

When choosing a pole, it is important to check that it does not yield a degenerate model. In particular we need to check that no term is able to face every stack. If it were the case, such a term could be used as a proof of $\bot$.

Definition 4.5.33. A pole $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$ is said to be consistent if for every closed term $t \in \Lambda^*$ there is a stack π such that $t * \pi \notin \perp\!\!\!\perp$.

In this thesis, another property will be required of our poles. As we are in the presence of an equivalence relation over terms, we will need our poles to be closed under this relation in some sense. This will allow us to derive the same properties for equivalent terms, and handle them uniformly.

Definition 4.5.34. Given an equivalence relation $R \subseteq \Lambda \times \Lambda$, a pole $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$ is said to be R-extensional if for every closed terms $t, u \in \Lambda^*$ such that $t \, R \, u$, and for every stack $\pi \in \Pi$, we have $t * \pi \in \perp\!\!\!\perp$ if and only if $u * \pi \in \perp\!\!\!\perp$.

From now on, and until the end of this chapter, we will only consider poles that are both $\equiv$-extensional and R-saturated for some reduction relation $R \subseteq (\Lambda \times \Pi) \times (\Lambda \times \Pi)$ such that $(\succ) \subseteq R$. This information will be kept implicit most of the time. Note that R, much like $(\equiv)$ is a parameter of our type system and realizability semantics.

Theorem 4.5.35. Let $\Phi \subseteq \Lambda^*_\iota$ be a set of closed values and $t, u \in \Lambda^*$ be two closed terms. With an $\equiv$-extensional pole, if $t \in \Phi^{\perp\perp}$ and $t \equiv u$ then $u \in \Phi^{\perp\perp}$.

Proof. By definition we need to take $\pi \in \Phi^\perp$ and show that $u * \pi \in \perp\!\!\!\perp$. Since $t \in \Phi^{\perp\perp}$ we have $t * \pi \in \perp\!\!\!\perp$ and hence we can conclude using the $\equiv$-extensionality of our pole since we have $t \equiv u$. □

In our realizability model, the well-formed closed types will be interpreted by the elements of a set defined according to their sort. Such a set can be seen as the interpretation, in the model, of the sort themselves.

Definition 4.5.36. To every sort $s \in \mathcal{S}$ we associate a set $[\![s]\!]$ defined as follows.

$$[\![\iota]\!] = \Lambda^*_\iota \qquad [\![\tau]\!] = \Lambda^* \qquad [\![\sigma]\!] = \Pi^*$$
$$[\![o]\!] = \{P \in \mathcal{P}(\Lambda^*_\iota/\equiv) \mid \square \in P\} \qquad [\![s \to r]\!] = [\![r]\!]^{[\![s]\!]}$$

Note that a type of sort o will be interpreted by a set of closed values containing the special value $\square$. It is also required for this set to be closed under the equivalence relation $(\equiv)$. This means that for all $\Phi \in [\![o]\!]$ and for all closed values $v, w \in \Lambda^*_\iota$ such that $v \in \Phi$ and $v \equiv w$ we also have $w \in \Phi$.

Remark 4.5.37. The presence of the value $\square$ in the interpretation of types of sort o is not important for the current chapter, nor for Chapter 5. It will however play a crucial

role in Chapter 6, where the semantical interpretation of propositions will be required to be non-empty.

The semantical interpretation of types usually relies on a substitution (or interpretation function) to interpret free variables. Here, we will use another common method consisting in extending the syntax of types with the elements of the model. We will thus substitute free variables with such elements of the model, which will allow us to interpret closed types only. Of course, an open type can always be made into a closed types by replacing its free variables by elements of the interpretation of the corresponding sorts.

Definition 4.5.38. We extend the syntax of types with the elements of the model. As a consequence, we will consider that $[\![s]\!] \subseteq \mathcal{F}$ for every sort $[\![s]\!]$. Note that this is already true for $s = \iota$, $s = \tau$ and $s = \sigma$. We will often use the letter Φ to denote an element of the model in the syntax. Our system is also extended with the following sorting rule.

$$\frac{\Phi \in [\![s]\!]}{\Sigma \vdash \Phi : s}$$

Definition 4.5.39. Given a sorting context Σ, we call a *valuation* over Σ a finite map ρ such that for all $\chi \in \mathrm{dom}(\Sigma)$ we have $\rho(\chi) \in [\![\Sigma(\chi)]\!]$. If $A \in \mathcal{F}$ is a type and $s \in \mathcal{S}$ is a sort such that $\Sigma \vdash A : s$ then we denote $A\rho$ the type formed by applying ρ to A.

Lemma 4.5.40. Let Σ_1 and Σ_2 be two sorting contexts such that $\mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) = \emptyset$. Let $A \in \mathcal{F}$ be a type and $s \in \mathcal{S}$ be a sort such that $\Sigma_1, \Sigma_2 \vdash A : s$. If ρ is a valuation over Σ_1 such that $\mathrm{dom}(\rho) \cap \mathrm{dom}(\Sigma_2) = \emptyset$ then $\Sigma_2 \vdash A\rho : s$.

Proof. Simple induction on the derivation of $\Sigma_1, \Sigma_2 \vdash A : s$ by replacing the variables of Σ_1 by their value in ρ. During the induction Σ_2 will grow when going through the rules that extend the context. The proof for the other rules can be handled immediately by induction hypothesis. The only interesting case is the axiom

$$\frac{}{\Sigma_1, \Sigma_2 \vdash \chi : s}$$

for which there are two cases. Either $\chi \in \mathrm{dom}(\Sigma_2)$ and the derivation remains an axiom, or $\chi \in \mathrm{dom}(\Sigma_1)$. In this second case we have $\chi \in \mathrm{dom}(\rho)$ and thus $\chi\rho = \rho(\chi) \in [\![s]\!]$. As a consequence the derivation becomes the following.

$$\frac{\rho(\chi) \in [\![s]\!]}{\Sigma_2 \vdash \rho(\chi) : s}$$

□

Lemma 4.5.41. Let Σ be a sorting context, $A \in \mathcal{T}$ be a type and $s \in \mathcal{S}$ be a sort such that we have $\Sigma \vdash A : s$. If ρ is a valuation over Σ then $\vdash A\rho : s$.

Proof. Direct consequence of Lemma 4.5.40 by taking $\Sigma_1 = \Sigma$ and $\Sigma_2 = \emptyset$. □

We can now give the interpretation of our type constructors in our model. In particular, the elements of the model such as values, terms or stacks will be interpreted as themselves.

Definition 4.5.42. To every closed type $A \in \mathcal{T}$ we associate a set $[\![A]\!]$ called its *interpretation*. It is defined inductively on the structure of A as follows.

$$\begin{aligned}
[\![\Phi]\!] &= \Phi \\
[\![(\chi^s \mapsto A)]\!] &= \Phi \mapsto [\![A[\chi := \Phi]]\!] \\
[\![A(B)]\!] &= [\![A]\!]([\![B]\!]) \\
[\![A \Rightarrow B]\!] &= \{\lambda x.t \mid \forall v \in [\![A]\!] \setminus \{\Box\}, t[x := v] \in [\![B]\!]^{\perp\perp}\} \cup \{\Box\} \\
[\![t \in A]\!] &= \{v \in [\![A]\!] \mid v \equiv t\} \cup \{\Box\} \\
[\![A \upharpoonright t \equiv u]\!] &= \{v \in [\![A]\!] \mid t \equiv u\} \cup \{\Box\} \\
\left[\!\left[\{(l_i : A_i)_{i \in I}\}\right]\!\right] &= \left\{\{(l_i = v_i)_{i \in I}\} \mid \forall i \in I, v_i \in [\![A_i]\!] \setminus \{\Box\}\right\} \cup \{\Box\} \\
\left[\!\left[[(C_i : A_i)_{i \in I}]\right]\!\right] &= \cup_{i \in I} \{C_i[v] \mid v \in [\![A_i]\!] \setminus \{\Box\}\} \cup \{\Box\} \\
[\![\forall \chi^s.A]\!] &= \cap_{\Phi \in [\![s]\!]} [\![A[\chi := \Phi]]\!] \\
[\![\exists \chi^s.A]\!] &= \cup_{\Phi \in [\![s]\!]} [\![A[\chi := \Phi]]\!]
\end{aligned}$$

Remark 4.5.43. We have $[\![A \upharpoonright t \equiv u]\!] = [\![A]\!]$ if $t \equiv u$ and $[\![A \upharpoonright t \equiv u]\!] = \{\Box\}$ otherwise. We also have $[\![t \in A]\!] = \{\Box\}$ if there is no $v \in [\![A]\!]$ such that $v \equiv t$.

Theorem 4.5.44. Let Σ be a sorting context, $A \in \mathcal{T}$ be a type and $s \in \mathcal{S}$ be a sort. If we have a derivation of $\Sigma \vdash A : s$ and if ρ is a valuation over Σ then $[\![A\rho]\!] \in [\![s]\!]$.

Proof. The proof is done by induction on the derivation of $\Sigma \vdash A : s$. We reason by case on the last rules used in the deduction tree. In the case of the rule

$$\frac{\Phi \in [\![s]\!]}{\Sigma \vdash \Phi : s}$$

the proof is trivial. For the first eighteen rules of Figure 4.1, the proof is immediate using the induction hypothesis. The remaining cases are treated below. Note that we recall the corresponding rules after the proof of each case.

- In the case of the arrow type, we need to show $[\![(A \Rightarrow B)\rho]\!] \in [\![o]\!]$. By induction hypothesis, we know $[\![A\rho]\!] \in [\![o]\!]$ and $[\![B\rho]\!] \in [\![o]\!]$, which give us $[\![A\rho]\!] \subseteq \Lambda^*_\iota$ and $[\![B\rho]\!]^{\perp\perp} \subseteq \Lambda^*$. Consequently, $[\![A\rho \Rightarrow B\rho]\!]$ is well-defined and $[\![A\rho \Rightarrow B\rho]\!] \subseteq \Lambda^*_\iota$. By definition, we also have $\Box \in [\![A\rho \Rightarrow B\rho]\!]$ so it only remains to show that $[\![A\rho \Rightarrow B\rho]\!]$ is closed under $(\equiv)$. According to Theorem 3.5.39 we know that the only value that is equal to $\Box$ is itself. As a consequence, we only have to consider values that are equivalent to a λ-abstraction of $[\![A\rho \Rightarrow B\rho]\!]$. Let us take $\lambda x.t \in [\![A\rho \Rightarrow B\rho]\!]$ and a value $w \in \Lambda^*_\iota$ such that $\lambda x.t \equiv w$. According to Theorem 3.5.43 we have $w = \lambda y.u$ for some $y \in \mathcal{V}_\iota$ and $u \in \Lambda$. To show that $\lambda y.u \in [\![A\rho \Rightarrow B\rho]\!]$ we take a value $v \in [\![A\rho]\!] \setminus \{\Box\}$ and we show $u[y := v] \in [\![B\rho]\!]^{\perp\perp}$. Since $\lambda x.t \in [\![A\rho \Rightarrow B\rho]\!]$ we know that $t[x := v] \in [\![B\rho]\!]^{\perp\perp}$ and thus it is enough to show $u[y := v] \equiv t[x := v]$ according to Theorem 4.5.35. Now, since $\lambda x.t \equiv \lambda y.u$ and $(\equiv)$ is a congruence (Definition 3.2.8) we have $(\lambda x.t)\ v \equiv (\lambda y.u)\ v$. Moreover, since $(\equiv)$ is a compatible equivalence relation (Definition 3.2.10) then we can use Lemma 3.3.16 to obtain $(\lambda x.t)\ v \equiv t[x := v]$ and $(\lambda y.u)\ v \equiv u[y := v]$ since we know that $v \neq \Box$ and v is closed. We can thus conclude using the transitivity of $(\equiv)$.
$$\frac{\Sigma \vdash A : o \qquad \Sigma \vdash B : o}{\Sigma \vdash A \Rightarrow B : o}$$
- In the case of the product type, we need to show that $\Phi = \left[\!\!\left[\{(l_i : A_i)_{i \in I}\}\rho\right]\!\!\right] \in [\![o]\!]$. For all $i \in I$, the induction hypothesis tells us that $[\![A_i\rho]\!] \in [\![o]\!]$, which implies $[\![A_i\rho]\!] \subseteq \Lambda^*_\iota$. As a consequence, we obtain $\Phi \subseteq \Lambda^*_\iota$. We also know that $\Box \in \Phi$ so it only remains to show that Φ is closed under $(\equiv)$. According to Theorem 3.5.39 we know that the only value that is equal to $\Box$ is itself. As a consequence, we only have to consider values that are records. Let us take $\{(l_i = v_i)_{i \in I}\} \in \Phi$ and a value $w \in \Lambda^*_\iota$ such that $\{(l_i = v_i)_{i \in I}\} \equiv w$ and show that $w \in \Phi$. According to Theorem 3.5.42, w must be of the form $\{(l_i = w_i)_{i \in I}\}$ with $v_i \equiv w_i$ for all $i \in I$. We can thus conclude since $[\![A_i\rho]\!]$ is closed under $(\equiv)$.
$$\frac{\{\Sigma \vdash A_i : o\}_{i \in I}}{\Sigma \vdash \{(l_i : A_i)_{i \in I}\} : o}$$
- In the case of the sum type, we can use the same reasoning as for the product type to obtain that $\Phi = \left[\!\!\left[[(C_i : A_i)_{i \in I}]\rho\right]\!\!\right] \subseteq \Lambda^*_\iota$. By definition, we also have $\Box \in \Phi$ so we only have to show that Φ is closed under $(\equiv)$. As it is defined as a union, it is enough to show that all of its components are themselves closed under $(\equiv)$. In particular, it is the case for $\{\Box\}$ according to Theorem 3.5.39. Let us now take $i \in I$ and show that $\{C_i[v] \mid v \in [\![A_i\rho]\!]\}$ is closed under $(\equiv)$. This follows from Theorem 3.5.41 since $[\![A_i\rho]\!]$ is closed under $(\equiv)$.
$$\frac{\{\Sigma \vdash A_i : o\}_{i \in I}}{\Sigma \vdash [(C_i : A_i)_{i \in I}] : o}$$
- In the case of the universal type, we need to show that $[\![(\forall \chi.A)\rho]\!] \in [\![o]\!]$. As we are free to rename χ we may assume that $\chi \notin \mathrm{dom}(\rho)$ and hence our goal rewrites $[\![\forall \chi.A\rho]\!] = \cap_{\Phi \in [\![s]\!]} [\![A\rho[\chi := \Phi]]\!] \in [\![o]\!]$. By induction hypothesis, we know that for

all $\Phi \in [\![s]\!]$ we have $[\![A\rho[\chi := \Phi]]\!] \in [\![o]\!]$. We can thus conclude since an arbitrary intersection of elements of $[\![o]\!]$ is trivially an element of $[\![o]\!]$.

$$\frac{\Sigma, \chi : s \vdash A : o}{\Sigma \vdash \forall \chi.A : o}$$

- In the case of the existential type, we can use a similar reasoning as for the universal type. We only need to remark that an arbitrary union of elements of $[\![o]\!]$ is trivially an element of $[\![o]\!]$.
$$\frac{\Sigma, \chi : s \vdash A : o}{\Sigma \vdash \exists \chi.A : o}$$
- In the case of the membership type, we need to show $[\![t\rho \in A\rho]\!] \in [\![o]\!]$. By induction hypothesis, we know $[\![t\rho]\!] = t\rho \in [\![\tau]\!] = \Lambda^*$ and $[\![A\rho]\!] \in [\![o]\!]$. As a consequence, the set $[\![t\rho \in A\rho]\!] = \{\nu \in [\![A\rho]\!] \mid \nu \equiv t\rho\} \cup \{\square\}$ is well-defined and we have $[\![t\rho \in A\rho]\!] \subseteq \Lambda^*_\iota$. It remains to show that $[\![t\rho \in A\rho]\!]$ is closed under $(\equiv)$ but this follows immediately by construction and using Theorem 3.5.39.
$$\frac{\Sigma \vdash t : \tau \quad \Sigma \vdash A : o}{\Sigma \vdash t \in A : o}$$
- In the case of the restriction type, we need to show $[\![A\rho \restriction t\rho \equiv u\rho]\!] \in [\![o]\!]$. Using the first induction hypothesis, we know that $[\![A\rho]\!] \in [\![o]\!]$. Now, using the second and third induction hypotheses we also know that $[\![t\rho]\!] = t\rho \in [\![\tau]\!] = \Lambda^*$ and that $[\![u\rho]\!] = u\rho \in [\![\tau]\!] = \Lambda^*$. Consequently, the equivalence $t \equiv u$ is well-defined, and thus $[\![A\rho \restriction t\rho \equiv u\rho]\!]$ is either equal to $[\![A\rho]\!]$ (which contains $\square$ already), in which case we can conclude immediately, or to $\{\square\}$. In this second case we obtain $\{\square\} \in [\![o]\!]$ using Theorem 3.5.39.
$$\frac{\Sigma \vdash A : o \quad \Sigma \vdash t : \tau \quad \Sigma \vdash u : \tau}{\Sigma \vdash A \restriction t \equiv u : o}$$

□

To conclude this section, we provide two lemmas that will allow us to show that the interpretation of a propositions is compatible with the equivalence relation $(\equiv)$. More precisely, we will show that substituting a value (resp. a term) with an equivalent value (resp. term) in a proposition does not change its semantical interpretation.

Lemma 4.5.45. Let Σ be a sorting context, $A \in \mathcal{F}$ be a type, $x \in \mathcal{V}_\iota$ be a λ-variable and ν_1, $\nu_2 \in \Lambda_\iota$ be values such that $\Sigma, x : \iota \vdash A : o$, $\Sigma \vdash \nu_1 : \iota$ and $\Sigma \vdash \nu_2 : \iota$. If ρ is a valuation over Σ and if $\nu_1\rho \equiv \nu_2\rho$ then we have $[\![A\rho[x := \nu_1\rho]]\!] = [\![A\rho[x := \nu_2\rho]]\!]$.

Proof. According to Theorem 4.5.44, we have $[\![\nu_1\rho]\!] = \nu_1\rho \in [\![\iota]\!]$ and $[\![\nu_2\rho]\!] = \nu_2\rho \in [\![\iota]\!]$ since we have $\Sigma \vdash \nu_1 : \iota$ and $\Sigma \vdash \nu_2 : \iota$. As a consequence, $\rho[x := \nu_1\rho]$ and $\rho[x := \nu_2\rho]$ are both valuations over the sorting context $\Sigma, x : \iota$, and thus we obtain $[\![A\rho[x := \nu_1\rho]]\!] \in [\![o]\!]$ and $[\![A\rho[x := \nu_2\rho]]\!] \in [\![o]\!]$ using Theorem 4.5.44 once again. We will now show, by induction

on the derivation of $\Sigma, x : \iota \vdash A : o$, that $[\![A\rho[x := v_1\rho]]\!] = [\![A\rho[x := v_2\rho]]\!]$. The only interesting cases (that are not immediate by induction hypothesis) are membership and restriction. In the case of membership

$$\frac{\Sigma \vdash t : \tau \quad \Sigma \vdash A : o}{\Sigma \vdash t{\in}A : o}$$

we need to show that $[\![(t{\in}A)\rho[x := v_1\rho]]\!] = [\![(t{\in}A)\rho[x := v_2\rho]]\!]$. By definition, we need to show $\{v \in [\![A\rho[x := v_1\rho]]\!] \mid v \equiv t\rho[x := v_1\rho]\} = \{v \in [\![A\rho[x := v_2\rho]]\!] \mid v \equiv t\rho[x := v_2\rho]\}$. As we know that $[\![A\rho[x := v_1\rho]]\!] = [\![A\rho[x := v_2\rho]]\!]$ by induction hypothesis, we only need to show that $t\rho[x := v_1\rho] \equiv t\rho[x := v_2\rho]$. This follows from the fact that $(\equiv)$ is a congruence since we have $v_1\rho \equiv v_2\rho$. In the case of restriction

$$\frac{\Sigma \vdash A : o \quad \Sigma \vdash t : \tau \quad \Sigma \vdash u : \tau}{\Sigma \vdash A \restriction t \equiv u : o}$$

we have to prove $[\![(A \restriction t \equiv u)\rho[x := v_1\rho]]\!] = [\![(A \restriction t \equiv u)\rho[x := v_2\rho]]\!]$. By definition, we need to show that $t\rho[x := v_1\rho] \equiv u\rho[x := v_1\rho]$ if and only if $t\rho[x := v_2\rho] \equiv u\rho[x := v_2\rho]$ since $[\![A\rho[x := v_1\rho]]\!] = [\![A\rho[x := v_2\rho]]\!]$ by induction hypothesis. We can then conclude since we have $t\rho[x := v_1\rho] \equiv t\rho[x := v_2\rho]$ and $u\rho[x := v_1\rho] \equiv u\rho[x := v_2\rho]$ using again the fact that $(\equiv)$ is a congruence. □

Lemma 4.5.46. Let Σ be a sorting context, $A \in \mathcal{T}$ be a type, $a \in \mathcal{V}_\tau$ be a term variable and $u_1, u_2 \in \Lambda_\iota$ be terms such that $\Sigma, a : \tau \vdash A : o$, $\Sigma \vdash u_1 : \tau$ and $\Sigma \vdash u_2 : \tau$. If ρ is a valuation over Σ and if $u_1\rho \equiv u_2\rho$ then $[\![A\rho[a := u_1\rho]]\!] = [\![A\rho[a := u_2\rho]]\!]$.

Proof. The proof is very similar to that of Lemma 4.5.45. □

4.6 Adequacy

Now that the interpretation of our types in our model has been specified, we need to show that our typing rules actually agree with the semantics. Intuitively, we need to check that whenever our type system can be used to prove that a term t has type A, then t is indeed in the term level interpretation of A. This will be summarised in the following theorem, but we first need the following definition. It will play a similar role as Definition 4.1.3 for typing contexts.

Definition 4.6.47. Let Σ be a sorting context, ρ be a valuation over Σ and Γ be a typing context such that $\Sigma \vdash \Gamma$ is derivable. We say that ρ realizes Γ and we write $\rho \Vdash \Gamma$ if for every $x \in \mathrm{dom}(\Gamma) \cap \mathcal{V}_\iota$ we have $\rho(x) \in [\![(\Gamma(x))\rho]\!] \setminus \{\square\}$ and for every $\alpha \in \mathrm{dom}(\Gamma) \cap \mathcal{V}_\sigma$ we have $\rho(\alpha) \in [\![(\Gamma(x))\rho]\!]^\perp$.

Theorem 4.6.48. Let Σ be a sorting context, Γ be a typing context, Ξ be an equational context and $A \in \mathcal{T}$ be a type. Let ρ be a valuation over Σ such that $\rho \Vdash \Gamma$ and $\rho \Vdash \Xi$.
- If t is a term such that $\Sigma \mid \Gamma ; \Xi \vdash t : A$ is derivable, then $t\rho \in [\![A\rho]\!]^{\perp\perp}$.
- If v is a value such that $\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} v : A$ is derivable, then $v\rho \in [\![A\rho]\!] \setminus \{\Box\}$.

Proof. We proceed by induction on the derivation of the judgment $\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} v : A$ or $\Sigma \mid \Gamma ; \Xi \vdash t : A$, and we reason by case on the last used rule. Note that the deduction rules are recalled below the proof of the corresponding case.
- In the case of (Ax), we immediately have $x\rho = \rho(x) \in [\![A\rho]\!] \setminus \{\Box\}$ since $\rho \Vdash \Gamma, x : A$.
$$\frac{}{\Sigma , x : \iota \mid \Gamma, x : A ; \Xi \vdash_{\mathrm{val}} x : A}\,\mathrm{Ax}$$
- If the last used rule is ($\uparrow$) then we need to show $v\rho \in [\![A\rho]\!]^{\perp\perp}$. By induction hypothesis we know $v\rho \in [\![A\rho]\!]$, hence we can conclude using Lemma 4.5.29.
$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} v : A}{\Sigma \mid \Gamma ; \Xi \vdash v : A}\uparrow$$
- If the last used rule is ($\Rightarrow_{\mathrm{i}}$) then we need to show $(\lambda x.t)\rho \in [\![(A \Rightarrow B)\rho]\!] \setminus \{\Box\}$. As we are free to rename x we can assume $(\lambda x.t)\rho = \lambda x.t\rho$ and our goal hence rewrites as $\lambda x.t\rho \in [\![A\rho \Rightarrow B\rho]\!]$. By definition, we need to take $v \in [\![A\rho]\!] \setminus \{\Box\}$ and prove $(t\rho)[x := v] \in [\![B\rho]\!]^{\perp\perp}$. We can thus conclude by induction hypothesis using the valuation $\rho[x := v]$ since we know that $v \neq \Box$.
$$\frac{\Sigma , x : \iota \mid \Gamma, x : A ; \Xi \vdash t : B}{\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} \lambda x.t : A \Rightarrow B}\Rightarrow_{\mathrm{i}}$$
- If the last used rule is ($\Rightarrow_{\mathrm{e}}$) then we need to show $(t\,u)\rho = t\rho\,u\rho \in [\![B\rho]\!]^{\perp\perp}$. Let us take $\pi \in [\![B\rho]\!]^{\perp}$ and show $t\rho\,u\rho * \pi \in \bot\!\!\!\bot$. Since $t\rho\,u\rho * \pi \succ u\rho * [t\rho]\pi$ and $\bot\!\!\!\bot$ is saturated, it is enough to show $u\rho * [t\rho]\pi \in \bot\!\!\!\bot$. As $u\rho \in [\![A\rho]\!]^{\perp\perp}$ by induction hypothesis, it only remains to show $[t\rho]\pi \in [\![A\rho]\!]^{\perp}$. Let us take $v \in [\![A\rho]\!]$ and show that $v * [t\rho]\pi \in \bot\!\!\!\bot$. If $v = \Box$ then we have $\Box * [t\rho]\pi \succ \Box * \pi$ so it is enough to show $\Box * \pi \in \bot\!\!\!\bot$ since $\bot\!\!\!\bot$ is saturated. This is immediate as $\Box \in [\![B\rho]\!]$. Now, if $v \neq \Box$ then we have $v * [t\rho]\pi \succ t\rho * v \cdot \pi$ and since $\bot\!\!\!\bot$ is saturated, it is enough to show $t\rho * v \cdot \pi \in \bot\!\!\!\bot$. By induction hypothesis $t\rho \in [\![A\rho \Rightarrow B\rho]\!]^{\perp\perp}$, hence it only remains to show that we have $v \cdot \pi \in [\![A\rho \Rightarrow B\rho]\!]^{\perp}$. Let us now take $w \in [\![A\rho \Rightarrow B\rho]\!]$ and show $w * v \cdot \pi \in \bot\!\!\!\bot$. If $w = \Box$ then we have $\Box * v \cdot \pi \succ \Box * \pi$ and it is again enough that $\Box * \pi \in \bot\!\!\!\bot$ as $\bot\!\!\!\bot$ is saturated. If $w = \lambda x.f$ then we have $\lambda x.f * v \cdot \pi \succ f[x := v] * \pi$ and as $\bot\!\!\!\bot$ is saturated, it is enough to show $f[x := v] * \pi \in \bot\!\!\!\bot$. Since $\pi \in [\![B\rho]\!]^{\perp}$ it only remains to show $f[x := v] \in [\![B\rho]\!]^{\perp\perp}$, but this is true by definition of $[\![A\rho \Rightarrow B\rho]\!]$ since $v \in [\![A\rho]\!]$ and we have $v \neq \Box$.
$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \Rightarrow B \qquad \Sigma \mid \Gamma ; \Xi \vdash u : A}{\Sigma \mid \Gamma ; \Xi \vdash t\,u : B}\Rightarrow_{\mathrm{e}}$$

- If the last used rule is (μ) then we need to show that $(\mu\alpha.t)\rho \in [\![A\rho]\!]^{\perp\perp}$. As we are free to rename α, we can assume that $(\mu\alpha.t)\rho = \mu\alpha.t\rho$ and thus our goal rewrites as $\mu\alpha.t\rho \in [\![A\rho]\!]^{\perp\perp}$. Let us now take a stack $\pi \in [\![A\rho]\!]^{\perp}$ and show $\mu\alpha.t\rho * \pi \in \perp\!\!\!\perp$. Since $\mu\alpha.t\rho * \pi \succ t\rho[\alpha := \pi] * \pi$ and $\perp\!\!\!\perp$ is saturated, it is enough to show that we have $t\rho[\alpha := \pi] * \pi \in \perp\!\!\!\perp$. We can then conclude by induction hypothesis with the valuation $\rho[\alpha := \pi]$.
$$\frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash t : A}{\Sigma \mid \Gamma; \Xi \vdash \mu\alpha.t : A}\mu$$
- If the last used rule is $([-])$ then we need to show $([\alpha]t)\rho = [\rho(\alpha)]t\rho \in [\![B\rho]\!]^{\perp\perp}$. Let us take $\pi \in [\![B\rho]\!]^{\perp}$ and show $[\rho(\alpha)]t\rho * \pi \in \perp\!\!\!\perp$. Since $[\rho(\alpha)]t\rho * \pi \succ t\rho * \rho(\alpha)$ and $\perp\!\!\!\perp$ is saturated, it is enough to show $t\rho * \rho(\alpha) \in \perp\!\!\!\perp$. By induction hypothesis we know $t\rho \in [\![A\rho]\!]^{\perp\perp}$ and hence we only need to show $\rho(\alpha) \in [\![A\rho]\!]^{\perp}$. This is immediate since $\rho \Vdash \Gamma, \alpha : A^{\perp}$.
$$\frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash t : A}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash [\alpha]t : B}[-]$$
- If the last used rule is $(\forall_i)$ then we need to show $\nu\rho \in [\![(\forall\chi^s.A)\rho]\!] \setminus \{\square\}$. As we are free to rename χ we can show $\nu\rho \in [\![\forall\chi^s.A\rho]\!] \setminus \{\square\} = \cap_{\Phi \in [\![s]\!]} [\![A\rho[\chi := \Phi]]\!] \setminus \{\square\}$. Hence we take $\Phi \in [\![s]\!]$ and show $\nu\rho \in [\![A\rho[\chi := \Phi]]\!] \setminus \{\square\}$. We have $\nu\rho = \nu\rho[\chi := \Phi]$ since χ cannot appear in ν as otherwise the conclusion judgment would not be well-formed. We can thus conclude by induction hypothesis using the valuation $\rho[\chi := \Phi]$.
$$\frac{\Sigma, \chi : s \mid \Gamma; \Xi \vdash_{\overline{val}} \nu : A}{\Sigma \mid \Gamma; \Xi \vdash_{\overline{val}} \nu : \forall\chi^s.A}\forall_i$$
- If the last used rule is $(\forall_e)$ then we need to show $t\rho \in [\![(A[\chi := B])\rho]\!]^{\perp\perp}$. By induction hypothesis we have $t\rho \in [\![(\forall\chi^s.A)\rho]\!]^{\perp\perp}$, and as a consequence we need to show that $[\![(\forall\chi^s.A)\rho]\!]^{\perp\perp} \subseteq [\![(A[\chi := B])\rho]\!]^{\perp\perp}$. According to Lemma 4.5.32, it is enough to show $[\![(\forall\chi^s.A)\rho]\!] \subseteq [\![(A[\chi := B])\rho]\!]$. As we are free to rename χ our goal rewrites as $[\![\forall\chi^s.A\rho]\!] \subseteq [\![A\rho[\chi := B\rho]]\!]$. Since $[\![\forall\chi^s.A\rho]\!] = \cap_{\Phi \in [\![s]\!]} [\![A\rho[\chi := \Phi]]\!]$ and $B\rho \in [\![s]\!]$ we can conclude.
$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : \forall\chi^s.A \qquad \Sigma \vdash B : s}{\Sigma \mid \Gamma; \Xi \vdash t : A[\chi := B]}\forall_e$$
- If the last used rule is $(\exists_i)$ then we need to show $t\rho \in [\![(\exists\chi^s.A)\rho]\!]^{\perp\perp}$. By induction hypothesis we have $t\rho \in [\![(A[\chi := B])\rho]\!]^{\perp\perp}$, and as a consequence we need to show that $[\![(A[\chi := B])\rho]\!]^{\perp\perp} \subseteq [\![(\exists\chi^s.A)\rho]\!]^{\perp\perp}$. According to Lemma 4.5.32 it is enough to show $[\![(A[\chi := B])\rho]\!] \subseteq [\![(\exists\chi^s.A)\rho]\!]$ and as we are free to rename χ our goal rewrites as $[\![A\rho[\chi := B\rho]]\!] \subseteq [\![\exists\chi^s.A\rho]\!]$. Since $[\![\exists\chi^s.A\rho]\!] = \cup_{\Phi \in [\![s]\!]} [\![A\rho[\chi := \Phi]]\!]$ and $B\rho \in [\![s]\!]$ we can conclude.
$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : A[\chi := B] \qquad \Sigma \vdash B : s}{\Sigma \mid \Gamma; \Xi \vdash t : \exists\chi^s.A}\exists_i$$

- If the last used rule is $(\exists_e)$ then we need to show $t\rho \in [\![C\rho]\!]^{\perp\perp}$. Since we are free to rename χ we have $\rho(x) \in [\![(\exists\chi^s.A)\rho]\!] \setminus \{\square\} = [\![\exists\chi^s.A\rho]\!] \setminus \{\square\}$ using $\rho \Vdash \Gamma, x : \exists\chi^s.A$. By definition, the means that $\rho(x) \in \cup_{\Phi \in [\![s]\!]} [\![A\rho[\chi := \Phi]]\!]$ and thus there must be $\Phi \in [\![s]\!]$ such that $\rho(x) \in [\![A\rho[\chi := \Phi]]\!]$. We can hence conclude using the induction hypothesis with the valuation $\rho[\chi := \Phi]$.

$$\frac{\Sigma, x : \iota, \chi : s \mid \Gamma, x : A; \Xi \vdash t : C}{\Sigma, x : \iota \mid \Gamma, x : \exists\chi^s.A; \Xi \vdash t : C}\exists_e$$

- If the last used rule is $(\in_e)$ then we need to show $u\rho \in [\![C\rho]\!]^{\perp\perp}$. To apply the induction hypothesis, we need to check that $\rho(x) \in [\![A\rho]\!] \setminus \{\square\}$ and that $\rho(x) \equiv t\rho$, provided $\rho(x) \in [\![t\rho \in A\rho]\!] \setminus \{\square\}$. By definition $[\![t\rho \in A\rho]\!] = \{v \in [\![A\rho]\!] \mid v \equiv t\rho\} \cup \{\square\}$ so we must indeed have $\rho(x) \in [\![A\rho]\!]$ and $\rho(x) \equiv t\rho$ since $v \neq \square$.

$$\frac{\Sigma, x : \iota \mid \Gamma, x : A; \Xi, x \equiv t \vdash u : C}{\Sigma, x : \iota \mid \Gamma, x : t \in A; \Xi \vdash u : C}\in_e$$

- In the case of $(\in_i)$ we need to show that $v\rho \in [\![v\rho \in A\rho]\!] \setminus \{\square\}$. By induction hypothesis we have $v\rho \in [\![A\rho]\!] \setminus \{\square\}$ so we only have to prove $v\rho \equiv v\rho$, which follows from the reflexivity of $(\equiv)$.

$$\frac{\Sigma \mid \Gamma; \Xi \vdash_{\text{val}} v : A}{\Sigma \mid \Gamma; \Xi \vdash_{\text{val}} v : v \in A}\in_i$$

- If the last used rule is $(\restriction_i)$ then we need to show $t\rho \in [\![A\rho \restriction u_1\rho \equiv u_2\rho]\!]^{\perp\perp}$. Using the right premise we know that $u_1\rho \equiv u_2\rho$ and hence $[\![A\rho \restriction u_1\rho \equiv u_2\rho]\!] = [\![A\rho]\!]$ by definition. Consequently we have $[\![A\rho \restriction u_1\rho \equiv u_2\rho]\!]^{\perp\perp} = [\![A\rho]\!]^{\perp\perp}$ by Lemma 4.5.32 and hence we can conclude since $t\rho \in [\![A\rho]\!]^{\perp\perp}$ by induction hypothesis.

