Towards proved programs
Part III: Dependently-typed programming

MPRI 2-4-2 (Version Fri Feb 6 12:33:16 CET 2015)

Yann **Régis-Gianas**
yrg@pps.univ-paris-diderot.fr

Paris 7	-	PPS
INRIA	-	$\pi.r^2$

Contents

Contents

Motivations

If types are to be used as specifications, they must refer to values.
With GADTs, we used a type-level encoding of these values:

```
type zero
type succ α
type nat α =
   | Z :  nat zero
   | S :  forall α. nat α → nat (succ α)

type vector α =
| Nil :  vector zero
| Cons :  forall α. α × vector α → vector (succ α)
```

This a rather indirect (and inefficient) way of talking about natural numbers.
Isn't it?

Types that depend on **values**

For the moment, we only considered types that may depend on types. Yet, there is no obvious reason not to allow arbitrary term to be used inside types:

```
type nat = Z : nat | S : nat → nat

type vector : nat → ⋆ → ⋆ =
| Nil : forall α. vector Z α
| Cons : forall α. forall (n : nat). α × vector n α → vector (S n) α
```

Look at the red part of `Cons` type: instead of a universal quantification over types, we wrote a universal quantification over all values of type `nat`. These quantification is sometimes written using a Π but it really behaves at the level of type like system F's symbol ∀.

This unusual type is **dependent on values**. This is a **dependent type**.

How to program with dependent types?

Preliminary definition

```
let rec add = function
| Z → fun n → n
| S k → fun n → S (add k n)
```

Notice that:

- `add Z n` reduces to `n`
- `add (S k) n` reduces to `S (add k n)`

Example (in an imaginary O'CAML with dependent types)

```
let rec concat
: forall α. forall (n m : nat).
  vector n α → vector m α → vector (add n m) α
= fun n m v1 v2 →
  match v1 with
  | Nil → v2
  | Cons [h] (x, xs) → Cons [add h m] (x, concat h xs v2)
```

How to design a type system that would accept this program?

Example (in an imaginary O'Caml with dependent types)

```
let rec concat
: forall α. forall (n m : nat).
  vector n α → vector m α → vector (add n m) α
= fun n m v1 v2 →
  match v1 with
  | Nil → v2
  | Cons [h] (x, xs) → Cons [add h m] (x, concat h xs v2)
```

In the first branch, using a similar reasoning as for GADTs, we should be able to refine the value of `n` to `Z` because `Nil : vector Z` α. Thus, we are expecting a result of type `vector (add Z m)` α.
How to conclude that `v2` has type `vector (add Z m)` α ?

Example (in an imaginary O'CAML with dependent types)

```
let rec concat
: forall α. forall (n m : nat).
  vector n α → vector m α → vector (add n m) α
= fun n m v1 v2 →
  match v1 with
  | Nil → v2
  | Cons [h] (x, xs) → Cons [add h m] (x, concat h xs v2)
```

We shall add a **conversion rule** that will justify the equivalence between this type and the type `vector m` α *modulo term reduction*. In this example, this allows us to internalize the fact that `add Z m` reduces to `m` inside the type equivalence.

Example (in an imaginary O'Caml with dependent types)

```
let rec concat
: forall α. forall (n m : nat).
  vector n α → vector m α → vector (add n m) α
= fun n m v1 v2 →
  match v1 with
  | Nil → v2
  | Cons [h] (x, xs) → Cons [add h m] (x, concat h xs v2)
```

In the second branch, the recursive call to `concat` produces a value of type `vector (add h m)` α. Therefore, the result has type `vector (S (add h m))` α. Yet, we are expecting a value of type `vector (add (S h) m)` α. Again, these two types are convertible to each other!

Curry-Howard correspondence

Now that types may depend on values, they can be considered as specifications. The Curry-Howard correspondence states a deep connection between programs and proofs:

Programming language	**Logic**
type	formula
term	proof
reduction	proof normalization

This connection can be explored fruitfully by looking for the image of some programming mechanisms on the logical side, or conversely, by looking for the image of somes reasoning laws on the computational side.

Does that necessary mean that *programm-ing* is *prov-ing* and reciprocally?

Why dependently typed programming matters? [McKinna, 2006]

Functional programmers usually start designing sofware by writing down type signatures. Inductive data type definitions are particularly useful to refine these specifications because they give a declarative decomposition of the problem as well as an induction scheme to reason on it. Most of the time, functional programmers say that once they have found the right data type definitions, the actual program is easy to write.

Following that idea, dependently-typed programmers advocate that the more expressive these definitions can be, the better!

Example

The usual data type for integer lists is a poor specification tool:

```
let rec sort : list → list =
function
  | [] → []
  | x :: xs → insert x (sort xs)

let rec insert : nat → list → list = ...
```

Here the data type (only) prevents the programmer to forget a case and forces the function to produce a list.

Example

Assume that `permutation_sorted_list l` is a dependent type to denote the sorted permutation of the elements of list `l`.

```
let rec sort : forall (l: list). permutation_sorted_list l =
function
  | [] → []
  | x :: xs → insert x xs (sort xs)

let rec insert :
  forall (x: nat, l: list, ls: permutation_sorted_list l).
  permutation_sorted_list (x :: l) = ...
```

Here the dependent type not only ensures that a list is returned but that this list is sorted and contains the same elements as the input one.

Contents

Formalizing a first-order dependent type system

(This part is highly inspired of *Advanced Topics in Types and Programming Languages*, chapter 2.)

We extend the syntax of simply typed lambda calculus types with quantification over values, also called **dependent product**.

