

$\Pi\Sigma$: Dependent Types Without the Sugar

Thorsten Altenkirch¹, Nils Anders Danielsson¹,
Andres Löh², and Nicolas Oury³

¹ School of Computer Science, University of Nottingham

² Institute of Information and Computing Sciences, Utrecht University

³ Division of Informatics, University of Edinburgh

Abstract. The recent success of languages like Agda and Coq demonstrates the potential of using dependent types for programming. These systems rely on many high level features like datatype definitions, pattern matching and implicit arguments to facilitate the use of the language. However, these features complicate the metatheoretical study and are a potential source of bugs.

To address these issues we introduce $\Pi\Sigma$, a dependently typed core language. It is small enough for metatheoretical study and the typechecker is small enough to be formally verified. In this language there is only one mechanism for recursion—used for types, functions and infinite objects—and an explicit mechanism to control unfolding, based on lifted types. Furthermore structural equality is used consistently for values and types; this is achieved by a new notion of α -equality for recursive definitions. We show, by translating several high-level constructions, that $\Pi\Sigma$ is suitable as a core language for dependently typed programming. We also provide a formal description of the language and its typing rules.

1 Introduction

Dependently typed programming offers programmers a flexible path towards formally verified programs and, at the same time, opportunities to increase a programmer’s productivity by offering new ways of structuring programs and proofs (Altenkirch et al. 2005). This is witnessed by the increasing popularity of systems like Agda (Norell 2007), but also the fact that dependently typed *programming* is becoming more popular in the Coq community (Coq Development Team 2009), for instance using some recent extensions (Sozeau 2008). An alternative to moving to full blown dependent types as present in Agda and Coq is to add dependently typed features without giving up a traditional view of the distinction between values and types. This is exemplified by the presence of GADTs in Haskell, and by more experimental systems like Ω mega (Sheard 2005), ATS (Cui et al. 2005), and She (McBride 2009).

Dependently typed languages offer a number of high level features to reduce the complexity of programming in such a rich type discipline and at the same time improve the readability of the code, such as:

Datatype definitions A convenient syntax to define dependently typed families inductively and/or coinductively.

Pattern matching Agda offers a very powerful mechanism for dependently typed pattern matching. To some degree this can be emulated in Coq by using Sozeau’s new tactic Program (2008).

Hidden parameters In dependently typed programs datatypes are often indexed. The index parameters can often be inferred using unification, which means that the programmer does not have to write them out manually. This provides an alternative to polymorphic type inference á la Hindley-Milner.

These features, while important for the usability of dependently typed languages, complicate the metatheoretic study and can be the source of subtle bugs in the type checker. To address such problems, we can use a core language which is small enough to allow metatheoretic study. A verified type checker for the core language can also provide a trusted core in the implementation of a full language.

Coq makes use of a core language, the Calculus of (Co)Inductive Constructions (CCIC, Giménez 1996). However, this calculus is quite complex: it includes the schemes for strictly positive datatype definitions and the accompanying recursion principles. Furthermore, it is unclear whether the advanced features of Agda such as dependently typed pattern matching, the flexible use of mixed induction/coinduction and induction-recursion can be easily translated into CCIC or a similar calculus. (One could argue that a core language is less useful if the translation from the full language is difficult to understand.)

In the present paper we suggest a different approach: we propose a core language that is specifically designed in such a way that we can easily translate the high-level features required for convenient dependently typed programming; on the other hand, we postpone the question of totality. Totality is important for dependently typed programs, partly because non-terminating proofs are not very useful, and partly for reasons of efficiency: if a certain type has at most one total value, then total code of that type does not need to be run at all. However, we believe that it can be beneficial to separate the verification of totality from the functional specification of the code. A future version of our core language may have support for free-standing certificates of totality (and the related notions of positivity and stratification).

The core language proposed in this paper is called $\Pi\Sigma^4$ and is based on a small collection of basic features:

- Dependent function types (Π -types) and dependent product types (Σ -types).
- A (very) impredicative universe of types with **Type : Type**.
- Finite sets (enumerations) using reusable and scopeless labels.
- A general mechanism for mutual recursion, allowing the encoding of advanced concepts such as induction-recursion.
- Lifted types, which are used to control recursion. These types offer a convenient way to represent mixed inductive/coinductive definitions.
- A definitional equality which is structural for all definitions, whether types or programs, enabled by a novel definition of α -equality for recursive definitions.

⁴ The present version is a simplification of a previous implementation, described in an unpublished draft (Altenkirch and Oury 2008).

We have implemented an interactive type checker/interpreter for $\Pi\Sigma$ in Haskell.⁵

Outline of the paper

We introduce $\Pi\Sigma$ by giving examples showing how high-level dependently typed programming constructs can be represented (Sect. 2). We then specify the operational semantics (Sect. 3), develop the equational theory (Sect. 4), and present the type system (Sect. 5). Finally we conclude and discuss further work (Sect. 6).

Related work

We already mentioned the core language CCIC (Giménez 1996). Another dependently typed core language, that of Epigram (Chapman et al. 2006), is basically a framework for implementing a total type theory, based on elimination combinators. Augustsson’s influential language Cayenne (1998) is like $\Pi\Sigma$ a partial language, but is not a core language. The idea to use a partial core language was recently and independently suggested by Coquand et al. (2009), who propose a language called Mini-TT, which is also related to Coquand’s Calculus of Definitions (2008). Mini-TT uses a nominal equality, unlike $\Pi\Sigma$ ’s structural equality, and unfolding of recursive definitions is not controlled explicitly by using lifted types, but by not unfolding inside patterns and sum types.

The use of lifted types is closely related to the use of suspensions to encode non-strictness in strict languages (Wadler et al. 1998).