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : A \quad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma; \Xi \vdash t : A \restriction u_1 \equiv u_2}\restriction_i$$

- If the last used rule is $(\restriction_e)$ then we need to show $t\rho \in [\![C\rho]\!]^{\perp\perp}$. To be able to apply the induction hypothesis, we only have to show $\rho(x) \in [\![A\rho]\!] \setminus \{\square\}$ and $u_1\rho \equiv u_2\rho$ under the assumption that $\rho(x) \in [\![A\rho \restriction u_1\rho \equiv u_2\rho]\!] \setminus \{\square\} \neq \emptyset$. This immediately follows from the definition of $[\![A\rho \restriction u_1\rho \equiv u_2\rho]\!]$ since it contains $\rho(x)$ and $\rho(x) \neq \square$.

$$\frac{\Sigma, x : \iota \mid \Gamma, x : A; \Xi, u_1 \equiv u_2 \vdash t : C}{\Sigma, x : \iota \mid \Gamma, x : A \restriction u_1 \equiv u_2; \Xi \vdash t : C}\restriction_e$$

- If the last used rule is $(\times_i)$ then we need to show $\{(l_i = v_i\rho)_{i \in I}\} \in \left[\!\!\left[\{(l_i : A_i\rho)_{i \in I}\}\right]\!\!\right] \setminus \{\square\}$. By definition it is enough to show that $v_i\rho \in [\![A_i\rho]\!] \setminus \{\square\}$ for all $i \in I$. This exactly corresponds to the induction hypotheses.

$$\frac{[\Sigma \mid \Gamma; \Xi \vdash_{\text{val}} v_i : A_i]_{i \in I}}{\Sigma \mid \Gamma; \Xi \vdash_{\text{val}} \{(l_i = v_i)_{i \in I}\} : \{(l_i : A_i)_{i \in I}\}}\times_i$$

- If the last used rule is $(\times_e)$ then we need to show $(\nu.l_k)\rho = \nu\rho.l_k \in [\![A_k\rho]\!]^{\perp\perp}$. Let us take $\pi \in [\![A_k\rho]\!]^{\perp}$ and show $\nu\rho.l_k * \pi \in \perp\!\!\!\perp$. By induction hypothesis we know that $\nu\rho \in \left[\!\!\left[\{(l_i : A_i)_{i\in I}\}\rho\right]\!\!\right] \setminus \{\Box\}$ and thus $\nu\rho = \{(l_i = \nu_i)_{i\in I}\}$ with $\nu_i \in [\![A_i]\!] \setminus \{\Box\}$ for all $i \in I$. Since $\{(l_i = \nu_i\rho)_{i\in I}\}.l_k * \pi \succ \nu_k\rho * \pi$ and $\perp\!\!\!\perp$ is saturated, it is enough to show $\nu_k\rho * \pi \in \perp\!\!\!\perp$. This is immediate since $\pi \in [\![A_k\rho]\!]^{\perp}$ and $\nu_k\rho \in [\![A_k\rho]\!] \setminus \{\Box\}$.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} \nu : \{(l_i : A_i)_{i\in I}\} \quad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash \nu.l_k : A_k}\,{\scriptstyle \times_e}$$

- If the last used rule is $(+_i)$ then we need to show $C_k[\nu\rho] \in \left[\!\!\left[[(C_i : A_i\rho)_{i\in I}]\right]\!\!\right] \setminus \{\Box\}$. By definition, it is enough to show that $\nu\rho \in [\![A_k\rho]\!] \setminus \{\Box\}$ since $k \in I$. This is exactly the induction hypothesis.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} \nu : A_k \quad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} C_k[\nu] : [(C_i : A_i)_{i\in I}]}\,{\scriptstyle +_i}$$

- If the last used rule is $(+_e)$ then we need to show $[\nu \mid (C_i[x_i] \to t_i)_{i\in I}]\rho \in [\![B\rho]\!]^{\perp\perp}$. As we are free to rename x_i for all $i \in I$ our goal rewrites as $[\nu\rho \mid (C_i[x_i] \to t_i\rho)_{i\in I}] \in [\![B\rho]\!]^{\perp\perp}$. Let us take $\pi \in [\![B\rho]\!]^{\perp}$ and show $[\nu\rho \mid (C_i[x_i] \to t_i\rho)_{i\in I}] * \pi \in \perp\!\!\!\perp$. By the first induction hypothesis $\nu\rho \in \left[\!\!\left[[(C_i : A_i\rho)_{i\in I}]\right]\!\!\right] \setminus \{\Box\}$ so it must be that $\nu\rho = C_k[w]$ for some $k \in I$ and $w \in [\![A_k\rho]\!] \setminus \{\Box\}$. As $\perp\!\!\!\perp$ is saturated and $[C_k[w] \mid (C_i[x_i] \to t_i\rho)_{i\in I}] * \pi \succ t_k\rho[x_k := w] * \pi$ it is enough to show $t_k\rho[x_k := w] * \pi \in \perp\!\!\!\perp$. Consequently, we only have to show that $t_k\rho[x_k := w] \in [\![B\rho]\!]^{\perp\perp}$ since $\pi \in [\![B\rho]\!]^{\perp}$. Note that neither B, A_k nor ν may contain the variable x_k as otherwise some judgment would not be well-formed. As a consequence, we have $[\![B\rho[x_k := w]]\!] = [\![B\rho]\!]$, $[\![A_k\rho[x_k := w]]\!] = [\![A_k\rho]\!]$ and $\nu\rho[x_k := w] = \nu\rho$. To be able to conclude using the induction hypothesis, we need to show that the valuation $\rho[x_k := w]$ realizes $\Gamma, x_k : A_k ; \Xi, \nu \equiv C_k[x_k]$. As ρ realizes $\Gamma ; \Xi$ and as $w \in [\![A_k\rho]\!] \setminus \{\Box\}$ it only remains to show $\nu\rho \equiv C_k[w]$, which follows from the reflexivity of $(\equiv)$.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} \nu : [(C_i : A_i)_{i\in I}] \quad [\Sigma, x_i : \iota \mid \Gamma, x_i : A_i ; \Xi, \nu \equiv C_i[x_i] \vdash t_i : B]_{i\in I}}{\Sigma \mid \Gamma ; \Xi \vdash [\nu \mid (C_i[x_i] \to t_i)_{i\in I}] : B}\,{\scriptstyle +_e}$$

- If the last used rule is $(\equiv_{\iota,\iota})$ then we need to show $(\nu[x := w_2])\rho \in [\![(A[x := w_2])\rho]\!] \setminus \{\Box\}$. As we are free to rename x our goal rewrites as $\nu\rho[x := w_2\rho] \in [\![A\rho[x := w_2\rho]]\!] \setminus \{\Box\}$. According to our second premise we have $w_1\rho \equiv w_2\rho$ and thus Lemma 4.5.45 gives us $[\![A\rho[x := w_2\rho]]\!] = [\![A\rho[x := w_1\rho]]\!]$. We also obtain $\nu\rho[x := w_2\rho] \equiv \nu\rho[x := w_1\rho]$ using the fact that $(\equiv)$ is a congruence. Now, since we know that $[\![A\rho[x := w_1\rho]]\!]$ is closed under $(\equiv)$, it only remains to show that we have $\nu\rho[x := w_1\rho] \in [\![A\rho[x := w_1\rho]]\!] \setminus \{\Box\}$. To be able to conclude by induction hypothesis, we need to show that $\rho \Vdash \Gamma[x := w_1]$. As we know that $\rho \Vdash \Gamma[x := w_2]$ we only need to show that for every type B appearing in Γ we have $[\![(B[x := w_1])\rho]\!] = [\![(B[x := w_2])\rho]\!]$. As we are again free to rename x, this rewrites as $[\![B\rho[x := w_1\rho]]\!] = [\![B\rho[x := w_2\rho]]\!]$. We can then conclude the proof using Lemma 4.5.45 once more.

$$\frac{\Sigma \mid \Gamma[x := w_1]; \Xi \vdash_{\mathrm{val}} v[x := w_1] : A[x := w_1] \qquad \Xi \vdash w_1 \equiv w_2}{\Sigma \mid \Gamma[x := w_2]; \Xi \vdash_{\mathrm{val}} v[x := w_2] : A[x := w_2]} \equiv_{\iota,\iota}$$

– If the last used rule is $(\equiv_{\tau,\iota})$ then the proof is similar to the previous case, using the fact that $\Phi_1 = \Phi_2$ implies $\Phi_1^{\perp\perp} = \Phi_2^{\perp\perp}$ (i.e., Lemma 4.5.32).

$$\frac{\Sigma \mid \Gamma[x := w_1]; \Xi \vdash t[x := w_1] : A[x := w_1] \qquad \Xi \vdash w_1 \equiv w_2}{\Sigma \mid \Gamma[x := w_2]; \Xi \vdash t[x := w_2] : A[x := w_2]} \equiv_{\tau,\iota}$$

– If the last used rule is $(\equiv_{\iota,\tau})$ then the proof is the same as for the $(\equiv_{\iota,\iota})$ case, using Lemma 4.5.46 instead of Lemma 4.5.45.

$$\frac{\Sigma \mid \Gamma[a := u_1]; \Xi \vdash_{\mathrm{val}} v[a := u_1] : A[a := u_1] \qquad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma[a := u_2]; \Xi \vdash_{\mathrm{val}} v[a := u_2] : A[a := u_2]} \equiv_{\iota,\tau}$$

– If the last used rule is $(\equiv_{\tau,\tau})$ then the proof is the same as for the $(\equiv_{\tau,\iota})$ case, using again Lemma 4.5.46 instead of Lemma 4.5.45.

$$\frac{\Sigma \mid \Gamma[a := u_1]; \Xi \vdash t[a := u_1] : A[a := u_1] \qquad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma[a := u_2]; \Xi \vdash t[a := u_2] : A[a := u_2]} \equiv_{\tau,\tau}$$

□

Thanks to adequacy (Theorem 4.6.48), it is possible to prove properties of our language and type system by choosing appropriate poles. For instance, we can easily check that closed, typed terms normalise using a pole containing only processes reducing to a final state. Note that here, we need to fix the parameters of the system. We will consider $(\equiv)$ to be $(\equiv_{>})$ and R to be $(>)$ in the following theorem.

Theorem 4.6.49. Every closed, typed term normalises. More precisely, for all $t \in \Lambda^*$ such that $\vdash t : A$ is derivable there is $v \in \Lambda^*_\iota \setminus \{\Box\}$ such that $t * \varepsilon \succ^* v * \varepsilon$.

Proof. We consider the pole $\perp\!\!\!\perp = \{p \in \Lambda * \Pi \mid \exists\, v \in \Lambda_\iota,\ p \succ^* v * \varepsilon\}$ which is trivially saturated. We will first check that it is also $\equiv_{>}$-extensional. Let us suppose that $t \equiv_{>} u$ and that $t * \pi \in \perp\!\!\!\perp$. By definition of $\perp\!\!\!\perp$, there must be a value v such that $t * \pi \succ^* v * \varepsilon$, and thus we have $t * \pi \Downarrow_{>}$. Since $t \equiv_{>} u$ we can deduce $u * \pi \Downarrow_{>}$, and thus there must be a value w such that $u * \pi \succ w * \varepsilon$. This exactly means that $u * \pi \in \perp\!\!\!\perp$.

We can now apply Theorem 4.6.48 with the pole $\perp\!\!\!\perp$ and the empty substitution ρ_{id} to obtain $t \in [\![A]\!]^{\perp\perp}$. By definition, this means that $t * \xi \in \perp\!\!\!\perp$ for every stack $\xi \in [\![A]\!]^{\perp}$. In particular, we have $t * \varepsilon \in \perp\!\!\!\perp$ as we trivially have $\varepsilon \in [\![A]\!]^{\perp}$. This exactly means that there is a value $v \in \Lambda_\iota$ such that $t * \varepsilon \succ^* v * \varepsilon$. It remains to show that v is closed and different from $\Box$. This is immediate as none of our reduction rules may introduce $\Box$ or a free variable, and a closed typable term contain neither. □

Using similar techniques, it will be possible to prove a stronger safety property in Chapter 6. In particular, we will show that terms of type A reduce to values of $[\![A]\!]$, provided that A is a *pure data type*. Roughly, this will corespond to types that do not contain arrows. In particular, we will use the fact that the value interpretation of such a type does not depend on the choice of the pole.

Remark 4.6.50. We do not prove any type safety result here as our realizability model and type system will be altered in Chapters 6 and 5. In particular, the reduction relation ($\twoheadrightarrow$) that will be introduced in Chapter 5 will allow for simpler proofs.

4.7 Typing stacks

In our system, the typing rule for the named terms of the $\lambda\mu$-calculus is limited to stack variables. As a consequence, terms can only contain terms of the form $[\pi]t$ when π is a stack variable.

$$\frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash t : A}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash [\alpha]t : B}[-]$$

It is however possible to generalise this typing rule to accept more stacks, provided that they can be typed in some sense.

Definition 4.7.51. A stack judgement is a tuple of a typing context Γ, an equational context Ξ, a stack $\pi \in \Pi$ and a type $A \in \mathcal{T}$ that is denoted $\Gamma ; \Xi \vdash \pi : A^{\perp}$. We say that such a judgment is well-formed under the sorting context Σ, and we write $\Sigma \mid \Gamma ; \Xi \vdash \pi : A^{\perp}$, if and only if we have $\Sigma \vdash \Gamma$, $\Sigma \vdash \Xi$, $\Sigma \vdash \pi : \sigma$ and $\Sigma \vdash A : o$.

Using a stack judgment, the typing rule for named terms can replaced by the following, provided that we have enough rules for typing stacks.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \qquad \Sigma \mid \Gamma ; \Xi \vdash \pi : A^{\perp}}{\Sigma \mid \Gamma ; \Xi \vdash [\pi]t : B}[-]$$

To recover our previous typing rule for named terms, we need at least a rule for handling stack variables. This typing rule is very similar to the axiom rule (Ax), as it refers to the context.

$$\frac{}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash \alpha : A^{\perp}}\mathrm{Ax}^{\perp}$$

Using ($\mathrm{Ax}^{\perp}$), the old ([–]) rule can be derived as follows.

$$\frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash t : A \qquad \dfrac{}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp}; \Xi \vdash \alpha : A^{\perp}}\mathrm{Ax}^{\perp}}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A ; \Xi \vdash [\alpha]t : B}[-]$$

In addition to the ($Ax^{\perp}$) rule, we provide two rules for stacks formed by pushing a value on a stack, or stack frames. Note that it is also possible to give a typing rule for the empty stack. We will not use it here as it requires all the processes of the form $v * \varepsilon$ to be in the pole.

$$\frac{\Sigma \mid \Gamma; \Xi \vdash_{\overline{\text{val}}} v : A \qquad \Sigma \mid \Gamma; \Xi \vdash \pi : B^{\perp}}{\Sigma \mid \Gamma; \Xi \vdash v \cdot \pi : A \Rightarrow B^{\perp}}\,\cdot\text{-}\cdot$$

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : A \Rightarrow B \qquad \Sigma \mid \Gamma; \Xi \vdash \pi : B^{\perp}}{\Sigma \mid \Gamma; \Xi \vdash [t]\pi : A^{\perp}}\,[-]\text{-}$$

Semantically, we will interpret stack judgments in a similar way as value and term judgments. The new adequacy lemma will then involve the three forms of judgments.

Lemma 4.7.52. Let $A, B \in \mathcal{F}$ be two types such that $\vdash A : o$ and $\vdash B : o$. If $v \in [\![A]\!] \setminus \{\Box\}$ and $\pi \in [\![B]\!]^{\perp}$ then $v \cdot \pi \in [\![A \Rightarrow B]\!]^{\perp}$.

Proof. Let us take a value $w \in [\![A \Rightarrow B]\!]$ and show that $w * v \cdot \pi \in \perp\!\!\!\perp$. If $w = \Box$ then we have $\Box * v \cdot \pi \succ \Box * \pi$ and thus it is enough to show that $\Box * \pi \in \perp\!\!\!\perp$ since $\perp\!\!\!\perp$ is saturated. This is immediate as $\pi \in [\![B]\!]^{\perp}$ and $\Box \in [\![B]\!]$. If $w = \lambda x.t$ then since $\perp\!\!\!\perp$ is saturated and $\lambda x.t * v \cdot \pi \succ t[x := v] * \pi$ it is enough to show $t[x := v] * \pi \in \perp\!\!\!\perp$. As $\pi \in [\![B]\!]^{\perp}$ this amounts to showing that $t[x := v] \in [\![B]\!]^{\perp\perp}$, but this follows by definition of $[\![A \Rightarrow B]\!]$ as $v \in [\![A]\!] \setminus \{\Box\}$. □

Theorem 4.7.53. Let Σ be a sorting context, Γ be a typing context, Ξ be an equational context and $A \in \mathcal{F}$ be a type. Let ρ be a valuation over Σ such that $\rho \Vdash \Gamma$ and $\rho \Vdash \Xi$.

- If $\Sigma \mid \Gamma; \Xi \vdash t : A$ is derivable, then $t\rho \in [\![A\rho]\!]^{\perp\perp}$.
- If $\Sigma \mid \Gamma; \Xi \vdash \pi : A^{\perp}$ is derivable, then $\pi\rho \in [\![A\rho]\!]^{\perp}$.
- If $\Sigma \mid \Gamma; \Xi \vdash_{\overline{\text{val}}} v : A$ is derivable, then $v\rho \in [\![A\rho]\!] \setminus \{\Box\}$.

Proof. As for Theorem 4.6.48, the proof is done by induction of the derivation of the typing judgments. For all the rules of Figure 4.2 but ([−]) the proof is exactly the same as for Theorem 4.6.48. Four new cases are displayed below.

- If the last used rule is ([−]) then we need to show $[\pi\rho]t\rho \in [\![B\rho]\!]^{\perp\perp}$. Let us take $\xi \in [\![B\rho]\!]^{\perp}$ and show $[\pi\rho]t\rho * \pi \in \perp\!\!\!\perp$. Since $[\pi\rho]t\rho * \pi \succ t\rho * \pi\rho$ and $\perp\!\!\!\perp$ is saturated, it is enough to show $t\rho * \pi\rho \in \perp\!\!\!\perp$. This is immediate since $t\rho \in [\![A\rho]\!]^{\perp\perp}$ by induction hypothesis and $\pi\rho \in [\![A\rho]\!]^{\perp}$.

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : A \qquad \Sigma \mid \Gamma; \Xi \vdash \pi : A^{\perp}}{\Sigma \mid \Gamma; \Xi \vdash [\pi]t : B}\,[-]$$

- If the last used rule is ($Ax^{\perp}$) then we immediately get $\alpha\rho = \rho(\alpha) \in [\![A\rho]\!]^{\perp}$ since we know that we have $\rho \Vdash \Gamma, \alpha : A^{\perp}$.

$$\frac{}{\Sigma\,,\,\alpha : \sigma \mid \Gamma,\, \alpha : A^{\perp};\, \Xi \vdash \alpha : A^{\perp}}\,\mathrm{Ax}^{\perp}$$

- If the last used rule is $(-\cdot-)$ then we need to show $\nu\rho\,.\,\pi\rho \in [\![A\rho \Rightarrow B\rho]\!]^{\perp}$. This follows from Lemma 4.7.52 since we have $\nu\rho \in [\![A\rho]\!] \setminus \{\square\}$ and $\pi\rho \in [\![B\rho]\!]^{\perp}$ by induction hypothesis.
$$\frac{\Sigma \mid \Gamma;\, \Xi \vdash_{\mathrm{val}} \nu : A \qquad \Sigma \mid \Gamma;\, \Xi \vdash \pi : B^{\perp}}{\Sigma \mid \Gamma;\, \Xi \vdash \nu\,.\,\pi : A \Rightarrow B^{\perp}}\,-\cdot-$$
- If the last used rule is $([-]-)$ then we need to show $[t\rho]\pi\rho \in [\![A\rho]\!]^{\perp}$. Let us take $\nu \in [\![A\rho]\!]$ and show $\nu * [t\rho]\pi\rho \in \perp\!\!\!\perp$. If $\nu = \square$ then we have $\square * [t\rho]\pi\rho \succ \square * \pi\rho$ and since $\perp\!\!\!\perp$ is saturated it is enough to show $\square * \pi\rho \in \perp\!\!\!\perp$. This is immediate as $\square \in [\![B\rho]\!]$ and $\pi\rho \in [\![B\rho]\!]^{\perp}$ by induction hypothesis. We can now suppose that $\nu \neq \square$ and thus $\nu \in [\![A\rho]\!] \setminus \{\square\}$. As $\perp\!\!\!\perp$ is saturated and $\nu * [t\rho]\pi\rho \succ t\rho * \nu\,.\,\pi\rho$ it is enough to show $t\rho * \nu\,.\,\pi\rho \in \perp\!\!\!\perp$. By induction hypothesis we know $t\rho \in [\![A\rho \Rightarrow B\rho]\!]^{\perp\perp}$ and thus it is enough to show $\nu\,.\,\pi\rho \in [\![A\rho \Rightarrow B\rho]\!]^{\perp}$. This follows from Lemma 4.7.52 since we have $\nu \in [\![A\rho]\!] \setminus \{\square\}$ and $\pi\rho \in [\![B\rho]\!]^{\perp}$.
$$\frac{\Sigma \mid \Gamma;\, \Xi \vdash t : A \Rightarrow B \qquad \Sigma \mid \Gamma;\, \Xi \vdash \pi : B^{\perp}}{\Sigma \mid \Gamma;\, \Xi \vdash [t]\pi : A^{\perp}}\,[-]-$$

□

Of course, Theorem 4.6.49 can still be proved in the extended system with exactly the same proof, but using the extended adequacy lemma.

5 A model for a semantical value restriction

In this chapter, we consider the encoding of dependent types (i.e., a form of typed quantification) into our system. However, the expressiveness of such constructs is considerably limited by the value restriction. To solve this issue we introduce the notion of *semantical value restriction*, which allows the system to accept many more programs. Obtaining a model justifying *semantical value restriction* will require us to change our notions of reduction and observational equivalence.

5.1 Dependent function types

It is possible to encode dependent types into our system to obtain a form of quantification over the values (or terms) of a given type. This encoding relies on the well-known relativised quantification scheme, and it was suggested by Alexandre Miquel.

Definition 5.1.1. Let $A, B \in \mathcal{T}$ be two types, $x \in \mathcal{V}_\iota$ be a λ-variable and $a \in \mathcal{V}_\tau$ be a term variable. We will use the following notations for representing dependent function types ranging over values and terms respectively.

$$\Pi_{x\in A}B \coloneqq \forall x.(x\in A \Rightarrow B) \qquad\qquad \Pi_{a\in A}B \coloneqq \forall a.(a\in A \Rightarrow B)$$

Of course, we do not need to give additional sorting rules since $\Pi_{x\in A}B$ and $\Pi_{a\in A}B$ are only syntactic sugars. However, we can use the following typing rules to work with dependent functions more easily.

$$\frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash t : B[y \coloneqq x]}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} \lambda x.t : \Pi_{y\in A}B}\,\Pi_{\iota,i}$$

$$\frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash t : B[a \coloneqq x]}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} \lambda x.t : \Pi_{a\in A}B}\,\Pi_{\tau,i}$$

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : \Pi_{x \in A} B \qquad \Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : A}{\Sigma \mid \Gamma; \Xi \vdash t\ v : B[x := v]}\,\Pi_{\iota,e}$$

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : \Pi_{a \in A} B \qquad \Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : A}{\Sigma \mid \Gamma; \Xi \vdash t\ v : B[a := v]}\,\Pi_{\tau,e}$$

Note that both elimination rules require the value restriction on their second premise. In other words, dependent functions can only be applied to values. The four new typing rules can immediately be used to extend the type system as they are derivable. Hence, we do not have to extend our adequacy lemma (Theorem 4.6.48 or Theorem 4.7.53).

Lemma 5.1.2. The typing rules for the dependent function types are derivable.

Proof. The derivations for each of the new rules is given below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, x : \iota \mid \Gamma, x : A; \Xi \vdash t : B[y := x]}{\Sigma, y : \iota, x : \iota \mid \Gamma, x : A; \Xi \vdash t : B[y := x]}\mathrm{Wk}}{\Sigma, y : \iota, x : \iota \mid \Gamma, x : A; \Xi, x \equiv y \vdash t : B[y := x]}\mathrm{Wk}}{\Sigma, y : \iota, x : \iota \mid \Gamma, x : A; \Xi, x \equiv y \vdash t : B}\equiv_{\tau,\iota}}{\Sigma, y : \iota, x : \iota \mid \Gamma, x : y{\in}A; \Xi \vdash t : B}\in_e}{\Sigma, y : \iota \mid \Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.t : y{\in}A \Rightarrow B}\Rightarrow_i}{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.t : \forall y.(y{\in}A \Rightarrow B)}\forall_i}{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.t : \Pi_{y \in A} B}\mathrm{Def}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, x : \iota \mid \Gamma, x : A; \Xi \vdash t : B[a := x]}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : A; \Xi \vdash t : B[a := x]}\mathrm{Wk}}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : A; \Xi, x \equiv a \vdash t : B[a := x]}\mathrm{Wk}}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : A; \Xi, x \equiv a \vdash t : B}\equiv_{\tau,\tau}}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : a{\in}A; \Xi \vdash t : B}\in_e}{\Sigma, a : \tau \mid \Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.t : a{\in}A \Rightarrow B}\Rightarrow_i}{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.t : \forall a.(a{\in}A \Rightarrow B)}\forall_i}{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.t : \Pi_{a \in A} B}\mathrm{Def}$$

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma; \Xi \vdash t : \Pi_{x \in A} B}{\Sigma \mid \Gamma; \Xi \vdash t : \forall x.(x{\in}A \Rightarrow B)}\mathrm{Def}}{\Sigma \mid \Gamma; \Xi \vdash t : v{\in}A \Rightarrow B[x := v]}\forall_e \qquad \dfrac{\dfrac{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : A}{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : v{\in}A}\in_i}{\Sigma \mid \Gamma; \Xi \vdash v : v{\in}A}\uparrow}{\Sigma \mid \Gamma; \Xi \vdash t\ v : B[x := v]}\Rightarrow_e$$

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma ; \Xi \vdash t : \Pi_{a \in A} B}{\Sigma \mid \Gamma ; \Xi \vdash t : \forall a.(a \in A \Rightarrow B)}\,\text{Def}}{\Sigma \mid \Gamma ; \Xi \vdash t : \nu \in A \Rightarrow B[a := \nu]}\,\forall_e \qquad \dfrac{\dfrac{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} \nu : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} \nu : \nu \in A}\,\in_i}{\Sigma \mid \Gamma ; \Xi \vdash \nu : \nu \in A}\,\uparrow}{\Sigma \mid \Gamma ; \Xi \vdash t\ \nu : B[a := \nu]}\,\Rightarrow_e$$

□

Remark 5.1.3. Note that in the typing rules we consider, the variable bound by the dependent type (i.e., x or a respectively) does not appear free in the type A. In fact, this restriction is not necessary, and we could adapt the rules accordingly. However, the considered types would not correspond to dependent functions anymore.

Our encoding of the dependent products makes sense with respect to the semantics. Indeed, their interpretation is similar to the arrow type as it contains functions, but the type of their body depends on the value of the input.

Lemma 5.1.4. If $A, B \in \mathcal{T}$ are types such that $\vdash A : o$ and $x : \iota \vdash B : o$ are derivable then we have the following.

$$[\![\Pi_{x \in A} B]\!] = \left\{\lambda x.t \mid \forall\, \nu \in [\![A]\!] \setminus \{\square\},\, t[x := \nu] \in [\![B[x := \nu]]\!]^{\perp\perp}\right\} \cup \{\square\}$$

Similarly, if $a : \tau \vdash B : o$ is derivable then we have the following.

$$[\![\Pi_{a \in A} B]\!] = \left\{\lambda x.t \mid \forall\, \nu \in [\![A]\!] \setminus \{\square\},\, t[x := \nu] \in [\![B[a := \nu]]\!]^{\perp\perp}\right\} \cup \{\square\}$$

Proof. The proof is done using simple equational reasoning starting from the definition of $[\![\forall x.(x \in A \Rightarrow B)]\!]$ and $[\![\forall a.(a \in A \Rightarrow B)]\!]$ respectively.

$$\begin{aligned}
[\![\Pi_{x \in A} B]\!] &= [\![\forall x.(x \in A \Rightarrow B)]\!] \\
&= \textstyle\bigcap_{\Phi \in [\![\iota]\!]} [\![\Phi \in A \Rightarrow B[x := \Phi]]\!] \\
&= \textstyle\bigcap_{\Phi \in [\![\iota]\!]} \{\lambda x.t \mid \forall\, \nu \in [\![\Phi \in A]\!] \setminus \{\square\},\, t[x := \nu] \in [\![B[x := \Phi]]\!]^{\perp\perp}\} \cup \{\square\} \\
&= \{\lambda x.t \mid \forall\, \Phi \in [\![\iota]\!], \forall\, \nu \in \{w \in [\![A]\!] \mid w \equiv \Phi\} \setminus \{\square\},\, t[x := \nu] \in [\![B[x := \Phi]]\!]^{\perp\perp}\} \cup \{\square\} \\
&= \{\lambda x.t \mid \forall\, \nu \in [\![A]\!] \setminus \{\square\},\, t[x := \nu] \in [\![B[x := \nu]]\!]^{\perp\perp}\} \cup \{\square\}
\end{aligned}$$

The proof for $[\![\forall a.(a \in A \Rightarrow B)]\!]$ is similar but it requires Lemma 4.5.46. □

5.2 The limits of the value restriction

In languages like OCaml, the value restriction is not so problematic. Indeed, it is only required on the typing rule for polymorphism, and programmers almost never notice it as they mostly define functions (which are values). Moreover, if an instance of the value restriction is encountered, one can always use a dummy λ-abstraction (or an η-expansion) to transform a term into a value. A common example of this situation arises when working

with combinators (e.g., parser combinators) and partial application. As an example, let us consider the code below (written in OCaml syntax).

```
(* type 'a gr                              *)
(* val any : char gr                       *)
(* val seq : 'a gr → 'b gr → ('a * 'b) gr *)

let mono = seq any
let poly = fun g → seq any g
```

Here, `t gr` represents a parser (or grammar) returning a value of type `t`. The atomic parser `any` reads one character on the parsed stream and returns its value. The combinator `seq` takes as input two parsers and puts them in sequence to obtain a new parser. The return value of this new parser is a couple of the return values of the parsers it is build with. Now, if `seq` is partially applied with the parser `any`, the expected result is a combinator taking as input a parser `g`, and returning a parser for the sequence build with `any` and `g`. Of course, we want this new combinator to be as generic as possible, so that it can be applied to any parser, with any return type. However, the combinator `mono` defined above is only weakly polymorphic. This means that we will only be able to apply it to parsers of one fixed (but yet unknown) type. Of course, `seq any` is not a value, and the value restriction applies. To solve this lack of generality, we need to rely on an η-expansion as in `poly`, which has the expected type `'a gr -> (char, 'a) gr`.

As the definition of `seq` is probably something like `fun p1 p2 -> body`, the value restriction is actually not required in `mono`. Indeed, the evaluation of `seq any` would instantly reduce to a value (without any side-effect) in one β-reduction. This means that `seq any` could actually be considered a value. However, as discussed in Chapter 1, the value restriction is a very simple and elegant way to ensure the soundness of the type system. For this reason, the limitations discussed here do not pose a big enough problem to motivate the design of a more complex criterion in usual ML-like languages.

In our system, however, the value restriction is not only required on the introduction rule for the universal quantifier, but also on the introduction rule for the membership predicate. As it is used to derive the $(\Pi_{\iota,e})$ and $(\Pi_{\tau,e})$ rules, the value restriction is enforced on the argument of dependent functions. It is indeed necessary as an unrestricted $(\in_i)$ rule breaks the consistency (and the type safety) of our system. To support this claim, we will consider the system in which $(\in_i)$ has been replaced by the following (unrestricted) typing rule.

$$\frac{\Sigma \mid \Gamma, \Xi \vdash t : A}{\Sigma \mid \Gamma, \Xi \vdash t : t \in A}\in_{i,\bot}$$

In this system, the following (unrestricted) typing rule for the elimination of the dependent function type can be derived (see the typing derivation below the rule).

$$\frac{\Sigma \mid \Gamma, \Xi \vdash t : \Pi_{a \in A} B \qquad \Sigma \mid \Gamma, \Xi \vdash u : A}{\Sigma \mid \Gamma, \Xi \vdash t\ u : B[a := u]}\Pi_{e,\perp}$$

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma ; \Xi \vdash t : \Pi_{a \in A} B}{\Sigma \mid \Gamma ; \Xi \vdash t : \forall a.(a \in A \Rightarrow B)}\text{Def}}{\Sigma \mid \Gamma ; \Xi \vdash t : u \in A \Rightarrow B[a := u]}\forall_e \qquad \dfrac{\Sigma \mid \Gamma ; \Xi \vdash u : A}{\Sigma \mid \Gamma ; \Xi \vdash u : u \in A}\in_{i,\perp}}{\Sigma \mid \Gamma ; \Xi \vdash t\ u : B[a := u]}\Rightarrow_e$$

For convenience, we will also introduce a strong application rule that can also be derived directly using $(\in_{i,\perp})$. This typing rule will be used to keep track of the argument used for a function, while typing the function itself.

$$\frac{\Sigma \mid \Gamma ; \Sigma \vdash t : u \in A \Rightarrow B \qquad \Sigma \mid \Gamma ; \Sigma \vdash u : A}{\Sigma \mid \Gamma ; \Sigma \vdash t\ u : B}\Rightarrow_{e,\in,\perp}$$

$$\dfrac{\Sigma \mid \Gamma ; \Xi \vdash t : u \in A \Rightarrow B \qquad \dfrac{\Sigma \mid \Gamma ; \Xi \vdash u : A}{\Sigma \mid \Gamma ; \Xi \vdash u : u \in A}\in_{i,\perp}}{\Sigma \mid \Gamma ; \Xi \vdash t\ u : B}\Rightarrow_e$$

We will now build a counter-example to the consistency of the system extended with the $(\in_{i,\perp})$ rule for the membership predicate. We will construct a typable term that reduces to a value that does not belong to the expected type. We consider the term $t\ u$ defined thanks to the following subterms.

$$\begin{aligned} t &= \lambda f.(\lambda\_.v\ (f\ F[\{\}]))\ (f\ T[\{\}]) \\ v &= \lambda y.[y \mid F[\_] \to C_1[\{\}] \mid T[\_] \to C_0[\{\}]] \\ u &= \mu\alpha.\lambda x.[x \mid F[\_] \to T[\{\}] \mid T[\_] \to [\alpha]\lambda\_.F[\{\}]] \end{aligned}$$

In many reduction steps, we have $t\ u * \varepsilon \succ^* C_1[\{\}] * \varepsilon$. To obtain our counter-example, we will show that $t\ u$ has type $[C_0 : \{\} \mid C_1 : \{\} \upharpoonright u\ F[\{\}] \equiv F[\{\}]]$. For every stack $\pi \in \Pi$ we have the following reduction sequence.

$$\begin{aligned} u\ F[\{\}] * \pi &\succ F[\{\}] * [u]\pi \\ &\succ u * F[\{\}] \cdot \pi \\ &\succ \lambda x.[x \mid F[\_] \to T[\{\}] \mid T[\_] \to [F[\{\}] \cdot \pi]\lambda\_.F[\{\}]] * F[\{\}] \cdot \pi \\ &\succ [F[\{\}] \mid F[\_] \to T[\{\}] \mid T[\_] \to [F[\{\}] \cdot \pi]\lambda\_.F[\{\}]] * \pi \\ &\succ T[\{\}] * \pi \end{aligned}$$

As a consequence, Lemma 3.3.14 tells us that $u\ F[\{\}] \equiv T[\{\}]$. As a consequence, it cannot be that $u\ F[\{\}] \equiv F[\{\}]$ and thus the type $[C_0 : \{\} \mid C_1 : \{\} \upharpoonright u\ F[\{\}] \equiv F[\{\}]]$ is equivalent to $[C_0 : \{\}]$ (i.e., they have the same semantical interpretation). This means that if we manage to show

that $t\ u$ has type $[C_0 : \{\} \mid C_1 : \{\} \upharpoonright u\ F[\{\}] \equiv F[\{\}]]$, then we will have a contradiction to the type safety as $t\ u$ reduces to $C_1[\{\}]$, which is not contained in $[\![[C_0 : \{\}]]\!]$.

In the typing derivation displayed below, the type B is defined as $[T : \{\} \mid F : \{\}]$, the type C is defined as $[C_0 : \{\} \mid C_1 : \{\} \upharpoonright f\ F[\{\}] \equiv F[\{\}]]$, the context Γ_1 is defined as $f : B \Rightarrow B$, the context Γ_2 is defined as $f : B \Rightarrow B, y : B, y \equiv f\ F[\{\}]$ and Γ_3 is defined as $\alpha : B \Rightarrow B^{\perp}, x : B$. Note that the proof is split into three pieces so that it may fit in one page. The first two pieces, labeled p_1 and p_2 respectively, should be plugged in the main part of the proof (i.e., the third piece) at corresponding premises.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2, y \equiv F[\{\}] \vdash_{val} \{\} : \{\}}\times_i \quad y \equiv f\ F[\{\}], y \equiv F[\{\}] \vdash f\ F[\{\}] \equiv F[\{\}]}{\Gamma_2, y \equiv F[\{\}] \vdash_{val} \{\} : \{\} \upharpoonright f\ F[\{\}] \equiv F[\{\}]}\upharpoonright_e}{\Gamma_2, y \equiv F[\{\}] \vdash_{val} C_1[\{\}] : C}+_e \quad \dfrac{\dfrac{\dfrac{}{\Gamma_2, y \equiv T[\{\}] \vdash_{val} \{\} : \{\}}\times_i}{\Gamma_2, y \equiv T[\{\}] \vdash_{val} C_0[\{\}] : C}+_e}{\Gamma_2, y \equiv T[\{\}] \vdash C_0[\{\}] : C}\uparrow}{\Gamma_2 \vdash_{val} y : B}Ax \quad \Gamma_2, y \equiv F[\{\}] \vdash C_1[\{\}] : C\uparrow}{f : B \Rightarrow B, y : B, y \equiv f\ F[\{\}] \vdash [y \mid F[\_] \to C_1[\{\}] \mid T[\_] \to C_0[\{\}]] : C}+_e}{f : B \Rightarrow B, y : (f\ F[\{\}]) \in B \vdash [y \mid F[\_] \to C_1[\{\}] \mid T[\_] \to C_0[\{\}]] : C}\in_e}{f : B \Rightarrow B \vdash_{val} \lambda y.[y \mid F[\_] \to C_1[\{\}] \mid T[\_] \to C_0[\{\}]] : (f\ F[\{\}]) \in B \Rightarrow C}\Rightarrow_i}{}p_1$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash_{val} x : B}Ax \quad \dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash_{val} \{\} : \{\}}\times_i}{\Gamma_3 \vdash_{val} T[\{\}] : B}+_i}{\Gamma_3 \vdash T[\{\}] : B}\uparrow \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash_{val} \{\} : \{\}}\times_i}{\Gamma_3 \vdash_{val} F[\{\}] : B}+_i}{\Gamma_3 \vdash F[\{\}] : B}\uparrow}{\Gamma_3 \vdash_{val} \lambda\_.F[\{\}] : B \Rightarrow B}\Rightarrow_i}{\Gamma_3 \vdash \lambda\_.F[\{\}] : B \Rightarrow B}\uparrow}{\Gamma_3 \vdash [\alpha]\lambda\_.F[\{\}] : B}[-]}{\alpha : B \Rightarrow B^{\perp}, x : B \vdash [x \mid F[\_] \to T[\{\}] \mid T[\_] \to [\alpha]\lambda\_.F[\{\}]] : B}+_e}{\alpha : B \Rightarrow B^{\perp} \vdash_{val} \lambda x.[x \mid F[\_] \to T[\{\}] \mid T[\_] \to [\alpha]\lambda\_.F[\{\}]] : B \Rightarrow B}\Rightarrow_i}{\alpha : B \Rightarrow B^{\perp} \vdash \lambda x.[x \mid F[\_] \to T[\{\}] \mid T[\_] \to [\alpha]\lambda\_.F[\{\}]] : B \Rightarrow B}\uparrow}{\vdash \mu\alpha.\lambda x.[x \mid F[\_] \to T[\{\}] \mid T[\_] \to [\alpha]\lambda\_.F[\{\}]] : B \Rightarrow B}\mu}{}p_2$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_{val} \nu : (f\ F[\{\}]) \in B \Rightarrow C}p_1}{\Gamma_1 \vdash \nu : (f\ F[\{\}]) \in B \Rightarrow C}\uparrow \quad \dfrac{\dfrac{}{\Gamma_1 \vdash_{val} f : B \Rightarrow B}Ax}{\Gamma_1 \vdash f : B \Rightarrow B}\uparrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_{val} \{\} : \{\}}\times_i}{\Gamma_1 \vdash_{val} F[\{\}] : B}+_i}{\Gamma_1 \vdash F[\{\}] : B}\uparrow}{\Gamma_1 \vdash f\ F[\{\}] : B}\Rightarrow_e}{\Gamma_1 \vdash \nu\ (f\ F[\{\}]) : C}\Rightarrow_{e,\in,\perp} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash_{val} f : B \Rightarrow B}Ax}{\Gamma_1 \vdash f : B \Rightarrow B}\uparrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_{val} \{\} : \{\}}\times_i}{\Gamma_1 \vdash_{val} T[\{\}] : B}+_i}{\Gamma_1 \vdash T[\{\}] : B}\uparrow}{\Gamma_1 \vdash f\ T[\{\}] : B}\Rightarrow_e}{\Gamma_1 \vdash_{val} \lambda\_.\nu\ (f\ F[\{\}]) : B \Rightarrow C}\Rightarrow_i}{\Gamma_1 \vdash \lambda\_.\nu\ (f\ F[\{\}]) : B \Rightarrow C}\uparrow}{f : B \Rightarrow B \vdash (\lambda\_.\nu\ (f\ F[\{\}]))\ (f\ T[\{\}]) : [C_0 : \{\} \mid C_1 : \{\} \upharpoonright f\ F[\{\}] \equiv F[\{\}]]}\Pi_{\tau,i}}{\vdash_{val} \lambda f.(\lambda\_.\nu\ (f\ F[\{\}]))\ (f\ T[\{\}]) : \Pi_{\alpha \in B \Rightarrow B}[C_0 : \{\} \mid C_1 : \{\} \upharpoonright \alpha\ F[\{\}] \equiv F[\{\}]]}\uparrow \quad \dfrac{}{\vdash u : B \Rightarrow B}p_2}{\vdash \lambda f.(\lambda\_.\nu\ (f\ F[\{\}]))\ (f\ T[\{\}]) : \Pi_{\alpha \in B \Rightarrow B}[C_0 : \{\} \mid C_1 : \{\} \upharpoonright \alpha\ F[\{\}] \equiv F[\{\}]]}\Pi_{e,\perp}}{\vdash (\lambda f.(\lambda\_.\nu\ (f\ F[\{\}]))\ (f\ T[\{\}]))\ u : [C_0 : \{\} \mid C_1 : \{\} \upharpoonright u\ F[\{\}] \equiv F[\{\}]]}$$

As we have shown that the unrestricted introduction rule for the membership type is unsound, there is no hope of encoding an unrestricted elimination rule for the dependent function type into our system. Indeed, only the membership types are able to link the world of terms and the world of types thanks to their semantics. The limitation imposed by the value restriction on dependent functions leads to an expressiveness problem. Indeed, it completely forbids the composition of dependent functions which is very common in practice (especially for building proofs). In Chapter 7 we will consider several examples including proofs on the natural numbers and the concatenation of vectors (i.e., lists of fixed length) which could not be accepted under the usual syntactic restriction. For such examples to work in our system, we need another criterion accepting more programs, while still preserving soundness.

5.3 Semantical value restriction

As discussed in the previous section, value restriction is an issue in the presence of the membership type (and thus the dependent function type). To solve the related expressiveness problem, the author introduced the notion of *semantical value restriction* [*Lepigre 2016*]. The main idea is to relax the restriction to allow terms "behaving like values", and not syntactic values only. Thanks to our notion of observational equivalence, this property can be expressed easily in the syntax. Indeed, we will substitute a typing rule like

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : \Pi_{a \in A} B \qquad \Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A}{\Sigma \mid \Gamma ; \Xi \vdash t\ v : B[a \coloneqq v]}\,\Pi_{\tau,e}$$

with the following rule, where the restriction to values on the second premise is replaced by a third premise involving equivalence.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : \Pi_{a \in A} B \qquad \Sigma \mid \Gamma ; \Xi \vdash u : A \qquad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash t\ u : B[a \coloneqq u]}$$

This third premise requires the term u to be equivalent to some value v. Of course, it is perfectly possible for u not to be a syntactic value itself.