We write $\tau_1 \rightarrow \tau_2$ for $\forall(x : \tau_1).\tau_2$ if x is free in τ_2.

$$\begin{array}{rcll} t & ::= & x \mid \lambda x : \tau.t \mid t\ t & \text{Terms} \\ \tau & ::= & \alpha \mid \forall x : \tau.\tau \mid \tau\ t & \text{Types} \end{array}$$

Types may not be well-formed

The syntax of types allows ill-formed types to be written. Indeed, term-level variables or type-level variables may not be correctly bound in the context or terms in indices may not be well-typed. We introduce kinds to ensure well-formedness of types.

$$K \quad ::= \quad \star \mid \forall (x : \tau).\, K \quad \text{Kinds}$$

Finally, typing environments now relate type variables to their declared kinds.

$$\Gamma \quad ::= \quad \bullet \mid \Gamma, x : \tau \mid \Gamma, \alpha : K \quad \text{Contexts}$$

Judgments

We need three mutually defined judgments, one for each syntactic category:

$\Gamma \vdash K$	"Kind K is well-formed in context Γ"
$\Gamma \vdash \tau :: K$	"Type τ has kind K in context Γ"
$\Gamma \vdash t : \tau$	"Term t has type τ in context Γ"

Well-formed kinds

$$\frac{}{\Gamma \vdash \star} \qquad\qquad \frac{\Gamma \vdash \tau : \star \qquad \Gamma, x : \tau \vdash K}{\Gamma \vdash \forall (x : \tau).\ K}$$

Kinds must only capture types that denotes **families of types**. In other words, types must only be parameterized by terms.

Well-formed types

The first two rules are straightforward adaptation of rules from STLC:

$$\frac{\alpha :: K \in \Gamma \qquad \Gamma \vdash K}{\Gamma \vdash \alpha :: K} \qquad\qquad \frac{\Gamma \vdash \tau_1 :: \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 :: \star}{\Gamma \vdash \forall (x : \tau_1).\ \tau_2 :: \star}$$

The third rule is trickier since it performs a term substitution in the kind K. This is reminescent of system type application except that a term variable is considered instead of a type variable:

$$\frac{\Gamma \vdash \tau :: \forall (x : \tau').\ K \qquad \Gamma \vdash t : \tau'}{\Gamma \vdash \tau t :: [x \mapsto t]K}$$

Well-typed terms

Here are the typing rules for simply typed lambda calculus:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$$

Well-typed terms

Here is how type dependencies of values can be tracked down:

$$\frac{x : \tau \in \Gamma \qquad \Gamma \vdash \tau :: \star}{\Gamma \vdash x : \tau} \qquad\qquad \frac{\Gamma \vdash \tau_1 :: \star \qquad \Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).t : \forall(x : \tau_1).\tau_2}$$

$$\frac{\Gamma \vdash t_1 : \forall(x : \tau_1).\tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]\tau_2}$$

There are two main differences:

- Extra checks ensure that type annotations make sense.
- An application of an effective argument to a function is recorded in the type of this application by an explicit instantiation of the function output type with this effective argument.

Example

Let us define the following context Γ:

$$\begin{array}{rcl}
\alpha & :: & \star \\
\mathsf{x} & : & \alpha \\
\mathsf{nil} & : & \mathsf{vec}\ \mathsf{zero} \\
\mathsf{cons} & : & \forall (n : nat).\ \mathsf{vec}\ n \to \alpha \to \mathsf{vec}\ (\mathsf{succ}\ n) \\
\mathsf{hd} & : & \forall (n : nat).\ \mathsf{vec}\ (\mathsf{succ}\ n) \to \alpha \\
\mathsf{concat} & : & \forall (n\ m : nat).\ \mathsf{vec}\ n \to \mathsf{vec}\ m \to \mathsf{vec}\ (\mathsf{add}\ n\ m)
\end{array}$$

. . . and the abbreviations $S \equiv \mathsf{cons}\ \mathsf{x}\ \mathsf{nil}$, $1 \equiv \mathsf{succ}\ \mathsf{zero}$, $2 \equiv \mathsf{succ}\ 1$, then:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash \mathsf{hd}\ 1 : [n \mapsto 1](\mathsf{vec}\ (\mathsf{succ}\ n) \to \alpha)} \qquad \dfrac{\ldots}{\Gamma \vdash \mathsf{concat}\ S\ S : [m \mapsto 1][n \mapsto 1](\mathsf{vec}\ (\mathsf{add}\ n\ m))}}{\Gamma \vdash \mathsf{hd}\ 1\ (\mathsf{concat}\ S\ S) : \alpha}$$

Example

Let us define the following context Γ:

$$\begin{array}{rcl}
\alpha & :: & \star \\
\mathsf{x} & : & \alpha \\
\mathsf{nil} & : & \mathsf{vec\ zero} \\
\mathsf{cons} & : & \forall(n : nat).\ \mathsf{vec}\ n \rightarrow \alpha \rightarrow \mathsf{vec\ (succ}\ n) \\
\mathsf{hd} & : & \forall(n : nat).\ \mathsf{vec\ (succ}\ n) \rightarrow \alpha \\
\mathsf{concat} & : & \forall(n\ m : nat).\ \mathsf{vec}\ n \rightarrow \mathsf{vec}\ m \rightarrow \mathsf{vec\ (add}\ n\ m)
\end{array}$$

...and the abbreviations $S \equiv \mathsf{cons\ x\ nil}$, $1 \equiv \mathsf{succ\ zero}$, $2 \equiv \mathsf{succ}\ 1$, then:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash \mathsf{hd}\ 1 : \mathsf{vec}\ 2} \qquad \dfrac{\ldots}{\Gamma \vdash \mathsf{concat}\ S\ S : \mathsf{vec\ (add}\ 1\ 1)}}{\Gamma \vdash \mathsf{hd}\ 1\ (\mathsf{concat}\ S\ S) : \alpha}$$

Example

Let us define the following context Γ:

$$\begin{array}{rcl}
\alpha & :: & \star \\
\mathsf{x} & : & \alpha \\
\mathsf{nil} & : & \mathsf{vec\ zero} \\
\mathsf{cons} & : & \forall(n : nat).\ \mathsf{vec}\ n \to \alpha \to \mathsf{vec}\ (\mathsf{succ}\ n) \\
\mathsf{hd} & : & \forall(n : nat).\ \mathsf{vec}\ (\mathsf{succ}\ n) \to \alpha \\
\mathsf{concat} & : & \forall(n\ m : nat).\ \mathsf{vec}\ n \to \mathsf{vec}\ m \to \mathsf{vec}\ (\mathsf{add}\ n\ m)
\end{array}$$

…and the abbreviations $S \equiv \mathsf{cons\ x\ nil}$, $1 \equiv \mathsf{succ\ zero}$, $2 \equiv \mathsf{succ}\ 1$, then:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash \mathsf{hd}\ 1 : \mathsf{vec}\ 2} \qquad \dfrac{\dfrac{\ldots}{\Gamma \vdash \mathsf{concat}\ S\ S : \mathsf{vec}\ (\mathsf{add}\ 1\ 1)} \qquad \dfrac{\ldots}{\mathsf{vec}\ (\mathsf{add}\ 1\ 1) \equiv \mathsf{vec}\ 2}}{\Gamma \vdash \mathsf{concat}\ S\ S : \mathsf{vec}\ 2}}{\Gamma \vdash \mathsf{hd}\ 1\ (\mathsf{concat}\ S\ S) : \alpha}$$

We also need **conversion rules**.

Conversion rules for types and terms

Since kinds also refer to values, a conversion rule for types is necessary.

$$\frac{\Gamma \vdash \tau :: K' \qquad \Gamma \vdash K \equiv K'}{\Gamma \vdash \tau :: K} \qquad\qquad \frac{\Gamma \vdash t : \tau' \qquad \Gamma \vdash \tau \equiv \tau' :: \star}{\Gamma \vdash t : \tau}$$

Which equivalence between types (and kinds)?

Ideally, leibniz equality would be the best choice. Yet, allowing any provable equality to be used in the conversion would make type-checking undecidable.

Another less expressive equality is the definitional equality (also called intensional equality). As $\beta\eta-$equivalence of terms is decidable in simply typed lambda calculus, it can be internalized inside the conversion rule.

We will formalize such a type equivalence in a typed setting. Our judgment for types will carry a kind and our judgment for terms will carry a type.

Kind equivalence

Kind equivalence is similar to the equality of first-order terms with binders modulo equivalence of type annotations.

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \star \qquad \Gamma, x : \tau_1 \vdash K_1 \equiv K_2}{\Gamma \vdash \forall(x : \tau_1).\ K_1 \equiv \forall(x : \tau_2).\ K_2}$$

$$\frac{\Gamma \vdash K}{\Gamma \vdash K \equiv K} \qquad \frac{\Gamma \vdash K_2 \equiv K_1}{\Gamma \vdash K_1 \equiv K_2} \qquad \frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3}$$

Type equivalence

Type equivalence is similar to equality between first-order term with binders modulo term equivalence on type family indices.

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \forall(x : \tau).\ K \qquad \Gamma, t_1 \equiv t_2 :: \tau}{\Gamma \vdash \tau_1 t_1 \equiv \tau_2 t_2 :: [x \mapsto t_1]K}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' :: \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 \equiv \tau_2' :: \star}{\Gamma \vdash \forall(x : \tau_1).\tau_2 \equiv \forall(x : \tau_1').\tau_2' :: \star}$$

$$\frac{\Gamma \vdash \tau :: K}{\Gamma \vdash \tau \equiv \tau :: K} \qquad \frac{\Gamma \vdash \tau' \equiv \tau :: K}{\Gamma \vdash \tau \equiv \tau' :: K} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: K \qquad \Gamma \vdash \tau_2 \equiv \tau_3 :: K}{\Gamma \vdash \tau_1 \equiv \tau_3 :: K}$$

In the rule for applications, notice that we have to apply a substitution to the kind, just as we did in the typing rule for application.

Term equivalence : structural rules

One subset of the type equivalence rules focuses on the syntactic equivalence between types. Again, this is very similar to first order equality in presence of binders.

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \star \qquad \Gamma, x : \tau_1 \vdash t_1 \equiv t_2 : \tau}{\Gamma \vdash \lambda(x : \tau_1).t_1 \equiv \lambda(x : \tau_2).t_2 : \forall(x : \tau_1).\tau}$$

$$\frac{\Gamma \vdash t_1 \equiv t_1' : \forall(x : \tau_1).\tau_2 \qquad \Gamma \vdash t_2 \equiv t_2' : \tau_1}{\Gamma \vdash t_1 t_2 \equiv t_1' t_2' : [x \mapsto t_2]\tau_2}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t \equiv t : \tau} \qquad \frac{\Gamma \vdash t' \equiv t : \tau}{\Gamma \vdash t \equiv t' : \tau} \qquad \frac{\Gamma \vdash t_1 \equiv t_2 : \tau \qquad \Gamma \vdash t_2 \equiv t_3 : \tau}{\Gamma \vdash t_1 \equiv t_3 : \tau}$$

Term equivalence : reduction-aware rules

$\beta-$equivalence is internalized in term equivalence by the following rule:

$$\frac{\Gamma, x : \tau_1 \vdash t_1 : \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (\lambda(x : \tau_1).t_1)t_2 \equiv [x \mapsto t_2]t_1 : [x \mapsto t_2]\tau_2}$$

η-equivalence by the other one:

$$\frac{\Gamma \vdash t : \tau_2 \qquad x \notin FV(t)}{\Gamma \vdash \lambda(x : \tau_1).t\ x \equiv t : \forall(x : \tau_1).\tau_2}$$

How to turn these rules into a type-checking algorithm?

Many of these rules are **not** syntax-directed. Do you have an idea of an equivalent syntax-directed formalization of dependent types?