Remark 5.3.5. Semantical value restriction is a strict relaxation of value restriction. Indeed, value restriction exactly corresponds to a version of semantical value restriction in which we would only be able to use reflexivity to show that two terms are equivalent.

In the syntax, semantical value restriction will be presented as a simple extension of our type system with the following, seemingly obvious, typing rule.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash v : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A}\,{\downarrow}$$

It should not be confused with our (↑) rule, which premise is a value judgment and which conclusion is a term judgment. Our new (↓) rule allows us to transform a term judgment into a value judgment, provided that the considered term is a value. The $(\equiv_{\tau,\tau})$ rule can then be used to obtain a proof of $\Sigma \mid \Gamma ; \Xi \vdash v : A$ from a proof of $\Sigma \mid \Gamma ; \Xi \vdash t : A$, provided that $\Xi \vdash t \equiv v$ can be proved. This technique can be used, for example, to derive a relaxed version of the membership introduction typing rule.

Lemma 5.3.6. The following typing rule is derivable in our system extended with (↓).

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \qquad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash t : t \in A} \in_{i,\tau}$$

Proof. We can use the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma ; \Xi \vdash t : A \qquad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash v : A} \equiv_{\tau,\tau}}{\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} v : A} \downarrow}{\Sigma \mid \Gamma ; \Xi \vdash_{\mathrm{val}} v : v \in A} \in_{i}}{\Sigma \mid \Gamma ; \Xi \vdash v : v \in A} \uparrow \qquad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash t : t \in A} \equiv_{\tau,\tau}$$

□

We can then derive a relaxed version of the elimination rule for the dependent function type. Again, it requires proving that a term is equivalence to some value.

Lemma 5.3.7. The following typing rule is derivable in our system extended with (↓).

$$\frac{\Sigma \mid \Gamma ; \Sigma \vdash t : \Pi_{a \in A} B \qquad \Sigma \mid \Gamma ; \Sigma \vdash u : A \qquad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash t\ u : B[a \coloneqq u]} \Pi_{e,\tau}$$

Proof. Using Lemma 5.3.6 we can use the following derivation in the system extended with the $(\in_{i,\tau})$ rule.

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma ; \Sigma \vdash t : \Pi_{a \in A} B}{\Sigma \mid \Gamma ; \Sigma \vdash t : \forall a.(a \in A \Rightarrow B)} \mathrm{Def}}{\Sigma \mid \Gamma ; \Sigma \vdash t : u \in A \Rightarrow B[a \coloneqq u]} \forall_e \qquad \dfrac{\Sigma \mid \Gamma ; \Sigma \vdash u : A \qquad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma ; \Sigma \vdash u : u \in A} \in_{i,\tau}}{\Sigma \mid \Gamma ; \Xi \vdash t\ u : B[a \coloneqq u]} \Rightarrow_e$$

□

Additionally, semantical value restriction allows us to derive a strong typing rule for general application. It can be seen as a relaxed form of the following rule (which can be derived easily in the initial system).

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : v{\in}A \Rightarrow B \qquad \Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : A}{\Sigma \mid \Gamma; \Xi \vdash t\ v : B}{\Rightarrow_{e,\in}}$$

The aim of such a rule is to keep track of the argument that will be applied to a function, when typing the function itself. This is useful, in particular, when this argument is used in a case analysis. However, in its restricted form, this typing rule is not very useful since a term like $(\lambda x.[x \mid (C_i[x_i] \to t_i)_{i\in I}])\ v$ is equivalent to $[v \mid (C_i[x_i] \to t_i)_{i\in I}]$, and thus the same effect can be obtained using the $(\equiv_{\tau,\tau})$. With semantical value restriction, we can derive a stronger, relaxed rule.

Lemma 5.3.8. The following typing rule is derivable in our system extended with $(\downarrow)$.

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : u{\in}A \Rightarrow B \qquad \Sigma \mid \Gamma; \Xi \vdash u : A \qquad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma; \Xi \vdash t\ u : B}{\Rightarrow_{e,\in,\tau}}$$

Proof. Using again Lemma 5.3.6 to obtain the $(\in_{i,\tau})$ rule, we can use the following derivation.

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : u{\in}A \Rightarrow B \qquad \dfrac{\Sigma \mid \Gamma; \Xi \vdash u : A \qquad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma; \Xi \vdash u : u{\in}A}{\in_{i,\tau}}}{\Sigma \mid \Gamma; \Xi \vdash t\ u : B}{\Rightarrow_e}$$

□

The $(\Rightarrow_{e,\in,\tau})$ rule can be used to obtain the following rule for a generalised form of case analysis ranging over terms (and not only values).

$$\frac{\Sigma \mid \Gamma; \Xi \vdash t : [(C_i : A_i)_{i\in I}] \qquad [\Sigma, x_i : \iota \mid \Gamma, x_i : A_i; \Xi, t \equiv C_i[x_i] \vdash t_i : B]_{i\in I} \qquad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma; \Xi \vdash (\lambda x.[x \mid (C_i[x_i] \to t_i)_{i\in I}])\ t : B}{+_{e,\tau}}$$

Without such a rule, it would be impossible to preserve the equivalences of the form $t \equiv C_i[x_i]$ in the premises. We would only know that $x \equiv C_i[x_i]$, which is not enough since we would have no way of linking x to t. The derivation of $(+_{e,\tau})$ is given below. Note that we will omit the sorting contexts for the proof to be more concise. We will also use the notation Δ for the context $\Gamma, x : [(C_i : A_i)_{i\in I}]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta; \Xi, x \equiv t \vdash_{\mathrm{val}} x : [(C_i : A_i)_{i\in I}]}{\mathrm{Ax}} \qquad \left[\dfrac{\dfrac{\Delta, x_i : A_i; \Xi, t \equiv C_i[x_i] \vdash t_i : B}{\Delta, x_i : A_i; \Xi, x \equiv t; t \equiv C_i[x_i] \vdash t_i : B}{\mathrm{Wk}}}{\Delta, x_i : A_i; \Xi, x \equiv t; x \equiv C_i[x_i] \vdash t_i : B}{\equiv_{\tau,\tau}}\right]_{i\in I}}{\Delta; \Xi, x \equiv t \vdash [x \mid (C_i[x_i] \to t_i)_{i\in I}] : B}{+_e}}{\Delta; \Xi \vdash [x \mid (C_i[x_i] \to t_i)_{i\in I}] : B}{\in_e}}{\Gamma; \Xi \vdash_{\mathrm{val}} \lambda x.[x \mid (C_i[x_i] \to t_i)_{i\in I}] : t{\in}[(C_i : A_i)_{i\in I}] \Rightarrow B}{\Rightarrow_i}}{\Gamma; \Xi \vdash \lambda x.[x \mid (C_i[x_i] \to t_i)_{i\in I}] : t{\in}[(C_i : A_i)_{i\in I}] \Rightarrow B}{\uparrow} \qquad \Gamma; \Xi \vdash t : [(C_i : A_i)_{i\in I}] \qquad \Xi \vdash t \equiv v}{\Gamma; \Xi \vdash (\lambda x.[x \mid (C_i[x_i] \to t_i)_{i\in I}])\ t : B}{\Rightarrow_{e,\in,\tau}}$$

Of course, the derivation of all the new typing rules introduced in this section is conditioned to the soundness of semantical value restriction. As a consequence, we need to adapt our model so that the (↓) rule is adequate. As we will see in the next sections, the required modifications are highly non-trivial and require changing our notions of reduction and equivalence.

5.4 Semantics for semantical value restriction

As mentioned in the previous section, semantical value restriction can be enabled in our system by extending it with the (↓) typing rule (recalled below).

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash v : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\text{val}}} v : A}\downarrow$$

In order to give a semantical justification to this rule (i.e., to show that it is adequate), we need to find a model in which the following property holds for every $\Phi \in [\![o]\!]$.

$$\Phi^{\perp\perp} \cap \Lambda_\iota^* \subseteq \Phi$$

This property is not true in general, and in particular it is not true in our current model. A counter-example is given in the following theorem.

Theorem 5.4.9. If we choose $(\equiv_>)$ as our equivalence relation and $(>)$ as our reduction relation then there is a pole $\bot\!\!\!\bot$ and a set of values $\Phi \in [\![o]\!]$ such that $\Phi^{\perp\perp} \cap \Lambda_\iota^*$ contains strictly more values that Φ.

Proof. Let us consider the pole $\bot\!\!\!\bot = \{p \in \Lambda \times \Pi \mid \exists\, v \in \Lambda_\iota,\, p \succ^* v * \varepsilon\}$ and show that the set $\Phi = \{v \in \Lambda_\iota^* \mid v \equiv_> \lambda x.\Omega\} \cup \{\square\}$ is suitable. As we have $\Lambda_\iota^* \subseteq \Lambda_\iota^{*\perp\perp}$ by Lemma 4.5.29 it is enough to show $\Phi^\perp = \Lambda_\iota^{*\perp}$ since in this case we get $\Lambda_\iota^* \subseteq \Phi^{\perp\perp} = \Lambda_\iota^{*\perp\perp}$. Of course, there are many values that are in Λ_ι^*, but not in Φ. For example, we have $\{\} \in \Lambda_\iota^*$ but $\{\} \not\equiv_> \lambda x.\Omega$ according to Theorem 3.4.35.

Since $\Phi \subseteq \Lambda_\iota^*$ we must have $\Phi^\perp \supseteq \Lambda_\iota^{*\perp}$ by Lemma 4.5.30, so it only remains to show that $\Phi^\perp \subseteq \Lambda_\iota^{*\perp}$. Let us take a stack $\pi \in \Phi^\perp$. By definition, we know $w * \pi \in \bot\!\!\!\bot$ for all $w \in \Phi$. Let us take a value $v_0 \in \Lambda_\iota^*$ and show that $v_0 * \pi \in \bot\!\!\!\bot$. If $v_0 = \square$ then this is immediate since $\square \in \Phi$ so we can assume $v_0 \neq \square$. We reason by case on the form of the stack π.

- If $\pi = \varepsilon$, then $v_0 * \pi \in \bot\!\!\!\bot$ by definition of $\bot\!\!\!\bot$.
- If $\pi = \alpha$ for some $\alpha \in \mathcal{V}_\sigma$ then it cannot be that $\pi \in \Phi^\perp$. If it were the case, we would have $\lambda x.\Omega * \alpha \in \bot\!\!\!\bot$, which cannot be true since this process is blocked.
- If $\pi = w \cdot \xi$ for some value $w \in \Lambda_\iota$ and stack $\xi \in \Pi$ then it cannot be that $\pi \in \Phi^\perp$. If it were the case, we would have $\lambda x.\Omega * w \cdot \xi \succ \Omega * \xi \in \bot\!\!\!\bot$, which cannot be true since this process is non-terminating.

- If $\pi = [t]\xi$ for some term $t \in \Lambda$ and stack $\xi \in \Pi$ then we consider the reduction of the process $t * z \,.\, \xi$ where $z \in \mathcal{V}_\iota$ is a fresh λ-variable. We know that $t * z \,.\, \xi$ cannot be non-terminating, as otherwise $(t * z \,.\, \xi)[z \coloneqq \lambda x.\Omega] = t * \lambda x.\Omega \,.\, \xi$ would also be non-terminating according to Lemma 2.6.44. This would contradict $[t]\xi \in \Phi^\perp$ since we have $\lambda x.\Omega * [t]\xi \succ t * \lambda x.\Omega \,.\, \xi$. Consequently, there is $q \in \Lambda \times \Pi$ such that $z * [t]\xi \succ q$. We now reason by case analysis following Lemma 2.6.46. If q is final, then $q[z \coloneqq v]$ is also final by Lemma 2.6.44, and thus $v * [t]\xi \in \perp\!\!\!\perp$. In all the other cases but $z * [t_0]\pi_0$ the process $q[z \coloneqq \lambda x.\Omega]$ is either still blocked or non-terminating, which contradicts the fact that $[t]\xi \in \Phi^\perp$. In the last case we can iterate the proof in a similar way as for Theorem 3.1.7. In the case where also get that $\lambda x.\Omega * [t]\xi \in \perp\!\!\!\perp$ since this process is non-terminating. □

The idea now is to use our equivalence relation $(\equiv_\succ)$ to extend the reduction relation $(\succ)$ with a new, surprising reduction rule. It will reduce processes having the form $\delta_{v,w} * \pi$ to $v * \pi$ in the case where $v \not\equiv_\succ w$, and remain stuck otherwise. With such a reduction rule, the definitions of reduction and equivalence become interdependent. Consequently, we need to be very careful so that everything remains well-defined. We will rely on a stratified construction of both reduction and equivalence.

Definition 5.4.10. For every $i \in \mathbb{N}$ we define two relations $(\twoheadrightarrow_i)$ and $(\equiv_i)$ as follows.

$$\begin{aligned}(\twoheadrightarrow_i) &= (\succ) \cup \left\{(\delta_{v,w} * \pi, v * \pi) \mid \exists\, j < i\,,\, v \not\equiv_j w\right\}\\ (\equiv_i) &= \left\{(t, u) \mid \forall\, j \leqslant i\,, \forall\, \pi \in \Pi, \forall\, \rho \in \mathcal{S}\,,\, t\rho * \pi \Downarrow_j \Leftrightarrow\, u\rho * \pi \Downarrow_j\right\}\end{aligned}$$

Here, all the relations are well-defined as there is no circularity. In particular, we have $(\twoheadrightarrow_0) = (\succ)$ since there is no natural number that is strictly smaller than 0. This implies that we also have $(\equiv_0) = (\equiv_\succ)$

Lemma 5.4.11. For every $i \in \mathbb{N}$, the relation $(\equiv_i)$ is an equivalence relation.

Proof. Immediate. □

We can then define our actual reduction relation and equivalence relation as a union and an intersection over the previously defined relations.

Definition 5.4.12. We define a reduction relation $(\twoheadrightarrow)$ and an equivalence relation $(\equiv)$.

$$(\twoheadrightarrow) = \bigcup_{i \in \mathbb{N}} (\twoheadrightarrow_i) \qquad\qquad (\equiv) = \bigcap_{i \in \mathbb{N}} (\equiv_i)$$

Remark 5.4.13. We have $(\twoheadrightarrow_i) \subseteq (\twoheadrightarrow_{i+1})$ and $(\equiv_{i+1}) \subseteq (\equiv_i)$. Consequently, the construction of $(\twoheadrightarrow_i)_{i\in\mathbb{N}}$ and $(\equiv_i)_{i\in\mathbb{N}}$ converges. In fact, $(\twoheadrightarrow)$ and $(\equiv)$ form a fixpoint at ordinal ω. Surprisingly, this property will not be explicitly required in the following.

Lemma 5.4.14. The relation $(\equiv)$ is an equivalence relation.

Proof. Immediate using Lemma 5.4.11 since an intersection of equivalence relations is itself an equivalence relation. □

For convenience, the definition of our new reduction and equivalence relations can be expressed in the following way, where $(\not\equiv)$ denotes the negation of $(\equiv)$.

$$\begin{aligned}
(\equiv) &= \left\{(t, u) \mid \forall i \in \mathbb{N}, \forall \pi \in \Pi, \forall \rho \in \mathcal{S}, t\rho * \pi \Downarrow_i \Leftrightarrow u\rho * \pi \Downarrow_i\right\} \\
(\not\equiv) &= \left\{(t, u), (u, t) \mid \exists i \in \mathbb{N}, \exists \pi \in \Pi, \exists \rho \in \mathcal{S}, t\rho * \pi \Downarrow_i \wedge u\rho * \pi \Uparrow_i\right\} \\
(\twoheadrightarrow) &= (\succ) \cup \{(\delta_{v,w} * \pi, v * \pi) \mid v \not\equiv w\}
\end{aligned}$$

Note that the definition of $(\twoheadrightarrow)$ corresponds exactly to what we aimed for: an extension of $(\succ)$ with a reduction rule for δ-like terms carrying two non-equivalent values.

Theorem 5.4.15. Let $\perp\!\!\!\perp \subseteq \Lambda\times\Pi$ be a $\twoheadrightarrow$-saturated set of processes such that for every $p \in \perp\!\!\!\perp$ we have $p \Downarrow_{\twoheadrightarrow}$ and $\square * \varepsilon \in \perp\!\!\!\perp$. If $\Phi \in [\![o]\!]$ then we have the following property.

$$\Phi^{\perp\perp} \cap \Lambda_\iota \subseteq \Phi$$

Proof. We need to show that for every value $v \in \Phi^{\perp\perp}$ we also have $v \in \Phi$. We are going to show the contrapositive, so let us assume $v \notin \Phi$ and show $v \notin \Phi^{\perp\perp}$. By definition, we need to find a stack $\pi \in \Phi^\perp$ such that $v * \pi \notin \perp\!\!\!\perp$. We will take $\pi = [\lambda x.\delta_{x,v}][\square]\varepsilon$ and show that it is suitable. We first need to prove that $\pi \in \Phi^\perp$ so we take $w \in \Phi$ and we show $w * \pi \in \perp\!\!\!\perp$. If $w = \square$ then $\square * \pi \twoheadrightarrow \square * [\square]\varepsilon \twoheadrightarrow \square * \varepsilon \in \perp\!\!\!\perp$ and we can thus conclude since $\perp\!\!\!\perp$ is $\twoheadrightarrow$-saturated. Otherwise, if $w \neq \square$ then we have the following.

$$w * \pi \twoheadrightarrow \lambda x.\delta_{x,v} * w\,.\,[\square]\varepsilon \twoheadrightarrow \delta_{w,v} * [\square]\varepsilon \twoheadrightarrow w * [\square]\varepsilon \twoheadrightarrow \square * w\,.\,\varepsilon \twoheadrightarrow \square * \varepsilon \in \perp\!\!\!\perp$$

Note that we have $v \not\equiv w$ since $v \notin \Phi$ and Φ is closed under $(\equiv)$. We can thus conclude using again the fact that $\perp\!\!\!\perp$ is $\twoheadrightarrow$-saturated. It now remains to show that $v * \pi_0 \notin \perp\!\!\!\perp$. It cannot be that $v = \square$ since we assumed $v \notin \Phi$. As a consequence, we have the following.

$$v * \pi \twoheadrightarrow \lambda x.\delta_{x,v} * v\,.\,[\square]\varepsilon \twoheadrightarrow \delta_{v,v} * [\square]\varepsilon \Uparrow_{\twoheadrightarrow}$$

Here, $\delta_{v,v} * [\square]\varepsilon$ is blocked since $v \equiv v$ by reflexivity of $(\equiv)$ and thus $v * \pi \notin \perp\!\!\!\perp$. □

Remark 5.4.16. Theorem 5.4.15 only gives us the required property of the model for poles of terminating processes (i.e., processes that eventually reduce to a final states). This limitation is not a problem here as we will only consider poles having this property.

5.5 Final instance of our model

The reduction relation ($\twoheadrightarrow$) and the equivalence relation ($\equiv$) give us an essential property for semantical value restriction. However, we need to verify some properties before being able to fix the parameters of our model definitively. There is no problem in adopting ($\twoheadrightarrow$) as our reduction relation as it contains ($\succ$). Nevertheless, we need to check that our equivalence relation ($\equiv$) is a congruence and that it is compatible with ($\equiv_{\succ}$). Let us first show that it is indeed a congruence.

Theorem 5.5.17. Let $t, u \in \Lambda$ be two terms and $\rho \in \mathcal{S}$ be a substitution. If we have $t \equiv u$ then $t\rho \equiv u\rho$.

Proof. Let us take $i_0 \in \mathbb{N}$, $\rho_0 \in \mathcal{S}$ and $\pi_0 \in \Pi$ and prove $(t\rho)\rho_0 * \pi_0 \Downarrow_{i_0} \Leftrightarrow (u\rho)\rho_0 * \pi_0 \Downarrow_{i_0}$, which can be rewritten as $t(\rho_0 \circ \rho) * \pi_0 \Downarrow_{i_0} \Leftrightarrow u(\rho_0 \circ \rho) * \pi_0 \Downarrow_{i_0}$. We can thus conclude by definition of $t \equiv u$ using i_0, the substitution $\rho_0 \circ \rho$ and the stack π_0. □

Theorem 5.5.18. Let $v_1, v_2 \in \Lambda_\iota$ be values, $t \in \Lambda$ be a term and $x \in \mathcal{V}_\iota$ be a λ-variable. If $v_1 \equiv v_2$ then we have $t[x := v_1] \equiv t[x := v_2]$.

Proof. We are going to prove the contrapositive so we suppose $t[x := v_1] \not\equiv t[x := v_2]$ and we show $v_1 \not\equiv v_2$. Let us first assume that neither v_1 nor v_2 is equal to $\square$ or to a λ-variables. By definition, we know that there $i \in \mathbb{N}$, $\pi \in \Pi$ and $\rho \in \mathcal{S}$ such that we have $(t[x := v_1])\rho * \pi \Downarrow_i$ and $(t[x := v_2])\rho * \pi \Uparrow_i$ (up to symmetry). As x is bound we can rename it so that $(t[x := v_1])\rho = t\rho[x := v_1\rho]$ and $(t[x := v_2])\rho = t\rho[x := v_2\rho]$. To finish the proof, we need to find $i_0 \in \mathbb{N}$, $\pi_0 \in \Pi$ and $\rho_0 \in \mathcal{S}$ such that $v_1\rho_0 * \pi_0 \Downarrow_{i_0}$ and $v_2\rho_0 * \pi_0 \Uparrow_{i_0}$ (up to symmetry). We can take $i_0 = i$, $\pi_0 = [\lambda x.t\rho]\pi$ and $\rho_0 = \rho$ since by definition we have $v_1\rho * [\lambda x.t\rho]\pi \twoheadrightarrow_i^* t\rho[x := v_1\rho] * \pi \Downarrow_i$ and $v_2\rho * [\lambda x.t\rho]\pi \twoheadrightarrow_i^* t\rho[x := v_2\rho] * \pi \Uparrow_i$. Note that here, it is essential that $v_1\rho$ and $v_2\rho$ are not equal to $\square$ or to some λ-variable as otherwise the first reduction steps could not be taken.

It remains to show that $v_1 \not\equiv v_2$ in the cases where v_1, v_2 or both are equal to $\square$ or a λ-variable. First, we can assume that $v_1 \neq v_2$ as otherwise we would immediately get a contradiction with $t[x := v_1] \not\equiv t[x := v_2]$ by reflexivity of ($\equiv$). As a consequence, we cannot have $v_1 = v_2 = \square$ or $v_1 = v_2 = x \in \mathcal{V}_\iota$. Now, in all the other cases we must have $v_1 \not\equiv_{\succ} v_2$ according to Theorems 3.5.39 and 3.5.40 so we get $v_1 \not\equiv v_2$. □

Lemma 5.5.19. Let $p \in \Lambda\times\Pi$ be a process, $t \in \Lambda$ be a term and $a \in \mathcal{V}_\tau$ be a term variable. If we have $p[a := t] \Downarrow_k$ for some $k \in \mathbb{N}$ then there is a blocked process $q \in \Lambda\times\Pi$ such that $p \succ^* q$ and either

- $q = v * \varepsilon$ for some value $v \in \Lambda_\iota$,
- $q = a * \pi$ for some stack $\pi \in \Pi$,
- $k \neq 0$ and $q = \delta_{v,w} * \pi$ for some values $v, w \in \Lambda_\iota$ and $\pi \in \Pi$. Moreover, in this case we know that $v[a := t] \not\equiv_j w[a := t]$ for some $j < k$.

Proof. If p is non-terminating then so is $p[a := t]$ according to Lemma 2.6.44. Since $(\succ) \subseteq (\twoheadrightarrow_k)$ this contradicts $p[a := t] \Downarrow_k$ and thus there must be a blocked process $q \in \Lambda\times\Pi$ such that $p \succ^* q$. Using Theorem 2.5.42 we obtain $p[a := t] \succ^* q[a := t]$, which tells us that $q[a := t] \Downarrow_k$. This means that q cannot be stuck, as otherwise $q[a := t]$ would also be stuck by Lemma 2.6.44 and this would contradict $q[a := t] \Downarrow_k$. Let us now suppose that $p = \delta_{v,w} * \pi$ for some $v, w \in \Lambda_\iota$ and $\pi \in \Pi$. Since $\delta_{v\rho,w\rho} * \pi \Downarrow_k$ there must be $j < k$ (and thus $k \neq 0$) such that $v\rho \not\equiv_j w\rho$, otherwise we would obtain a contradiction. According to Lemma 2.6.46 it remains to rule the following forms for q, where $b \neq a$.

$$x.l_k * \pi \qquad x * v\,.\,\pi \qquad [x \mid (C_i[x_i] \to t_i)_{i\in I}] * \pi$$
$$x * [t]\pi \qquad R_{x,u} * \pi \qquad b * \pi \qquad v * \alpha$$

If q was of one of these forms, then $q[a := t]$ would still be blocked, which would again contradict $q[a := t] \Downarrow_k$. □

Lemma 5.5.20. Let $u_1, u_2, t \in \Lambda$ be three terms and $a \in \mathcal{V}_\tau$ be a term variable. For all $k \in \mathbb{N}$, if $u_1 \equiv_k u_2$ then $t[a := u_1] \equiv_k t[a := u_2]$.

Proof. We do a proof by induction on k. If $k = 0$ then this property exactly corresponds to Theorem 3.1.7. Let us now take $k > 0$, suppose $u_1 \equiv_k u_2$ and show $t[a := u_1] \equiv_k t[a := u_2]$. By definition, we need to take $\pi \in \Pi$, $\rho \in \mathcal{S}$ and show the following.

$$(t[a := u_1])\rho * \pi \Downarrow_k \Leftrightarrow (t[a := u_2])\rho * \pi \Downarrow_k$$

Since a is bound we are free to rename it so we may assume $(t[a := u_1])\rho = t\rho[a := u_1\rho]$, $(t[a := u_2])\rho = t\rho[a := u_2\rho]$ and $a \notin \mathrm{FV}_\tau(\pi) \cup \mathrm{FV}_\tau(t_1) \cup \mathrm{FV}_\tau(t_2)$. By symmetry, we can thus suppose $t\rho[a := u_1\rho] * \pi \Downarrow_k$ and show $t\rho[a := u_2\rho] * \pi \Downarrow_k$.

We will now build a sequence $(t_i, \pi_i, l_i)_{i\in I}$ defined in such a way that for all $i \in I$ we have $t\rho[a := u_1\rho] * \pi \twoheadrightarrow_k^* t_i[a := u_1\rho] * \pi_i[a := u_1\rho]$ in l_i steps. We will also require $(l_i)_{i\in I}$ to be increasing and to have a strictly increasing subsequence. Under this condition our sequence will be finite. If it was infinite, $t\rho[a := u_1\rho] * \pi$ would be non-terminating, and this would contradict $t\rho[a := u_1\rho] * \pi \Downarrow_k$. As a consequence, our sequence has a finite

number $n+1$ of elements (for some $n \in \mathbb{N}$), and we can denote it $(t_i, \pi_i, l_i)_{i \leqslant n}$. To show that $(l_i)_{i \leqslant n}$ has a strictly increasing subsequence, we will ensure that it does not have three equal consecutive values.

To define (t_0, π_0, l_0) we consider the reduction of the process $t\rho * \pi$. Since we have $(t\rho * \pi)[a := u_1\rho] = t\rho[a := u_1] * \pi \Downarrow_k$ we can apply Lemma 5.5.19 to obtain a blocked process p such that $t\rho * \pi \succ^j p$. We thus take $t_0 * \pi_0 = p$ and $l_0 = j$. According to Theorem 2.5.42 we have $(t\rho * \pi)[a := u_1\rho] \succ^j t_0[a := u_1\rho] * \pi_0[a := u_1\rho]$. Consequently, we can deduce that $(t\rho * \pi)[a := u_1\rho] \twoheadrightarrow_k^* t_0[a := u_1\rho] * \pi_0[a := u_1\rho]$ in $l_0 = j$ steps.

To define $(t_{i+1}, \pi_{i+1}, l_{i+1})$ we consider the process $t_i * \pi_i$. By construction we know that $t\rho[a := u_1\rho] * \pi \twoheadrightarrow_k^* t_i[a := u_1\rho] * \pi_i[a := u_1\rho]$ in l_i steps. According to Lemma 5.5.19, $t_i * \pi_i$ can only be of three different shapes.

- If $t_i * \pi_i = \nu * \varepsilon$ for some $\nu \in \Lambda_\iota$ then the sequence ends with $n = i$.
- If $t_i = a$ then we consider the process $t_i[a := u_1\rho] * \pi_i$. By construction we know $(t_i[a := u_1\rho] * \pi_i)[a := u_1\rho] \Downarrow_k$ and Lemma 5.5.19 gives us a blocked process p such that $t_i[a := u_1\rho] * \pi_i \succ^j p$. By Theorem 2.5.42 $(t_i[a := u_1\rho] * \pi_i)[a := u_1\rho] \succ^j p[a := u_1\rho]$, and hence $t_i[a := u_1\rho] * \pi_i[a := u_1\rho] \twoheadrightarrow_k^* p[a := u_1\rho]$ in j steps. We then take as a definition $t_{i+1} * \pi_{i+1} = p$ and $l_{i+1} = l_i + j$.

  Now, is it possible to have $j = 0$? This can only happen when $t_i[a := u_1\rho] * \pi_i$ is of one of the three forms of Lemma 5.5.19. It cannot be of the form $a * \pi$ as we assumed that a does not appear in $u_1\rho$. If it is of the form $\nu * \varepsilon$, then we reached the end of the sequence with $i = n$ so there is no problem. We only have to be careful when $t_i[a := u_1\rho] = \delta_{\nu,w}$. In this case, we will make sure that we always have $l_{i+2} > l_{i+1}$ (see the following case).
- If $t_i = \delta_{\nu,w}$ for some $\nu, w \in \Lambda_\iota$ then we know $\nu[a := u_1\rho] \not\equiv_m w[a := u_1\rho]$ for some $m < k$. Hence, we have $t_i[a := u_1\rho] * \pi_i = \delta_{\nu[a:=u_1\rho],w[a:=u_1\rho]} * \pi_i \twoheadrightarrow_k \nu[a := u_1\rho] * \pi_i$ by definition. Moreover, $t_i[a := u_1\rho] * \pi_i[a := u_1\rho] \twoheadrightarrow_k \nu[a := u_1\rho] * \pi_i[a := u_1\rho]$ by definition of $(\twoheadrightarrow_k)$. Since $t[a := u_1\rho] * \pi \twoheadrightarrow_k^* t_i[a := u_1\rho] * \pi_i[a := u_1\rho]$ in l_i steps we get that $t[a := u_1\rho] * \pi \twoheadrightarrow_k^* \nu[a := u_1\rho] * \pi_i[a := u_1\rho]$ in $l_i + 1$ steps, and hence we have $(\nu[a := u_1\rho] * \pi_i)[a := u_1\rho] = \nu[a := u_1\rho] * \pi_i[a := u_1\rho] \Downarrow_k$.

  We now consider the reduction of the process $\nu[a := u_1\rho] * \pi_i$. According to Lemma 5.5.19 there is a blocked process p such that $\nu[a := u_1\rho] * \pi_i \succ^j p$. Using Theorem 2.5.42 we obtain $\nu[a := u_1\rho] * \pi_i[a := u_1\rho] \succ^j p[a := u_1\rho]$ from which we can deduce that we have $\nu[a := u_1\rho] * \pi_i[a := u_1\rho] \twoheadrightarrow_k^* p[a := u_1\rho]$ in j steps. We then take $t_{i+1} * \pi_{i+1} = p$ and $l_{i+1} = l_i + j + 1$ (and thus $l_{i+1} > l_i$).

Intuitively $(t_i, \pi_i, l_i)_{i \leqslant n}$ mimics the reduction of the process $t[a := u_1\rho] * \pi$ while making explicit every substitution of a and every reduction of a δ-like state.

To end the proof we will show that for all $i \leqslant n$ we have $t_i[a := u_2\rho] * \pi_i[a := u_2\rho] \Downarrow_k$. For $i = 0$ this will give us $t[a := u_2\rho] * \pi \Downarrow_k$, which is the expected result. As by construction $t_n * \pi_n = \nu * \varepsilon$, we have $t_n[a := u_2\rho] * \pi_n[a := u_2\rho] = \nu[a := u_2\rho] * \varepsilon$ from which we get $t_n[a := u_2\rho] * \pi_n[a := u_2\rho] \Downarrow_k$. We now suppose that $t_{i+1}[a := u_1\rho] * \pi_{i+1}[a := u_2\rho] \Downarrow_k$

for $0 \leqslant i < n$ and show that $t_i[a := u_1\rho] * \pi_i[a := u_2\rho] \Downarrow_k$. By construction $t_i * \pi_i$ can be of two shapes since only $t_n * \pi_n$ can be of the form $v * \varepsilon$.

- If $t_i = a$ then we have $u_1\rho * \pi_i \twoheadrightarrow_k^* t_{i+1} * \pi_{i+1}$. As a consequence, Theorem 2.5.42 gives us $u_1\rho * \pi_i[a := u_2\rho] \twoheadrightarrow_k t_{i+1}[a := u_2\rho] * \pi_i[a := u_2\rho]$ from which we can deduce that we have $u_1\rho * \pi_i[a := u_2\rho] \Downarrow_k$ by induction hypothesis. Since $u_1 \equiv_k u_2$ by hypothesis, we obtain $u_2\rho * \pi_i[a := u_2\rho] = (t_i * \pi_i)[a := u_2\rho] \Downarrow_k$.
- If $t_i = \delta_{v,w}$ then we have $v * \pi_i \twoheadrightarrow_k t_{i+1} * \pi_{i+1}$. As a consequence, Theorem 2.5.42 gives us $v[a := u_2\rho] * \pi_i[a := u_2\rho] \twoheadrightarrow_k t_{i+1}[a := u_2\rho] * \pi_{i+1}[a := u_2\rho]$. Using the induction hypothesis we obtain $v[a := u_2\rho] * \pi_i[a := u_2\rho] \Downarrow_k$. It remains to show that we have $\delta_{v[a:=u_2\rho],w[a:=u_2\rho]} * \pi_i[a := u_2\rho] \twoheadrightarrow_k^* v[a := u_2\rho] * \pi_i[a := u_2\rho]$. We need to find $j < k$ such that $v[a := u_2\rho] \not\equiv_k w[a := u_2\rho]$. By construction, there is $m < k$ such that we have $v[a := u_1\rho] \not\equiv_m w[a := u_1\rho]$, and we will show $v[a := u_2\rho] \not\equiv_m w[a := u_2\rho]$. Using the global induction hypothesis twice, we obtain that $v[a := u_1\rho] \equiv_m v[a := u_2\rho]$ and that $w[a := u_1\rho] \equiv_m w[a := u_2\rho]$. Now if we suppose $v[a := u_2\rho] \equiv_m w[a := u_2\rho]$ then we have $v[a := u_1\rho] \equiv_m v[a := u_2\rho] \equiv_m w[a := u_2\rho] \equiv_m v[a := u_1\rho]$, which contradicts $v[a := u_1\rho] \not\equiv_m w[a := u_1\rho]$. Hence we must have $v[a := u_2\rho] \not\equiv_m w[a := u_2\rho]$. □

Theorem 5.5.21. Let $u_1, u_2, t \in \Lambda$ be three terms and $a \in \mathcal{V}_\tau$ be a term variable. If $u_1 \equiv u_2$ then $t[a := u_1] \equiv t[a := u_2]$.

Proof. We suppose that $u_1 \equiv u_2$ which means that $u_1 \equiv_i u_2$ for all $i \in \mathbb{N}$. We need to show that $t[a := u_1] \equiv t[a := u_2]$ so we take $i_0 \in \mathbb{N}$ and show $t[a := u_1] \equiv_{i_0} t[a := u_2]$. By hypothesis we have $u_1 \equiv_{i_0} u_2$ and hence we can conclude using Lemma 5.5.20. □

Theorem 5.5.22. The relation $(\equiv)$ is a congruence.

Proof. Combination of Theorem 5.5.17, Theorem 5.5.18 and Theorem 5.5.21. □

Now that we have proved $(\equiv)$ to be a congruence, it remains to show that it is compatible with $(\equiv_\succ)$. Intuitively, this will provide us with sufficient conditions for proving equivalences based on Chapter 3.

Theorem 5.5.23. For every terms $t, u \in \Lambda$ such that $t \equiv u$ we have $t \equiv_\succ u$.

Proof. Immediate by definition. □

Theorem 5.5.24. Let $t, u \in \Lambda$ be two terms. If for every stack $\pi \in \Pi$ there is a process $p \in \Lambda\times\Pi$ such that both $t * \pi \succ^* p$ and $u * \pi \succ^* p$, then $t \equiv u$.

Proof. By definition, we need to take $i_0 \in \mathbb{N}$, $\pi_0 \in \Pi$ and $\rho_0 \in \mathcal{S}$ and show that we have $t\rho_0 * \pi_0 \Downarrow_{i_0} \Leftrightarrow u\rho_0 * \pi_0 \Downarrow_{i_0}$. Since $(\succ) \subseteq (\twoheadrightarrow_{i_0})$, it is enough to find a common reduct of $t\rho_0 * \pi_0$ and $u\rho_0 * \pi_0$ using $(\succ)$. We consider a renaming substitution σ mapping the variables of $\mathrm{FV}(\pi_0)$ to distinct fresh variables. Note that σ has an inverse σ^{-1} mapping $\sigma(\chi)$ to χ for all $\chi \in \mathrm{dom}(\tau)$. We thus obtain $t\rho_0 * \pi_0 = (t\rho_0 * \pi_0\sigma)\sigma^{-1} = (t * \pi_0\sigma)(\rho_0 \circ \sigma^{-1})$ and $u\rho_0 * \pi_0 = (u * \pi_0\sigma)(\rho_0 \circ \sigma^{-1})$. By hypothesis, $t * \pi_0\sigma$ and $u * \pi_0\sigma$ have a common reduct p_0. We can thus use Lemma 2.5.42 to obtain $t\rho_0 * \pi_0 = (t * \pi_0\sigma)(\rho_0 \circ \sigma^{-1}) \succ^* p_0(\rho_0 \circ \sigma^{-1})$ and $u\rho_0 * \pi_0 = (u * \pi_0\sigma)(\rho_0 \circ \sigma^{-1}) \succ^* p_0(\rho_0 \circ \sigma^{-1})$. □

Theorem 5.5.25. Let $t_1, t_2 \in \Lambda$ be two terms such that $x \in \mathrm{FV}_\iota(t_1) \cap \mathrm{FV}_\iota(t_2)$. If there is a term $u \in \Lambda$ such that for all terms $v \in \Lambda_\iota$ and stacks $\pi \in \Pi$ we have $t_1[x := v] * [u]\pi \succ^* v * \pi$ and $t_2[x := v] * [u]\pi \succ^* v * \pi$, and if $t_1[x := v_1] \equiv t_2[x := v_2]$ for some values $v_1, v_2 \in \Lambda_\iota$, then we have $v_1 \equiv v_2$.

Proof. Let us take $v_1, v_2 \in \Lambda_\iota$ such that $v_1 \not\equiv v_2$ and show that $t_1[x := v_1] \not\equiv t_2[x := v_2]$. By definition there is $i_0 \in \mathbb{N}$, $\pi_0 \in \Pi$ and $\rho_0 \in \mathcal{S}$ such that $v_1\rho_0 * \pi_0 \Downarrow_{i_0}$ and $v_2\rho_0 * \pi_0 \not\Downarrow_{i_0}$ (up to symmetry). We need to find $i_1 \in \mathbb{N}$, $\pi_1 \in \Pi$ and $\rho_1 \in \mathcal{S}$ such that $(t_1[x := v_1])\rho_1 * \pi_1 \Downarrow_{i_1}$ and $(t_2[x := v_2])\rho_1 * \pi_1 \not\Downarrow_{i_1}$. We consider a renaming substitution σ mapping the free variable of π_0 to distinct fresh variables, and we denote σ^{-1} its inverse. We will now show that $i_1 = i_0$, $\pi_1 = [u]\pi_0\sigma$ and $\rho_1 = \rho_0 \circ \sigma^{-1}$ are suitable. According to our main hypothesis, we have $t_1[x := v_1] * \pi_1 \succ^* v_1 * \pi_0\sigma$ and $t_2[x := v_2] * \pi_1 \succ^* v_2 * \pi_0\sigma$. We can thus use Lemma 2.5.42 to obtain $(t_1[x := v_1])\rho_0 * [u]\pi_0 = (t_1[x := v_1] * \pi_1)\rho_1 \succ^* (v_1 * \pi_0\sigma)\rho_1 = v_1\rho_0 * \pi_0 \Downarrow_{i_0}$, and similarly $(t_2[x := v_2])\rho_0 * \pi_0 \succ^* v_2\rho_0 * \pi_0 \not\Downarrow_{i_0}$. □

Theorem 5.5.26. The relation $(\equiv)$ is compatible with $(\equiv_\succ)$.