How to turn these rules into a type-checking algorithm?

Many of these rules are **not** syntax-directed. Do you have an idea of an equivalent syntax-directed formalization of dependent types?

As usual, we have to remove the non syntax-directed rules and try to inline their "essence" in strategic rules in order to prove that they are admissible in the syntax-directed system.

Algorithmic kind formation

The rules for kinds are unchanged.

$$\frac{\Gamma \vdash_a \tau :: \star \qquad \Gamma, x : \tau \vdash_a K}{\Gamma \vdash_a \forall (x : \tau).\ K} \qquad\qquad \frac{}{\Gamma \vdash_a \star}$$

Algorithmic kinding

As promised, we remove the conversion rule of types since it was not syntax-directed. It is inlined in the rule for the formation of type family.

$$\frac{\alpha :: K \in \Gamma}{\Gamma \vdash_a \alpha :: K} \qquad \frac{\Gamma \vdash_a \tau_1 :: \star \qquad \Gamma, (x : \tau_1) \vdash_a \tau_2 :: \star}{\Gamma \vdash_a \forall(x : \tau_1).\tau_2 :: \star}$$

$$\frac{\Gamma \vdash_a \tau :: \forall(x : \tau').\ K \qquad \Gamma \vdash_a t : \tau'' \qquad \Gamma \vdash_a \tau' \equiv \tau''}{\Gamma \vdash_a \tau t : [x \mapsto t]K}$$

Algorithmic typing

We use the same idea to remove the conversion rule for terms.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_a x : \tau} \qquad \frac{\Gamma \vdash_a \tau_1 :: \star \qquad \Gamma, (x : \tau_1) \vdash_a t : \tau_2}{\Gamma \vdash_a \lambda(x : \tau_1).t : \forall(x : \tau_1).\tau_2}$$

$$\frac{\Gamma \vdash_a t_1 : \forall(x : \tau_1).\tau_2 \qquad \Gamma \vdash_a t_2 : \tau_1' \qquad \Gamma \vdash_a \tau_1 \equiv \tau_1'}{\Gamma \vdash_a t_1 t_2 : [x \mapsto t_2]\tau_2}$$

Algorithmic kind equivalence

We remove non syntax-directed rules, that is reflexivity, transitivity and symmetry.

$$\frac{}{\Gamma \vdash_a \star \equiv \star} \qquad \frac{\Gamma \vdash_a \tau_1 \equiv \tau_2 \qquad \Gamma, (x : \tau_1) \vdash_a K_1 \equiv K_2}{\Gamma \vdash_a \forall (x : \tau_1).\ K_1 \equiv \forall (x : \tau_2).\ K_2}$$

Algorithmic type equivalence

Again, we remove non syntax-directed rules.

$$\frac{}{\Gamma \vdash_a \alpha \equiv \alpha} \qquad \frac{\Gamma \vdash_a \tau_1 \equiv \tau_1' \qquad \Gamma, (x : \tau_1) \vdash_a \tau_2 \equiv \tau_2'}{\Gamma \vdash_a \forall(x : \tau_1).\tau_2 \equiv \forall(x : \tau_1').\tau_2'}$$

$$\frac{\Gamma \vdash_a \tau_1 \equiv \tau_2 \qquad \Gamma \vdash_a t_1 \equiv t_2}{\Gamma \vdash_a \tau_1 t_1 \equiv \tau_2 t_2}$$

How to check for the equivalence of terms?

The most direct way of testing for the equivalence of two terms is to normalize them and compare their normal forms. Fortunately, the existence of a unique normal form follows from the fact that the reduction relation of this programming language is **strongly normalizing** and **confluent** given the following definition of strong β-reduction:

$$\frac{t_1 \rightarrow_\beta t_2}{\lambda x : \tau_1 . t_1 \rightarrow_\beta \lambda x : \tau_1 . t_2} \qquad \frac{t_1 \rightarrow_\beta t_1'}{t_1\ t_2 \rightarrow_\beta t_1'\ t_2} \qquad \frac{t_2 \rightarrow_\beta t_2'}{t_1\ t_2 \rightarrow_\beta t_1\ t_2'}$$

$$\frac{}{(\lambda (x : \tau_1) . t_1) t_2 \rightarrow_\beta [x \mapsto t_2] t_1}$$

However, equivalence of terms can be decided more efficiently by alternating some steps of reduction and the verification that the two terms can coincide. In case of failure, this algorithm will respond faster.

Weak head normal forms

Weak head reduction only applies $\beta-$reduction in the head position:

$$\frac{t_1 \rightarrow_{wh} t_1'}{t_1\ t_2 \rightarrow_{wh} t_1'\ t_2} \qquad \frac{}{(\lambda(x:\tau_1).t_1)t_2 \rightarrow_{wh} [x \mapsto t_2]t_1}$$

Theorem

If $\Gamma \vdash t : \tau$, then there exists a unique term t', written $whnf(t)$ such that $t \rightarrow^{\star}_{wh} t' \not\rightarrow_{wh}$.