Proof. Combination of Theorem 5.5.23, Theorem 5.5.24 and Theorem 5.5.25. □

Now that we have fixed our reduction relation to be $(\twoheadrightarrow)$ and our equivalence to be $(\equiv)$, the adequacy lemma is still valid. However, we can now extend our type system with the typing rule that we aimed for.

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash v : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\overline{\mathrm{val}}} v : A}\downarrow$$

We can thus extend the adequacy lemma provided that our pole $\perp\!\!\!\perp$ contains only terminating processes and that $\Box * \varepsilon \in \perp\!\!\!\perp$ (i.e., requirements of Theorem 5.4.15).

Theorem 5.5.27. Let Σ be a sorting context, Γ be a typing context, Ξ be an equational context and $A \in \mathcal{T}$ be a type. Let ρ be a valuation over Σ such that $\rho \Vdash \Gamma$ and $\rho \Vdash \Xi$.

- If $\Sigma \mid \Gamma ; \Xi \vdash t : A$ is derivable, then $t\rho \in [\![A\rho]\!]^{\perp\perp}$.
- If $\Sigma \mid \Gamma ; \Xi \vdash \pi : A^\perp$ is derivable, then $\pi\rho \in [\![A\rho]\!]^{\perp}$.

- If $\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} v : A$ is derivable, then $v\rho \in [\![A\rho]\!] \setminus \{\square\}$.

Proof. As for Theorems 4.6.48 and 4.7.53, the proof is done by induction of the derivation of the typing judgments. For all the rules of Figure 4.2 and the rules for typing stacks given in Theorem 4.7.53 the proof does not change. We only have to be concerned with our new (↓) rule. We need to show $v\rho \in [\![A\rho]\!] \setminus \{\square\}$, knowing that $v\rho \in [\![A\rho]\!]^{\perp\perp}$ by induction hypothesis. According to Theorem 5.4.15, every value of $[\![A\rho]\!]^{\perp\perp}$ is also in $[\![A\rho]\!]$. As a consequence, it only remains to show $v\rho \neq \square$. In the case where v is a λ-abstraction, a variant or a record this is immediate. If $v = x \in \mathcal{V}_\iota$ then this is also immediate since by definition $\rho(x) \neq \square$ for all $x \in \text{dom}(\rho)$. Finally, it cannot be that $v = \square$ since it cannot be introduced by any of our typing rules. In particular, it cannot be brought into focus by rules handling equivalence as this would require replacing a term or a value that was equivalent to $\square$. However, $\square$ is only equivalent to itself according to Theorem 3.5.39. □

We have now obtained a model allowing the use of semantical value restriction. We will see in the next section that the presence of coercion rules between value and terms judgments will allow us to make the type system a lot simpler by only considering one form of judgments.

5.6 Derived type system

Now that our type system contains both the (↑) and the (↓) typing rules, it is always possible to switch between term and value judgments (at least when the subject of the considered judgment is a value). As a consequence, we will simply forget about the value judgments and only work using term judgments. The obtained system will be simpler, in particular it will have less typing rules.

Note that all the typing rules of our new system will be derivable in the current one. As a consequence, we will not need to go through a new adequacy lemma, nor a modification of our semantics. In fact, the derivation of our new typing rules will mostly consist in composing the current typing rules with (↑) on the conclusion. For example, we will derive the new arrow introduction rule ($\Rightarrow_{i,\tau}$) as follows.

$$\dfrac{\dfrac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash t : A}{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} \lambda x.t : A \Rightarrow B}\Rightarrow_i}{\Sigma \mid \Gamma ; \Xi \vdash \lambda x.t : A \Rightarrow B}\uparrow$$

As for value judgments appearing in a premise, we will simply precompose them with the (↓) rule. For example, we will derive the new product elimination rule ($\times_{e,\tau}$) as follows.

$$\dfrac{\dfrac{\Sigma \mid \Gamma ; \Xi \vdash v : \{(l_i : A_i)_{i \in I}\}}{\Sigma \mid \Gamma ; \Xi \vdash_{\text{val}} v : \{(l_i : A_i)_{i \in I}\}}\downarrow \qquad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash v.l_k : A_k}\times_e$$

$$\frac{}{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash x : A}\,\mathrm{Ax}_{\tau} \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \Rightarrow B \quad \Sigma \mid \Gamma ; \Xi \vdash u : A}{\Sigma \mid \Gamma ; \Xi \vdash t\, u : B}\,\Rightarrow_{e}$$

$$\frac{}{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp} ; \Xi \vdash \alpha : A^{\perp}}\,\mathrm{Ax}^{\perp} \qquad \frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi \vdash t : B}{\Sigma \mid \Gamma ; \Xi \vdash \lambda x.t : A \Rightarrow B}\,\Rightarrow_{\iota,\tau}$$

$$\frac{\Sigma, \alpha : \sigma \mid \Gamma, \alpha : A^{\perp} ; \Xi \vdash t : A}{\Sigma \mid \Gamma ; \Xi \vdash \mu\alpha.t : A}\,\mu \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \quad \Sigma \mid \Gamma ; \Xi \vdash \pi : A^{\perp}}{\Sigma \mid \Gamma ; \Xi \vdash [\pi]t : B}\,[-]$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash v : A \quad \Sigma \mid \Gamma ; \Xi \vdash \pi : B^{\perp}}{\Sigma \mid \Gamma ; \Xi \vdash v \cdot \pi : A \Rightarrow B^{\perp}}\,\cdot\text{-}\cdot_{\tau} \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \Rightarrow B \quad \Sigma \mid \Gamma ; \Xi \vdash \pi : B^{\perp}}{\Sigma \mid \Gamma ; \Xi \vdash [t]\pi : A^{\perp}}\,[-]\text{-}$$

$$\frac{\Sigma, \chi : s \mid \Gamma ; \Xi \vdash t : A \quad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash t : \forall \chi^{s}.A}\,\forall_{\iota,\tau} \qquad \frac{\Sigma, x : \iota, \chi : s \mid \Gamma, x : A ; \Xi \vdash t : C}{\Sigma, x : \iota \mid \Gamma, x : \exists \chi^{s}.A ; \Xi \vdash t : C}\,\exists_{e}$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : \forall \chi^{s}.A \quad \Sigma \vdash B : s}{\Sigma \mid \Gamma ; \Xi \vdash t : A[\chi := B]}\,\forall_{e} \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A[\chi := B] \quad \Sigma \vdash B : s}{\Sigma \mid \Gamma ; \Xi \vdash t : \exists \chi^{s}.A}\,\exists_{i}$$

$$\frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi, x \equiv t \vdash u : C}{\Sigma, x : \iota \mid \Gamma, x : t \in A ; \Xi \vdash u : C}\,\in_{e} \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \quad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma ; \Xi \vdash t : A \restriction u_1 \equiv u_2}\,\restriction_{i}$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash t : A \quad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma ; \Xi \vdash t : t \in A}\,\in_{\iota,\tau} \qquad \frac{\Sigma, x : \iota \mid \Gamma, x : A ; \Xi, u_1 \equiv u_2 \vdash t : C}{\Sigma, x : \iota \mid \Gamma, x : A \restriction u_1 \equiv u_2 ; \Xi \vdash t : C}\,\restriction_{e}$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash v : \{(l_i : A_i)_{i \in I}\} \quad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash v.l_k : A_k}\,\times_{e,\tau} \qquad \frac{\Sigma \mid \Gamma ; \Xi \vdash v : A_k \quad k \in I}{\Sigma \mid \Gamma ; \Xi \vdash C_k[v] : [(C_i : A_i)_{i \in I}]}\,+_{i,\tau}$$

$$\frac{[\Sigma \mid \Gamma ; \Xi \vdash v_i : A_i]_{i \in I}}{\Sigma \mid \Gamma ; \Xi \vdash \{(l_i = v_i)_{i \in I}\} : \{(l_i : A_i)_{i \in I}\}}\,\times_{i,\tau}$$

$$\frac{\Sigma \mid \Gamma ; \Xi \vdash v : [(C_i : A_i)_{i \in I}] \quad [\Sigma, x_i : \iota \mid \Gamma, x_i : A_i ; \Xi, v \equiv C_i[x_i] \vdash t_i : B]_{i \in I}}{\Sigma \mid \Gamma ; \Xi \vdash [v \mid (C_i[x_i] \to t_i)_{i \in I}] : B}\,+_{e,\tau}$$

$$\frac{\Sigma \mid \Gamma[x := w_1] ; \Xi \vdash t[x := w_1] : A[x := w_1] \quad \Xi \vdash w_1 \equiv w_2}{\Sigma \mid \Gamma[x := w_2] ; \Xi \vdash t[x := w_2] : A[x := w_2]}\,\equiv_{\tau,\iota}$$

$$\frac{\Sigma \mid \Gamma[a := u_1] ; \Xi \vdash t[a := u_1] : A[a := u_1] \quad \Xi \vdash u_1 \equiv u_2}{\Sigma \mid \Gamma[a := u_2] ; \Xi \vdash t[a := u_2] : A[a := u_2]}\,\equiv_{\tau,\tau}$$

Figure 5.3 – Derived typing rules.

The full set of our new rules is displayed in Figure 5.3. Note that some rules, like $(\Rightarrow_e)$ for example, remain unchanged. All the modified rules but $(\forall_{i,\tau})$ and $(\in_{i,\tau})$ can be derived immediately as demonstrated above. The rules $(\forall_{i,\tau})$ and $(\in_{i,\tau})$ are instances of the semantical value restriction. The former can be derived as follows, and a derivation for the latter was given in a previous section.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, \chi : s \mid \Gamma; \Xi \vdash t : A \quad \Xi \vdash t \equiv v}{\Sigma, \chi : s \mid \Gamma; \Xi \vdash v : A}\equiv_{\tau,\tau}}{\Sigma, \chi : s \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : A}\downarrow}{\Sigma \mid \Gamma; \Xi \vdash_{\mathrm{val}} v : \forall \chi^s.A}\forall_i}{\Sigma \mid \Gamma; \Xi \vdash v : \forall \chi^s.A}\uparrow \quad \Xi \vdash t \equiv v}{\Sigma \mid \Gamma; \Xi \vdash t : \forall \chi^s.A}\equiv_{\tau,\tau}$$

With our new set of typing rules, it is very easy to derive introduction and elimination rules for the dependent product type. Using the $(\in_{i,\tau})$ rule, semantical value restriction is immediately propagated to the elimination rules of the dependent product type.

$$\dfrac{\Sigma, x : \iota \mid \Gamma, x : x \in A; \Xi \vdash t : B[a := x]}{\Sigma \mid \Gamma; \Xi \vdash \lambda x.t : \Pi_{a \in A} B}\Pi_i$$

$$\dfrac{\Sigma \mid \Gamma; \Xi \vdash t : \Pi_{a \in A} B \quad \Sigma \mid \Gamma; \Xi \vdash u : A \quad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma; \Xi \vdash t\, u : B[a := u]}\Pi_e$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, x : \iota \mid \Gamma, x : x \in A; \Xi \vdash t : B[a := x]}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : x \in A; \Xi \vdash t : B[a := x]}\mathrm{Wk}}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : x \in A; \Xi, x \equiv a \vdash t : B[a := x]}\mathrm{Wk} \quad \dfrac{}{\Xi, x \equiv a \vdash x \equiv a}\top}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : a \in A; \Xi, x \equiv a \vdash t : B}\equiv_{\tau,\tau}}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : A; \Xi, x \equiv a \vdash t : B}\in_e}{\Sigma, a : \tau, x : \iota \mid \Gamma, x : a \in A; \Xi \vdash t : B}\in_e}{\Sigma, a : \tau \mid \Gamma; \Xi \vdash \lambda x.t : a \in A \Rightarrow B}\Rightarrow_i \quad \dfrac{}{\Xi \vdash \lambda x.t \equiv \lambda x.t}\top}{\Sigma \mid \Gamma; \Xi \vdash \lambda x.t : \forall a.(a \in A \Rightarrow B)}\forall_i}{\Sigma \mid \Gamma; \Xi \vdash \lambda x.t : \Pi_{a \in A} B}\mathrm{Def}$$

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma; \Xi \vdash t : \Pi_{a \in A} B}{\Sigma \mid \Gamma; \Xi \vdash t : \forall a.(a \in A \Rightarrow B)}\mathrm{Def}}{\Sigma \mid \Gamma; \Xi \vdash t : u \in A \Rightarrow B[a := u]}\forall_e \quad \dfrac{\Sigma \mid \Gamma; \Xi \vdash u : A \quad \Xi \vdash u \equiv v}{\Sigma \mid \Gamma; \Xi \vdash u : u \in A}\forall_e}{\Sigma \mid \Gamma; \Xi \vdash t\, u : B[a := u]}\Rightarrow_e$$

Remark 5.6.28. Dually, it is possible to encode a form of dependent pair type using existential quantification and product types. As for the dependent functions, the membership predicate is used to bridge the world of terms and the world of types.

$$\Sigma_{x\in A}B \coloneqq \exists x.\{l_1 : x\in A\,;\, l_2 : B\} \qquad\qquad \Sigma_{a\in A}B \coloneqq \exists a.\{l_1 : a\in A\,;\, l_2 : B\}$$

As dependent pair types are based on records (which can only contain values in our system), their use is rather limited. It seems however possible to rely on a dependent product type to obtain satisfactory typing rules based on terms of the following form.

$$(\lambda x.\lambda y.\{l_1 = x\,;\, l_2 = y\})\ t_1\ t_2$$

5.7 Understanding our new equivalence

To better understand our new definition of equivalence, we can compare it to another equivalence relation with a more intuitive definition. The definition of this new relation, denoted $(\equiv_{\twoheadrightarrow})$, will be very similar to that of $(\equiv_{>})$ (see Chapter 3). It can be seen as the observational equivalence induced by $(\twoheadrightarrow)$

Definition 5.7.29. The relation $(\equiv_{\twoheadrightarrow}) \subseteq \Lambda\times\Lambda$ is defined as follows.

$$(\equiv_{\twoheadrightarrow}) = \left\{(t\,,\,u) \mid \forall\,\pi\in\Pi, \forall\,\rho\in\mathcal{S}\,,\, t\rho * \pi \Downarrow_{\twoheadrightarrow} \Leftrightarrow\ u\rho * \pi \Downarrow_{\twoheadrightarrow}\right\}$$

Lemma 5.7.30. The relation $(\equiv_{\twoheadrightarrow}) \subseteq \Lambda\times\Lambda$ is an equivalence relation.

Proof. Immediate. □

Theorem 5.7.31. If $t, u \in \Lambda$ are two terms such that $t \equiv u$ then $t \equiv_{\twoheadrightarrow} u$. In other words, we have $(\equiv) \subseteq (\equiv_{\twoheadrightarrow})$

Proof. Let us suppose that $t \equiv u$, take $\pi_0 \in \Pi$ and take $\rho_0 \in \mathcal{S}$. By symmetry we can assume that $t\rho_0 * \pi_0 \Downarrow_{\twoheadrightarrow}$ and show that $u\rho_0 * \pi_0 \Downarrow_{\twoheadrightarrow}$. By definition there is $i_0 \in \mathbb{N}$ such that $t\rho_0 * \pi_0 \Downarrow_{i_0}$. Since $t \equiv u$ we know that for all $i \in \mathbb{N}$, $\pi \in \Pi$ and $\rho \in \mathcal{S}$ we have $t\sigma * \pi \Downarrow_i \Leftrightarrow u\sigma * \pi \Downarrow_i$. This is true in particular for $i = i_0$, $\pi = \pi_0$ and $\rho = \rho_0$. We hence obtain $u\rho_0 * \pi_0 \Downarrow_{i_0}$ which give us $u\rho_0 * \pi_0 \Downarrow_{\twoheadrightarrow}$. □

Remark 5.7.32. The converse implication is not true in general, that is we do not have $(\equiv) \supseteq (\equiv_{\twoheadrightarrow})$. A counter-example is given by the terms $t = \delta_{\lambda x.x,\{\}}$ and $u = \lambda x.x$ since $t \equiv_{\twoheadrightarrow} u$ but $t \not\equiv_{>} u$ (and thus $t \not\equiv u$). More generally, if $p, q \in \Lambda\times\Pi$ are processes, having $p \Downarrow_{\twoheadrightarrow} \Rightarrow q \Downarrow_{\twoheadrightarrow}$ does not always imply $p \Downarrow_i \Rightarrow q \Downarrow_i$ for every natural number $i \in \mathbb{N}$.

As shown by the Theorem 5.7.31, our $(\equiv)$ relation is more fine-grained (i.e., it discriminates more terms than $(\equiv_{\twoheadrightarrow})$). However, its formulation does not really provide more intuition on the behaviour of our equivalence relation. Indeed, the definition of $(\twoheadrightarrow)$ still involves $(\equiv)$, an thus the definition of $(\equiv_{\twoheadrightarrow})$ is still very subtle, even if its statement remains relatively simple.

In the end, what really matters to us is for our equivalence relation to be compatible with the notion of reduction. And in fact, the only part of the reduction relation that will matter in practice is $(>)$. Indeed, δ-like terms are only provided in the system for obtaining a well-behaved semantics. In particular, we do not want to expose them to the users of our implementation. It is thus enough for $(\equiv)$ to be compatible with $(\equiv_{>})$.

Remark 5.7.33. Allowing the user to work with and reason about δ-like terms would not be such a bad idea. Indeed, it could allow the encoding of mathematical objects into pure terms of the language. In particular, it would be interesting to investigate the possibility of using the following alternative reduction rules for δ-like processes.

$$\begin{array}{lcll} \delta_{v,w} * \pi & \twoheadrightarrow & C_0[\{\}] * \pi & \text{when } v \not\equiv w \\ \delta_{v,w} * \pi & \twoheadrightarrow & C_1[\{\}] * \pi & \text{when } v \equiv w \end{array}$$

They would allow the definition of non-computable functions in our language by giving complete access to our equivalence relation. This would allow, for example, the definition of functions like $\lambda x.\lambda y.\delta_{x,y}$ (i.e., a general equality function).

6 Introducing subtyping into the system

In this chapter, we reformulate the definition of our system to account for subtyping. The main idea is to transform the typing rules that do not have algorithmic contents into subtyping rules. For instance, quantifiers, fixpoints, membership types and equality types will be handled using subtyping.

6.1 Interests of subtyping

There is no denying that polymorphism and type abstraction are essential features for programming in a generic way. They lead to programs that are shorter, more modular, easier to understand and hence more reliable. Although subtyping provides similar perspectives, it is considerably less widespread among programming languages. Practical languages only rely on limited forms of subtyping for their module system [*MacQueen 1984*], or for the use of polymorphic variants [*Garrigue 1998*]. Overall, subtyping is useful for both product types (e.g., records or modules) and sum types (e.g., polymorphic variants). It provides canonical injections between a type and its subtypes. For example, the natural numbers may be defined as a subtype of the integers.

The downside of subtyping is that it is difficult to incorporate in complex systems like Haskell or OCaml. For example, OCaml provides polymorphic variants [*Garrigue 1998*] for which complex annotated types are inferred. For instance, one would expect the following OCaml function to be given the type `[`T | `F] → [`T | `F]`.

```
let neg = function `T → `F | `F → `T
```

Indeed, the variance of the arrow type conveys enough information: neg can be applied to elements of any subtype of `[`T | `F]` (e.g., `[`T]`) and produces elements of any supertype of `[`T | `F]` (e.g. `[`T | `F | `M]`). OCaml infers the type `[<`T | `F] → [>`T |`

`F] in which subtypes and supertypes are explicitly tagged. This is not very natural and hides a complex mechanism involving polymorphic type variables. More discussion on the limitations of OCaml's polymorphic variants, can be found in [*Castagna 2016*], for example.

In this thesis, we will show that it is possible to design a practical type system based on subtyping for our language. It allows for a rather straight-forward implementation following the typing and subtyping rules that will be given in the following sections. In particular, the typing and subtyping procedures are directed by the syntax of terms and types respectively. The ideas presented here were introduced in a joint work with Christophe Raffalli [*Lepigre 2017*].

6.2 Symbolic witnesses and local subtyping

Several related technical innovations are required to include subtyping into our system. In particular, we will need to generalise the usual subtyping relation $A \subset B$ (meaning "A is a subtype of B") using a *local subtyping* relation $t \in A \subset B$. It will be interpreted as "if t has type A then it also has type B". Usual subtyping is then recovered using choice operators inspired from *Hilbert's Epsilon and Tau functions* [*Hilbert 1934*]. In our system, the choice operator $\varepsilon_{x \in A}(t \notin B)$ denotes a value v of type A such that $t[x := v]$ does not have type B. If no such term exists, then an arbitrary term of type A can be chosen. We can then take $\varepsilon_{x \in A}(x \notin B) \in A \subset B$ as a definition of $A \subset B$.

Remark 6.2.1. Of course, for the choice operator $\varepsilon_{x \in A}(t \notin B)$ to be well-defined we will need the interpretation of every type to be non-empty. It is the case in [*Lepigre 2017*] as the model is based on Girard's reducibility candidates [*Girard 1972*, *Girard 1989*]. Here, we will use the special value $\square$ as it is contained in the interpretation of all types by construction. This is its very purpose.

Choice operators can be used to replace the notion of free variables, and hence suppress the need for typing contexts. The contexts will then be limited to an equational context containing closed terms. Intuitively, $\varepsilon_{x \in A}(t \notin B)$ denotes a counterexample to the fact that $\lambda x.t$ has type $A \Rightarrow B$. Consequently, we will use the following rule for typing λ-abstractions.

$$\frac{\Xi \vdash t[x := \varepsilon_{x \in A}(t \notin B)] : B}{\Xi \vdash \lambda x.t : A \Rightarrow B}$$

It can be read as a proof by contradiction as its premise is only valid when there is no value v of type A such that $t[x := v]$ does not have type B. The axiom rule is then replaced by the following typing rule for choice operators.

$$\frac{}{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : A}$$

Obviously, the same trick can be used for μ-variables and μ-abstractions. As choice operators for values, choice operators for stacks will always need to be interpreted (even if there is no stack satisfying the property they carry). As a consequence, we will need to make sure that the set of stacks associated to each type contains at least one element.

The use of choice operators and the elimination of typing contexts will play an essential role in the definition of our type system. Indeed, they will allow us to handle quantifiers using our local subtyping relation only. As a consequence, we will work with syntax-directed typing rules and most of the work will be done using subtyping. We will thus introduce two new type constructors $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ corresponding to choice operators for picking a type satisfying the denoted properties. For example, $\varepsilon_X(t \notin B)$ is a type such that the term t does not have type $B[X := \varepsilon_X(t \notin B)]$. Intuitively, $\varepsilon_X(t \notin B)$ is a counter-example to the judgment "t has type $\forall X.B$". Hence, to show that t has type $\forall X.B$ it will be enough to show that it has type $B[X := \varepsilon_X(t \notin B)]$. As a consequence, the introduction rule for the universal quantifier is subsumed by the following local subtyping rule.

$$\frac{\Xi \vdash t \in A \subset B[X := \varepsilon_X(t \notin B)] \qquad \Xi \vdash t \equiv v}{\Xi \vdash t \in A \subset \forall X.B}$$

Note that it includes a premise stating that the term t carried by the judgments should be equivalent to a value. This corresponds to the semantical value restriction condition in our new system with subtyping.

In conjunction with local subtyping, choice operators allow the derivation of many valid permutations of quantifiers and connectives. For example, subtyping relations like

$$\forall X.\forall Y.A \subset \forall Y.\forall X.A \qquad\qquad \{l_1 : \forall X.A\,;\, l_2 : \forall X.B\} \subset \forall X.\{l_1 : A\,;\, l_2 : B\}$$

can be easily obtained thanks to our syntax-directed subtyping rules. In particular, they do not include a transitivity rule, and this is a good thing since such a rule could not be implemented. Indeed, it would require the system to guess an intermediate type. Transitivity is generally admissible in subtyping systems. In our system however, it is an open problem whether a form of transitivity is admissible. However, type annotations of the form $((t : A) : B) : C$ can always be used to decompose a proof of $t : C$ into proofs of $t : A$, $t \in A \subset B$ and $t \in B \subset C$. Such annotations are also required in the implementations of systems having a transitivity rule. Indeed, without annotations the system would need to guess the intermediate types A and B.

6.3 Typing and subtyping rules

We will now give the formal definition of our new type system with subtyping. We will reuse some of the formalism given in Chapter 4, but modifications will be required. For

instance, we will need to extend the language of values, stacks and formulas to include choice operators.

Definition 6.3.2. We extend the language of formulas $\mathcal{F}$ with new constructors $\varepsilon_{\chi:s}(t \in A)$ and $\varepsilon_{\chi:s}(t \notin A)$ representing choice operators. They will be made available in the syntax, and will provide an alternative presentation of quantifiers. Note that our system needs to be extended with the following two sorting rules.

$$\frac{\Sigma \vdash t : \tau \quad \Sigma, \chi : s \vdash A : o}{\Sigma \vdash \varepsilon_{\chi:s}(t \in A) : s} \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma, \chi : s \vdash A : o}{\Sigma \vdash \varepsilon_{\chi:s}(t \notin A) : s}$$

To introduce value and stack witnesses into our system, we will need to make a distinction between values, terms, stacks and formulas that may contain value and stack witnesses and those that may not.

Definition 6.3.3. We extend the syntax of values with a new constructor $\varepsilon_{x \in A}(t \notin B)$, where $x \in \mathcal{V}_\iota$ is a λ-variables, $t \in \Lambda$ is a term and $A, B \in \mathcal{F}$ are propositions. Similarly, the syntax of stacks is extended with a new constructor $\varepsilon_{\alpha \in \neg A}(t \notin A)$, where $\alpha \in \mathcal{V}_\sigma$ is a μ-variable, $t \in \Lambda$ is a term and $A \in \mathcal{F}$ is a proposition. Note that our system needs to be extended with the following two sorting rules.

$$\frac{x : \iota \vdash A : o \quad x : \iota \vdash t : \tau \quad x : \iota \vdash B : o}{\Sigma \vdash \varepsilon_{x \in A}(t \notin B) : \iota} \qquad \frac{\alpha : \sigma \vdash A : o \quad \alpha : \sigma \vdash t : \tau}{\Sigma \vdash \varepsilon_{\alpha \in \neg A}(t \notin A) : \sigma}$$

It is important to note that no other variable than x may be bound in A, t or B in the first rule. Similarly, only α can be bound in t and B in the second one. These restrictions are required for the definition of our semantics.

Definition 6.3.4. We will refer to values, terms and stacks that may contain value and stack witnesses as *raw* values, *raw* terms and *raw* stacks. The corresponding sets will be denoted Λ^+_ι, Λ^+ and Π^+ respectively.

Before going into our new typing and subtyping rules, we will extend the syntax of formulas one more time. Indeed, our current system (and its semantics) does not allow for all the basic subtyping relations that we could hope for. For instance, if $I_1 \subseteq I_2$ then it is possible to show that $[(C_i : A_i)_{i \in I_1}]$ is a subtype of $[(C_i : A_i)_{i \in I_2}]$ in the system. However, the corresponding relation on product types is not satisfied by our semantics. Intuitively, the elements of the type $\{(l_i : A_i)_{i \in I}\}$ must be of the form $\{(l_i = v_i)_{i \in I}\}$. In particular, they cannot have additional record fields. One solution to this problem would be to amend the semantics of product types. Instead, we will keep our original, *strict* product type, and introduce another *extensible* product type.

Definition 6.3.5. The syntax of formulas $\mathcal{F}$ is extended with an *extensible product type* constructors $\{(l_i : A_i)_{i\in I}; \cdots\}$. Our system again requires a new sorting rule.

$$\frac{\{\Sigma \vdash A_i : o\}_{i\in I}}{\Sigma \vdash \{(l_i : A_i)_{i\in I}; \cdots\} : o}$$

We will now give the new typing and subtyping rules of our system, which will contain three (and in fact four) new forms of judgments.

Definition 6.3.6. A *general typing judgment* is a triple of an equational context Ξ, a raw term $t \in \Lambda^+$ and a formula $A \in \mathcal{F}$ that is denoted $\Xi \vdash t : A$. A *general stack judgment* is a triple of an equational context Ξ, a raw stack $\pi \in \Pi^+$ and a formula $A \in \mathcal{F}$ that is denoted $\Xi \vdash \pi : \neg A$. A *pointed subtyping judgment* is a quadruple of an equational context Ξ, a raw term $t \in \Lambda^+$ and two formulas $A, B \in \mathcal{F}$ that is denoted $\Xi \vdash t \in A \subset B$. We will use the notation $\Xi \vdash A \subset B$ when the term t is equal to $\varepsilon_{x\in A}(x \notin B)$.

To keep track of the special value $\square$ in our judgments, we will use inequalities of the form $v \neq \square$. They will be defined in terms of an equivalence, so we do not need to extend our definitions formally.

Definition 6.3.7. Given a closed value $v \in \Lambda^*_\iota$ we will use the notation $v \neq \square$ for the syntactic equivalence $(\lambda x.\{\})\ v \equiv \{\}$ in equational contexts.

As shown by the following lemma, this notation provides the right intuition as it agrees with the semantics.

Lemma 6.3.8. Given a closed value $v \in \Lambda^*_\iota$ we have $(\lambda x.\{\})\ v \equiv \{\}$ if and only if v is different from $\square$.

Proof. Let us first assume that $v \neq \square$ and show $(\lambda x.\{\})\ v \equiv \{\}$. Since v is closed, it cannot be a λ-variable and so we can conclude immediately using Theorem 3.3.16. For the other direction we show the contrapositive so we assume $v = \square$ and we show $(\lambda x.\{\})\ v \not\equiv \{\}$. According to Theorem 3.3.17 we have $(\lambda x.\{\})\ \square \equiv \square$ and so it is enough to show $\square \not\equiv \{\}$. This follows from Theorem 3.5.39. □

Definition 6.3.9. A general typing judgment, general stack judgment or pointed subtyping judgment is said to be valid if it can be derived using the rules of Figures 6.4 and 6.5.

There is nothing too surprising about our new typing rules. Note however that, in the case where our equivalence decision procedure is unable to prove a premise of the form $\Xi \vdash v \equiv t$ in a local subtyping rule, one can always fallback to the (Gen) rule.

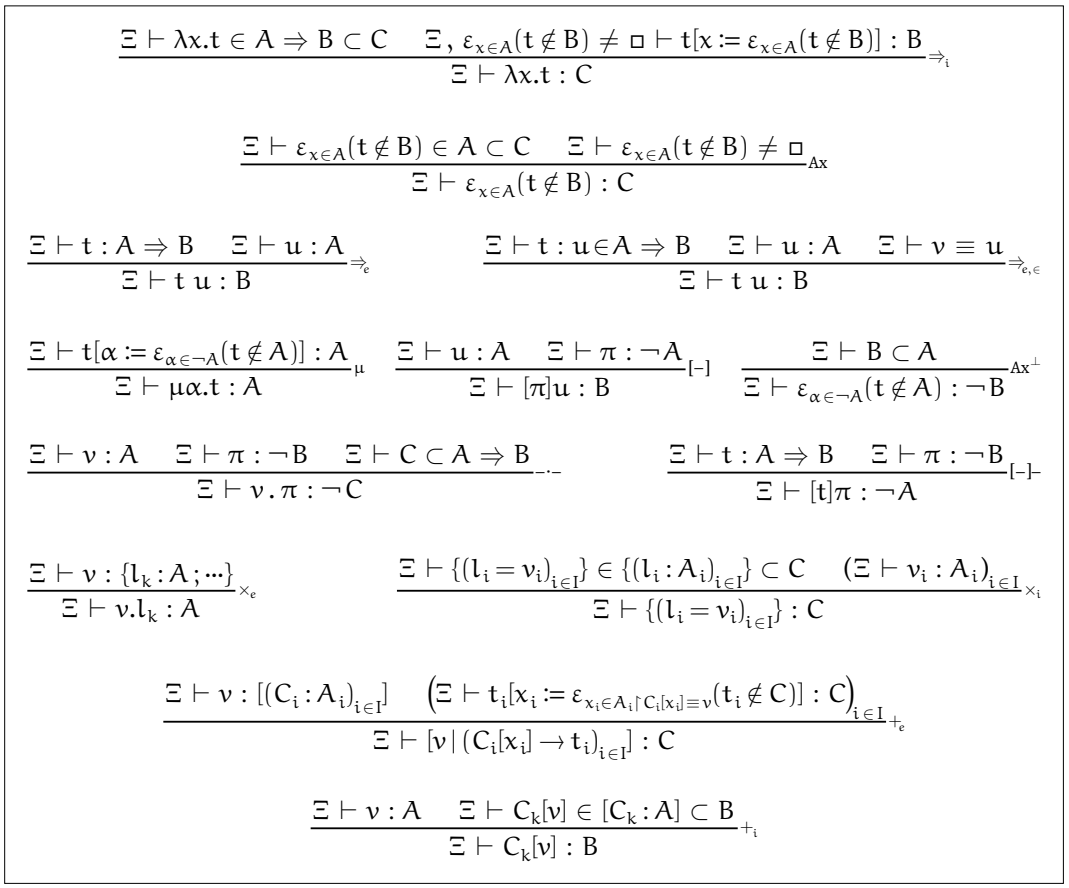

Figure 6.4 – Typing rules for terms and stacks.

6.4 Semantics of subtyping

We will now adapt our model to work with our new typing and subtyping rules. The main problem that we need to solve is the interpretation of raw terms, values and stacks. In particular, we need to provide an interpretation to our choice operators. In [*Lepigre 2017*], the choice operator $\varepsilon_{x\in A}(t \notin B)$ is interpreted using a value $v \in [\![A]\!]$ such that $t[x := v]$ is in the semantics of B. In the case where no such value exists, an arbitrary member of $[\![A]\!]$ is chosen. Here, a crucial point is that the set $[\![A]\!]$ should not be empty. It is the case here as by construction we have $\square \in [\![A]\!]$ for every proposition A. This special value $\square$ will thus be understood as an undefined value witness.

$$\frac{\Xi, w \neq \square \vdash w \in A_2 \subset A_1 \quad \Xi, w \neq \square \vdash t\ w \in B_1 \subset B_2 \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in A_1 \Rightarrow B_1 \subset A_2 \Rightarrow B_2}{\scriptstyle \Rightarrow \quad w = \varepsilon_{x \in A_2}(t\ x \notin B_2)}$$

$$\frac{I_1 \subset I_2 \quad (\Xi \vdash (\lambda x.[x \mid C_i[x_i] \to x_i])\ t \in A_i \subset B_i)_{i \in I_1} \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in [(C_i : A_i)_{i \in I_1}] \subset [(C_i : B_i)_{i \in I_2}]}{\scriptstyle +}$$

$$\frac{(\Xi \vdash (\lambda x.x.l_i)\ t \in A_i \subset B_i)_{i \in I} \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in \{(l_i : A_i)_{i \in I}\} \subset \{(l_i : B_i)_{i \in I}\}}{\scriptstyle \times} \qquad \frac{(\Xi \vdash \rho_1(\chi) \equiv \rho_2(\chi))_{\chi \in FV(A)}}{\Xi \vdash t : A\rho_1 \subset A\rho_2}{\scriptstyle Ax_{\subset}}$$

$$\frac{I_2 \subset I_1 \quad (\Xi \vdash (\lambda x.x.l_i)\ t \in A_i \subset B_i)_{i \in I_2} \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in \{(l_i : A_i)_{i \in I_1}; \cdots\} \subset \{(l_i : B_i)_{i \in I_2}; \cdots\}}{\scriptstyle \times_{ext}} \qquad \frac{\Xi \vdash A \subset B}{\Xi \vdash t : A \subset B}{\scriptstyle Gen}$$

$$\frac{I_2 \subset I_1 \quad (\Xi \vdash (\lambda x.x.l_i)\ t \in A_i \subset B_i)_{i \in I_2} \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in \{(l_i : A_i)_{i \in I_1}\} \subset \{(l_i : B_i)_{i \in I_2}; \cdots\}}{\scriptstyle \times_{\subset}}$$

$$\frac{\Xi \vdash t \in A[\chi := C] \subset B}{\Xi \vdash t \in \forall \chi^s.A \subset B}{\scriptstyle \forall_l} \qquad \frac{\Xi \vdash t \in A \subset B[\chi := \varepsilon_{\chi:s}(t \notin B)] \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t \in A \subset \forall \chi^s.B}{\scriptstyle \forall_r}$$

$$\frac{\Xi \vdash t \in A \subset B[\chi := C]}{\Xi \vdash t \in A \subset \exists \chi^s.B}{\scriptstyle \exists_r} \qquad \frac{\Xi \vdash t \in A[\chi := \varepsilon_{\chi:s}(t \in A)] \subset B \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in \exists \chi^s.A \subset B}{\scriptstyle \exists_l}$$

$$\frac{\Xi, u_1 \equiv u_2 \vdash t \in A \subset B \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t \in A \restriction u_1 \equiv u_2 \subset B}{\scriptstyle \restriction_l} \qquad \frac{\Xi \vdash t \in A \subset B \quad \Xi \vdash u_1 \equiv u_2}{\Xi \vdash t \in A \subset B \restriction u_1 \equiv u_2}{\scriptstyle \restriction_r}$$

$$\frac{\Xi, t \equiv u \vdash t \in A \subset B \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in u \in A \subset B}{\scriptstyle \in_l} \qquad \frac{\Xi \vdash t \in A \subset B \quad \Xi \vdash t \equiv u \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in A \subset u \in B}{\scriptstyle \in_r}$$

Figure 6.5 – Local subtyping rules.

Lemma 6.4.10. Let $p \in \Lambda \times \Pi$ be a process such that $\square \notin p$. If there is $q \in \Lambda \times \Pi$ such that $p \twoheadrightarrow q$ then $\square \notin q$.

Proof. Most reduction rules of $(\twoheadrightarrow)$ only build a new process using components of the process p being evaluated. Hence, they cannot make $\square$ appear it was not already present in p. The remaining rules are related to binders, and obviously if t, ν and π do not contain $\square$, then neither do $t[x := \nu]$ or $t[\alpha := \pi]$. □

Similarly, if a variable does not appear in a process then it cannot appear during reduction. In particular, the evaluation of a closed process never produces an open process.

In the semantics, raw values, raw terms and raw stacks will be interpreted using values, terms and stacks with the same structure. The underlying choice operators will thus be replaced by elements of the corresponding syntactic category.

Definition 6.4.11. Given a raw value $v \in \Lambda_\iota^+$ (resp. raw term $t \in \Lambda^+$, raw stack $\pi \in \Pi^+$), we denote $[\![v]\!] \in \Lambda_\iota$ (resp. $[\![t]\!] \in \Lambda$, $[\![\pi]\!] \in \Pi$) its semantical interpretation. It is defined inductively as follows.

$$
\begin{array}{rclcrcl}
[\![x]\!] & = & x & & [\![[t]\pi]\!] & = & [[\![t]\!][\![\pi]\!] \\
[\![\lambda x.t]\!] & = & \lambda x.[\![t]\!] & & \left[\!\!\left[\{(l_i = v_i)_{i\in I}\}\right]\!\!\right] & = & \{(l_i = [\![v_i]\!])_{i\in I}\} \\
[\![C[v]]\!] & = & C[[\![v]\!]] & & [\![\varepsilon_{x\in A}(t \notin B)]\!] & = & v \in [\![A[x := v]]\!] \setminus \{\square\} \mid [\![t[x := v]]\!] \notin [\![B]\!]^{\perp\perp} \\
[\![\square]\!] & = & \square & & [\![\varepsilon_{x\in A}(t \notin B)]\!] & = & \square \text{ otherwise} \\
[\![a]\!] & = & a & & [\![\mu\alpha.t]\!] & = & \mu\alpha.[\![t]\!] \\
[\![t\ u]\!] & = & [\![t]\!]\ [\![u]\!] & & [\![[\pi]t]\!] & = & [[\![\pi]\!][\![t]\!] \\
[\![v.l_k]\!] & = & [\![v]\!].l_k & & \left[\!\!\left[[v \mid (C_i[x_i] \to t_i)_{i\in I}]\right]\!\!\right] & = & [[\![v]\!] \mid (C_i[x_i] \to [\![t_i]\!])_{i\in I}] \\
[\![Y_{t,v}]\!] & = & Y_{[\![t]\!],[\![v]\!]} & & [\![\delta_{v,w}]\!] & = & \delta_{[\![v]\!],[\![w]\!]} \\
[\![R_{v,t}]\!] & = & R_{[\![v]\!],[\![t]\!]} & & [\![\varepsilon]\!] & = & \varepsilon \\
[\![\alpha]\!] & = & \alpha & & [\![\varepsilon_{\alpha\in\neg A}(t \notin A)]\!] & = & \pi \in [\![A[\alpha := \pi]]\!]^{\perp} \mid [\![t[\alpha := \pi]]\!] \notin [\![B]\!]^{\perp\perp} \\
[\![v\,.\,\pi]\!] & = & [\![v]\!]\,.\,[\![\pi]\!] & & [\![\varepsilon_{\alpha\in\neg A}(t \notin A)]\!] & = & [\square]\varepsilon \text{ otherwise}
\end{array}
$$

It is important to note that, from now on, raw values, raw terms and raw stacks may appear in types of any sort. However, up to the interpretation of such raw syntactic elements, the semantics of our types will remain the same as in Chapter 4. Indeed, we will consider that a raw value, raw term or raw stack is equal to its interpretation (and thus to a value, term and stack respectively). We will however need to account for witnesses of the form $\varepsilon_{\chi:s}(t \in A)$ and $\varepsilon_{\chi:s}(t \notin A)$ in their semantical interpretation. Intuitively, these witnesses will be understood as formulas of the corresponding sort satisfying the denoted property.