Algorithmic term equivalence

$$\frac{\Gamma \vdash_a whnf(t_1) \equiv_{wh} whnf(t_2)}{\Gamma \vdash_a t_1 \equiv t_2} \qquad \frac{}{\Gamma \vdash_a x \equiv_{wh} x}$$

$$\frac{\Gamma, x : \tau \vdash_a t_1 \equiv t_2}{\Gamma \vdash_a \lambda(x : \tau).t_1 \equiv_{wh} \lambda(x : \tau).t_2} \qquad \frac{\Gamma \vdash_a t_1 \equiv_{wh} t_1' \qquad \Gamma \vdash_a t_2 \equiv_{wh} t_2'}{\Gamma \vdash_a t_1 t_2 \equiv_{wh} t_1' t_2'}$$

$$\frac{\Gamma, x : \tau \vdash_a t_1\ x \equiv t_2}{\Gamma \vdash_a t_1 \equiv_{wh} \lambda(x : \tau).t_2} \quad t_1 \text{ is not a lambda abstraction.}$$

$$\frac{\Gamma, x : \tau \vdash_a t_1 \equiv t_2\ x}{\Gamma \vdash_a \lambda(x : \tau).t_1 \equiv_{wh} t_2} \quad t_2 \text{ is not a lambda abstraction.}$$

Show that $(\lambda f.\lambda x.f\ x)(\lambda x.x) \equiv \lambda x.x$ and
that $(\lambda f.\lambda x.f\ x\ x)(\lambda x.x) \not\equiv (\lambda f.f)(\lambda x.x)$

Termination of type-checking

Theorem

If $\Gamma \vdash t_1 : \tau_1$ and $\Gamma \vdash t_2 : \tau_2$, then the backwards search for a derivation of $\Gamma \vdash_a t_1 \equiv t_2$ always terminates.

Proof.

By using a lexical order that first considers the length of reduction sequences and then the size of terms. $\square$

Soundness of algorithmic type-checking

Theorem (Soundness)

1. *If* $\Gamma \vdash_a K$ *holds then* $\Gamma \vdash K$ *holds.*
2. *If* $\Gamma \vdash_a \tau :: K$ *holds then* $\Gamma \vdash \tau :: K$ *holds.*
3. *If* $\Gamma \vdash_a t : \tau$ *holds then* $\Gamma \vdash t : \tau$ *holds.*
4. *If* $\Gamma \vdash_a K$, $\Gamma \vdash_a K'$ *and* $\Gamma \vdash_a K \equiv K'$ *hold then* $\Gamma \vdash K \equiv K'$ *holds.*
5. *If* $\Gamma \vdash_a \tau :: K$, $\Gamma \vdash_a \tau' :: K$ *and* $\Gamma \vdash_a \tau \equiv \tau' :: K$ *hold then* $\Gamma \vdash \tau \equiv \tau' :: K$ *holds.*
6. *If* $\Gamma \vdash_a t : \tau$, $\Gamma \vdash_a t' :: \tau'$ *and* $\Gamma \vdash_a t \equiv t' :: \tau$ *hold then* $\Gamma \vdash t \equiv t' :: \tau$ *holds.*

Proof.

By induction over the derivations of the algorithmic type system. $\square$

Completeness of algorithmic type-checking

Theorem (Completeness)

1. *If* $\Gamma \vdash K$ *holds then* $\Gamma \vdash_a K$ *holds.*
2. *If* $\Gamma \vdash \tau :: K$ *holds then* $\Gamma \vdash_a \tau :: K$ *holds.*
3. *If* $\Gamma \vdash t : \tau$ *holds then* $\Gamma \vdash_a t : \tau$ *holds.*
4. *If* $\Gamma \vdash K$, $\Gamma \vdash K'$ *and* $\Gamma \vdash K \equiv K'$ *hold then* $\Gamma \vdash_a K \equiv K'$ *holds.*
5. *If* $\Gamma \vdash \tau :: K$, $\Gamma \vdash \tau' :: K$ *and* $\Gamma \vdash \tau \equiv \tau' :: K$ *hold then* $\Gamma \vdash_a \tau \equiv \tau' :: K$ *holds.*
6. *If* $\Gamma \vdash t : \tau$, $\Gamma \vdash t' : \tau'$ *and* $\Gamma \vdash t \equiv t' :: \tau$ *hold then* $\Gamma \vdash_a t \equiv t' : \tau$ *holds.*

Proof.

This is a technical proof based on a logical relation [Coquand, 1996; Harper and Pfenning, 2005]. □

Type preservation

Theorem

If $\Gamma \vdash t : \tau$ and $t \to_\beta t'$ then $\Gamma \vdash t' : \tau$.

Proof.

By the usual syntactic way on the algorithmic type system. □

Sigma types

In a programming language with dependent types, it is natural to pack several values whose types are dependent to each other. For instance, it is useful to pack a vector and its length together. This packaging can be done using a **dependent pair data constructor** whose second component's type refers to the value of the first component.

We extend the syntax of terms with typed pairs and the two related projections:

$$t ::= \ldots \mid (t, t : \Sigma x : \tau.\tau) \mid t.1 \mid t.2$$

and also the syntax of types:

$$\tau ::= \ldots \mid \Sigma x : \tau.\tau$$

Sigma types

Strong β-reduction now reduces inside the components of a pair and projections have their own rules:

$$\frac{}{(t_1, t_2 : \tau).i \rightarrow_\beta t_i} \qquad \frac{t \rightarrow_\beta t'}{t.i \rightarrow_\beta t'.i}$$

$$\frac{t_1 \rightarrow_\beta t_1'}{(t_1, t_2 : \tau) \rightarrow_\beta (t_1', t_2 : \tau)} \qquad \frac{t_2 \rightarrow_\beta t_2'}{(t_1, t_2 : \tau) \rightarrow_\beta (t_1, t_2' : \tau)}$$

Weak $\beta-$reduction is also extended with two extra rules:

$$\frac{}{(t_1, t_2 : \tau).i \rightarrow_{wh} t_i} \qquad \frac{t \rightarrow_{wh} t'}{t.i \rightarrow_{wh} t'.i}$$

Kinding and typing

Sigma types must be checked for well-kindness:

$$\frac{\Gamma \vdash \tau_1 :: \star \qquad \Gamma; (x : \tau_1) \vdash \tau_2 :: \star}{\Gamma \vdash \Sigma x : \tau_1 . \tau_2 :: \star}$$