Definition 6.4.12. We extend the definition of the interpretation of types (Definition 4.5.42) in such a way that:

- $[\![\varepsilon_{\chi:s}(t \in A)]\!] = \Phi$ such that $\Phi \in [\![s]\!]$ and, if possible, $[\![t]\!] \in [\![A[\chi := \Phi]]\!]^{\perp\perp}$,
- $[\![\varepsilon_{\chi:s}(t \notin A)]\!] = \Phi$ such that $\Phi \in [\![s]\!]$ and, if possible, $[\![t]\!] \notin [\![A[\chi := \Phi]]\!]^{\perp\perp}$.

Note that for every sort $s \in \mathcal{S}$ it is easy to see that $[\![s]\!] \neq \emptyset$. As a consequence, the interpretation of a type is always well-defined.

Remark 6.4.13. Although this is not explicitly mentioned, the interpretation of our choice operators should be compatible with the definition of $(\equiv)$. For instance, if $[\![t]\!] \equiv [\![u]\!]$ then we require $[\![\varepsilon_{x\in A}(t \notin B)]\!] = [\![\varepsilon_{x\in A}(u \notin B)]\!]$ (and similarly for the other forms of choice operators). This is possible since $[\![t[x := v]]\!] \in [\![B]\!]^{\perp\perp}$ if and only if $[\![u[x := v]]\!] \in [\![B]\!]^{\perp\perp}$ as the interpretation of our types is closed under $(\equiv)$. Note that this means that we will also have $[\![\varepsilon_{x\in A}(t \notin B)]\!] = \square$ if and only if $[\![\varepsilon_{x\in A}(u \notin B)]\!] = \square$ since the interpretation of our types is closed under $(\equiv)$.

Definition 6.4.14. We also extend Definition 4.5.42 with an interpretation for the extensible product type. It is defined as follows.

$$\left[\!\left[\{(l_i : A_i)_{i\in I}; \cdots\}\right]\!\right] = \left\{\{(l_i = v_i)_{i\in K}\} \mid I \subseteq K \wedge \forall i \in I,\, v_i \in [\![A_i]\!] \setminus \{\square\}\right\} \cup \{\square\}$$

We will now give the interpretation of our different forms of judgments in the semantics, and prove the adequacy of the semantics with respect to the typing rules given in Figures 6.4 and 6.5.

Definition 6.4.15. Let $t \in \Lambda^+$ be a raw term, $\pi \in \Pi^+$ be a raw stack and $A, B \in \mathcal{T}$ be two types. We will write $t \Vdash A$ if $[\![t]\!] \in [\![A]\!]^{\perp\perp}$, $\pi \Vdash A$ if $[\![\pi]\!] \in [\![A]\!]^{\perp}$ and $t \Vdash A \subset B$ if $t \Vdash A$ implies $t \Vdash B$.

Lemma 6.4.16. Let $A, B \in \mathcal{T}$ be two types such that $\varepsilon_{x\in A}(x \notin B) \Vdash A \subset B$. In this case we have $[\![A]\!] \subseteq [\![B]\!]$.

Proof. We proceed by case on the definition of $v = [\![\varepsilon_{x\in A}(x \notin B)]\!]$. If $v \neq \square$ then we have $v \in [\![A]\!]$ and $v \notin [\![B]\!]^{\perp\perp}$. This is a contradiction since $v \notin [\![B]\!]$ by Theorem 4.5.29. If $v = \square$ then for all $w \in [\![A]\!] \setminus \{\square\}$ we have $w \in [\![B]\!]^{\perp\perp}$ and $w \in [\![B]\!]$ thanks to Theorem 5.4.15. Moreover, $\square \in [\![A]\!]$ and $\square \in [\![B]\!]$ so we indeed have $[\![A]\!] \subseteq [\![B]\!]$. □

Theorem 6.4.17. Let Ξ be an equational context, $A, B \in \mathcal{T}$ be closed types and $t \in \Lambda^+$ be a raw term. If for all $(t_1, t_2) \in \Xi$ the equivalence $[\![t_1]\!] \equiv [\![t_2]\!]$ holds then we have the following.

- If $\Xi \vdash t : A$ is valid then we have $t \Vdash A$. Moreover, if t is a value then $[\![t]\!] \neq \square$.
- If $\Xi \vdash \pi : \neg A$ is valid then we have $\pi \Vdash A$.
- If $\Xi \vdash t \in A \subset B$ is valid then we have $t \Vdash A \subset B$.

Proof. We do a proof by induction on the structure of the proof of $\Xi \vdash t : A$, the proof of $\Xi \vdash \pi : \neg A$ and the proof of $\Xi \vdash t \in A \subseteq B$ respectively. We consider the last rules used in the proof.

- In the case of the $(\Rightarrow_i)$ rule, we need to show that $\lambda x.t \Vdash C$. According to the first induction hypothesis, it is enough to show $\lambda x.t \Vdash A \Rightarrow B$. Let us now suppose that there is a value $v \in [\![A]\!] \setminus \{\Box\}$ such that $[\![t[x := v]]\!] \notin [\![B]\!]^{\bot\bot}$. In this case we get a contradiction with our induction hypothesis since we must have $[\![\varepsilon_{x\in A}(x \notin B)]\!] \neq \Box$ and $[\![t[x := \varepsilon_{x\in A}(x \notin B)]]\!] \notin [\![B]\!]^{\bot\bot}$. As a consequence, we know that $[\![t[x := v]]\!] \in [\![B]\!]^{\bot\bot}$ for all $v \in [\![A]\!] \setminus \{\Box\}$, which exactly means that $[\![\lambda x.t]\!] \in [\![A \Rightarrow B]\!]$. We can thus conclude using Theorem 4.5.29.

$$\frac{\Xi \vdash \lambda x.t \in A \Rightarrow B \subset C \quad \Xi, \varepsilon_{x\in A}(t \notin B) \neq \Box \vdash t[x := \varepsilon_{x\in A}(t \notin B)] : B}{\Xi \vdash \lambda x.t : C} \Rightarrow_i$$

- In the case of the $(\Rightarrow_e)$ rule, we need to show that $t\ u \Vdash B$, which is the same as $[\![t]\!]\,[\![u]\!] \in [\![B]\!]^{\bot\bot}$. We thus take $\pi \in [\![B]\!]^{\bot}$ and we show that $[\![t]\!]\,[\![u]\!] * \pi \in \perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated, it is enough to show $[\![u]\!] * [[\![t]\!]]\pi \in \perp\!\!\!\perp$. As $[\![u]\!] \in [\![A]\!]^{\bot\bot}$ by our second induction hypothesis, we will prove that $[[\![t]\!]]\pi \in [\![A]\!]^{\bot}$. We thus take $v \in [\![A]\!]$ and show that we have $v * [[\![t]\!]]\pi \in \perp\!\!\!\perp$. If $v = \Box$ then we have $\Box * [[\![t]\!]]\pi \twoheadrightarrow \Box * \pi$ and since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated it is enough to show $\Box * \pi \in \perp\!\!\!\perp$, which is immediate since $\pi \in [\![B]\!]^{\bot}$ and $\Box \in [\![B]\!]$. Now, if $v \neq \Box$ then $v * [[\![t]\!]]\pi \twoheadrightarrow [\![t]\!] * v.\pi$ and as $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated it is enough to show $[\![t]\!] * v.\pi \in \perp\!\!\!\perp$. Since $[\![t]\!] \in [\![A \Rightarrow B]\!]^{\bot\bot}$ according to the first induction hypothesis, we only have to show that $v.\pi \in [\![A \Rightarrow B]\!]^{\bot\bot}$ so we take $w \in [\![A \Rightarrow B]\!]$ and show that $w * v.\pi \in \perp\!\!\!\perp$. If $w = \Box$ then $\Box * v.\pi \twoheadrightarrow \Box * \pi$ and it is enough to show $\Box * \pi \in \perp\!\!\!\perp$ as $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated. This is immediate since $\pi \in [\![B]\!]^{\bot}$ and $\Box \in [\![B]\!]$. If $w = \lambda x.f$ then $\lambda x.f * v.\pi \twoheadrightarrow f[x := v] * \pi$ and it is enough to show $f[x := v] * \pi \in \perp\!\!\!\perp$ since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated. As $\pi \in [\![B]\!]^{\bot}$ it only remains to show $f[x := v] \in [\![B]\!]^{\bot\bot}$, but this is true by definition of $[\![A \Rightarrow B]\!]$ since $v \in [\![A]\!] \setminus \{\Box\}$.

$$\frac{\Xi \vdash t : A \Rightarrow B \quad \Xi \vdash u : A}{\Xi \vdash t\ u : B} \Rightarrow_e$$

- In the case of the (Ax) rule, we need to $\varepsilon_{x\in A}(t \notin B) \Vdash C$. Using the induction hypothesis, it is enough to show $\varepsilon_{x\in A}(t \notin B) \Vdash A$. Moreover, according to Lemma 4.5.29 we only need to prove $[\![\varepsilon_{x\in A}(t \notin B)]\!] \in [\![A]\!]$, which follows by definition since $\Box \in [\![A]\!]$. Moreover, we obtain $[\![\varepsilon_{x\in A}(t \notin B)]\!] \neq \Box$ using the right premise with Lemma 6.3.8.

$$\frac{\Xi \vdash \varepsilon_{x\in A}(t \notin B) \in A \subset C \quad \Xi \vdash \varepsilon_{x\in A}(t \notin B) \neq \Box}{\Xi \vdash \varepsilon_{x\in A}(t \notin B) : C}\text{Ax}$$

- In the case of the $(\Rightarrow_{e,\in})$ rule, we need to show that $t\ u \Vdash B$, which is the same as $[\![t]\!]\,[\![u]\!] \in [\![B]\!]^{\bot\bot}$. We thus take $\pi \in [\![B]\!]^{\bot}$ and we show $[\![t]\!]\,[\![u]\!] * \pi \in \perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated, it is enough to show $[\![u]\!] * [[\![t]\!]]\pi \in \perp\!\!\!\perp$. Now, as our pole is $(\equiv)$-extensional and we know that $[\![u]\!] \equiv [\![v]\!]$ for a value v, it is enough to show that $[\![v]\!] * [[\![t]\!]]\pi \in \perp\!\!\!\perp$. If $[\![v]\!] = \Box$ then we can conclude as for the $(\Rightarrow_e)$ rule and otherwise we can show $[\![t]\!] * [\![v]\!].\pi \in \perp\!\!\!\perp$ as $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated. As $[\![t]\!] \in [\![u \in A \Rightarrow B]\!]^{\bot\bot}$ by

our first induction hypothesis, we only need to show $[\![v]\!] \cdot \pi \in [\![u \in A \Rightarrow B]\!]^{\perp}$. We thus take a value $w \in [\![u \in A \Rightarrow B]\!]$ and show $w * [\![v]\!] \cdot \pi \in \bot\!\!\!\bot$. If $w = \Box$ then we can conclude as for the $(\Rightarrow_e)$ rule. If $w = \lambda x.f$ then we can again take a reduction step and show that $f[x := [\![v]\!]] * \pi \in \bot\!\!\!\bot$, for which it is enough to show $f[x := [\![v]\!]] \in [\![B]\!]$. To conclude using the definition of $[\![u \in A \Rightarrow B]\!]$ we need to show $[\![v]\!] \in [\![u \in A]\!] \setminus \{\Box\}$. Since we have $[\![v]\!] \neq \Box$ and $[\![u]\!] \equiv [\![v]\!]$ we only need to show $[\![v]\!] \in [\![A]\!]$. Using Theorem 5.4.15 it is enough to show $[\![v]\!] \in [\![A]\!]^{\perp\perp}$, which follows from Theorem 4.5.35 as $[\![u]\!] \in [\![A]\!]^{\perp\perp}$ by our second induction hypothesis since $[\![u]\!] \equiv [\![v]\!]$.

$$\frac{\Xi \vdash t : u \in A \Rightarrow B \quad \Xi \vdash u : A \quad \Xi \vdash v \equiv u}{\Xi \vdash t\ u : B} \Rightarrow_{e,\in}$$

- In the case of the (μ) rule, we need to show that $\mu\alpha.t \Vdash A$, which is the same as $\mu\alpha.[\![t]\!] \in [\![A]\!]^{\perp\perp}$. We thus take $\pi \in [\![A]\!]^{\perp}$ and show $\mu\alpha.[\![t]\!] * \pi \in \bot\!\!\!\bot$. Since $\bot\!\!\!\bot$ is $(\twoheadrightarrow)$-saturated, it is enough to show $[\![t]\!][\alpha := \pi] * \pi = [\![t[\alpha := \pi]]\!] * \pi \in \bot\!\!\!\bot$ and as $\pi \in [\![A]\!]^{\perp}$ we only need to show $[\![t[\alpha := \pi]]\!] \in [\![A]\!]^{\perp\perp}$. Let us now suppose that there is a stack $\xi \in [\![A]\!]^{\perp}$ such that $[\![t[\alpha := \xi]]\!] \notin [\![A]\!]^{\perp\perp}$. In this case we can assume that we have $\xi = [\![\varepsilon_{\alpha \in \neg A}(t \notin A)]\!]$ but this contradicts the induction hypothesis which tells us that $[\![t[\alpha := \varepsilon_{\alpha \in \neg A}(t \notin A)]]\!] \in [\![A]\!]^{\perp\perp}$. As a consequence, we have $[\![t[\alpha := \xi]]\!] \in [\![A]\!]^{\perp\perp}$ for all $\xi \in [\![A]\!]^{\perp}$. In particular, this is true for $\xi = \pi$.

$$\frac{\Xi \vdash t[\alpha := \varepsilon_{\alpha \in \neg A}(t \notin A)] : A}{\Xi \vdash \mu\alpha.t : A}\mu$$

- In the case of the $([-])$ rule, we need to show that $[\pi]u \Vdash B$, which is the same as $[[\![\pi]\!]][\![u]\!] \in [\![B]\!]^{\perp\perp}$. We thus take $\xi \in [\![B]\!]^{\perp}$ and show that $[[\![\pi]\!]][\![u]\!] * \xi \in \bot\!\!\!\bot$. As $\bot\!\!\!\bot$ is $(\twoheadrightarrow)$-saturated, it is enough to show $[\![u]\!] * [\![\pi]\!] \in \bot\!\!\!\bot$ but this is immediate since we have $[\![u]\!] \in [\![A]\!]^{\perp\perp}$ and $[\![\pi]\!] \in [\![A]\!]^{\perp}$ by induction hypothesis.

$$\frac{\Xi \vdash u : A \quad \Xi \vdash \pi : \neg A}{\Xi \vdash [\pi]u : B}[-]$$

- In the case of the $(Ax^{\perp})$ rule, we need to show that $\varepsilon_{\alpha \in \neg A}(t \notin A) \Vdash B$, which is the same as $[\![\varepsilon_{\alpha \in \neg A}(t \notin A)]\!] \in [\![B]\!]^{\perp}$. By induction hypothesis, we know that $\varepsilon_{x \in B}(x \notin A) \Vdash B \subset A$ and thus we can apply Lemma 6.4.16 to get $[\![B]\!] \subseteq [\![A]\!]$. Using Lemma 4.5.30 we then obtain $[\![A]\!]^{\perp} \subseteq [\![B]\!]^{\perp}$ and thus it is enough to show $[\![\varepsilon_{\alpha \in \neg A}(t \notin A)]\!] \in [\![A]\!]^{\perp}$. We now reason by case on the definition of $[\![\varepsilon_{\alpha \in \neg A}(t \notin A)]\!]$. It is either defined to be a stack in $\pi \in [\![A]\!]^{\perp}$ (with some properties) in which case we can conclude immediately, or it is defined to be the stack $[\Box]\varepsilon$. In this second case we take $v \in [\![A]\!]$ and show $v * [\Box]\varepsilon \in \bot\!\!\!\bot$. If $v = \Box$ then we have $\Box * [\Box]\varepsilon \twoheadrightarrow \Box * \varepsilon$ and otherwise we have $v * [\Box]\varepsilon \twoheadrightarrow \Box * v \cdot \varepsilon \twoheadrightarrow \Box * \varepsilon$. We can then conclude since $\bot\!\!\!\bot$ is $(\twoheadrightarrow)$-saturated and $\Box * \varepsilon \in \bot\!\!\!\bot$.

$$\frac{\Xi \vdash B \subset A}{\Xi \vdash \varepsilon_{\alpha \in \neg A}(t \notin A) : \neg B}Ax^{\perp}$$

- In the case of the $(-\cdot-)$ rule, we need to show that $v \cdot \pi \Vdash C$, which is the same as $[\![v]\!] \cdot [\![\pi]\!] \in [\![C]\!]^{\perp}$. Using Lemmas 4.5.30 and 6.4.16 with the third induction hypothesis we have $[\![A \Rightarrow B]\!]^{\perp} \subseteq [\![C]\!]^{\perp}$. It is hence enough to show $[\![v]\!] \cdot [\![\pi]\!] \in [\![A \Rightarrow B]\!]^{\perp}$, so we take $w \in [\![A \Rightarrow B]\!]$ and show $w * [\![v]\!] \cdot [\![\pi]\!] \in \perp\!\!\!\perp$. If $w = \Box$ then it is enough to show $\Box * [\![\pi]\!] \in \perp\!\!\!\perp$ as $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated. This is immediate since $[\![\pi]\!] \in [\![B]\!]^{\perp}$ by the second induction hypothesis and $\Box \in [\![B]\!]$. If $w = \lambda x.f$ then it is enough to show that $f[x := [\![v]\!]] * [\![\pi]\!] \in \perp\!\!\!\perp$ as $(\twoheadrightarrow)$-saturated. Using the second induction hypothesis, we only need to show $f[x := [\![v]\!]] \in [\![B]\!]^{\perp\perp}$. As $[\![v]\!] \in [\![A]\!]^{\perp\perp}$ and $[\![v]\!] \neq \Box$ by our first induction hypothesis conclude by definition of $[\![A \Rightarrow B]\!]$ using Theorem 5.4.15.

$$\frac{\Xi \vdash v : A \quad \Xi \vdash \pi : \neg B \quad \Xi \vdash C \subset A \Rightarrow B}{\Xi \vdash v \cdot \pi : \neg C}\,{\scriptstyle -\cdot -}$$

- In the case of the $([-]-)$ rule, we need to show that $[t]\pi \Vdash A$, which is the same as $[[\![t]\!][\![\pi]\!] \in [\![A]\!]^{\perp}$. We thus take $v \in [\![A]\!]$ and show $v * [[\![t]\!][\![\pi]\!] \in \perp\!\!\!\perp$. If $v = \Box$ we have $\Box * [[\![t]\!][\![\pi]\!] \twoheadrightarrow \Box * [\![\pi]\!]$ and since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated we only need to show $\Box * [\![\pi]\!] \in \perp\!\!\!\perp$. This is immediate as $[\![\pi]\!] \in [\![B]\!]^{\perp}$ by our second induction hypothesis and $\Box \in [\![B]\!]$. If $v \neq \Box$ then we have $v * [[\![t]\!][\![\pi]\!] \twoheadrightarrow [\![t]\!] * v \cdot [\![\pi]\!]$ and since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated, it is enough to show $[\![t]\!] * v \cdot [\![\pi]\!] \in \perp\!\!\!\perp$. By the first induction hypothesis we have $[\![t]\!] \in [\![A \Rightarrow B]\!]^{\perp\perp}$ so we only need to show $v \cdot [\![\pi]\!] \in [\![A \Rightarrow B]\!]^{\perp}$. This can be done as in the proof of the previous case.

$$\frac{\Xi \vdash t : A \Rightarrow B \quad \Xi \vdash \pi : \neg B}{\Xi \vdash [t]\pi : \neg A}\,{\scriptstyle [-]-}$$

- In the case of the $(\times_e)$ rule, we need to show that $v.l_k \Vdash A$, which is the same as $[\![v]\!].l_k \in [\![A]\!]^{\perp\perp}$. We thus take $\pi \in [\![A]\!]^{\perp}$ and show that $[\![v]\!].l_k * \pi \in \perp\!\!\!\perp$. By induction hypothesis, we know that $[\![v]\!] \in [\![\{l_k : A\,; \cdots\}]\!]^{\perp\perp}$ and that $[\![v]\!] \neq \Box$. Moreover, using Theorem 5.4.15 we obtain $[\![v]\!] \in [\![\{l_k : A\,; \cdots\}]\!]$ and thus $[\![v]\!] = \{(l_i = w_i)_{i \in I}\}$ with $k \in I$ and $v_k \in [\![A]\!]$. As a consequence we only need to prove $w_k * \pi \in \perp\!\!\!\perp$ since $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated. This is immediate as $w_k \in [\![A]\!]$ and $\pi \in [\![A]\!]^{\perp}$.

$$\frac{\Xi \vdash v : \{l_k : A\,; \cdots\}}{\Xi \vdash v.l_k : A}\,{\scriptstyle \times_e}$$

- In the case of the $(\times_i)$ rule, we need to show $\{(l_i = v_i)_{i \in I}\} \Vdash C$. Using the first induction hypothesis, it is enough to show that $\{(l_i = [\![v_i]\!])_{i \in I}\} \in \left[\!\!\left[\{(l_i : A_i)_{i \in I}\}\right]\!\!\right]$. By definition, we need to show that $[\![v_i]\!] \in [\![A_i]\!] \setminus \{\Box\}$ for all $i \in I$. This follows by Theorem 5.4.15 using the induction hypotheses.

$$\frac{\Xi \vdash \{(l_i = v_i)_{i \in I}\} \in \{(l_i : A_i)_{i \in I}\} \subset C \quad (\Xi \vdash v_i : A_i)_{i \in I}}{\Xi \vdash \{(l_i = v_i)_{i \in I}\} : C}\,{\scriptstyle \times_i}$$

- In the case of the $(+_e)$ rule, we need to show $[v \mid (C_i[x_i] \to t_i)_{i \in I}] \Vdash C$, which is the same as $[[\![v]\!] \mid (C_i[x_i] \to [\![t_i]\!])_{i \in I}] \in [\![C]\!]^{\perp\perp}$. We thus take a stack $\pi \in [\![C]\!]^{\perp}$ and show that

$[[\nu]\,|\,(C_i[x_i] \to [\![t_i]\!])_{i\in I}] * \pi \in \perp\!\!\!\perp$. Using the first induction hypothesis (together with Theorem 5.4.15) we learn that $[\![\nu]\!] \in \left[\!\left[(C_i : A_i)_{i\in I}\right]\!\right]$ and that $[\![\nu]\!] \neq \Box$. As a consequence, $[\![\nu]\!] = C_k[w_k]$ for some $k \in I$ and $w_k \in [\![A_i]\!] \setminus \{\Box\}$. As $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated we only need to show $[\![t_k]\!][x_k := w_k] * \pi = [\![t_k[x_k := w_k]]\!] * \pi \in \perp\!\!\!\perp$. Let us assume that it is false and find a contradiction. Since $w_k \in [\![A_i]\!] \setminus \{\Box\}$ and $[\![\nu]\!] = C_k[w_k]$ we have $w_k \in [\![A_i \restriction C_k[w_k] \equiv \nu]\!] \setminus \{\Box\}$. As a consequence $\left[\!\left[\varepsilon_{x_k \in A_k \restriction C[x_k] \equiv \nu}(t_k \notin C)\right]\!\right] \neq \Box$ since w_k is a possible definition for the witness. This contradicts the induction hypothesis since it implies $\left[\!\left[t_k[x_k := \varepsilon_{x_k \in A_k \restriction C[x_k] \equiv \nu}(t_k \notin C)]\right]\!\right] \notin [\![C]\!]^{\perp\perp}$.

$$\frac{\Xi \vdash \nu : [(C_i : A_i)_{i\in I}] \qquad \left(\Xi \vdash t_i[x_i := \varepsilon_{x_i \in A_i \restriction C_i[x_i] \equiv \nu}(t_i \notin C)] : C\right)_{i\in I}}{\Xi \vdash [\nu\,|\,(C_i[x_i] \to t_i)_{i\in I}] : C}\,{+_e}$$

- In the case of the $(+_i)$ rule, we need to show $C_k[\nu] \Vdash B$. Using the second induction hypothesis and Lemma 4.5.29, it is enough to show $C_k[[\![\nu]\!]] \in [\![[C_k : A]]\!]$. By definition, it is enough to show $[\![\nu]\!] \in [\![A]\!] \setminus \{\Box\}$ which follows from the first induction hypothesis and Theorem 5.4.15.

$$\frac{\Xi \vdash \nu : A \qquad \Xi \vdash C_k[\nu] \in [C_k : A] \subset B}{\Xi \vdash C_k[\nu] : B}\,{+_i}$$

- In the case of the $(\mathrm{Ax}_\subset)$ rule, we need to show $t \Vdash A\rho_1 \subset A\rho_2$. It is enough to show $[\![A\rho_1]\!] = [\![A\rho_2]\!]$ since in this case we will have $[\![A\rho_1]\!]^{\perp\perp} = [\![A\rho_2]\!]^{\perp\perp}$ which implies our goal. For every λ-variable or term variable $\chi \in \mathrm{FV}(A)$ we have $[\![\rho_1(\chi)]\!] \equiv [\![\rho_2(\chi)]\!]$. As a consequence, we can conclude with a simple proof by induction using Theorems 5.5.18 and 5.5.21 to substitute one variable at a time.

$$\frac{(\Xi \vdash \rho_1(\chi) \equiv \rho_2(\chi))_{\chi \in \mathrm{FV}(A)}}{\Xi \vdash t : A\rho_1 \subset A\rho_2}\,\mathrm{Ax}_\subset$$

- In the case of the (Gen) rule, we need to show that $t \Vdash A \subset B$. Using the induction hypothesis with Lemma 6.4.16 we obtain $[\![A]\!] \subseteq [\![B]\!]$, and we can thus conclude using Lemma 4.5.32.

$$\frac{\Xi \vdash A \subset B}{\Xi \vdash t : A \subset B}\,\mathrm{Gen}$$

- In the case of the $(\Rightarrow)$ rule, we need to show that $t \Vdash A_1 \Rightarrow B_1 \subset A_2 \Rightarrow B_2$. We thus assume $[\![t]\!] \in [\![A_1 \Rightarrow B_1]\!]^{\perp\perp}$ and show that we have $[\![t]\!] \in [\![A_2 \Rightarrow B_2]\!]^{\perp\perp}$. Now, as $\perp\!\!\!\perp$ is $(\equiv)$-extensional and $[\![\nu]\!] \equiv [\![t]\!]$ for some value ν, we know that $[\![\nu]\!] \in [\![A_1 \Rightarrow B_1]\!]^{\perp\perp}$ and we only have to show that $[\![\nu]\!] \in [\![A_2 \Rightarrow B_2]\!]^{\perp\perp}$. With Theorem 5.4.15 we even have $[\![\nu]\!] \in [\![A_1 \Rightarrow B_1]\!]$ and we can show $[\![\nu]\!] \in [\![A_2 \Rightarrow B_2]\!]$. If $[\![\nu]\!] = \Box$ then this is immediate as we have $\Box \in [\![A_2 \Rightarrow B_2]\!]$ by definition. Let us now suppose that $[\![\nu]\!] = \lambda x.f$ and that for all $w \in [\![A_1]\!] \setminus \{\Box\}$ we have $f[x := w] \in [\![B_1]\!]^{\perp\perp}$. If we have $[\![t]\!]\, w \in [\![B_2]\!]^{\perp\perp}$ for all $w \in [\![A_2]\!] \setminus \{\Box\}$ then, since we have $[\![t]\!] \equiv \lambda x.f$, we can use Theorems 4.5.35 and 5.5.21 to get $(\lambda x.f)\, w \in [\![B_2]\!]^{\perp\perp}$. We can then use Theorem 3.3.16 to obtain $f[x := w] \in [\![B_2]\!]^{\perp\perp}$ for the same reason. This exactly means that we have

$[\![v]\!] = \lambda x.f \in [\![A_2 \Rightarrow B_2]\!]$. Finally, let us suppose that $[\![t]\!]\, w \notin [\![B_2]\!]^{\perp\perp}$ for some value $w \in [\![A_2]\!] \setminus \{\square\}$. We can assume that $w = \left[\!\left[\varepsilon_{x \in A_2}(t\ x \notin B_2)\right]\!\right]$ and thus the first induction hypothesis gives us $w \in [\![A_1]\!]$. We then obtain that $f[x := w] \in [\![B_1]\!]^{\perp\perp}$ by definition of $[\![A_1 \Rightarrow B_1]\!]$. Now, using again Theorems 3.3.16 and 5.5.21 we obtain $f[x := w] \equiv [\![t]\!]\, w$ and thus we get a contradiction with the second induction hypothesis.

$$\frac{\Xi, w \neq \square \vdash w \in A_2 \subset A_1 \quad \Xi, w \neq \square \vdash t\ w \in B_1 \subset B_2 \quad \Xi \vdash t \equiv v}{\Xi \vdash t \in A_1 \Rightarrow B_1 \subset A_2 \Rightarrow B_2}{\scriptstyle \Rightarrow \quad w = \varepsilon_{x \in A_2}(t\ x \notin B_2)}$$

- In the case of the $(+)$ rule, we need to show $t \Vdash [(C_i : A_i)_{i \in I_1}] \subset [(C_i : B_i)_{i \in I_2}]$. As in the case of the $(\Rightarrow)$ rule, we know that $[\![t]\!] \equiv [\![v]\!]$ so we can assume that we have $[\![v]\!] \in \left[\!\left[(C_i : A_i)_{i \in I_1}\right]\!\right]$ and show that $[\![v]\!] \in \left[\!\left[(C_i : B_i)_{i \in I_2}\right]\!\right]$. If $[\![v]\!] = \square$ then the proof is trivial as for the $(\Rightarrow)$ rule. As a consequence, we may assume that $[\![v]\!] = C_k[w]$ for some $k \in I_1$ and $w \in [\![A_k]\!] \setminus \{\square\}$. Thus, we only need to show that $w \in [\![B_k]\!]$. Let us now consider the term $[\![(\lambda x.[x \mid C_k[x_k] \to x_k])\ t]\!] = (\lambda x.[x \mid C_k[x_k] \to x_k])\ [\![t]\!]$. As we have $[\![t]\!] \equiv [\![v]\!] = C_k[w]$, we know $(\lambda x.[x \mid C_k[x_k] \to x_k])\ [\![t]\!] \equiv (\lambda x.[x \mid C_k[x_k] \to x_k])\ C_k[w]$. Using Theorems 3.3.16 and 3.3.15 we even obtain $(\lambda x.[x \mid C_k[x_k] \to x_k])\ [\![t]\!] \equiv w$. We can hence conclude using the induction hypothesis, the $(\equiv)$-extensionality of the pole and Theorem 5.4.15.

$$\frac{I_1 \subset I_2 \quad (\Xi \vdash (\lambda x.[x \mid C_i[x_i] \to x_i])\ t \in A_i \subset B_i)_{i \in I_1} \quad \Xi \vdash t \equiv v}{\Xi \vdash t \in [(C_i : A_i)_{i \in I_1}] \subset [(C_i : B_i)_{i \in I_2}]}+$$

- In the case of the $(\times)$ rule, we need to show $t \Vdash \{(l_i : A_i)_{i \in I}\} \subset \{(l_i : B_i)_{i \in I}\}$. As in the case of the $(\Rightarrow)$ rule, we know that $[\![t]\!] \equiv [\![v]\!]$ so we can assume that we have $[\![v]\!] \in \left[\!\left\{(l_i : A_i)_{i \in I}\right\}\!\right]$ and show that $[\![v]\!] \in \left[\!\left\{(l_i : B_i)_{i \in I}\right\}\!\right]$. Again, if $[\![v]\!] = \square$ then the proof is trivial. Hence, we may assume $[\![v]\!] = \{(l_i = v_i)_{i \in I}\}$ and $v_i \in [\![A_i]\!] \setminus \{\square\}$ for all index $i \in I$. As a consequence, we only need to show that $v_i \in [\![B_i]\!]$ for all $i \in I$. Let us take $k \in I$ and consider the term $[\![(\lambda x.x.l_k)\ t]\!] = (\lambda x.x.l_k)\ [\![t]\!]$. As we have $[\![t]\!] \equiv [\![v]\!] = \{(l_i = v_i)_{i \in I}\}$, we know $(\lambda x.x.l_k)\ [\![t]\!] \equiv (\lambda x.x.l_k)\ \{(l_i = v_i)_{i \in I}\}$. We then obtain $(\lambda x.x.l_k)\ [\![t]\!] \equiv v_k$ using Theorems 3.3.16 and 3.3.15. We can hence conclude using the induction hypothesis, the $(\equiv)$-extensionality of the pole and Theorem 5.4.15.

$$\frac{(\Xi \vdash (\lambda x.x.l_i)\ t \in A_i \subset B_i)_{i \in I} \quad \Xi \vdash t \equiv v}{\Xi \vdash t \in \{(l_i : A_i)_{i \in I}\} \subset \{(l_i : B_i)_{i \in I}\}}\times$$

- In the case of the $(\times_{ext})$ rule, the proof is similar to the $(\times)$ case. Since we know that $[\![t]\!] \equiv [\![v]\!]$ we can assume $[\![v]\!] \in \left[\!\left\{(l_i : A_i)_{i \in I_1}\right\}\!\right]$ and show that $[\![v]\!] \in \left[\!\left\{(l_i : B_i)_{i \in I_2}\right\}\!\right]$. If $[\![v]\!] = \square$ then the proof is trivial so we may assume that $[\![v]\!] = \{(l_i = v_i)_{i \in I}\}$ with $I_1 \subseteq I$ and $v_i \in [\![A_i]\!]$ for all index $i \in I_1$. As a consequence, we only need to show $v_i \in [\![B_i]\!]$ for all $i \in I_2$. Let us take $k \in I_2$ and consider the term $[\![(\lambda x.x.l_k)\ t]\!] = (\lambda x.x.l_k)\ [\![t]\!]$. As we have $[\![t]\!] \equiv [\![v]\!] = \{(l_i = v_i)_{i \in I}\}$, we know $(\lambda x.x.l_k)\ [\![t]\!] \equiv (\lambda x.x.l_k)\ \{(l_i = v_i)_{i \in I}\}$. Using

Theorems 3.3.16 and 3.3.15 we obtain $(\lambda x.x.l_k)\ [\![t]\!] \equiv \nu_k$. We can hence conclude using the induction hypothesis, the $(\equiv)$-extensionality of the pole and Theorem 5.4.15.

$$\frac{I_2 \subset I_1 \quad (\Xi \vdash (\lambda x.x.l_i)\ t \in A_i \subset B_i)_{i \in I_2} \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in \{(l_i : A_i)_{i \in I_1}; \cdots\} \subset \{(l_i : B_i)_{i \in I_2}; \cdots\}} \times_{\text{ext}}$$

- In the case of the $(\times_{\subset})$ rule, the proof is exactly the same as for $(\times_{\text{ext}})$ with $I = I_1$.

$$\frac{I_2 \subset I_1 \quad (\Xi \vdash (\lambda x.x.l_i)\ t \in A_i \subset B_i)_{i \in I_2} \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t \in \{(l_i : A_i)_{i \in I_1}\} \subset \{(l_i : B_i)_{i \in I_2}; \cdots\}} \times_{\subset}$$

- In the case of the $(\forall_l)$ rule, we need to show that $t \Vdash \forall \chi^s.A \subset B$. We thus suppose that $t \Vdash \forall \chi^s.A$ and show $t \Vdash B$. By induction hypothesis, it is enough to prove $t \Vdash A[\chi := C]$ and thus we will show $[\![\forall \chi^s.A]\!]^{\perp\perp} \subseteq [\![A[\chi := C]]\!]^{\perp\perp}$. According to Lemma 4.5.32, it is enough to show $[\![\forall \chi^s.A]\!] \subseteq [\![A[\chi := C]]\!]$, which is immediate by definition.

$$\frac{\Xi \vdash t \in A[\chi := C] \subset B}{\Xi \vdash t \in \forall \chi^s.A \subset B} \forall_l$$

- In the case of the $(\forall_r)$ rule, we need to show that $t \Vdash A \subset \forall \chi^s.B$. We thus suppose that $[\![t]\!] \in [\![A]\!]^{\perp\perp}$ and show that $[\![t]\!] \in [\![\forall \chi^s.B]\!]^{\perp\perp}$. We have $[\![t]\!] \in [\![B[\chi := \varepsilon_{\chi:s}(t \notin B)]]\!]^{\perp\perp}$ by induction hypothesis. Moreover, as the pole is $(\equiv)$-extensional and $[\![\nu]\!] \equiv [\![t]\!]$ for some value ν, we have $[\![\nu]\!] \in [\![B[\chi := \varepsilon_{\chi:s}(t \notin B)]]\!]^{\perp\perp}$ according to Theorem 4.5.35. For the same reason, it will be enough for us to show that $[\![\nu]\!] \in [\![\forall \chi^s.B]\!]^{\perp\perp}$. With Theorem 5.4.15 we even have $[\![\nu]\!] \in [\![B[\chi := \varepsilon_{\chi:s}(t \notin B)]]\!]$ and we can show $[\![\nu]\!] \in [\![\forall \chi^s.B]\!]$. We now suppose that there is $\Phi \in [\![s]\!]$ such that $[\![t]\!] \notin [\![B[\chi := \Phi]]\!]^{\perp\perp}$. We can thus assume that $[\![B[\chi := \varepsilon_{\chi:s}(t \notin B)]]\!] = [\![B[\chi := \Phi]]\!]$, which contradicts $[\![t]\!] \in [\![B[\chi := \Phi]]\!]^{\perp\perp}$. As a consequence, it must be that for every formula $\Phi \in [\![s]\!]$ we have $[\![t]\!] \in [\![B[\chi := \Phi]]\!]^{\perp\perp}$, or equivalently $[\![\nu]\!] \in [\![B[\chi := \Phi]]\!]$ using Theorems 4.5.35 and 5.4.15. This immediately implies that $[\![\nu]\!] \in [\![\forall \chi^s.B]\!]$.

$$\frac{\Xi \vdash t \in A \subset B[\chi := \varepsilon_{\chi:s}(t \notin B)] \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t \in A \subset \forall \chi^s.B} \forall_r$$

- In the case of the $(\exists_r)$ rule, the proof is similar as for the $(\forall_l)$ rule. We need to show that $t \Vdash A \subset \exists \chi^s.B$ so we suppose $t \Vdash A$ and show $t \Vdash \exists \chi^s.B$. Using the induction hypothesis, we know that $t \Vdash B[\chi := C]$. As a consequence, we only need to show $[\![B[\chi := C]]\!]^{\perp\perp} \subseteq [\![\exists \chi^s.B]\!]^{\perp\perp}$. This immediatly follows from the definition of $[\![\exists \chi^s.B]\!]$ using Lemma 4.5.32.

$$\frac{\Xi \vdash t \in A \subset B[\chi := C]}{\Xi \vdash t \in A \subset \exists \chi^s.B} \exists_r$$

- In the case of the $(\exists_l)$ rule, the proof is similar as for the $(\forall_r)$ rule. We need to show that $t \Vdash \exists \chi^s.A \subset B$ so we suppose $t \Vdash \exists \chi^s.A$ and show $t \Vdash B$. To be able to use the induction hypothesis, we need to show that $[\![t]\!] \in [\![A[\chi := \varepsilon_{\chi:s}(t \in A)]]\!]^{\perp\perp}$, provided

that $[\![t]\!] \in [\![\exists \chi^s.A]\!]^{\perp\perp}$. As we have $[\![v]\!] \equiv [\![t]\!]$ we can use Theorems 4.5.35 and 5.4.15, assume $[\![v]\!] \in [\![\exists \chi^s.A]\!]$ and show $[\![v]\!] \in [\![A[\chi := \varepsilon_{\chi:s}(t \in A)]]\!]$. We will now suppose, by contradiction, that for every element Φ of $[\![s]\!]$ we have $[\![t]\!] \notin [\![A[\chi := \Phi]]\!]^{\perp\perp}$ and thus $[\![v]\!] \notin [\![A[\chi := \Phi]]\!]$ using again Theorems 4.5.35 and 5.4.15. This is a contradiction since this exactly means that $[\![v]\!] \notin [\![\exists \chi^s.A]\!]$. Hence, there must be $\Phi \in [\![s]\!]$ such that $[\![t]\!] \in [\![A[\chi := \Phi]]\!]^{\perp\perp}$. We can thus suppose $[\![A[\chi := \varepsilon_{\chi:s}(t \in A)]]\!] = [\![A[\chi := \Phi]]\!]$ which gives us $[\![t]\!] \in [\![A[\chi := \varepsilon_{\chi:s}(t \in A)]]\!]^{\perp\perp}$. We can thus conclude the proof using Theorems 4.5.35 and 5.4.15 one more time.

$$\frac{\Xi \vdash t \in A[\chi := \varepsilon_{\chi:s}(t \in A)] \subset B \qquad \Xi \vdash t \equiv v}{\Xi \vdash t \in \exists \chi^s.A \subset B}\,{\scriptstyle \exists_l}$$

- In the case of the $(\restriction_l)$ rule, we need to show that $t \Vdash A \restriction u_1 \equiv u_2 \subset B$. We thus assume that we have $[\![t]\!] \in [\![A \restriction u_1 \equiv u_2]\!]^{\perp\perp}$ and we show $[\![t]\!] \in [\![B]\!]^{\perp\perp}$. As in the case of the $(\Rightarrow)$ rule, we know that $[\![t]\!] \equiv [\![v]\!]$ so we can assume that we have $[\![v]\!] \in [\![A \restriction u_1 \equiv u_2]\!]$ and we can show $[\![v]\!] \in [\![B]\!]$. Now, if $[\![v]\!] = \square$ then we can conclude immediately. Otherwise, we have $[\![v]\!] \in [\![A]\!]$ and $[\![u_1]\!] \equiv [\![u_2]\!]$ by definition of $[\![A \restriction u_1 \equiv u_2]\!]$. As a consequence, we get $[\![v]\!] \in [\![B]\!]$ by induction hypothesis (using $[\![u_1]\!] \equiv [\![u_2]\!]$).

$$\frac{\Xi, u_1 \equiv u_2 \vdash t \in A \subset B \qquad \Xi \vdash v \equiv t}{\Xi \vdash t \in A \restriction u_1 \equiv u_2 \subset B}\,{\scriptstyle \restriction_l}$$

- In the case of the $(\restriction_r)$ rule, we need to show that $t \Vdash A \subset B \restriction u_1 \equiv u_2$. We thus assume $t \Vdash A$ and show $t \Vdash B \restriction u_1 \equiv u_2$. Using the induction hypothesis, we know that $t \Vdash B$. We will thus conclude the proof by showing $[\![B \restriction u_1 \equiv u_2]\!] = [\![B]\!]$, which implies $[\![B \restriction u_1 \equiv u_2]\!]^{\perp\perp} = [\![B]\!]^{\perp\perp}$. This is immediate since $[\![u_1]\!] \equiv [\![u_2]\!]$ by hypothesis.

$$\frac{\Xi \vdash t \in A \subset B \qquad \Xi \vdash u_1 \equiv u_2}{\Xi \vdash t \in A \subset B \restriction u_1 \equiv u_2}\,{\scriptstyle \restriction_r}$$

- In the case of the $(\in_l)$ rule, we need to show that $t \Vdash u \in A \subset B$. We thus assume that we have $t \Vdash u \in A$ and show $t \Vdash B$. As in the case of the $(\Rightarrow)$ rule, we know that $[\![t]\!] \equiv [\![v]\!]$ so we can assume that we have $[\![v]\!] \in [\![u \in A]\!]$ and show $[\![v]\!] \in [\![B]\!]^{\perp\perp}$. As $[\![v]\!] \in [\![u \in A]\!]$, we have $[\![v]\!] \equiv [\![u]\!]$ and $[\![v]\!] \in [\![A]\!]$ by definition. We can thus conclude by using the induction hypothesis to get $[\![v]\!] \in [\![B]\!]$ since we know $[\![t]\!] \equiv [\![u]\!]$.

$$\frac{\Xi, t \equiv u \vdash t \in A \subset B \qquad \Xi \vdash t \equiv v}{\Xi \vdash t \in u \in A \subset B}\,{\scriptstyle \in_l}$$

- In the case of the $(\in_r)$ rule, we need to show that $t \Vdash A \subset u \in B$. We thus suppose that $t \Vdash A$ and show $t \Vdash u \in B$. Using the first induction hypothesis, we know that $t \Vdash B$. Now, as the pole is $(\equiv)$-extensional and $[\![v]\!] \equiv [\![t]\!]$ for some value v, we can use Lemma 4.5.35 and Theorem 5.4.15 to obtain $[\![v]\!] \in [\![B]\!]$. For a similar reason, it is enough to show that $[\![v]\!] \in [\![u \in B]\!]^{\perp\perp}$. By definition of $[\![u \in B]\!]$, it only remains

to show $[\![v]\!] \equiv [\![u]\!]$, which follows by transitivity of $(\equiv)$ knowing $[\![v]\!] \equiv [\![t]\!]$ and $[\![t]\!] \equiv [\![u]\!]$.

$$\frac{\Xi \vdash t \in A \subset B \quad \Xi \vdash t \equiv u \quad \Xi \vdash t \equiv v}{\Xi \vdash t \in A \subset u \in B}\,{\scriptstyle \in_r}$$

□

6.5 Completeness on pure data types

In the previous section, the semantics was shown to be adequate with respect to our typing rules. In other words, typed programs really are what they are expected to be in the semantics. Our adequacy lemma (Theorem 6.4.17) is thus a soundness result, and we are now going to wonder about completeness.

Although we cannot hope for a full completeness of our semantics, it is still possible to show that our system is complete when restricted to simple enough types. In particular, we will consider types that do not contain arrow (or function) types. They will not contain quantifiers either.

Definition 6.5.18. A type $A \in \mathcal{F}$ is said to be a *pure data type* if it is only constructed using sum types and strict product types. In other words, a pure data type is generated by the following BNF grammar. We will denote $\mathcal{F}_0$ the set of all the pure data types.

$$(\mathcal{F}_0) \qquad A, B ::= \{(l_i : A_i)_{i\in I}\} \mid [(C_i : A_i)_{i\in I}]$$

Here, pure data types are rather limited. However, if the system was extended with inductive types, then they could also be included in the definition. As a consequence, pure data types would contain, for example, unary natural numbers or lists.

Theorem 6.5.19. Let $A \in \mathcal{F}_0$ be a pure data type. For every value $v \in [\![A]\!] \setminus \{\square\}$ the judgment $\vdash v : A$ is derivable.

Proof. We do a proof by induction on the structure of A. If it is of the form $\{(l_i : A_i)_{i\in I}\}$ then by definition $v = \{(l_i = v_i)_{i\in I}\}$ with $v_i \in [\![A_i]\!] \setminus \{\square\}$ since $v \neq \square$. As a consequence, the induction hypotheses give us a proof of $\vdash v_k : A_k$ for all $k \in I$. We can thus build a proof of $\vdash \{(l_i = v_i)_{i\in I}\} : \{(l_i : A_i)_{i\in I}\}$ as follows.

$$\frac{\dfrac{}{\vdash \{(l_i = v_i)_{i\in I}\} \in \{(l_i : A_i)_{i\in I}\} \subset \{(l_i : A_i)_{i\in I}\}}\,{\scriptstyle \mathrm{Ax}_\subset} \qquad [\vdash v_i : A_i]_{i\in I}}{\vdash \{(l_i = v_i)_{i\in I}\} : \{(l_i : A_i)_{i\in I}\}}\,{\scriptstyle \times_i}$$

Note that the base case of our induction is the empty record type $\{\}$. In this case the above proof does not have any open premise.

If A is of the form $[(C_i : A_i)_{i\in I}]$ then by definition we have $v = C_k[w]$ with $k \in I$ and $w \in [\![A_k]\!] \setminus \{\square\}$ since $v \neq \square$. Our induction hypothesis hence gives us a derivation of $\vdash w : A_k$ that we can use to build a proof of $\vdash C_k[w] : [(C_i : A_i)_{i\in I}]$ as follows.

$$\dfrac{\vdash w : A_k \qquad \dfrac{\{k\} \subseteq I \qquad \dfrac{}{\vdash (\lambda x.[x \mid C_k[x_k] \to x_k])\ C_k[w] \in A_k \subset A_k}\mathrm{Ax}_{\subset} \qquad \vdash C_k[w] \equiv C_k[w]}{\vdash C_k[w] \in [C_k : A_k] \subset [(C_i : A_i)_{i\in I}]}+}{\vdash C_k[w] : [(C_i : A_i)_{i\in I}]}+_i$$

□

6.6 Normalisation, safety and consistency

Thanks to our new adequacy lemma (Theorem 6.4.17), we can now study the properties of our system. Although a normalisation result was already given in Chapter 4, we cannot reuse it since our type system and its semantics have been modified. We will however use very similar techniques, which consist in considering particular examples of poles.

In this thesis, the choice of a pole is more constrained than it usually is the framework of classical realizability (e.g., [*Krivine 2009*, *Miquel 2011*]). Indeed, the only property that is commonly required of a pole is to be saturated under the reduction relation of the abstract machine (i.e., to be closed under backward reduction). Here however, we need to ask for more properties. Our poles need to be $(\equiv)$-extensional for the semantics of our types to be closed under equivalence. Moreover, they must satisfy the conditions of our main theorem (Theorem 5.4.15). In particular they must only contain terminating processes and they must include the process $\square * \varepsilon$.

Theorem 6.6.20. Every closed, typed term normalises. More precisely, for every term $t \in \Lambda^*$ such that $\vdash t : A$ is derivable, there is $v \in \Lambda^*_\iota \setminus \{\square\}$ such that $t * \pi \twoheadrightarrow^* v * \varepsilon$.

Proof. We consider the pole $\perp\!\!\!\perp = \{p \in \Lambda * \Pi \mid \exists\, v \in \Lambda_\iota,\ p \twoheadrightarrow^* v * \varepsilon\}$ which is trivially saturated. Moreover, this pole only contains processes that reduce to a final state. Let us now verify that $\perp\!\!\!\perp$ is $\equiv$-extensional. We thus suppose that $t \equiv u$ and that $t * \pi \in \perp\!\!\!\perp$. By definition of $\perp\!\!\!\perp$, there must be a value v such that $t * \pi \twoheadrightarrow^* v * \varepsilon$. This means that there must be $k \in \mathbb{N}$ such that $t * \pi \twoheadrightarrow^*_k v * \varepsilon$, and hence we have $t * \pi \Downarrow_{\twoheadrightarrow}$. Since $t \equiv u$ we can deduce $u * \pi \Downarrow_{\twoheadrightarrow}$, and thus there must be a value w such that $u * \pi \twoheadrightarrow^*_k w * \varepsilon$. As a consequence, we have $u * \pi \twoheadrightarrow^* w * \varepsilon$, which gives us $u * \pi \in \perp\!\!\!\perp$.

We can now apply Theorem 6.4.17 with the pole $\perp\!\!\!\perp$ and obtain $t \Vdash A$. Since $t \in \Lambda^*$, it cannot contain any choice operator and we have $[\![t]\!] = t \in [\![A]\!]^{\perp\perp}$ by definition. This means that $t * \pi \in \perp\!\!\!\perp$ for every stack $\pi \in [\![A]\!]^\perp$. In particular, we have $t * \varepsilon \in \perp\!\!\!\perp$ as we trivially have $\varepsilon \in [\![A]\!]^\perp$. This exactly means that there is a value $v \in \Lambda_\iota$ such that $t * \varepsilon \twoheadrightarrow^* v * \varepsilon$. It remains to show that v is closed and different from $\square$. This follows from the fact that $t * \varepsilon$

is closed and that it does not contain $\square$ since none of our reduction rules can introduce free variables or the value $\square$. $\square$

Now that we have proved normalisation, we will show that our system is type safe. In other words, a typed program is expected to reduce to a value of the corresponding type. For instance, a program which type corresponds to some encoding of the natural numbers is expected to evaluate to a value representing a natural number. As usual in classical realizability, we do not prove type safety for all types. In particular, safety is never proved for types containing function arrows. We will here restrict ourselves to pure data types.

Remark 6.6.21. The restriction to pure data types is not a problem in practice. For example, functions can only be observed through their application. In particular, placing a function in a well-typed context will only allow us to observe well-typed output. Similar arguments apply to all the type constructors that are not directly considered for type safety.

Even when we restrict to pure data types, the proof of type safety is subtle in our system. The difficult part consist in showing that the pole defined from the value level interpretation of a pure data type is $\equiv$-extensional. This is in fact possible thanks (again) to our $\delta_{v,w}$ term constructor.

Theorem 6.6.22. For every closed term $t \in \Lambda^*_\iota$ such that $\vdash t : A$ for a pure data type $A \in \mathcal{F}_0$, there is a value $v \in [\![A]\!] \setminus \{\square\}$ such that $t * \varepsilon \twoheadrightarrow^* v * \varepsilon$.

Proof. We consider the pole $\perp\!\!\!\perp_A = \{p \in \Lambda * \Pi \mid \exists\, v \in [\![A]\!],\, p \twoheadrightarrow^* v * \varepsilon\}$ which is trivially saturated. Moreover, this pole only contains processes that reduce to a final state. Note that the set $[\![A]\!]$ can be used in the definition of $\perp\!\!\!\perp_A$ because A is a pure data type. In particular, the type A does not contain arrow types. If they did, the definition would be circular as the pole is used in the interpretation of the arrow type. Let us now verify that $\perp\!\!\!\perp_A$ is $\equiv$-extensional. We thus suppose that $t \equiv u$ and that $t * \pi \in \perp\!\!\!\perp_A$. By definition there must be a value $v \in [\![A]\!]$ such that $t * \pi \twoheadrightarrow^* v * \varepsilon$. Now, since $t \equiv u$ there must be $w \in \Lambda_\iota$ such that $u * \pi \twoheadrightarrow^* w * \varepsilon$ and it remains to show that $w \in [\![A]\!]$. To conclude, it is enough to show $w \equiv v$ as we know that $[\![A]\!]$ is closed under $(\equiv)$. Let us now apply the substitution $\rho = [\varepsilon := [\lambda x.\delta_{v,x}]\varepsilon]$ to $t * \pi$. We thus obtain $(t * \pi)\rho \twoheadrightarrow^* v\rho * [\lambda x.\delta_{v,x}]\varepsilon$ and since A is a pure data types, it is easy to see that v cannot contain terms (since they only appear in λ-abstractions) nor stacks (since they only appear in terms). As a consequence, we have $(t * \pi)\rho \twoheadrightarrow^* v * [\lambda x.\delta_{v,x}]\varepsilon \twoheadrightarrow^2 \delta_{v,v} * \varepsilon$, which is stuck. As a consequence, we know that $t\rho * \pi\rho \notin \perp\!\!\!\perp$. Now, since we have $t \equiv u$, we can apply Theorem 4.5.35 and obtain that $u\rho * \pi\rho \notin \perp\!\!\!\perp$. By applying reduction steps, we obtain that $w\rho * [\lambda x.\delta_{v,x}]\varepsilon$ is not in $\perp\!\!\!\perp$ either. Now, since we have $w\rho * [\lambda x.\delta_{v,x}]\varepsilon \twoheadrightarrow^2 \delta_{v,w\rho} * \varepsilon$ then we also know that $\delta_{v,w\rho} * \varepsilon \notin \perp\!\!\!\perp$. This can only be true if $w\rho \equiv v$. Now, since $[\![A]\!]$ is closed under $(\equiv)$ then it must be that

$w\rho \in [\![A]\!]$. As mentioned above, the elements of $[\![A]\!]$ cannot contain the empty stack symbol ε and thus we have $w\rho = w$, which gives $w \in [\![A]\!]$.

We can now apply Theorem 6.4.17 with the pole $\bot\!\!\!\bot_A$ and obtain $t \Vdash A$. Since $t \in \Lambda^*$ it cannot contain any choice operator, and thus we have $[\![t]\!] = t \in [\![A]\!]^{\bot\bot}$ by definition. This means that $t * \pi \in \bot\!\!\!\bot_A$ for every stack $\xi \in [\![A]\!]^{\bot}$. In particular, we have $t * \varepsilon \in \bot\!\!\!\bot_A$ as we trivially have $\varepsilon \in [\![A]\!]^{\bot}$. This exactly means that there is a value $v \in [\![A]\!]$ such that $t * \varepsilon \twoheadrightarrow^* v * \varepsilon$. Moreover, $v \neq \square$ since a typed term cannot contain $\square$. □

One of the applications of our type safety theorem is to show the consistency of the system. In particular, we can immediately show that the type $[]$ (i.e., the empty sum type) is empty. In other words, there should be no typable program of type $[]$.

Theorem 6.6.23. There is no closed term $t \in \Lambda^*$ such that $\vdash t : []$ is derivable.

Proof. Let us assume that there a term $t \in \Lambda^*$ that $\vdash t : []$. As $[]$ is a pure data type we can apply Theorem 6.6.22 to obtain a value $v \in [\![[]]\!]$ such that $t * \varepsilon \twoheadrightarrow^* v * \varepsilon$. However, we have $[\![[]]\!] = \{\square\}$ so it must be that $v = \square$. This is a contradiction since we know that the process $t * \varepsilon$ does not contain $\square$, and thus $v * \varepsilon$ cannot contain $\square$ either according to Lemma 6.4.10. □

In our system, there are many ways of building an empty type. As a consequence, Theorem 6.6.23 is not enough for ensuring the consistency of the system as it only considers the type $[]$. However, the other forms of empty type can be handled using typing. Let us consider the type $\forall X.X$, which also denotes an empty type. Let us suppose that we have a term t such that $\vdash t : \forall X.X$ and t does not contain $\square$. We can then build the following typing derivation, which gives a term of type $[]$.

$$\dfrac{\dfrac{}{\vdash \lambda x.x \in (\forall X.X) \Rightarrow [] \subset (\forall X.X) \Rightarrow []}\,\mathrm{Ax}_{\subset} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \varepsilon_{x \in \forall X.X}(x \notin []) \in [] \subset []}\,\mathrm{Ax}_{\subset}}{\vdash \varepsilon_{x \in \forall X.X}(x \notin []) \in \forall X.X \subset []}\,\forall_l \quad \vdash \varepsilon_{x \in \forall X.X}(x \notin []) : []}{\vdash \lambda x.x : (\forall X.X) \Rightarrow []}\,\Rightarrow_i \qquad \vdash t : \forall X.X}{\vdash (\lambda x.x)\ t : []}\,\Rightarrow_e$$

The existence of such a term contradicts Theorem 6.6.23 and thus there cannot exist terms such as t. Similar proofs can be made for other forms of empty types.

6.7 Toward (co-)inductive types and recursion

The type system described in this chapter does not yet contain all the ingredients required for a practical programming language and proof system. Indeed, it lacks inductive types and does not allow recursion. In this section, we will hint toward the inclusion of these features. To this aim, we will rely on the approach described in [*Lepigre 2017*] by Christophe Raffalli and the author. In particular, we will extend the system with typing rules allowing the construction of infinite typing and subtyping derivations. We will then rely on a notion of *syntactic ordinals* to show that they are well-founded and thus correct.

In the system, ordinals will be handled using another atomic sort, which will automatically provide us with quantification over ordinals. In our types, ordinals will be used to annotate inductive (and coinductive) types with a size information. As a consequence, they will allow us to extend our system with sized types [*Hughes 1996*, *Abel 2008*, *Lepigre 2017*]. In the semantics, syntactic ordinals will be interpreted using actual ordinals.

Definition 6.7.24. We denote by κ the sort of *syntactic ordinals*. From now on, we will consider that it is contained in our set of atomic sorts $\mathcal{S}_0$ and thus $\kappa \in \mathcal{S}$.

Definition 6.7.25. The set of all the syntactic ordinals is generated from a set of ordinal variables $\mathcal{V}_\kappa = \{\theta, \eta, \zeta ...\}$ using the following BNF grammar.

$$(\mathcal{O}) \qquad \tau, \upsilon ::= \theta \mid \infty \mid \tau{+}1 \mid \varepsilon_{\theta<\tau}(t \in A) \qquad \theta \in \mathcal{V}_\kappa, t \in \Lambda^+, A \in \mathcal{F}$$

Our syntactic ordinals are formed using variable, the constant ∞, the successor symbol and choice operators of the form $\varepsilon_{\theta<\tau}(t \in A)$. Note that we need to extend the system with the following three sorting rules.

$$\frac{}{\Sigma \vdash \infty : \kappa} \qquad \frac{\Sigma \vdash \tau : \kappa}{\Sigma \vdash \tau{+}1 : \kappa} \qquad \frac{\Sigma \vdash \tau : \kappa \quad \Sigma, \theta : \kappa \vdash t : \tau \quad \Sigma, \theta : \kappa \vdash A : o}{\Sigma \vdash \varepsilon_{\theta<\tau}(t \in A) : \kappa}$$

In the model, syntactic ordinals will be interpreted using actual ordinals, as is done in [*Lepigre 2017*]. In particular, ∞ will be interpreted by an ordinal that is large enough to ensure the convergence of all fixpoints in the semantics of our types.

Definition 6.7.26. Given a syntactic ordinal τ, we denote $[\![\tau]\!]$ the ordinal corresponding to its interpretation. In particular, we define $[\![\infty]\!]$ to be the set of all the ordinals smaller or equal to the cardinal of the set $\mathcal{P}([\![o]\!])$. The interpretation of the other syntactic ordinals is defined inductively as follows.

$$[\![o]\!] = o \qquad [\![\varepsilon_{\theta<\tau}(t \in A)]\!] = o \mid [\![t[\theta := o]]\!] \in [\![A[\theta := o]]\!]^{\perp\perp}$$
$$[\![\tau+1]\!] = [\![\tau]\!]+1 \qquad [\![\varepsilon_{\theta<\tau}(t \in A)]\!] = 0 \text{ otherwise}$$

We will use the notation $[\![\mathcal{O}]\!]$ for the ordinal $[\![\infty]\!] + \omega$, which is the set of all the ordinals that can be represented in our syntax.

Remark 6.7.27. As in the previous chapter, the multiple extensions of the language that are given in this chapter are not independent. As our syntactic elements are all defined mutually inductively, every single modification leads to global changes. In particular, the definition of types (and thus of raw terms) is affected.

We will now extend the syntax of our types with two constructors denoting the least or greatest fixpoint of a parametric type. Intuitively, these types will allow us to construct inductive and coinductive types. As we are using sized types, our fixpoint constructors will have the peculiarity of carrying an ordinal. In particular, a fixpoint carrying the ordinal ∞ will correspond to usual (not sized) types.

Definition 6.7.28. We extend the syntax of formulas $\mathcal{F}$ with a least fixpoint constructor $\mu_\tau X.A$ and a greatest fixpoint constructor $\nu_\tau X.A$. Both of these constructors carry an ordinal. Note that we need to extend our system with the following sorting rules.

$$\frac{\Sigma, X : o \vdash A : o \quad \Sigma \vdash \tau : \kappa}{\Sigma \vdash \mu_\tau X.A : o} \qquad \frac{\Sigma, X : o \vdash A : o \quad \Sigma \vdash \tau : \kappa}{\Sigma \vdash \nu_\tau X.A : o}$$

Moreover, we will implicitly assume that in these constructions, the variable X only appears positively in A (i.e., it is in covariant position).

Definition 6.7.29. In the semantics, sized inductive and coinductive types are interpreted in the usual way, as pre-fixpoints and post-fixpoints respectively.

$$[\![\mu_\tau X.A]\!] = \bigcup_{o<[\![\tau]\!]} [\![X \mapsto A]\!]^o(\{\square\}) \qquad [\![\nu_\tau X.A]\!] = \bigcap_{o<[\![\tau]\!]} [\![X \mapsto A]\!]^o(\Lambda_\iota)$$

Before giving the new subtyping rule for handling inductive and coinductive type, we need to extend the context of our judgments with a so-called positivity context.

Definition 6.7.30. A *positivity context* is a list of syntactic ordinals assumed to be positive. For convenience, we will represent such contexts using comma-separated lists of syntactic ordinals generated by the following BNF grammar.

$$\gamma ::= \emptyset \mid \gamma, \tau \qquad \tau \in \mathcal{O}$$

$$\frac{i \leqslant 0}{\gamma \vdash \tau \leqslant_i \tau} \qquad \frac{\gamma \vdash \tau \leqslant_{i+1} \upsilon}{\gamma \vdash \tau+1 \leqslant_i \upsilon} \qquad \frac{\gamma \vdash \tau \leqslant_{i-1} \upsilon}{\gamma \vdash \tau \leqslant_i \upsilon+1}$$

$$\frac{\gamma, \tau \vdash \tau \leqslant_{i-1} \upsilon}{\gamma, \tau \vdash \varepsilon_{\theta<\tau}(t \in A) \leqslant_i \upsilon} \qquad \frac{\gamma \vdash \tau \leqslant_i \upsilon}{\gamma \vdash \varepsilon_{\theta<\tau}(t \in A) \leqslant_i \upsilon}$$

Figure 6.6 – Rules of ordinal ordering.

$$\frac{\gamma ; \Xi \vdash t \in A \subset B[X := \mu_\infty X.B]}{\gamma ; \Xi \vdash t \in A \subset \mu_\infty X.B}\mu_{r,\infty} \qquad \frac{\gamma ; \Xi \vdash t \in A[X := \nu_\infty X.A] \subset B}{\gamma ; \Xi \vdash t \in \nu_\infty X.A \subset B}\nu_{l,\infty}$$

$$\frac{\gamma ; \Xi \vdash t \in A \subset B[X := \mu_\upsilon X.B] \quad \gamma \vdash \upsilon < \tau}{\gamma ; \Xi \vdash t \in A \subset \mu_\tau X.B}\mu_r$$

$$\frac{\gamma ; \Xi \vdash t \in A[X := \nu_\upsilon X.A] \subset B \quad \gamma \vdash \upsilon < \tau}{\gamma ; \Xi \vdash t \in \nu_\tau X.A \subset B}\nu_l$$

$$\frac{\gamma, \tau ; \Xi \vdash t \in A[X := \mu_{\varepsilon_{\theta<\tau}(t \in A[X:=\mu_\theta X.A])} X.A] \subset B \quad \Xi \vdash \nu \equiv t}{\gamma ; \Xi \vdash t \in \mu_\tau X.A \subset B}\mu_l$$

$$\frac{\gamma, \tau ; \Xi \vdash t \in A \subset B[X := \nu_{\varepsilon_{\theta<\tau}(t \notin B[X:=\nu_\theta X.B])} X.B] \quad \Xi \vdash \nu \equiv t}{\gamma ; \Xi \vdash t \in A \subset \nu_\tau X.B}\nu_r$$

Figure 6.7 – Local subtyping rules for inductive and coinductive types.

We will say that a positivity context γ is valid if the interpretation of every syntactic ordinal τ of γ is strictly positive (i.e., $[\![\tau]\!] > 0$).

In the system, positivity contexts will be necessary for the derivation of ordering judgments on syntactic ordinals. They will be used to make sure that fixpoints can be unfolded in subtyping judgments.

Definition 6.7.31. An *ordering judgment* is a tuple of a positivity context γ, syntactic ordinals τ and υ and an integer $i \in \mathbb{Z}$ denoted $\gamma \vdash \tau \leqslant_i \upsilon$. An ordering judgment is said to be valid if it can be derived using the deduction rules given in Figure 6.6. Note that we will write $\gamma \vdash \tau < \upsilon$ when $i = 1$.

Definition 6.7.32. We extend our system with the six subtyping rules given in Figure 6.7. Note that all the other rules need to be modified to contain a positivity context. However, there is no difficulty in doing so as they only need to transmit this context.

Intuitively, the $(\mu_{r,\infty})$ rule allows the unrolling of a least fixpoint on the right side of the subtyping relation. This rule can only be applied when the limit of the fixpoint has been reached, and it is always the case with the ordinal ∞. When the ordinal is too small to make the fixpoint converge, the (μ_r) rule can be applied. However, it requires finding an ordinal υ that is strictly smaller than the ordinal τ carried by the fixpoint. In particular, this ensures that τ was not equal to 0, and thus that the fixpoint can be unrolled. When a least fixpoint appears on the left side of a pointed subtyping relation, the (μ_l) rule can be applied. In this case, a choice operator is used to obtain some ordinal such that the left part of the judgment is satisfied. If no such ordinal exist, then 0 is chosen and thus the premise of the rule is immediate. The three rules for handling the greatest fixpoint constructor are dual to those for the least fixpoint constructor.

Handling recursion requires providing a typing rule for our fixpoint term constructor. However, this is not enough because in practice we will need (part of) our positivity contexts to be communicated between part of the typing trees to be able to establish that an infinite typing proof is well-founded. As a consequence, we also introduce two new type connectives that will be used for this purpose. In particular, they will allow us to give stronger rules for the typing of λ-abstractions and pattern matchings.

Definition 6.7.33. We extend the syntax of formulas $\mathcal{F}$ with two new type constructors $A \restriction \gamma$ and $\gamma \hookrightarrow A$ called positivity restriction and positivity implication. The former is a variant of our restriction constructor for equivalences and the latter denotes an implication (with no algorithmic contents) depending on a positivity context. Note that we need to extend the system with the following sorting rules.

$$\frac{\Sigma \vdash A : o \quad (\Sigma \vdash \tau : \kappa)_{\tau \in \gamma}}{\Sigma \vdash A \restriction \gamma : o} \qquad \frac{\Sigma \vdash A : o \quad (\Sigma \vdash \tau : \kappa)_{\tau \in \gamma}}{\Sigma \vdash \gamma \hookrightarrow A : o}$$

Definition 6.7.34. In the semantics, positivity restriction and positivity implication are interpreted as follows.

$$[\![A \restriction \gamma]\!] = [\![A]\!] \text{ when } \forall \tau \in \gamma, [\![\tau]\!] > 0 \qquad [\![A \restriction \gamma]\!] = \{\Box\} \text{ otherwise}$$
$$[\![\gamma \hookrightarrow A]\!] = [\![A]\!] \text{ when } \forall \tau \in \gamma, [\![\tau]\!] > 0 \qquad [\![\gamma \hookrightarrow A]\!] = \Lambda_\iota \text{ otherwise}$$

Remark 6.7.35. It would be perfectly possible to add an implication connective similar to $\gamma \hookrightarrow A$ but depending on an equivalence instead of a list of ordinals. It is in fact included in the implementation of the system.

$$\frac{\gamma, \gamma_0 ; \Xi \vdash t \in A \subset B \quad \Xi \vdash v \equiv t}{\gamma ; \Xi \vdash t \in A \restriction \gamma_0 \subset B}\restriction_{l,\kappa} \qquad \frac{\gamma ; \Xi \vdash t \in A \subset B \quad \gamma_0 \subset \gamma}{\gamma ; \Xi \vdash t \in A \subset B \restriction \gamma_0}\restriction_{r,\kappa}$$

$$\frac{\gamma ; \Xi \vdash t \in A \subset B \quad \gamma_0 \subseteq \gamma}{\gamma ; \Xi \vdash t \in \gamma_0 \hookrightarrow A \subset B}\hookrightarrow_{l,\kappa} \qquad \frac{\gamma, \gamma_0 ; \Xi \vdash t \in A \subset B}{\gamma ; \Xi \vdash t \in A \subset \gamma_0 \hookrightarrow B}\hookrightarrow_{r,\kappa}$$

$$\frac{\gamma ; \Xi \vdash \lambda x.t \in \gamma_0 \hookrightarrow (A \Rightarrow B) \subset C \quad \gamma, \gamma_0 ; \Xi \vdash t[x := \varepsilon_{x \in A}(t \notin B)] : B}{\gamma ; \Xi \vdash \lambda x.t : C}\Rightarrow_{i,\kappa}$$

$$\frac{\gamma ; \Xi \vdash v : A \quad \gamma ; \Xi \vdash v \in A \subseteq [(C_i : A_i)_{i \in I}] \restriction \gamma_0 \quad \left(\gamma, \gamma_0 ; \Xi \vdash t_i[x_i := \varepsilon_{x_i \in A_i \restriction C[x_i] \equiv v}(t_i \notin C)] : C\right)_{i \in I}}{\gamma ; \Xi \vdash [v \mid (C_i[x_i] \to t_i)_{i \in I}] : C}+_{e,\kappa}$$

$$\frac{\gamma, \gamma_0 ; \Xi \vdash t[x := \lambda x.Y_{\lambda r.t,x}] : A \Rightarrow B \quad \gamma ; \Xi \vdash \lambda x.Y_{\lambda r.t,x} \in \gamma_0 \hookrightarrow (A \Rightarrow B) \subset C}{\gamma ; \Xi \vdash \lambda x.Y_{\lambda r.t,x} : C}Y$$

Figure 6.8 – Typing and subtyping rules for the handling of recursion.

We will now extend our system with a last set of typing and local subtyping rules for handling recursion. We will then prove that all the rules introduced in this section are adequate.

Definition 6.7.36. We extend the system with the seven rules of Figure 6.8.

We can now put everything together and prove yet another adequacy lemma. However, we first need to give the interpretation of ordinal ordering judgments.

Lemma 6.7.37. If the judgment $\gamma \vdash \tau_1 \leqslant_i \tau_2$ is derivable and for every syntactic ordinal $\tau \in \gamma$, we have $[\![\tau]\!] > 0$, then the following holds.
- If $i \geqslant 0$ then $[\![\tau_1]\!] + i \leqslant [\![\tau_2]\!]$,
- if $i \leqslant 0$ then $[\![\tau_1]\!] \leqslant [\![\tau_2]\!] + i$.

Here, we use the notation $o + i$ for the i-th successor of the ordinal o. Note that in particular, $\gamma \vdash \tau_1 < \tau_2$ is derivable then we have $[\![\tau_1]\!] < [\![\tau_2]\!]$.

Proof. A complete proof is given in [*Lepigre 2017, Lemma 1.10*]. □

Theorem 6.7.38. Let γ be a positivity context, Ξ be an equational context, $A, B \in \mathcal{F}$ be closed types and $t \in \Lambda^+$ be a closed raw term. If for every syntactic ordinal $\tau \in \gamma$ we have $[\![\tau]\!] > 0$ and if for every $(t_1, t_2) \in \Xi$ the equivalence $[\![t_1]\!] \equiv [\![t_2]\!]$ holds then we have the following.
- If $\gamma ; \Xi \vdash t : A$ is valid then we have $t \Vdash A$. Moreover, if t is a value $[\![t]\!] \neq \square$.

- If $\gamma\,;\,\Xi \vdash \pi : \neg A$ is valid then we have $\pi \Vdash A$.
- If $\gamma\,;\,\Xi \vdash t \in A \subset B$ is valid then we have $t \Vdash A \subset B$.

Proof. The proof proceeds as for Theorem 6.4.17 with the exception of the management of the positivity context. For all the rules that were given in previous sections of the current chapter, the adaptation is immediate as the positivity context is only transmitted. As a consequence, we only consider the rules that were introduced in this section.