A dependent pair is assigned a sigma type. Notice the substitution that relates the value of the first component to the type of the second component:

$$\frac{\Gamma \vdash \Sigma x : \tau_1 . \tau_2 :: \star \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma; (x : \tau_1) \vdash t_2 : [x \mapsto t_1]\tau_2}{(t_1, t_2 : \Sigma x : \tau_1 . \tau_2) : \Sigma x : \tau_1 . \tau_2}$$

$$\frac{\Gamma \vdash t : \Sigma x : \tau_1 . \tau_2}{\Gamma \vdash t.1 : \tau_1} \qquad\qquad \frac{\Gamma \vdash t : \Sigma x : \tau_1 . \tau_2}{\Gamma \vdash t.2 : [x \mapsto t.1]\tau_2}$$

Term equivalence

Reduction rules for projection must be taken into account by the equivalence:

$$\frac{\Gamma \vdash \Sigma x : \tau_1 . \tau_2 :: \star \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : [x \mapsto t_1]\tau_2}{\Gamma \vdash (t_1, t_2 : \Sigma x : \tau_1 . \tau_2).1 \equiv t_1 : \tau_1}$$

$$\frac{\Gamma \vdash \Sigma x : \tau_1 . \tau_2 :: \star \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : [x \mapsto t_1]\tau_2}{\Gamma \vdash (t_1, t_2 : \Sigma x : \tau_1 . \tau_2).2 \equiv t_2 : [x \mapsto t_1]\tau_2}$$

From two projections of a term t, it is possible to form t using the constructor for pairs. This rule is called "Surjective pairing".

$$\frac{\Gamma \vdash t : \Sigma x : \tau_1 . \tau_2}{\Gamma \vdash (t.1, t.2 : \Sigma x : \tau_1 . \tau_2) \equiv t : \Sigma x : \tau_1 . \tau_2}$$

Algorithmic kinding and typing

The rule to check for the well-kindness of a type is very similar to the rule for the product:

$$\frac{\Gamma \vdash_a \tau_1 :: \star \qquad \Gamma; (x : \tau_1) \vdash_a \tau_2 :: \star}{\Gamma \vdash_a \Sigma x : \tau_1 . \tau_2 :: \star}$$

To type-check a dependent pair, we first compute the type τ_1' of the left component. We check that it is compatible with the expected type τ_1 for x. Then, t_1 can be used safely to instanciate x in τ_2.

$$\frac{\begin{array}{c}\Gamma \vdash_a \Sigma x : \tau_1 . \tau_2 :: \star \qquad \Gamma \vdash_a t_1 : \tau_1' \qquad \Gamma; (x : \tau_1) \vdash_a t_2 : \tau_2' \\ \Gamma \vdash_a \tau_1 \equiv \tau_1' \qquad \Gamma \vdash_a [x \mapsto t_1]\tau_2 \equiv \tau_2'\end{array}}{\Gamma \vdash_a (t_1, t_2 : \Sigma x : \tau_1 . \tau_2) : \Sigma x : \tau_1 . \tau_2}$$

$$\frac{\Gamma \vdash_a t : \Sigma x : \tau_1 . \tau_2}{\Gamma \vdash_a t.1 : \tau_1} \qquad \frac{\Gamma \vdash_a t : \Sigma x : \tau_1 . \tau_2}{\Gamma \vdash_a t.2 : [x \mapsto t.1]\tau_2}$$

Algorithmic type and term equivalence

Two dependent sum types are equivalent if the two types of the first component are compatible as well as the two types of the second component.

$$\frac{\Gamma \vdash_a \tau_1 \equiv \tau_1' \qquad \Gamma; (x : \tau_1) \vdash_a \tau_2 \equiv \tau_2'}{\Gamma \vdash_a \Sigma x : \tau_1.\tau_2 \equiv \Sigma x : \tau_1'.\tau_2'}$$

To check for equivalence between two dependent pairs, it is sufficient to check for the equivalence of each component separately. The equivalence between the types is implied by this equivalence.

$$\frac{\Gamma \vdash_a t_i \equiv t_i'}{\Gamma \vdash_a (t_1, t_2 : \tau) \equiv_{wh} (t_1', t_2' : \tau')}$$

$$\frac{\Gamma \vdash_a t_i \equiv t.i \qquad t \text{ is not a pair}}{\Gamma \vdash_a (t_1, t_2 : \tau) \equiv_{wh} t} \qquad \frac{\Gamma \vdash_a t.i \equiv t_i \qquad t \text{ is not a pair}}{\Gamma \vdash_a t \equiv_{wh} (t_1, t_2 : \tau)}$$

Pure Type Systems

As you have probably noticed, when a new construction is added into the syntax, a lot of rules have to be written to define well-formedness and equivalence at the level of types and at the level of terms. Fortunately, it is possible to reformulate the same type system in a syntax that comprises both types and terms.

In a Pure Type System [Barendregt, 1991], the syntax of terms is:

$$\begin{array}{rcl} \tau, t & ::= & s \mid x \mid \lambda x : t.t \mid t\ t \mid \forall x : t.t \\ s & ::= & \square \mid \star \\ \Gamma & ::= & \bullet \mid \Gamma; x : t \end{array}$$

The sorts s are used to discriminate between types and terms.

Typing rules

The following set of typing rules is parameterized by the products that are allowed at the level of types.

$$\Gamma \vdash \star : \square \qquad \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash \tau_1 : s_i \qquad \Gamma; x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \forall (x : \tau_1). \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \forall (x : \tau_1). \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \ t_2 : [x \mapsto t_2]\tau_2} \qquad \frac{\Gamma \vdash \tau_1 : s_i \qquad \Gamma; (x : \tau_1) \vdash \tau_2 : s_j}{\Gamma \vdash \forall (x : \tau_1). \tau_2 : s_j}$$

$$\frac{\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma \vdash \tau_2 : s}{\Gamma \vdash t : \tau_2}$$

where $(s_i, s_j) \in \{(\star, \star), (\star, \square)\}$

The lambda cube[1]

$$\begin{tikzcd}
 & F_\omega \arrow[rr] & & \lambda\mathrm{C} \\
F \arrow[ur] \arrow[rr] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda_{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

System	*Authorized products*
Simply typed lambda calculus	$(\star, \star)$
First order dependent types	$(\star, \star), (\star, \Box)$
System F	$(\star, \star), (\Box, \star)$
System F^ω	$(\star, \star), (\Box, \star), (\Box, \Box)$
Calculus of Construction	$(\star, \star), (\star, \Box), (\Box, \star), (\Box, \Box)$

[1]Drawing from Gilles Dowek's lecture notes.