- In the case of the $(\mu_{r,\infty})$ rule, we need to show that $t \Vdash A \subset \mu_\infty X.B$. By induction hypothesis, we know that $t \Vdash A \subset B[X := \mu_\infty X.B]$ so we can conclude immediately as we know that $[\![\mu_\infty X.B]\!] = [\![B[X := \mu_\infty X.B]]\!]$ as the fixpoint has been reached.

$$\frac{\gamma\,;\,\Xi \vdash t \in A \subset B[X := \mu_\infty X.B]}{\gamma\,;\,\Xi \vdash t \in A \subset \mu_\infty X.B}\,\mu_{r,\infty}$$

- In the case of the $(\nu_{r,\infty})$ rule, the proof is dual to the $(\mu_{r,\infty})$ case.

$$\frac{\gamma\,;\,\Xi \vdash t \in A[X := \nu_\infty X.A] \subset B}{\gamma\,;\,\Xi \vdash t \in \nu_\infty X.A \subset B}\,\nu_{l,\infty}$$

- In the case of the (μ_r) rule, we need to show that $t \Vdash A \subset \mu_\tau X.B$. According to Lemma 6.7.37 we have $[\![\upsilon]\!] < [\![\tau]\!]$ and thus we have $[\![B[X := \mu_\upsilon X.B]]\!] = [\![\mu_{\upsilon+1} X.B]\!] \subseteq [\![\mu_\tau X.B]\!]$. We can thus conclude using Lemma 4.5.32 and the induction hypothesis. It is important that X only appears positively in B to obtain the inclusion directly.

$$\frac{\gamma\,;\,\Xi \vdash t \in A \subset B[X := \mu_\upsilon X.B] \qquad \gamma \vdash \upsilon < \tau}{\gamma\,;\,\Xi \vdash t \in A \subset \mu_\tau X.B}\,\mu_r$$

- In the case of the (ν_l) rule, the proof is dual to the (μ_r) case.

$$\frac{\gamma\,;\,\Xi \vdash t \in A[X := \nu_\upsilon X.A] \subset B \qquad \gamma \vdash \upsilon < \tau}{\gamma\,;\,\Xi \vdash t \in \nu_\tau X.A \subset B}\,\nu_l$$

- In the case of the (μ_l) rule, we need to show that $t \Vdash \mu_\tau X.A \subset B$. If we have $[\![\tau]\!] = 0$ then we can conclude immediately as in this case $[\![\mu_\tau X.A]\!] = \{\square\} \subseteq [\![B]\!]$ and thus $[\![\mu_\tau X.A]\!]^{\perp\perp} \subseteq [\![B]\!]^{\perp\perp}$ by Lemma 4.5.32. We can thus suppose that $[\![\tau]\!] > 0$ in the following so that we can use the induction hypothesis. Let now suppose that we have $[\![t]\!] \in [\![\mu_\tau X.A]\!]^{\perp\perp}$ and show $[\![t]\!] \in [\![B]\!]^{\perp\perp}$. By our second hypothesis, we have a value v such that $[\![t]\!] \equiv [\![v]\!]$. As a consequence we can work at the value level thanks to Theorem 5.4.15. We can thus suppose that $[\![v]\!] \in [\![\mu_\tau X.A]\!]$ and show $[\![v]\!] \in [\![B]\!]$. By definition of $[\![\mu_\tau X.A]\!]$ we know that there is an ordinal $o < [\![\tau]\!]$ such that $v \in [\![X \mapsto A]\!]^o(\{\square\})$. As a consequence, the choice operator $\varepsilon_{\theta<\tau}(v \in A[X := \mu_\theta X.A]) = \varepsilon_{\theta<\tau}(t \in A[X := \mu_\theta X.A])$ is well-defined and thus we have $[\![v]\!] \in \left[\!\!\left[A[X := \mu_{\varepsilon_{\theta<\tau}(t \in A[X := \mu_\theta X.A])} X.A]\right]\!\!\right]$. We can hence apply the induction hypothesis (again using Theorem 5.4.15) to get $[\![v]\!] \in [\![B]\!]$.

$$\frac{\gamma, \tau ; \Xi \vdash t \in A[X \coloneqq \mu_{\varepsilon_{\theta<\tau}(t \in A[X:=\mu_\theta X.A])} X.A] \subset B \quad \Xi \vdash \nu \equiv t}{\gamma ; \Xi \vdash t \in \mu_\tau X.A \subset B}\mu_l$$

- In the case of the (ν_r) rule, the proof is dual to the (μ_l) case.

$$\frac{\gamma, \tau ; \Xi \vdash t \in A \subset B[X \coloneqq \nu_{\varepsilon_{\theta<\tau}(t \notin B[X:=\nu_\theta X.B])} X.B] \quad \Xi \vdash \nu \equiv t}{\gamma ; \Xi \vdash t \in A \subset \nu_\tau X.B}\nu_r$$

- In the case of the $(\restriction_{l,\kappa})$ rule, the proof is similar as for $(\restriction_l)$ Theorem 6.4.17.

$$\frac{\gamma, \gamma_0 ; \Xi \vdash t \in A \subset B \quad \Xi \vdash \nu \equiv t}{\gamma ; \Xi \vdash t \in A \restriction \gamma_0 \subset B}\restriction_{l,\kappa}$$

- In the case of the $(\restriction_{r,\kappa})$ rule, the proof is similar as for $(\restriction_r)$ in Theorem 6.4.17.

$$\frac{\gamma ; \Xi \vdash t \in A \subset B \quad \gamma_0 \subset \gamma}{\gamma ; \Xi \vdash t \in A \subset B \restriction \gamma_0}\restriction_{r,\kappa}$$

- In the case of the $(\hookrightarrow_{l,\kappa})$ rule, we need to show $t \Vdash \gamma_0 \hookrightarrow A \subset B$. Thanks to our second premise we know that all the ordinals in γ_0 are positive. As a consequence we know that $[\![\gamma_0 \hookrightarrow A]\!] = [\![A]\!]$. We can thus conclude using Theorem 4.5.32 and the induction hypothesis.

$$\frac{\gamma ; \Xi \vdash t \in A \subset B \quad \gamma_0 \subseteq \gamma}{\gamma ; \Xi \vdash t \in \gamma_0 \hookrightarrow A \subset B}\hookrightarrow_{l,\kappa}$$

- In the case of the $(\hookrightarrow_{r,\kappa})$ rule, we need to show $t \Vdash A \subset \gamma_0 \hookrightarrow B$. We thus suppose that $[\![t]\!] \in [\![A]\!]^{\perp\perp}$ and show $[\![t]\!] \in [\![\gamma_0 \hookrightarrow B]\!]^{\perp\perp}$. If the ordinals of γ_0 are not all positive then the proof is immediate as $[\![\gamma_0 \hookrightarrow B]\!] = \Lambda^*_\iota$. If all the ordinals are positive then we can immediately conclude using the induction hypothesis.

$$\frac{\gamma, \gamma_0 ; \Xi \vdash t \in A \subset B}{\gamma ; \Xi \vdash t \in A \subset \gamma_0 \hookrightarrow B}\hookrightarrow_{r,\kappa}$$

- In the case of the $(\Rightarrow_{i,\kappa})$ rule, we need to show $\lambda x.t \Vdash C$. Using the first induction hypothesis it is enough to show $\lambda x.t \Vdash \gamma_0 \hookrightarrow (A \Rightarrow B)$. Let us first assume that some ordinal of γ_0 is equal to 0. In this case $[\![\gamma_0 \hookrightarrow (A \Rightarrow B)]\!] = \Lambda^*_\iota$ and thus we have $\lambda x.[\![t]\!] \in [\![\gamma_0 \hookrightarrow (A \Rightarrow B)]\!]$ so we can conclude with Lemma 4.5.29. Now, if γ_0 only contains positive ordinals then we have $[\![\gamma_0 \hookrightarrow (A \Rightarrow B)]\!] = [\![A \Rightarrow B]\!]$ and we can use the second induction to conclude the proof as in the $(\Rightarrow_i)$ case.

$$\frac{\gamma ; \Xi \vdash \lambda x.t \in \gamma_0 \hookrightarrow (A \Rightarrow B) \subset C \quad \gamma, \gamma_0 ; \Xi \vdash t[x \coloneqq \varepsilon_{x \in A}(t \notin B)] : B}{\gamma ; \Xi \vdash \lambda x.t : C}\Rightarrow_{i,\kappa}$$

- In the case of the $(+_{e,\kappa})$ rule, we need to show $[\nu \,|\, (C_i[x_i] \to t_i)_{i \in I}] \Vdash C$. By our first induction hypothesis we have $[\![\nu]\!] \in [\![A]\!]^{\perp\perp}$ and $[\![\nu]\!] \neq \Box$. Thanks to our second induction hypothesis combined with Theorem 5.4.15 we obtain $[\![\nu]\!] \in \left[\!\!\left[[(C_i : A_i)_{i \in I}] \restriction \gamma_0\right]\!\!\right]$ and thus we know that the ordinals of γ_0 are all positive as otherwise this would imply

$[\![\nu]\!] \neq \square$. As a consequence, we have $\left[\!\left[[(C_i : A_i)_{i\in I}] \restriction \gamma_0\right]\!\right] = \left[\!\left[[(C_i : A_i)_{i\in I}]\right]\!\right]$ and we can finish the proof as in the case of $(+_e)$ rule since we can use the remaining induction hypotheses.

$$\frac{\gamma\,;\,\Xi \vdash \nu : A \qquad \gamma\,;\,\Xi \vdash \nu \in A \subseteq [(C_i : A_i)_{i\in I}] \restriction \gamma_0 \qquad \left(\gamma, \gamma_0\,;\,\Xi \vdash t_i[x_i := \varepsilon_{x_i \in A_i \restriction C[x_i] \equiv \nu}(t_i \notin C)] : C\right)_{i\in I}}{\gamma\,;\,\Xi \vdash [\nu \mid (C_i[x_i] \to t_i)_{i\in I}] : C}\,{}_{+_{e,\kappa}}$$

– In the case of the (Y) rule, we need to show $\lambda x.Y_{\lambda r.t,x} \Vdash C$. Using the second induction hypothesis it is enough to show $\lambda x.Y_{\lambda r.t,x} \Vdash \gamma_0 \hookrightarrow (A \Rightarrow B)$. As for the $(\Rightarrow_{i,\kappa})$ case, if γ_0 contains some ordinal that is equal to 0 then we can conclude immediately. We can thus assume that all the ordinals of γ_0 are positive, which means that we can use the right induction hypothesis and that we have $[\![\gamma_0 \hookrightarrow (A \Rightarrow B)]\!] = [\![A \Rightarrow B]\!]$. We need to prove $\lambda x.Y_{\lambda r.[\![t]\!],x} \in [\![A \Rightarrow B]\!]^{\perp\perp}$ for which it is enough to show $\lambda x.Y_{\lambda r.[\![t]\!],x} \in [\![A \Rightarrow B]\!]$ according to Theorem 5.4.15. By definition, we need to take a value $\nu \in [\![A]\!] \setminus \{\square\}$ and show $Y_{\lambda r.[\![t]\!],\nu} \in [\![B]\!]^{\perp\perp}$. We thus take $\pi \in [\![B]\!]^{\perp}$ and we prove $Y_{\lambda r.[\![t]\!],\nu} * \pi \in \perp\!\!\!\perp$. As $\perp\!\!\!\perp$ is $(\twoheadrightarrow)$-saturated and we have $Y_{\lambda r.[\![t]\!],\nu} * \pi \twoheadrightarrow^* [\![t[r := \lambda x.Y_{\lambda r.t,x}]]\!] * \nu \cdot \pi$ we only need to show $[\![t[r := \lambda x.Y_{\lambda r.t,x}]]\!] * \nu \cdot \pi \in \perp\!\!\!\perp$. According to our first induction hypothesis it only remains to show that we have $\nu \cdot \pi \in [\![A \Rightarrow B]\!]^{\perp}$. This follows easily from the fact that $\nu \in [\![A]\!] \setminus \{\square\}$ and that $\pi \in [\![B]\!]^{\perp}$.

$$\frac{\gamma, \gamma_0\,;\,\Xi \vdash t[x := \lambda x.Y_{\lambda r.t,x}] : A \Rightarrow B \qquad \gamma\,;\,\Xi \vdash \lambda x.Y_{\lambda r.t,x} \in \gamma_0 \hookrightarrow (A \Rightarrow B) \subset C}{\gamma\,;\,\Xi \vdash \lambda x.Y_{\lambda r.t,x} : C}\,{}_{Y}$$

□

Even with all the new rules introduced in this section, the system is still not quite ready to be usable. In fact, a notion of circular proofs and an associated notion of well-foundedness needs to be introduced in the system. It is possible to apply the framework defined in [*Lepigre 2017, Section 3*] to obtain circular proofs because the syntax used for ordinals is exactly the same as ours, and the system presented in the paper requires a very similar construction. After obtaining a notion of circular proofs, it would be possible to extend Theorem 6.7.38 so that it is proved by ordinal induction on the circular proofs, provided that they have been shown well-founded. We do not go into the details here for lack of time as this is still a work in progress.

7 Implementation and examples

In this last chapter, we consider examples of programs and proofs that can be written and manipulated using our prototype implementation. This restricted set of examples is not intended to give an exhaustive view of our language. Their only purpose is to demonstrate its expressiveness through a selected set of examples.

7.1 Concrete syntax and syntactic sugars

Although it is conveniently short, the abstract syntax that was used since Chapter 2 is not suitable for actual programming. Throughout this chapter, we will rely on the syntax used by the prototype implementation of the system. Some examples of programs written with this concrete syntax were already considered in Chapter 1. We will now give some elements of the translation from the concrete syntax to the abstract syntax.

Remark 7.1.1. Although we will not give the full details, it would be possible to give a precise definition of the translation of terms and types between the two syntaxes.

At the top level of the concrete syntax, there are three different ways of defining meta-variable: the definition of an expression of a given sort, the definition of a type and the definition of a value. The definition of an expression simply amounts to giving a name to some higher-order term of our language (i.e., to an element of $\mathcal{T}$ of an arbitrary sort). Examples of such definitions are given below.

```
def delta : ι           = fun x { x x }
def rapp⟨t:τ, u:τ⟩ : τ = u t
def omega : τ           = rapp⟨delta, delta⟩
def neg⟨a:o⟩ : o        = a ⇒ ∀x:o, x
```

Note that higher-order definitions can have arguments, which are given in angle brackets. Sort annotations are given for the arguments, as well as for the global (fully applied) expression. For example, the expression `rapp` (reversed application) given above has the sort $\tau \to \tau \to \tau$, and to be used as an element of sort τ it needs to be provided with two arguments of sort τ as in the definition of `omega`. Of course, any term of sort ι also has sort τ and the sort of expressions can be inferred most of the time (i.e., it does not always have to be provided by the user).

As it is very common to define types (i.e., expressions of sort o), a specific syntax is provided to do so. However, it is completely equivalent to using a standard definition as it is only syntactic sugar. As a consequence, the following two definitions of the type of booleans are equivalent.

```
type boolean = [True; False]
def boolean : o = [True; False]
```

Type definitions can also have arguments, and the `rec` or `corec` keyword can be used to make a type definition inductive or coinductive. For example, the type of lists and the type of streams can be defined as follows (equivalent higher-order definitions are also given).

```
type rec list⟨a:o⟩ = [Nil; Cons of {hd : a; tl : list}]
type corec stream⟨a:o⟩ = {} ⇒ {hd : a; tl : stream}

def list⟨a:o⟩ : o = μ list [Nil; Cons of {hd : a; tl : list}]
def stream⟨a:o⟩ : o = ν stream {} ⇒ {hd : a; tl : stream}
```

Note that the name of the constructors are uppercase identifiers, while lowercase identifiers are used everywhere else. In particular, constructor names and label names are not limited as in the abstract syntax.

Finally, values can be defined and type-checked using the `val` keyword. As for types, the `rec` keyword can be used and it makes the definition recursive. For example, we can define the following functions on lists, using the built-in type `bool` and the `option⟨a⟩` type from the standard library.

```
val is_empty : ∀a, list⟨a⟩ ⇒ bool =
  fun l {
    case l {
      Nil     → true
      Cons[_] → false
    }
  }
```

```
include lib.option
// type option⟨a⟩ = [None; Some of a]

val rec last : ∀a, list⟨a⟩ ⇒ option⟨a⟩ =
  fun l {
    case l {
      Nil     → None
      Cons[c] →
        case c.tl {
          Nil     → Some[c.hd]
          Cons[_] → last c.tl
        }
    }
  }
```

Note that the body of functions and the patterns of case analyses are wrapped in curly brackets (like in the Rust language). This will also be the case for conditionals, but they will only be introduced later.

Another important remark about our concrete syntax is that we allow terms which are not values everywhere, including in records and variants. This is in fact translated to the limited abstract syntax as discussed in Remark 2.5.34. For example, the term `Some[c.hd]` in the above example is translated to `(fun x { Some[x] }) c.hd` internally.

In the concrete syntax, λ-abstractions and μ-abstractions can be written using the syntax `fun x { t }` and `save k { t }` respectively. Note that they can take multiple arguments at once. For example, `fun x y { t }` is the same as `fun x { fun y { t } }`. A continuation (or stack) `k` can be restored using the syntax `restore k t`, which corresponds to named terms in the abstract syntax.

7.2 Encoding of strict product types

As discussed in the previous chapter, the most natural semantics for product types in presence of subtyping allows for extensible records. In other words, it is always possible to provide more fields than necessary. However, product types with a fixed set of fields often arise in practice. For this reason, we introduced so-called strict product types, which were used to state our type safety theorem (Theorem 6.6.22). In this section, we will show that it is possible to encode strict records into the system, without extending it with two different forms of product types.

Let us first consider the "unit" type, corresponding to the empty product. In our system, a first attempt at defining such a type would be to use an extensible record with no fields. However, as indicated by its name, this type contains many more elements that the empty record (written `{}` in our system).

```
type wrong_unit = { … }

// It is inhabited by the empty record.
val u : wrong_unit = {}

// And in fact by any record...
val u_aux : wrong_unit = {l = {}}
```

To avoid extending the system with strict record types, it is possible to use one of the following three encodings for the `unit` type. They rely on the membership type, optionally combined with existential quantification and restriction.

```
type unit1 = {} ∈ { … }
type unit2 = ∃x:ι, x ∈ ({ … } | x ≡ {})
type unit3 = ∃x:ι, (x ∈ { … }) | x ≡ {}
```

Note that these types are all equivalent semantically. In the implementation, we chose to use the definition `unit1` in place of the syntactic sugar `{}` (strict product type with no fields). We can thus use the following definition for a reasonable `unit` type.

```
type unit = {} ∈ { … }
// type unit = {}

// It is inhabited by the empty record.
val u : unit = {}

// But not by any other record.
// val fail : unit = {l = {}}
```

In fact, we can show that every value of `unit` is equivalent to the empty record `{}` with the following proof.

```
val true_unit : ∀x∈unit, x ≡ {} = fun x { {} }
```

The encoding of strict products is not limited to the unit type. A similar encoding can be used for any record type, and it is made accessible in the syntax using the strict product type notation. For instance, the type of pairs `pair⟨a,b⟩` can be encoded as follows.

```
type pair⟨a,b⟩ = ∃ x y:ι, {fst = x; snd = y} ∈ {fst : a; snd : b; … }
// type pair⟨a,b⟩ = {fst : a ; snd : b}
```

```
val couple : ∀ a b, a ⇒ b ⇒ pair⟨a,b⟩ =
  fun x y { {fst = x ; snd = y} }

val pi1 : ∀ a b, pair⟨a,b⟩ ⇒ a = fun p { p.fst }
val pi2 : ∀ a b, pair⟨a,b⟩ ⇒ b = fun p { p.snd }
```

As for the unit type, it is possible to show that the elements of a pair type are indeed records with exactly two fields `fst` and `snd`.

```
val true_pair : ∀a b, ∀p∈pair⟨a,b⟩, ∃x y:ι, p ≡ {fst = x ; snd = y} =
  fun p { {} }
```

7.3 Booleans and tautologies

After defining the (one element) `{}` (or `unit`) type in the previous section, we will now consider the (two elements) type of booleans. It can be encoded as usual using a polymorphic variant type as follows.

```
type boolean = [False of {}; True of {}]
// type boolean = [False; True]
```

Note that our variants need to have exactly one argument, which will here be `{}` on both the `True` and `False` constructors. However, if no argument type is specified, then `{}` is used implicitly.

In practice, booleans are often used together with conditional structures. There are several (non-equivalent) ways of defining them in our language. A first possibility is to define a condition function with three arguments as follows.

```
val cond_fun : ∀ a, boolean ⇒ a ⇒ a ⇒ a =
  fun c e1 e2 { case c { True → e1 | False → e2 } }
```

However, this function leads to both the expressions for the "then" and "else" branches to be evaluated, before a choice is made. This is not the semantics that is expected in practice, as it would be rather inefficient.

In our language, we can encode the usual conditional structures using a term macro `cond`, with three term arguments. This is made possible by the higher-order features of our language.

```
def cond⟨c:τ, e1:τ, e2:τ⟩ : τ =
  case c { True → e1 | False → e2 }
```

Although such a macro is not typed at the time of its definition (it is only given a sort), it is expanded and type-checked each time it is used. For instance, `cond` can be used to define the following alternative definition of `cond_fun`.

```
val cond_fun : ∀ a, boolean ⇒ a ⇒ a ⇒ a =
  fun c e1 e2 { cond⟨c, e1, e2⟩ }
```

In the following, we will prefer using the built-in `bool` type, rather that the `boolean` type defined above. Although they are completely equivalent, only the `bool` type will allow us to use the syntax for conditional structures. For example, the following definition of `cond_fun` can be given for `bool`.

```
val cond_fun : ∀ a, bool ⇒ a ⇒ a ⇒ a =
  fun c e1 e2 { if c { e1 } else { e2 } }
```

It is exactly equivalent to the one using the `boolean` type. The only difference lies in the name of the constructors, which are hidden to the user in the `bool` type. Only the conditional structures and the usual boolean constants can be accessed by the user.

Higher-order macros similar to `cond` can also be used to obtain conjunction and disjunction operators having the expected (lazy) semantics. This feature is in fact useful in many ways, we will see in a further section that it can even be used to encode a form of proof tactics.

```
def land⟨b1:τ, b2:τ⟩ =
  if b1 { b2 } else { false }

def lor⟨b1:τ, b2:τ⟩ =
  if b1 { true } else { b2 }
```

Before going further, we are going to define a set of usual boolean operators. We will then prove that they behave as expected in the system.

```
val not : bool ⇒ bool =
  fun a { if a { false } else { true } }

val or  : bool ⇒ bool ⇒ bool = fun a b { lor⟨a, b⟩ }
val and : bool ⇒ bool ⇒ bool = fun a b { land⟨a, b⟩ }
val imp : bool ⇒ bool ⇒ bool = fun a b { lor⟨not a, b⟩ }
val xor : bool ⇒ bool ⇒ bool = fun a b { if a { not b } else { b } }
val eq  : bool ⇒ bool ⇒ bool = fun a b { xor a (not b) }
```

We will now consider the law of the excluded middle (on booleans). It can be stated and proved as follows in the system.

```
val excl_mid : ∀x∈bool, or x (not x) ≡ true =
  fun b { if b { {} } else { {} } }
```

Remark 7.3.2. The law of the excluded middle on booleans is not to be confused with the term using control operators that was given in the introduction. Indeed, they live in completely different levels of the system.

As the type of booleans contain only finitely many elements, properties can always be proved by exhaustively listing the different cases. This is what was done above for `excl_mid`, but it was not too tedious as there were only two cases. This will not be the case anymore when considering properties with more parameters. For example, let us consider the reflexivity, commutativity of the `eq` function.

```
val eq_refl : ∀a∈bool, eq a a ≡ true =
  fun a { if a { {} } else { {} } }

val eq_comm : ∀a b∈bool, eq a b ≡ eq b a =
  fun a b {
    if a { if b { {} } else { {} } }
    else { if b { {} } else { {} } }
  }

val eq_asso : ∀a b c∈bool, eq (eq a b) c ≡ eq a (eq b c) =
  fun a b c {
    if a {
      if b { if c { {} } else { {} } }
      else { if c { {} } else { {} } }
    } else {
      if b { if c { {} } else { {} } }
      else { if c { {} } else { {} } }
    }
  }
```

To simplify the writing of such trivial proofs, it is possible to use term macros defined using higher-order definitions. Such macros can then be used to prove any tautology with a given number of arguments on booleans.

```
def auto1⟨a:τ⟩           : τ = if a { {} } else { {} }
def auto2⟨a:τ, b:τ⟩      : τ = if a { auto1⟨b⟩ } else { auto1⟨b⟩ }
def auto3⟨a:τ, b:τ, c:τ⟩ : τ = if a { auto2⟨b,c⟩ } else { auto2⟨b,c⟩ }

val eq_refl_auto : ∀a∈bool, eq a a ≡ true =
  fun a { auto1⟨a⟩ }

val eq_comm_auto : ∀a b∈bool, eq a b ≡ eq b a =
  fun a b { auto2⟨a,b⟩ }

val eq_asso_auto : ∀a b c∈bool, eq (eq a b) c ≡ eq a (eq b c) =
  fun a b c { auto3⟨a,b,c⟩ }
```

7.4 Unary natural numbers and totality

It is now time to consider a first example of data type with infinitely many elements: unary natural numbers. Their definition was already given in Chapter 1, together with some simple proofs. Let us start by recalling the definition of unary natural numbers, together with their addition and multiplication functions.

```
type rec nat = [Zero; Succ of nat]

val rec add : nat ⇒ nat ⇒ nat =
  fun n m {
    case n {
      Zero    → m
      Succ[k] → Succ[add k m]
    }
  }

val rec mul : nat ⇒ nat ⇒ nat =
  fun n m {
    case n {
      Zero    → Zero
      Succ[k] → add m (mul k m)
    }
  }
```

As `add` and `mul` are defined using recursion on their first argument, it is immediate to show `add Zero n ≡ n` and `mul Zero n ≡ Zero` for every `n`.

```
val add_zero_v : ∀v:ι, add Zero v ≡ v    = {}
val mul_zero_v : ∀v:ι, mul Zero v ≡ Zero = {}
```

Note that these properties are proved by quantifying over every possible value `v`. Whether it is a unary natural number or not is not important as `add Zero v` and `mul Zero v` can still be unfolded. In fact, the `add` function can always be given an arbitrary value as second argument as it is never considered in a case analysis.

Of course, it is possible to show similar properties by quantifying only on natural numbers using dependent functions.

```
val add_zero_n : ∀n∈nat, add Zero n ≡ n    = fun _ { {} }
val mul_zero_n : ∀n∈nat, mul Zero n ≡ Zero = fun _ { {} }
```

However, such definitions will never need to be used in practice since the more general `add_zero_v` and `mul_zero_v` have trivial proofs. This means that they can be obtained immediately by unfolding definitions, and thus it will never be necessary to invoke them. In particular, the user will never need to call `add_zero_n` or `mul_zero_n` to prove the corresponding equivalences.

Let us now consider the commutativity of the `add` function, which is a more interesting example. To obtain these properties, two lemmas are required. We need to prove that `add n Zero ≡ n` for every unary number `n`, and that `add n Succ[m] ≡ Succ[add n m]` for every unary numbers `n` and `m`. These two properties can be obtained easily using a case analysis and induction.

```
val rec add_n_zero : ∀n∈nat, add n Zero ≡ n =
  fun n {
    case n {
      Zero    → {}
      Succ[k] → let ih = add_n_zero k; {}
    }
  }

val rec add_succ : ∀n m∈nat, add n Succ[m] ≡ Succ[add n m] =
  fun n m {
    case n {
      Zero    → {}
      Succ[k] → let ih = add_succ k m; {}
    }
  }
```

A proof of the commutativity of add is then obtained in a similar way, but the two previously proved lemmas are used.

```
val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero    → let lem = add_n_zero m; {}
      Succ[k] → let ih  = add_comm k m;
                let lem = add_succ m k; {}
    }
  }
```

Note that in the system, a lemma is used by calling the corresponding function. Similarly, using an induction hypothesis corresponds to performing a recursive call. A proof can thus only be correct if it terminates on every possible input. Otherwise, obviously invalid proofs would be allowed by using a "non decreasing induction hypothesis".

With more complex examples, it is often required to establish the totality of functions. In other words, we need to show that the application of a function to any value (of the right type) produces a value. As an example, we will show that the add function is total.

```
val rec add_total : ∀n m∈nat, ∃v:ι, add n m ≡ v =
  fun n m {
    case n {
      Zero    → {}
      Succ[k] → let ih = add_total k m; {}
    }
  }
```

Using add_total twice, it is then possible to show that the addition function corresponds to an associative operation.

```
val rec add_asso : ∀n m p∈nat, add n (add m p) ≡ add (add n m) p =
  fun n m p {
    let tot_m_p = add_total m p;
    case n {
      Zero    → {}
      Succ[k] → let tot_k_m = add_total k m;
                let ih = add_asso k m p; {}
    }
  }
```

To conclude this section, we will prove the commutativity of the `mul` function. This result requires three intermediate lemmas. First, we need to show that `mul n Zero` $\equiv$ `Zero` for every unary number `n` and that `mul` is total. It is then necessary to show that `mul n Succ[m]` $\equiv$ `add (mul n m) n` for all `n` and `m`. These three properties are rather straight-forward to obtain, even if longer proofs are harder to read.

```
val rec mul_n_zero : ∀n∈nat, mul n Zero ≡ Zero =
  fun n {
    case n {
      Zero    → {}
      Succ[k] → let ih = mul_n_zero k; {}
    }
  }

val rec mul_total : ∀n m∈nat, ∃v:ι, mul n m ≡ v =
  fun n m {
    case n {
      Zero    → {}
      Succ[k] → let ih = mul_total k m;
                let lem = add_total m (mul k m); {}
    }
  }

val rec mul_succ : ∀n m∈nat, mul n Succ[m] ≡ add (mul n m) n =
  fun n m {
    case n {
      Zero    → {}
      Succ[k] →
        let lem = mul_succ k m;
        let tot = mul_total k m;
        let tot = add_total m (mul k m);
        let lem = add_succ (add m (mul k m)) k;
        let lem = add_asso m (mul k m) k;
        let tot = mul_total k Succ[m]; {}
    }
  }
```

The commutativity of `mul` then follows using yet another proof by induction, using each of the above lemmas.

```
val rec mul_comm : ∀n m∈nat, mul n m ≡ mul m n =
  fun n m {
    case n {
      Zero    → let lem = mul_n_zero m; {}
      Succ[k] → let ih  = mul_comm m k;
                let lem = mul_succ m k;
                let tot = mul_total k m;
                let lem = add_comm (mul k m) m; {}
    }
  }
```

One possible way for making a proof more readable is to give type annotations. They can be used to specify explicitly the equivalences that are being shown when using lemmas, but also for checking that some properties can be derived at a given point in the proof. We give below another version of `mul_comm` annotated in this way.

```
val rec mul_comm : ∀n m∈nat, mul n m ≡ mul m n =
  fun n m {
    case n {
      Zero    → let ded : mul Zero m ≡ Zero = {};
                let lem : mul m Zero ≡ Zero = mul_n_zero m; {}
      Succ[k] → let ded : mul Succ[k] m ≡ add m (mul k m) = {};
                let ih  : mul k m ≡ mul m k = mul_comm k m;
                let lem : mul m Succ[k] ≡ add (mul m k) m = mul_succ m k;
                let tot : (∃v:ι, mul k m ≡ v) = mul_total k m;
                let lem : add (mul k m) m ≡ add m (mul k m) =
                  add_comm (mul k m) m;
                {}
    }
  }
```

Here, we add type annotations on used lemmas to explicit the properties they prove. We also extend the proof with intermediate steps using local definitions. They allow us to check that some property can be deduced in the current context, while showing more reasoning steps. Note that the names chosen for the local definitions are never used, they can hence be seen as a form of comments (we could also write `_` to avoid giving explicit names). Only the equations that are transparently added to the context matter. Using this discipline, the proofs are not only more readable, but also easier to write.

Another, more satisfactory way of obtaining more readable proofs is to use again the higher-order layer of our system to define "tactics". We will here use a `t_deduce` tactic

to check that some equation holds in the current context, and a `t_show` tactic to prove a property using a given proof.

```
def t_deduce⟨f:o⟩ : τ = ({} : f)
def t_show⟨f:o, p:τ⟩ : τ = (p : f)
```

We can then modify our commutativity proof to obtain the following, which is a lot more readable than our original proof. Note that here, we use semicolons to put proof steps in sequence. It is encoded as usual using a dummy redex.

```
val rec mul_comm : ∀n m∈nat, mul n m ≡ mul m n =
  fun n m {
    case n {
      Zero    → t_deduce⟨mul Zero m ≡ Zero⟩;
                t_show⟨mul m Zero ≡ Zero, mul_n_zero m⟩
      Succ[k] → t_deduce⟨mul Succ[k] m ≡ add m (mul k m)⟩;
                t_show⟨mul k m ≡ mul m k, mul_comm k m⟩;
                t_show⟨mul m Succ[k] ≡ add (mul m k) m, mul_succ m k⟩;
                t_show⟨(∃v:ι, mul k m ≡ v), mul_total k m⟩;
                t_show⟨add (mul k m) m ≡ add m (mul k m), add_comm (mul k m) m⟩
    }
  }
```

We could even introduce syntactic sugar for our tactics into the parser of our language to obtain the following, very satisfactory proof.

```
val rec mul_comm : ∀n m∈nat, mul n m ≡ mul m n =
  fun n m {
    case n {
      Zero    → deduce mul Zero m ≡ Zero;
                show mul m Zero ≡ Zero using mul_n_zero m
      Succ[k] → deduce mul Succ[k] m ≡ add m (mul k m);
                show mul k m ≡ mul m k using mul_comm k m;
                show mul m Succ[k] ≡ add (mul m k) m
                  using mul_succ m k;
                show ∃v:ι, mul k m ≡ v using mul_total k m;
                show add (mul k m) m ≡ add m (mul k m)
                  using add_comm (mul k m) m
    }
  }
```

7.5 Lists and their vector subtypes

We will now consider the type of lists containing elements of a fixed type, given as a parameter. As usual, operations on lists will be polymorphic in this parameter.

```
type rec list⟨a:o⟩ = [Nil ; Cons of {hd : a ; tl : list}]
```

According to the above definition, a list is either empty (`Nil` constructor), or built using a smaller list and an element (`Cons` constructor). Note that the argument of the `Cons` constructor is formed using a product (or record) type with two elements. The label `hd` denotes the *head* of the list (i.e., its first element) and `tl` denotes its *tail*.

Remark 7.5.3. The type that is stored under the `tl` label is `list` and not `list⟨a⟩` due to the encoding of the "`type rec`" construct. It is formed using a higher-order function which body contains a fixpoint construction over a variable named `list`.

As for the natural number, it is possible to define the usual functions on lists, including `exists` or `rev_append` (see Chapter 1). Many more functions are given in the standard library of the prototype, together with its source code. Here, we will only focus on the `map` and `length` functions, which are given below.

```
val rec map : ∀a b:o, (a ⇒ b) ⇒ list⟨a⟩ ⇒ list⟨b⟩ =
  fun f l {
    case l {
      Nil     → Nil
      Cons[c] → Cons[{hd = f c.hd; tl = map f c.tl}]
    }
  }

val rec length : ∀a:o, list⟨a⟩ ⇒ nat =
  fun l {
    case l {
      Nil     → Zero
      Cons[c] → Succ[length c.tl]
    }
  }
```

The `map` function applies the function given as first argument to all the elements of the list given as second argument. The `length` function simply computes a unary natural number corresponding to the length of the list given as argument. We will now prove the totality of

these two functions because it will be needed later. Note that the totality of the map function can only be established ifs first argument is itself total.

```
// total⟨f,a⟩ means that f is total on the domain a.
def total⟨f:ι,a:o⟩ : o = ∀x∈a, ∃v:ι, f x ≡ v

val rec map_total : ∀a b:o, ∀f∈(a ⇒ b),
    total⟨f,a⟩ ⇒ ∀l∈list⟨a⟩, ∃v:ι, map f l ≡ v =
  fun fn ft ls {
    case ls {
      Nil     → {}
      Cons[c] →
        let lem = ft c.hd;
        let ih  = map_total fn ft c.tl; {}
    }
  }

val rec length_total : ∀a:o, ∀l∈list⟨a⟩, ∃v:ι, v ≡ length l =
  fun l {
    case l {
      Nil     → {}
      Cons[c] → let ind = length_total c.tl; {}
    }
  }
```

We will now show that two successive uses of map on a list, with two different functions, is the same as applying map once using the composition of the two functions. To do so, we will first need to show that the composition of two total functions is itself total.

```
val compose_total : ∀a b c:o, ∀f∈(a ⇒ b), ∀g∈(b ⇒ c),
    total⟨f,a⟩ ⇒ total⟨g,b⟩ ⇒ total⟨(fun x { g (f x) }), a⟩ =
  fun f g ftot gtot a {
    show ∃v:ι, f a ≡ v using ftot a;
    show ∃w:ι, g (f a) ≡ w using gtot (f a)
  }
```

We can then state and prove our lemma as follows, using a proof by induction together with our different totality results.

```
val map_map : ∀a b c:o, ∀f∈(a ⇒ b), ∀g∈(b ⇒ c), total⟨f,a⟩ ⇒ total⟨g,b⟩ ⇒
    ∀l∈list⟨a⟩, map g (map f l) ≡ map (fun x { g (f x) }) l =
  fun f g ftot gtot {
    fix fun map_map ls {
      case ls {
        Nil     → {}
        Cons[c] →
          let hd = c.hd; let tl = c.tl;
          let tgf = compose_total f g ftot gtot hd;
          let lem = ftot hd;
          let lem = map_total f ftot tl;
          let ind = map_map tl; {}
      }
    }
  }
```

Note that the proof by induction starts at the level of the "`fix`" keyword, which takes the fixpoint of the functions that immediately follows it. In fact, our "`val rec`" construct is exactly encoded in this way.

In our system, it is possible to encode the type of vectors (i.e., lists of a given length) using a restriction on the type of lists. In other words, vectors of length `s` will be defined as the type of all lists `l` such that `length l ≡ s`. The type of vectors will hence have two parameters. The former will give the type of the elements contained in the vectors and the latter will be the size parameter, in the form of a term.

```
type vec⟨a:o, s:τ⟩ = ∃l:ι, l∈(list⟨a⟩ | length l ≡ s)
```

We can then define a concatenation function `app` on vector. It produces a vector which length is the sum of the lengths of its two arguments. Note that the definition of `app` requires the use of `length_total`.

```
val rec app : ∀a:o, ∀m n:ι, vec⟨a, m⟩ ⇒ vec⟨a, n⟩ ⇒ vec⟨a, add m n⟩ =
  fun l1 l2 {
    case l1 {
      Nil     → l2
      Cons[c] → length_total c.tl;
                Cons[{hd = c.hd; tl = app c.tl l2}]
    }
  }
```

We can now define a ternary concatenation function on vectors as follows, using two calls to app. To be able to define this function, the totality of the add function is required.

```
val app3 : ∀a:o, ∀m n p:ι, vec⟨a,m⟩ ⇒ vec⟨a,n⟩ ⇒ vec⟨a,p⟩
                          ⇒ vec⟨a, add m (add n p)⟩ =
 fun l1 l2 l3 {
   let lem = add_total (length l2) (length l3);
   app l1 (app l2 l3)
 }
```

It is important to note that an element of vec⟨a,s⟩ can always be used as an element of list⟨a⟩, independently of s. In fact, vec⟨a,s⟩ is a subtype of list⟨a⟩.

```
val vec_to_list : ∀a:o, ∀s:τ, vec⟨a,s⟩ ⇒ list⟨a⟩ = fun x { x }
```

Note that we will never need to use the function vec_to_list to turn a vector into a list. A vector can be seen as a list directly, without relying on any form of coercion.

7.6 Sorted lists and insertion sort

We will now consider the insertion sort algorithm, and prove that it actually produces sorted lists. Before going further, we will start by defining a type ord⟨a⟩ that will be represent an ordering relation together with its properties.

```
type ord⟨a:o⟩ = ∃cmp:ι,
  { cmp : cmp ∈ (a ⇒ a ⇒ bool)
  ; tot : ∀x y∈a, ∃v:ι, cmp x y ≡ v
  ; dis : ∀x y∈a, or (cmp x y) (cmp y x) ≡ true }
```

If we ignore the leading existential, an ordering relation simply consist in a product (or record) type containing a comparison function, a proof of its totality and a proof that every element can be compared. The existential quantifier is only there to make the comparison function accessible in the types of the other fields.

Remark 7.6.4. An element of type ord⟨a⟩ does not really correspond to an ordering relation as transitivity is included. Although it could very well be given, it is not required for insertion sort.

We can then define our isort function in the usual way, using an intermediate function insert, inserting an element in a sorted list.

```
val rec insert : ∀a:o, ord⟨a⟩ ⇒ a ⇒ list⟨a⟩ ⇒ list⟨a⟩ =
  fun o x l {
    case l {
      Nil     → Cons[{hd = x; tl = Nil}]
      Cons[c] →
        let hd = c.hd; let tl = c.tl;
        if o.cmp x hd { Cons[{hd = x ; tl = l}] } else {
          let tl = insert o x tl;
          Cons[{hd = hd ; tl = tl}]
        }
    }
  }

val rec isort : ∀a:o, ord⟨a⟩ ⇒ list⟨a⟩ ⇒ list⟨a⟩ =
  fun o l {
    case l {
      Nil     → Nil
      Cons[c] → insert o c.hd (isort o c.tl)
    }
  }
```

Until the end of this section, our goal will be to show that for any ordering relation `o` and list `l`, the list `isort o l` is indeed sorted.

A first step in this direction consists in specifying what it means to be sorted. We hence define a boolean valued program taking as input an ordering relation and a list, and indicating whether the list is sorted.

```
val rec sorted : ∀a:o, ∀o∈ord⟨a⟩, ∀l∈list⟨a⟩, bool =
  fun o l {
    case l {
      Nil      → true
      Cons[c1] →
        let hd = c1.hd; let tl = c1.tl;
        case tl {
          Nil      → true
          Cons[c2] → let hd2 = c2.hd;
                     land⟨(o.cmp) hd hd2, sorted o tl⟩
        }
    }
  }
```

Most remarkably, we can even define the type of sorted lists using the restriction operator. Indeed, sorted lists are lists on which the `sorted` function returns `true`.

```
type slist⟨a:o,o:τ⟩ = ∃l:ι, l∈(list⟨a⟩ | sorted o l ≡ true)
```

At the end of this section we will be able to define another version of `isort` that can be given the type `∀a:o, ∀o∈ord⟨a⟩, list⟨a⟩ ⇒ slist⟨a,o⟩`.

To build our proof, we will first need to establish the totality of the `insert` and `isort` functions. This can be done straightforwardly with the following proofs.

```
val rec insert_total :
    ∀a:o, ∀o∈ord⟨a⟩, ∀x∈a, ∀l∈list⟨a⟩, ∃v:ι, insert o x l ≡ v =
  fun o x l {
    case l {
      Nil      → {}
      Cons[c1] →
        let hd = c1.hd; let tl = c1.tl;
        let lem = o.tot x hd;
        if o.cmp x hd {
          {}
        } else {
          let ih = insert_total o x tl; {}
        }
    }
  }

val rec isort_total :
    ∀a:o, ∀o∈ord⟨a⟩, ∀l∈list⟨a⟩, ∃v:ι, isort o l ≡ v =
  fun o l {
    case l {
      Nil     → {}
      Cons[c] →
        let ih  = isort_total o c.tl;
        let lem = insert_total o c.hd (isort o c.tl); {}
    }
  }
```

It is then necessary to show that inserting an element in a sorted list yields a sorted list. The proof of this lemma is not complicated either, but the case analysis is a bit tedious due to the lack of deep pattern matching.

```
val rec isorted : ∀a, ∀o∈ord⟨a⟩, ∀x∈a, ∀l∈slist⟨a,o⟩, sorted o (insert o x l) ≡ true =
  fun o x l {
    case l {
      Nil      → {}
      Cons[c1] →
        let lem = o.tot x c1.hd;
        if o.cmp x c1.hd { {} } else {
          let lem = o.tot c1.hd x;
          let lem = o.dis x c1.hd;
          case c1.tl {
            Nil      → {}
            Cons[c2] → let lem = insert_total o x c2.tl;
                       let lem = o.tot c1.hd c2.hd;
                       let lem = o.tot x c2.hd;
                       if o.cmp c1.hd c2.hd {
                         let lem = isorted o x c1.tl;
                         if o.cmp x c2.hd { {} } else { {} }
                       } else { ✂ }
          }
        }
    }
  }
```

Remark 7.6.5. The ✂ symbol (pronounced "scissors") can be used to mark a branch of the code as unreachable when there is an equational contradiction. Here, we must have `o.cmp c1.hd c2.hd ≡ true` as otherwise the hypothesis that `l` is sorted would be contradicted. Note that ✂ can be replaced by any term of the language as it will never be run.