Contents

Towards a dependently-typed programming language

There are a lot of active research on the metatheory of Pure Type Systems [Siles and Herbelin, 2010] because, despite of their apparent simplicity, a lot of technical details are very hard to figure out. As a consequence, extending them to handle **stateful computation or non termination**, is a challenge.

Currently, in the programming language community, this extension is considered from two different points of view:

- Curry-Howard centric: [Miquel; Nanevski et al., 2008; Lebresne, 2008]
  Extend the logic to cope with side-effects.
  What is a program with side-effects proving?
- Programming centric [Augustsson, 1998; Chen and Xi, 2005]:
  Assume a programming language with side-effets. How to prevent the impure terms to pollute the (logical) types?
  (This is required for the decidability of type-checking.)

Contents

Restricting type families

A direct way of preventing impure terms to flow into types is **to restrict the syntax** of type family application to have indexes only range in a set of pure terms.

What are these pure terms exactly? These are terms that are harmless to the decidability of type-checking. Non terminating terms are clearly impure. How to forbid them syntactically in types while preserving the substitutivity of typing?

From that perspective, the application rule is the most problematic in λLF:

$$\frac{\Gamma \vdash t_1 : \forall (x : \tau_1).\tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]\tau_2}$$

How to restrict the allowed t_2 here?

Restricting type families

A direct way of preventing impure terms to flow into types is **to restrict the syntax** of type family application to have indexes only range in a set of pure terms.

What are these pure terms exactly? These are terms that are harmless to the decidability of type-checking. Non terminating terms are clearly impure. How to forbid them syntactically in types while preserving the substitutivity of typing?

From that perspective, the application rule is the most problematic in λLF:

$$\frac{\Gamma \vdash t_1 : \forall (x : \tau_1).\tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]\tau_2}$$

How to restrict the allowed t_2 here? Forbid functions! They may hide non terminating terms!

A syntactic class for type family index

Dependent ML [Xi and Pfenning, 1999] introduces two distinct applications and abstractions.

The first, dependent, application forces the argument to have an **index sort** I:

$$\begin{array}{rcl} I & ::= & \mathrm{int} \mid \{x : \mathrm{int} \mid P\} \\ P & ::= & P \wedge P \mid i \leq i \\ i & ::= & x \mid q \mid q.i \mid i + i \\ & & \text{where } q \in \mathbb{Z} \end{array}$$

The syntax for terms is:

$$t ::= x \mid \lambda(x : I).t \mid \lambda(x : \tau).t \mid t\ [i] \mid t\ t$$

Any impure constant can be added at the level of terms (fixpoint combinator, references, ...) as long as it is not present in types.

Type-checking in DML

The dependent products are restricted to quantification over index sorts:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \forall(x : I).\ \tau \mid \tau[i] \mid \tau \rightarrow \tau \\ K & ::= & \star \mid \forall(x : I).\ K \end{array}$$

Most of the rules can be imported from simply typed λ-calculus. An interesting one deals with application of an index to a term:

$$\frac{\Gamma \vdash t : \forall(x : I).\tau \qquad \Gamma \models i : I}{\Gamma \vdash t[i] : [x \mapsto i]\tau}$$

In order to validate $\Gamma \models i : I$, a solver for linear arithmetic is necessary.

How to relate dynamic terms to static values?

In this setting, integers are only present at the level of index sorts. In order to handle *runtime integers*, another syntactic class of terms must be introduced.

$$\begin{array}{rcl} t & ::= & \ldots \mid k \mid \ldots \\ & & \text{with } k \in \mathbb{Z} \end{array}$$

As with GADTs, we relate these runtime integers with static integers using **singleton types** of the form "$Int(i)$", representing for each static integer i, the unique dynamic representation of i.

How to populate typing environments with assumptions about these integers?

How to relate dynamic tests to static assumptions?

When we are dynamically comparing an integer with another one, it would be nice to have the type system aware of the conclusion of this test.

It is possible to define special booleans tailored to that purpose:

$$\begin{array}{rcl} \mathsf{true} & : & \forall (x : \{x : \mathsf{int} \mid 1 \leq x\}).\ \mathsf{Bool}\ (x) \\ \mathsf{false} & : & \forall (x : \{x : \mathsf{int} \mid x \leq 0\}).\ \mathsf{Bool}\ (x) \\ \mathsf{leq} & : & \forall (x, y : \mathsf{int}).\ \mathsf{Int}\ (x) \rightarrow \mathsf{Int}\ (y) \rightarrow \mathsf{Bool}\ (1 + y - x) \end{array}$$

Then, it is straighforward to devise a special conditional expression to lift dynamic information to the static world:

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool}\ (i) \qquad \Gamma, 1 \leq i \vdash t_2 : \tau \qquad \Gamma, i \leq 0 \vdash t_3 : \tau}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \tau}$$

Benefits of DML

Even if DML only focuses on a very restricted set of pure terms, a lot of useful properties can be internalized inside the equivalence relation that are hard to handle with a definitional equality.

For instance, the commutativity of the addition or the neutrality of 0 with respect to addition are automatically taken into account by the type-checker.

Example (in an imaginary O'Caml with dependent types)

If we come back to the example of concatenation between two vectors indexed by their lengths. We have seen that a dependently typed programming language can type-check the following program:

```
let rec concat
: forall α. forall (n m : nat).
  vector n α → vector m α → vector (n + m) α
= fun n m v1 v2 →
  match v1 with
  | Nil → v2
  | Cons [h] (x, xs) → Cons [h + m] (x, concat h xs v2)
```

because definitional equality is able to show that $0 + n$ is convertible to n and $succ\ (h + m)$ is convertible to $(succ\ h) + m$. Yet, does it still work if we had done the recursion over `v2` instead of `v1`?

Example (in an imaginary O'CAML with dependent types)

No!

```
let rec concat
: forall α. forall (n m : nat).
  vector n α → vector m α → vector (n + m) α
= fun n m v1 v2 →
  match v2 with
  | Nil → v2
  | Cons [h] (x, xs) → Cons [h + m] (x, concat h xs v2)
```

because this time, $n + 0$ is not convertible n neither $succ\ (n + h)$ is convertible to $n + (succ\ h)$.

In DML, there is no distinction between the provability of $n + 0 = n$ and $0 + n = n$. Thus, the two versions of this program would be well-typed.

Extending the DML approach

The fundamental idea behind Dependent ML is the respect of the **phase distinction**, a clear separation between a compile-time and a run-time. The (so-called) *dynamic* terms of the programming language and the *static* terms of the language of types are syntactically disjoint. Singleton types are the only way to relate a static terms to a dynamic terms.

The ATS [Chen and Xi, 2005] programming language generalizes Dependent ML with GADTs, which makes possible the **dynamic representation of proof evidence** that some property over static values is satisfied. Eventually, ATS ends up to be both a programming language with effectful operations and a theorem proving language manipulating proof terms. This idea is also used in the Concoqtion language, a mix of O'Caml and Coq[Fogarty et al., 2007].

CoqMT [Strub, 2010] is an implementation of Coq that internalizes decision procedures for first order theories in the conversion rule, which makes more dependently-typed programs typable.

Weakness of ATS

The idea of preserving phase distinction thanks to a syntactic separation between static and dynamic terms allows effectful computations in the programming language.

However, this feature implies a syntactic heaviness because, for each runtime formal argument, two abstractions are needed to introduce both the runtime value and its static counterpart:

```
(* In ATS, "int" is both a sort and a type constructor *)
(* of type "int -> type".  *)
typedef Nat = [n: int | n ≥ 0] int n
(* [n:int | n >= 0] is an existential quantification.  *)

(* "nat" is the sort [n:int | n >= 0].  *)
fun fact2 {n: nat} (x:  int n):  Nat =
  if x > 0 then x nmul fact2 (x-1) else 1
(* assuming nmul has type "(Nat, Nat) -> Nat".  *)
```

Open issues related to dependently typed programming

Current dependently-typed programming languages suffer several defects:

- **Effectful computations**
- **Modularity**
- **Embarrassing proof terms**
- **Fragile proof terms**

Open issues related to dependently typed programming

Current dependently-typed programming languages suffer several defects:

- **Effectful computations** : It is not clear how to deal with non terminating or stateful computation. Should we use monadic-style encoding or effect system to encompass terms that may jeopardize the logic consistency?
- **Modularity**
- **Embarrassing proof terms**
- **Fragile proof terms**

Open issues related to dependently typed programming

Current dependently-typed programming languages suffer several defects:

- **Effectful computations**
- **Modularity** : What would be the *right* way of indexing a datatype? In a standard library, should we index the type of lists only by their length or should we also index them by the set of the elements they contain? Is there a most general way of indexing types? Should we index types at all?
- **Embarrassing proof terms**
- **Fragile proof terms**

Open issues related to dependently typed programming

Current dependently-typed programming languages suffer several defects:

- **Effectful computations**
- **Modularity**
- **Embarrassing proof terms** : Adjoining a proof term of $(P\ v)$ to a value v can be convenient because it makes the proof to be conveyed with the value and the property P available everywhere v is. Yet, it can also be an encumbrance because equality between values must implement proof-irrelevance. Furthermore, it is not always simple to define an erasure of the purely logical contents of a term.
- **Fragile proof terms**

Open issues related to dependently typed programming

Current dependently-typed programming languages suffer several defects:

- **Effectful computations**
- **Modularity**
- **Embarrassing proof terms**
- **Fragile proof terms** : When a proof term is interlaced with a program, it should be written in a way that is compatible with the efficient reduction strategy used for programs (like call-by-value for instance). The resulting proof term is highly dependent on the way the program is written. If the program is changed, even if the proof-term should prove the same theorem, it must often be changed too.

Bibliography I

▷ Lennart Augustsson. Cayenne, a language with dependent types. *SIGPLAN Not.*, 34:239–250, September 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/291251.289451.

Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005.

Thierry Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

▷ Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: 10.1145/1244381.1244400.

Bibliography II

▷ Robert Harper and Frank Pfenning. On equivalence and canonical forms in the lf type theory. *ACM Trans. Comput. Logic*, 6:61–101, January 2005. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1042038.1042041.

Sylvain Lebresne. A system f with call-by-name exceptions. In Luca Aceto, Ivan Damgård, Leslie Goldberg, Magnús Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 323–335. Springer Berlin / Heidelberg, 2008.

James McKinna. Why dependent types matter. In *POPL*, page 1, 2006.

▷ Alexandre Miquel. Classical modal realizability and side effects.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 229–240, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: http://doi.acm.org/10.1145/1411204.1411237.

Bibliography III

Vincent Siles and Hugo Herbelin. Equality is typable in semi-full pure type systems. In *Proceedings, 25th Annual IEEE Symposium on Logic in Computer Science (LICS '10), Edinburgh, UK, 11-14 July 2010*. IEEE Computer Society Press, 2010.

▷ Pierre-Yves Strub. Coq Modulo Theory. *19th EACSL Annual Conference on Computer Science Logic*, 08 2010.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.