We can then prove that `isort` produces lists that are sorted using a simple proof by induction as follows.

```
val rec isort_sorted : ∀a, ∀o∈ord⟨a⟩, ∀l∈list⟨a⟩, sorted o (isort o l) ≡ true =
  fun o l {
    case l {
      Nil     → {}
      Cons[c] → let lem = isort_total o c.tl;
                let ind = isort_sorted o c.tl;
                let lem = isorted o c.hd (isort o c.tl); {}
    }
  }
```

Finally, we can obtain a sorting function which return type indicates that the produced list is sorted. As for vectors, the type of sorted lists is a subtype of lists. As a consequence, a sorted list can be used as a list transparently.

```
val isort_full : ∀a, ∀o∈ord⟨a⟩, list⟨a⟩ ⇒ slist⟨a,o⟩ =
  fun o l {
    let tot = isort_total  o l;
    let lem = isort_sorted o l;
    isort o l
  }
```

7.7 Lookup function with an exception

We will consider an example of a program that relies on control operators as exceptions. We will take the common example of a lookup function on the type of lists. However, the type of our exception will carry a proof that the element that is looked for is in the list. To encode this property, we will rely on the `exists` function (given in Chapter 1).

```
val rec exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    case l {
      Nil     → false
      Cons[c] → if pred c.hd { true } else { exists pred c.tl }
    }
  }
```

Our lookup function (named `find`) can the be defined as follows, using a logical negation as the type of the exception.

```
val rec find : ∀a:o, ∀pred∈(a ⇒ bool), total⟨pred,a⟩ ⇒ ∀l∈list⟨a⟩,
    neg⟨exists pred l ≡ false⟩ ⇒ a =
  fun pred pred_tot l exc {
    case l {
      Nil     → exc {}
      Cons[c] →
        let lem = pred_tot c.hd;
        if pred c.hd { c.hd }
        else { find pred pred_tot c.tl exc }
    }
  }
```

Note that the exception exc can only be called if we are able to feed it with a proof that no element of the list satisfy the predicate pred. As a consequence, we are guaranteed that the exception cannot be raised if the list contains an element satisfying pred.

To conclude, let us give two examples of function defined using find. The first one will simply call find and wrap its result in the usual option⟨a⟩ type.

```
val find_opt :
    ∀a:o, ∀pred∈(a ⇒ bool), total⟨pred,a⟩ ⇒ list⟨a⟩ ⇒ option⟨a⟩ =
  fun pred pred_tot l {
    save a {
      some (find pred pred_tot l (fun _ { restore a none }))
    }
  }
```

The second one does the same, but it looks for an element satisfying the predicate into a list of lists.

```
val rec find_list :
    ∀a:o, ∀pred∈(a ⇒ bool), total⟨pred,a⟩ ⇒
    list⟨list⟨a⟩⟩ ⇒ option⟨a⟩ =
  fun pred pred_tot l {
    case l {
      Nil     → none
      Cons[c] →
        save a {
          let f = fun _ { restore a (find_list pred pred_tot c.tl) };
          some (find pred pred_tot c.hd f)
        }
    }
  }
```

Note that the recursive call of find_list is done inside the exception handler provided to find.

7.8 An infinite tape lemma on streams

To conclude this chapter, we will consider an example of program that can only be written in a classical setting (i.e., with control operators). We are going to define a function on streams of natural numbers, that extracts from its input a stream of odd numbers, or a stream of even numbers. First, we need to define odd and even numbers in our language.

```
val rec is_odd : nat ⇒ bool =
  fun n {
    case n {
      Zero    → false
      Succ[m] →
        case m {
          Zero    → true
          Succ[p] → is_odd p
        }
    }
  }

type odd  = {v∈nat | is_odd v ≡ true }
type even = {v∈nat | is_odd v ≡ false}
```

Note that here, we use a "set type" syntax similar to that of NuPrl [*Constable 1986*]. It is encoded as follows in our system.

```
type odd  = ∃v:ι, v∈(nat | is_odd v ≡ true )
type even = ∃v:ι, v∈(nat | is_odd v ≡ false)
```

Before going further, we need to establish that the `odd` function is total. It will be required when we decide whether a given number of the input stream is odd or even.

```
val rec odd_total : ∀n∈nat, ∃v:ι, is_odd n ≡ v =
  fun n {
    case n {
      Zero    → {}
      Succ[m] →
        case m {
          Zero    → {}
          Succ[p] → odd_total p
        }
    }
  }
```

We also need to define the type of streams, together a related type corresponding to streams with an explicit size annotation (or ordinal) `o`. Intuitively, this size annotation indicates the number of elements that are available in the stream.

```
type corec stream⟨a⟩ = {} ⇒ {hd : a; tl : stream}
type sized_stream⟨o,a⟩ = νo stream {} ⇒ {hd : a; tl : stream}
```

We can now defined the `itl_aux` function, which will be used to build our main infinite tape lemma function. Note that this function uses `abort`, which logically amounts to the *ex falso quodlibet* principle. Size annotations are also required on the type of `itl_aux`, for our type-checking algorithm to prove its termination.

```
val abort : ∀y, (∀x,x) ⇒ y = fun x { x }

val rec itl_aux :
    ∀a b, neg⟨sized_stream⟨a,even⟩⟩ ⇒
    neg⟨sized_stream⟨b,odd⟩⟩ ⇒ neg⟨stream⟨nat⟩⟩ =
  fun fe fo s {
    let hd = (s {}).hd;
    let tl = (s {}).tl;
    use odd_total hd;
    if is_odd hd {
      fo (fun _ {
        {hd = hd; tl = save o {
          abort (itl_aux fe (fun x { restore o x }) tl)}}
      })
    } else {
      fe (fun _ {
        {hd = hd; tl = save e {
          abort (itl_aux (fun x { restore e x }) fo tl)}}
      })
    }
  }
```

Intuitively, the `itl_aux` function looks at the head of its third argument (a stream of natural numbers). Depending on whether this number is odd or even, the function then calls one of its first two arguments, which corresponds to a partially constructed stream of even or odd numbers. The read number is then added to this stream, and a recursive call is made to continue the construction.

Remark 7.8.6. It is surprising that our prototype implementation is able to establish the termination of `itl_aux`. Indeed, at each call, an element is added to one of two streams. Moreover, this example does not satisfy the usually required semi-continuity condition (see, for example, [*Abel 2008*]).

Using `itl_aux`, it is then possible to define the `itl` function corresponding to our infinite tape lemma.

```
val itl : stream⟨nat⟩ ⇒ [InL of stream⟨even⟩; InR of stream⟨odd⟩] =
  fun s {
    save a {
      InL[save e { restore a InR[save o {
        abort (itl_aux (fun x { restore e x}) (fun x { restore o x }) s)
      } ] } ]
    }
  }
```

This function starts by saving two continuations, corresponding to the constructors `InL` and `InR` of the return type, and then calls `itl_aux` on the input stream. The very fact that we can write `itl` proves that it is possible to extract a stream of odd numbers or a stream of even numbers from any stream of natural numbers.

« Résumé substantiel » (en français)

Au cours des dernières années, les assistants à la preuves on fait des progrès considérables et ont atteint un grand niveau de maturité. Ils ont permit la certification de programmes complexes tels que des compilateurs et même des systèmes d'exploitation. Néanmoins, l'utilisation d'un assistant de preuve requiert des compétences techniques très particulières, qui sont très éloignées de celles requises pour programmer de manière usuelle. Pour combler cet écart, nous entendons concevoir un langage de programmation de style ML supportant la preuve de programmes. Il combine au sein d'un même outil la flexibilité de ML et le fin niveau de spécification offert par un assistant de preuve. Autrement dit, le système peut être utilisé pour programmer de manière fonctionnelle et fortement typée, tout en permettant l'obtention de nouvelles garanties au besoin.

On étudie donc un langage en appel par valeurs dont le système de type étend une logique d'ordre supérieur. Il comprend un type égalité (entre programmes non typés), un type de fonctions dépendantes, la logique classique et du sous-typage. La combinaison de l'appel par valeurs, des fonctions dépendantes et de la logique classique est connu pour poser des problèmes de cohérence. Pour s'assurer de la correction du système (cohérence logique et sûreté à l'exécution), on propose un cadre théorique basé sur la réalisabilité classique de Krivine. Le modèle repose sur une propriété essentielle qui lie les différents niveaux d'interprétation des types d'une manière novatrice.

On démontre aussi l'expressivité de notre système en se basant sur son implantation dans un prototype. Il peut être utilisé pour prouver des propriétés de programmes standards tels que la fonction « map » sur les listes ,ou le tri par insertion.

Chapitre 1, introduction

Depuis l'apparition des premiers ordinateurs, chaque génération de programmeurs à du faire face au problème de la fiabilité du code. Les langages statiquement typés tels que Java, Haskell, OCaml, Rust ou Scala ont attaqué ce problème avec des vérifications statiques, au moment de la compilation, pour détecter des programmes incorrects. Leur typage fort est particulièrement utile quand plusieurs objets incompatibles doivent être manipulés au même moment. Par exemple, un programme qui calcule une addition sur un booléen (ou une fonction) est immédiatement rejeté. Durant les dernières années, les avantages du typage statique ont même été reconnus au sein de la communauté des langages dynamiquement typés. Des systèmes de vérification statique du typage sont dorénavant disponibles pour Javascript [*Microsoft 2012*, *Facebook 2014*] ou Python [*Lehtosalo 2014*].

Dans les trente dernières années, des progrès significatifs ont été fait dans l'application de la théorie des types aux langages de programmation. La correspondance de Curry-

Howard, qui lie les systèmes de types des langages de programmation fonctionnels à la logique mathématique, a été explorée dans deux directions principales. D'un côté, les assistants à la preuve comme Coq ou Agda sont basés sur des logiques très expressives [*Coquand 1988*, *Martin-Löf 1982*]. Pour montrer leur cohérence logique, les langages de programmation sous-jacents doivent être restreints aux programmes qui peuvent être montrés terminant. Ils interdisent donc les formes de récursion les plus générales. De l'autre côté, les langages de programmation fonctionnelle comme OCaml ou Haskell sont adaptés à la programmation, car ils n'imposent pas de restriction sur la récursion. Cependant, ils sont basés sur des logiques qui ne sont pas cohérentes, ce qui implique qu'ils ne peuvent pas être utilisés pour démontrer des formules mathématiques.

Le but de ce travail est de fournir un environnement uniforme au sein duquel des programmes peuvent être écrits, spécifiés, et prouvés. L'idée est de combiner un langage de programmation à la ML complet, avec un système de type enrichi pour permettre la spécification de comportements calculatoires. Ce langage peut donc être utilisé comme ML pour programmer en tirant profit d'un typage statique fort, mais aussi comme un assistant à la preuve pour démontrer des propriétés de programmes ML. L'uniformité du système permet, en outre, de raffiner les programmes petit à petit, pour obtenir de plus en plus de garanties. En particulier, il n'y a pas de distinction syntaxique entre les programmes et les preuves dans le système. On peut donc mélanger preuves et programmes durant la construction de preuves ou de programmes. Par exemples, on peut utiliser des mécanismes de preuve au sein de programmes afin qu'ils portent des propriétés (par exemple, l'addition avec sa commutativité). Les programmes peuvent utiliser des mécanismes de preuve pour éliminer du code mort (ne pouvant pas être atteint à l'exécution).

Dans cette thèse, notre but premier est de mettre au point un système de type pour un langage de programmation fonctionnelle, utilisable en pratique. Parmi les nombreux choix techniques possibles, nous avons décidé de considérer un langage en appel par valeur similaire à OCaml ou SML, ces derniers ayant fait leurs preuves en terme d'efficacité et d'utilisation. Notre langage comporte des variants polymorphes [*Garrigue 1998*] et des types enregistrements à la SML, qui sont très pratiques pour encoder des types de données. Par exemple, le type des listes peut être défini et utilisé de la manière suivante.

```
type rec list⟨a⟩ = [Nil ; Cons of {hd : a ; tl : list}]

val rec exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    case l {
      Nil     → false
      Cons[c] → if pred c.hd { true } else { exists pred c.tl }
    }
  }
```

Ici, la fonction polymorphe `exists` prend comme paramètre un prédicat et une liste, et elle indique si (au moins) un élément de la liste satisfait le prédicat.

Le système présenté ici n'est pas seulement un langage de programmation, mais aussi un assistant à la preuve, et en particulier à la preuve de programmes. Son mécanisme de preuve est basé sur des types égalités de la forme `t ≡ u`, où `t` et `u` sont des programmes arbitraires du langage. Un tel type égalité est habité par (ou contient) `{}` (c'est à dire l'enregistrement vide) si l'équivalence dénotée est vraie, et il est vide sinon. Les équivalences sont gérées en utilisant une procédure partielle de décision, qui est dirigée par la construction de programmes. Un contexte d'équations est maintenu par l'algorithme de typage, afin de stocker les équivalences supposées correctes durant la construction de la preuve de typage. Ce contexte est étendu quand une nouvelle équation est apprise (par exemple, quand un lemme est appliqué), et une équation est prouvée en cherchant une contradiction (par exemple, quand deux variants différents sont supposés égaux).

Pour illustrer le fonctionnement des preuves, nous allons considérer l'exemple très simple des entiers naturels en représentation unaire (les nombres de Peano). Leur type est donné ci-dessous, avec la fonction d'addition correspondante, définie par récurrence sur son premier argument.

```
type rec nat = [Zero ; Succ of nat]

val rec add : nat ⇒ nat ⇒ nat =
  fun n m {
    case n { Zero → m | Succ[k] → Succ[add k m] }
  }
```

Comme premier exemple, nous allons montrer `add Zero n ≡ n` pour tout `n`. Pour exprimer cette propriété, on utilise le type `∀n:ι, add Zero n ≡ n`, où `ι` peut être vu comme l'ensemble de tous les programmes complètement évalués. Cette énoncé peut ensuite être démontré comme suit.

```
val add_z_n : ∀n:ι, add Zero n ≡ n = {}
```

Ici, la preuve est immédiate (c'est à dire, `{}`) comme `add Zero n ≡ n` se déduit directement de la définition de la fonction `add`. Notez que cette équivalence est vraie pour tout `n`, qu'il corresponde à un élément de `nat` ou pas. Par exemple, on peut montrer sans problème l'équivalence `add Zero true ≡ true`.

Regardons maintenant l'énoncé `∀n:ι, add n Zero ≡ n`. Bien qu'il soit très similaire à `add_z_n` en apparence, il ne peut pas être démontré. En effet, la relation `add n Zero ≡ n` n'est pas vraie quand `n` n'est pas un entier unaire. Dans ce cas, l'évaluation de `add n Zero` produit une erreur à l'exécution. En conséquence, on devra se reposer sur une forme de

quantification dont le domaine se limite aux entiers unaires. Ceci peut être réalisé avec le type `∀n∈nat, add n Zero ≡ n`, qui corresponds à une fonction (dépendante) prenant en entrée un entier `n`, et retournant une preuve de `add n Zero ≡ n`. Cette propriété peut ensuite être prouvée en utilisant de l'induction (programme récursif) et une analyse par cas (filtrage par motif).

```
val rec add_n_z : ∀n∈nat, add n Zero ≡ n =
  fun n {
    case n {
      Zero    → {}
      Succ[k] → let ih = add_n_z k; {}
    }
  }
```

Si `n` est `Zero`, alors on doit montrer `add Zero Zero ≡ Zero`, qui est immédiat par définition de `add`. Dans le cas où `n` est `Succ[k]` on doit montrer `add Succ[k] Zero ≡ Succ[k]`. Par définition de `add`, cette équation se réduit en `Succ[add k Zero] ≡ Succ[k]`. Il suffit donc de montrer `add k Zero ≡ k` en utilisant l'hypothèse d'induction (`add_n_z k`).

Chapitre 2, calcul non typé

Dans ce chapitre, on introduit le langage de programmation qui sera utilisé dans toute la suite de cette thèse. Sa sémantique opérationnelle est exprimée à l'aide d'une machine abstraite, qui nous permettra de considérer des opérations produisant des effets de bord. Comme tout langage de programmation fonctionnelle, notre système est base sur le λ-calcul. Créé par Alonzo Church dans les années trente, le λ-calcul [*Church 1941*] est un formalisme permettant la représentation de fonctions calculables, et en particulier de fonctions récursives. Comme l'a démontré Alan Turing, le λ-calcul est un *modèle de calcul universel* [*Turing 1937*].

Les termes du λ-calcul (appelés λ-termes) sont construits à partir d'un alphabet dénombrable de variables (appelées λ-variables) $\mathcal{V}_\iota = \{x, y, z...\}$. L'ensemble de tous les λ-termes est généré par la grammaire BNF suivante.

$$t, u ::= x \mid \lambda x.t \mid t\ u \qquad\qquad x \in \mathcal{V}_\iota$$

Un terme de la forme $\lambda x.t$ est appelé abstraction (ou λ-abstractions) et un terme de la forme $t\ u$ est appelé application. Le langage que nous considérons dans cette thèse est en fait exprimé sous la forme d'une machine abstraite comprenant quatre catégories syntaxiques (valeurs, termes, piles et processus) générées par la grammaire BNF suivante.

$$
\begin{array}{lll}
(\Lambda_\iota) & v, w & ::= x \mid \lambda x.t \mid C_k[v] \mid \{(l_i = v_i)_{i\in I}\} \mid \square \\
(\Lambda) & t, u & ::= a \mid v \mid t\ u \mid \mu\alpha.t \mid [\pi]t \mid v.l_k \mid [v \mid (C_i[x_i] \to t_i)_{i\in I}] \mid Y_{t,v} \mid R_{v,t} \mid \delta_{v,w} \\
(\Pi) & \pi, \rho & ::= \varepsilon \mid \alpha \mid v \cdot \pi \mid [t]\pi \\
(\Lambda\times\Pi) & p, q & ::= t * \pi
\end{array}
$$

Let termes et valeurs forment une variante du $\lambda\mu$-calculus [*Parigot 1992*], enrichit avec des éléments des langages à la ML (enregistrements et variants). Les valeurs de la forme $C_k[v]$ (avec $k \in \mathbb{N}$) correspondent à des variants (ou constructeurs). Un filtrage par motif peut être effectué sur les variants avec la syntaxe $[v \mid (C_i[x_i] \to t_i)_{i\in I}]$, où le motif $C_i[x_i]$ est associés au terme t_i pour tout i dans $I \subseteq_{\text{fin}} \mathbb{N}$. D'une manière similaire, les valeurs de la forme $\{(l_i = v_i)_{i\in I}\}$ correspondent à des enregistrements, qui sont des n-uplets avec des composantes nommées. L'opération de projection $v.l_k$ peut être utilisée pour accéder à la valeur stockées sous le label l_k dans v.

Les processus forment l'état interne de notre machine abstraite. On peut en fait voir un processus comme un terme, placé dans un contexte d'évaluation représenté par une pile. Intuitivement, la pile π du processus $t * \pi$ contient les paramètres qui seront fournit à t. Comme on est en appel par valeur, les piles stockent également les fonctions durant l'évaluation de leurs arguments. C'est pourquoi on a besoin de piles de la forme $[t]\pi$. La sémantique opérationnelle de notre langage est donnée par la relation $(\succ)$ définie sur les processus en utilisant les règles de réduction suivantes.

$$
\begin{array}{rcll}
t\ u * \pi & \succ & u * [t]\pi & \\
v * [t]\pi & \succ & t * v \cdot \pi & \text{si } v \notin \mathcal{V}_\iota \cup \{\square\} \\
\lambda x.t * v \cdot \pi & \succ & t[x := v] * \pi & \\
\mu\alpha.t * \pi & \succ & t[\alpha := \pi] * \pi & \\
[\xi]t * \pi & \succ & t * \xi & \\
\{(l_i = v_i)_{i\in I}\}.l_k * \pi & \succ & v_k * \pi & \text{si } k \in I \\
[C_k[v] \mid (C_i[x_i] \to t_i)_{i\in I}] * \pi & \succ & t_i[x_i := v] * \pi & \text{si } k \in I \\
Y_{t,v} * \pi & \succ & t\ (\lambda x.Y_{t,x})\ v * \pi & \\
R_{\{(l_i = v_i)_{i\in I}\},u} * \pi & \succ & u * \pi & \\
\square * [t]\pi & \succ & \square * \pi & \\
\square * v \cdot \pi & \succ & \square * \pi & \\
[\square \mid (C_i[x_i] \to t_i)_{i\in I}] * \pi & \succ & \square * \pi & \\
\square.l_k * \pi & \succ & \square * \pi &
\end{array}
$$

Les trois premières règles sont celles qui prennent en charge la β-réduction, c'est à dire l'évaluation standard des termes du λ-calcul. Quand la machine abstraite rencontre une application, la fonction est stockée sur la pile jusqu'à ce que son argument ait été complètement évalué. Une fois l'argument calculé, un valeur fait face à la pile contenant la fonction, on peut donc utiliser la second règle pour mettre la fonction en position d'évaluation et son

argument en position d'argument, prêt à être consommé dés que la fonction se sera évalué en une λ-abstraction. À ce moment là, on pourra réaliser une substitution sans capture en utilisant la troisième règle, pour que l'application prenne effet. Le but des règles suivantes est de prendre en charge l'évaluation des programmes formés avec les autres constructeurs du langage (effets, enregistrements, variants, récursion).

Chapitre 3, équivalence observationnelle

Dans ce chapitre, on introduit une relation d'équivalence sur les termes du langage. Deux termes sont considérés équivalent si et seulement si ils ont le même comportement observable en terme de calcul. Des propriétés générales sont ensuite obtenues pour toute relation d'équivalence satisfaisant certaines contraintes. Ces propriétés sont essentielles pour la définition de la sémantique de réalisabilité dans les chapitres suivants. De plus, elles nous permettent d'implanter une procédure partielle de décision pour l'équivalence de programmes.

L'idée principale de ce chapitre est de considérer une forme d'équivalence observationnelle. En d'autre termes, deux programmes seront considérés équivalent dés lors qu'ils ont le même comportement observable dans tous les contextes d'évaluation. Ici, on observera simplement, pour chaque pile, si le processus formé calcule une valeur ou produit une erreur (ou ne termine pas). On quantifiera également sur toutes les substitutions pour les variables libres, de manière à pouvoir comparer des termes ouverts. La relation $(\equiv_>) \subseteq \Lambda\times\Lambda$ est donc définie comme suit.

$$(\equiv_>) = \left\{(t\,,\,u) \mid \forall\,\pi \in \Pi, \forall\,\rho \in \mathcal{S}\,,\, t\rho * \pi \Downarrow_> \Leftrightarrow\ u\rho * \pi \Downarrow_>\right\}$$

Chapitre 4, système de type et sémantique

Dans ce chapitre, on présente un nouveau système de type, qui se distingue grâce à une notion d'équivalence de programme embarquée. Elle permet de spécifier des propriétés équationnelles de programmes, qui sont ensuite prouvées par des raisonnements équationnels sur les programmes. Nos types sont interprétés en utilisant des techniques standard de la réalisabilité classique, qui nous permettent de donner une justification sémantique à nos règles de typage.

L'interprétation des types est définie inductivement, de manière usuelle. Il faut quand même remarquer que, du fait de l'appel par valeur, l'interprétation du type des fonctions requiert trois niveaux d'interprétation (valeurs, piles, termes) liés par orthogonalité. Plus précisément, on définira l'interprétation $[\![A]\!]$ d'un type comme l'ensemble de ses valeurs,

et on obtiendra ensuite par orthogonalité un ensemble de piles $[\![A]\!]^{\perp}$, puis un ensemble de termes $[\![A]\!]^{\perp\perp}$ de la manière suivante.

$$[\![A]\!]^{\perp} = \left\{\pi \in \Pi \mid \forall\, v \in [\![A]\!], v * \pi \Downarrow_{>}\right\} \qquad [\![A]\!]^{\perp\perp} = \left\{t \in \Lambda \mid \forall\, \pi \in [\![A]\!]^{\perp}, t * \pi \Downarrow_{>}\right\}$$

En particulier, le type restriction, dénotant une conjonction sémantique sans contenu algorithmique, sera interprété comme $[\![A \upharpoonright t \equiv u]\!] = [\![A]\!]$ si on a $t \equiv u$ en accord avec la section précédente. Sinon, on prendra $[\![A \upharpoonright t \equiv u]\!] = [\![\bot]\!]$, c'est à dire la même interprétation que le type vide $\bot$.

Chapitre 5, restriction aux valeurs

Dans ce chapitre, nous considérons l'encodage des types dépendants, qui sont une forme de quantification typée dans notre système. Cependant, l'expressivité de ces derniers est considérablement limitée par la restriction aux valeurs. Pour résoudre ce problème, on introduit dans le système la notion de *restriction aux valeurs sémantique*, qui permet au système d'accepter bien plus de programmes. Obtenir un modèle justifiant la *restriction aux valeurs sémantique* nécessite de changer à la fois la sémantique opérationnelle et la définition de l'équivalence de programmes.

L'idée ici est de considérer qu'un terme t est une valeur, si il existe une valeur v telle que $t \equiv v$. On pourra donc donner des règles de typage relâchées, plus générales, qui auront une prémisse de la forme $\Xi \vdash t \equiv v$. Pour rendre notre modèle de réalisabilité compatible avec cette idée, il est absolument nécessaire que les différents niveaux d'interprétation des types satisfassent la relation suivante.

$$v \in [\![A]\!]^{\perp\perp} \Rightarrow v \in [\![A]\!]$$

En d'autre termes, si une valeur est présente dans l'interprétation d'un type au niveau des termes, alors elle était déjà présente au niveau des valeurs. Plus précisément, on demandera à ce que l'opération $[\![A]\!] \mapsto [\![A]\!]^{\perp\perp}$, qui peut être vue comme une forme de complétion, n'introduise pas de nouvelles valeurs.

Bien que cette propriété soit naturelle, elle n'est pas satisfaite en général dans les modèles de réalisabilité classique (en appel par valeurs). La définition d'un modèle ayant cette propriété est le résultat central de cette thèse [*Lepigre 2016*]. Il est obtenu en étendant la syntaxe des termes avec une opération permettant de tester des équivalences durant la réduction de la machine abstraite.

Chapitre 6, sous-typage

Dans ce chapitre, on reformule la définition de notre système de type pour inclure du sous-typage. L'idée principale est de transformer les règles de typage qui n'ont pas de contenu algorithmique en règles de sous-typage. En particulier, les quantificateurs, points fixes, appartenance et égalités seront gérés au sein du sous-typage.

Dans le cadre de cette thèse, le sous-typage à deux intérêts principaux. En premier lieux, il permet de donner un formulation du système qui est dirigée par la syntaxe. En d'autre termes, une et une seule règle de typage peut être appliquée pour un terme donné (peu importe le type correspondant), et une et une seule règle de sous-typage peut être appliquée étant donné deux types (modulo quelques subtilités). Il est en fait surprenant qu'on puisse obtenir un ensemble de règles aussi satisfaisant pour une implantation, malgré la très probable non décidabilité du typage et du sous-typage dans notre système (c'est une extension de System F [*Wells 1999*]).

Sur le plan technique, nous considérons une notion de sous-typage bien particulière (appelé sous-typage pointé) de la forme $t \in A \subset B$, et nous faisons appel à des opérateurs de choix inspirés des travaux de Hilbert (voir [*Lepigre 2017*]).

Chapitre 7, programmes et preuves

Dans ce dernier chapitre, nous considérons des exemples de programmes et de preuves qui peuvent être écrits et manipulés par le prototype que nous avons implanté. Cet ensemble restreint d'exemples n'a pas pour but de présenter le système de manière exhaustive. Ils visent seulement à démontrer l'expressivité du système, à travers une sélection d'exemples. Tous les exemples données dans ce chapitre ont été vérifiés par notre prototype à la production de ce document. Ils sont donc acceptés par notre implémentation sans qu'aucune modification soit nécessaire.

Une partie des exemples considérés concernent les listes, avec certains de leurs sous-types. En particulier, on considère le type des vecteurs (listes de taille fixée) et les listes triées. On démontre ainsi, par exemple, qu'il est possible de refléter par le typage qu'une fonction de tri (ici le tri par insertion) transforme une liste en une liste triée.

Pour finir le chapitre, on considère quelque exemples utilisant la logique classique. En particulier, on définit un programme qui est en fait extrait de la preuve classique d'un lemme sur les listes infinies d'entiers (ou « stream »). On définit ainsi une fonction qui, étant donné une liste infinie d'entiers, retourne une sous-liste infinie de nombre pairs, ou une sous-liste infinie de nombres impairs. Il n'est évidemment pas possible d'écrire un tel programme hors d'un cadre classique.

References

[*Abel 2008*] *Semi-Continuous Sized Types and Termination*, Andreas Abel, Logical Methods in Computer Science, Volume 4, Number 2.

[*Barendregt 1981*] *The Lambda Calculus - Its Syntax and Semantics*, Hendrik P. Barendregt, North-Holland.

[*Baro 2003*] *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML*, Sylvain Baro, Thèse de l'Université Paris Diderot.

[*Casinghino 2014*] *Combining Proofs and Programs in a Dependently Typed Language*, Chris Casinghino, Vilhelm Sjöberg and Stephanie Weirich, proceedings of POPL.

[*Castagna 2016*] *Set-Theoretic Types for Polymorphic Variants*, Giuseppe Castagna, Tommaso Petrucciani and Kim Nguyen, proceedings of ICFP.

[*Charguéraud 2010*] *Program verification through characteristic formulae*, Arthur Charguéraud, proceedings of ICFP.

[*Charguéraud 2011*] *Characteristic formulae for the verification of imperative programs*, Arthur Charguéraud, proceedings of ICFP.

[*Chlipala 2005*] *Strict bidirectional type checking*, Adam Chlipala, Leaf Petersen and Robert Harper, proceedings of TLDI.

[*Church 1936*] *Some properties of conversion*, Alonzo Church and John Barkley Rosser Sr., Transactions of the American Mathematical Society, Volume 36, Number 3.

[*Church 1941*] *The Calculi of Lambda-Conversion*, Alonzo Church, Annals of Mathematical Studies, Volume 6.

[*Constable 1986*] *Implementing Mathematics with the Nuprl proof development system*, Robert. L. Constable, S. F. Allen, H. M. Bromley et al., Prentice Hall.

[*CoqTeam 2004*] *The Coq Proof Assistant Reference Manual*, Coq Development Team, LogiCal Project.

[*Coquand 1988*] *The Calculus of Constructions*, Thierry Coquand and Gérard Huet, Information and Computation, Volume 76, Issue 2-3.

[*Damas 1982*] *Principal Type-Schemes for Functional Programs*, Luís Damas and Robin Milner, proceedings of POPL.

[*Facebook 2014*] *Flow - A static type checker for Javascript*, Facebook Inc., open source.

[*Filliâtre 2013*] *Why3 - Where Programs Meet Provers*, Jean-Christophe Filliâtre and Andrei Paskevich, proceedings of ESOP, Lecture Notes in Computer Science, Volume 7792.

[*Flanagan 1993*] *The Essence of Compiling with Continuations*, Cormac Flanagan, Amr Sabry, Bruce Duba et al., proceedings of PLDI.

[*Garrigue 1998*] *Programming with Polymorphic Variants*, Jacques Garrigue, proceedings of the ML Workshop.

[*Garrigue 2004*] *Relaxing the Value Restriction*, Jacques Garrigue, proceedings of FLOPS.

[*Girard 1972*] *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Jean-Yves Girard, Thèse de l'Université Paris VII.
[*Girard 1989*] *Proofs and Types*, Jean-Yves Girard, Paul Taylor and Yves Lafont, Cambridge University Press.
[*Griffin 1990*] *A Formulae-as-Types Notion of Control*, Timothy G. Griffin, proceedings of POPL.
[*Harper 1991*] *ML with callcc is unsound*, Robert Harper and Mark Lillibridge, Message posted to the SML mailing list.
[*Hilbert 1934*] *Grundlagen der Mathematik I*, David Hilbert and Paul Bernays, Grundlehren der mathematischen Wissenschaften.
[*Howe 1989*] *Equality in Lazy Computation Systems*, Douglas J. Howe, proceedings of LICS.
[*Huet 1997*] *The Zipper*, Gérard Huet, Journal of Functional Programming, Volume 7.
[*Hughes 1996*] *Proving the Correctness of Reactive Systems Using Sized Types*, John Hughes, Lars Pareto and Amr Sabry, proceedings of POPL.
[*Jia 2008*] *AURA: a Programming Language for Authorization and Audit*, L. Jia, J. Vaughan, Karl Mazurak et al., proceedings of ICFP.
[*Krivine 1990*] *Lambda-calcul, types et modèles*, Jean-Louis Krivine, Masson.
[*Krivine 2007*] *A call-by-name lambda-calculus machine*, Jean-louis Krivine, Higher Order and Symbolic Computation.
[*Krivine 2009*] *Realizability in Classical Logic*, Jean-Louis Krivine, Panoramas et Synthèses, Volume 27.
[*Lehtosalo 2014*] *mypy - Optional static typing for Python*, Jukka Lehtosalo, Open source project.
[*Lepigre 2016*] *A Classical Realizability Model for a Semantical Value Restriction*, R. Lepigre, proceedings of ESOP, Lecture Notes in Computer Science, Volume 9632.
[*Lepigre 2017*] *Practical Subtyping for System F with Sized (Co-)Induction*, Rodolphe Lepigre and Christophe Raffalli, Submitted.
[*Leroy 1991*] *Polymorphic Type Inference and Assignment*, Xavier Leroy and Pierre Weis, in the proceedings of POPL.
[*Leroy 1993*] *Polymorphism by Name for References and Continuations*, Xavier Leroy, in the proceedings of POPL.
[*Licata 2009*] *Positively Dependent Types*, Daniel Licata and Robert Harper, in the proceedings of PLPV.
[*MacQueen 1984*] *Modules for Standard ML*, David MacQueen, proceedings of LFP.
[*Manoury 1992*] *ProPre A Programming Language with Proofs*, Pascal Manoury, Michel Parigot and Marianne Simonot, Lecture Notes in Computer Science, Volume 624.
[*Martin-Löf 1982*] *Constructive Mathematics and Computer Programming*, Per Martin-Löf, Studies in Logic the Foundations of Mathematics, Volume 104.
[*Microsoft 2012*] *TypeScript - Javascript that scales*, Microsoft, Open source project.
[*Miquel 2001*] *Le Calcul des Constructions Implicites : Syntaxe et Sémantique*, Alexandre Miquel, Thèse de l'Université Paris VII.

[*Miquel 2007*] *Classical Program Extraction in the Calculus of Constructions*, Alexandre Miquel, proceedings of CSL.
[*Miquel 2011*] *Existential witness extraction in classical realizability and via a negative translation*, Alexandre Miquel, Logical Methods in Computer Science.
[*Mitchell 1991*] *An Extension of System F with Subtyping*, L. Cardelli, S. Martini, J. C. Mitchell et al., proceedings of TACS.
[*Mitchell 1996*] *Foundations for Programming Languages*, John C. Mitchell, MIT Press.
[*Moggi 1989*] *Computational Lambda-Calculus and Monads*, Eugenio Moggi, in the proceedings of LICS.
[*Munch 2009*] *Focalisation and Classical Realisability*, Guillaume Munch-Maccagnoni, in the proceedings of CSL.
[*Norell 2008*] *Dependently Typed Programming in Agda*, Ulf Norell, Lecture notes from the Summer School in Advanced FP.
[*Owre 1996*] *PVS: Combining Specification, Proof Checking and Model Checking*, S. Owre, S. Rajan, J. Rushby et al., Lecture Notes In Computer Science.
[*Parigot 1992*] *Lambda-Mu-calculus: an algorithmic interpretation of classical natural deduction*, Michel Parigot, proceedings of LPAR.
[*Raffalli 1998*] *System F-eta*, Christophe Raffalli.
[*Raffalli 1999*] *An optimized complete semi-algorithm for system F-eta*, Christophe Raffalli.
[*Reynolds 1974*] *Towards a Theory of Type Structure*, John C. Reynolds, proceedings Colloque sur la Programmation.
[*Régis-Gianas 2007*] *Des types aux assertions logiques : preuve automatique ou assistée de propriétés sur les programmes fonctionnels*, Yann Régis-Gianas, Thèse de l'Université Paris Diderot.
[*Swamy 2011*] *Secure Distributed Programming with Value-Dependent Types*, Nikhil Swamy, Juan Chen, C. Fournet et al., proceedings of ICFP.
[*Tarditi 1996*] *TIL: A Type-Directed Optimizing Compiler for ML*, D. Tarditi, G. Morrisett, P. Cheng et al., proceedings of PLDI.
[*Tiuryn 1996*] *The Subtyping Problem for Second-Order Types is Undecidable*, Jerzy Tiuryn and Pawel Urzyczyn, Proceedings of LICS.
[*Tiuryn 2002*] *The Subtyping Problem for Second-Order Types is Undecidable*, Jerzy Tiuryn and Pawel Urzyczyn, Information and Computation, Volume 179.
[*Tofte 1990*] *Type Inference for Polymorphic References*, Mads Tofte, Information and Computation, Volume 89, Issue 1.
[*Turing 1937*] *Computability and Lambda-Definability*, Alan Turing, Journal of Symbolic Logic.
[*Wadler 2003*] *Call-by-value is dual to call-by-name*, Philip Wadler, SIGPLAN Notices 38(9).
[*Wells 1994*] *Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable*, Joe B. Wells, Proceedings of LICS.
[*Wells 1999*] *Typability and type checking in System F are equivalent and undecidable*, Joe B. Wells, Annals of Pure and Applied Logic, Volume 98.

[*Wright 1994*] *A Syntactic Approach to Type Soundness*, A. K. Wright and M. Felleisen, Information and Computation, Volume 15, Issue 1.
[*Wright 1995*] *Simple Imperative Polymorphism*, Andrew K. Wright, Lisp and Symbolic Computation, Volume 8, Number 4.
[*Xi 1999*] *Dependent Types in Practical Programming*, Hongwei Xi and Frank Pfenning, in the proceedings of POPL.
[*Xi 2003*] *Applied Type System: Extended Abstract*, Hongwei Xi, proceedings of TYPES.
[*de Groote 1994*] *On the Relation between the Lambda-Mu-Calculus and the Syntactic Theory of Sequential Control*, Philippe de Groote, Lecture Notes in Computer Science.

Résumé

Au cours des dernières années, les assistants de preuves on fait des progrès considérables et ont atteint un grand niveau de maturité. Ils ont permit la certification de programmes complexes tels que des compilateurs et même des systèmes d'exploitation. Néanmoins, l'utilisation d'un assistant de preuve requiert des compétences techniques très particulières, qui sont très éloignées de celles requises pour programmer de manière usuelle. Pour combler cet écart, nous entendons concevoir un langage de programmation de style ML supportant la preuve de programmes. Il combine au sein d'un même outil la flexibilité de ML et le fin niveau de spécification offert par un assistant de preuve. Autrement dit, le système peut être utilisé pour programmer de manière fonctionnelle et fortement typée tout en autorisant l'obtention de nouvelles garanties au besoin.

On étudie donc un langage en appel par valeurs dont le système de type étend une logique d'ordre supérieur. Il comprend un type égalité entre les programmes non typés, un type de fonction dépendant, la logique classique et du sous-typage. La combinaison de l'appel par valeurs, des fonctions dépendantes et de la logique classique est connu pour poser des problèmes de cohérence. Pour s'assurer de la correction du système (cohérence logique et sûreté à l'exécution), on propose un cadre théorique basé sur la réalisabilité classique de Krivine. Le modèle repose sur une propriété essentielle qui lie les différent niveaux d'interprétation des types d'une manière novatrice.

On démontre aussi l'expressivité de notre système en se basant sur son implantation dans un prototype. Il peut être utilisé pour prouver des propriétés de programmes standards tels que la fonction « map » sur les listes ou le tri par insertion.

Abstract

In recent years, proof assistant have reached an impressive level of maturity. They have led to the certification of complex programs such as compilers and operating systems. Yet, using a proof assistant requires highly specialised skills and it remains very different from standard programming. To bridge this gap, we aim at designing an ML-style programming language with support for proofs of programs, combining in a single tool the flexibility of ML and the fine specification features of a proof assistant. In other words, the system should be suitable both for programming (in the strongly-typed, functional sense) and for gradually increasing the level of guarantees met by programs, on a by-need basis.

We thus define and study a call-by-value language whose type system extends higher-order logic with an equality type over untyped programs, a dependent function type, classical logic and subtyping. The combination of call-by-value evaluation, dependent functions and classical logic is known to raise consistency issues. To ensure the correctness of the system (logical consistency and runtime safety), we design a theoretical framework based on Krivine's classical realizability. The construction of the model relies on an essential property linking the different levels of interpretation of types in a novel way.

We finally demonstrate the expressive power of our system using our prototype implementation, by proving properties of standard programs like the map function on lists or the insertion sort.