Acknowledgements

We would like to thank Andreas Abel, Thierry Coquand and Simon Peyton Jones for discussions related to the $\Pi\Sigma$ project. We would also like to thank Darin Morrison, who added Unicode support and implemented an Emacs mode, and members of the Functional Programming Laboratory in Nottingham, who have given feedback on our work.

2 $\Pi\Sigma$ by example

In this section, we first briefly introduce the syntax of $\Pi\Sigma$ (Sect. 2.1). The rest of the section demonstrates how to encode a number of high-level features of dependently typed programming languages in $\Pi\Sigma$: (co)datatypes (Sects. 2.2 and 2.3), equality (Sect. 2.4), families of datatypes (Sect. 2.5), and finally universes (Sect. 2.6).

2.1 Syntax overview

The syntax of $\Pi\Sigma$ is defined as follows:

⁵ The package `pisigma` is available from Hackage (<http://hackage.haskell.org>).

Terms $t, u, \sigma, \tau ::= \mathbf{Type} \mid (x : \sigma) \rightarrow \tau \mid \lambda x \rightarrow t \mid x \mid t \ t$
 $\mid (x : \sigma) * \tau \mid (t, t) \mid \mathbf{split} \ t \ \mathbf{with} \ (x, x) \rightarrow t$
 $\mid \{\bar{l}\} \mid 'l \mid \mathbf{case} \ t \ \mathbf{of} \ \{\bar{l} \rightarrow t^l\} \mid \uparrow \sigma \mid [t] \mid !t \mid \mathbf{let} \ \Gamma \ \mathbf{in} \ t$
 Contexts $\Gamma, \Delta ::= \varepsilon \mid \Gamma; x : t \mid \Gamma; x = t$

While there is no syntactic difference between terms and types, we use the metavariables σ and τ to highlight positions where terms play the role of types. We are writing \bar{a}^s for an s -separated sequence of as . The language supports the following concepts:

Type The type of types is **Type**. Since $\Pi\Sigma$ is partial anyway due to the use of general recursion, we also assume **Type** : **Type**.

Dependent functions We use the same notation as Agda, writing $(x : \sigma) \rightarrow \tau$ for dependent function types.

Dependent products We write $(x : \sigma) * \tau$ for dependent products. Elements are constructed using tuple notation: (t, t) . The eliminator **split** t **with** $(x, y) \rightarrow u$ deconstructs the scrutinee t binding its components to x and y and then evaluates u .

Enumerations Enumerations are finite sets written $\{\bar{l}\}$. The labels l do not interfere with identifiers, can be reused and have no scope. To disambiguate them from identifiers we use $'l$ to construct an element of an enumeration. The eliminator **case** t **of** $\{\bar{l} \rightarrow t^l\}$ analyzes the scrutinee t and chooses the matching branch.

Lifting A lifted type $\uparrow \sigma$ contains boxed terms $[t]$. Definitions are not unfolded inside boxes. If a box is forced using $!$, then evaluation can continue.

Let A program Γ (either top level or local by using **let**) is a context, i. e., a sequence of declarations $x : \sigma$ and (possibly recursive) definitions $x = u$. Definitions and declarations may occur in any order, subject to the following constraints:

- Before a variable can be defined, it must first be declared.
- Every declaration and definition has to type check with respect to the previous declarations and definitions.

To simplify the presentation, $\Pi\Sigma$ —despite being a core language—allows a modicum of syntactic sugar: A non-dependent function type can be written as $\sigma \rightarrow \tau$, a non-dependent product as $\sigma * \tau$ (both ‘ \rightarrow ’ and ‘ $*$ ’ associate to the right). Several variables may be bound at once in λ abstractions $(\lambda x_1 \ x_2 \ \dots \ x_n \rightarrow t)$, function types $((x_1 \ x_2 \ \dots \ x_n : \sigma) \rightarrow \tau)$ and product types $((x_1 \ x_2 \ \dots \ x_n : \sigma) * \tau)$. We can also combine a declaration and a subsequent definition: instead of $x : \sigma; x = t$; we write $x : \sigma = t$.

2.2 Datatypes

$\Pi\Sigma$ does not have a builtin mechanism to define datatypes. Instead, we rely on its more primitive features—finite types, Σ -types and recursion—to model datatypes. As a simple example, consider the declaration of (Peano) natural numbers and addition. We represent *Nat* as a recursively defined Σ -type whose

first component is a tag (*zero* or *suc*), indicating which constructor we are using, and whose second component gives the type of the constructor arguments:

$$Nat : \mathbf{Type} = (l : \{zero\ suc\}) * \mathbf{case} \ l \ \mathbf{of} \ \{zero \rightarrow Unit \mid suc \rightarrow [Nat]\};$$

In the case of *zero* we use a one element type *Unit* which is defined as $Unit : \mathbf{Type} = \{unit\}$. The recursive occurrence of *Nat* is placed inside a box (i. e., $[Nat]$ rather than *Nat*). Boxing prevents infinite unfolding during evaluation. Evaluation is performed while testing type equality, and boxing is often essential to keep the type checker from diverging. Using the above representation we can derive the constructors $zero : Nat = ('zero, 'unit)$ and $suc : Nat \rightarrow Nat = \lambda i \rightarrow ('suc, i)$. Addition can then be defined as follows:

$$\begin{aligned} add &: Nat \rightarrow Nat \rightarrow Nat; \\ add &= \lambda m \ n \rightarrow \mathbf{split} \ m \ \mathbf{with} \ (lm, m') \rightarrow !\mathbf{case} \ lm \ \mathbf{of} \ \begin{cases} zero \rightarrow [n] \\ suc \rightarrow [suc \ (add \ m' \ n)] \end{cases}; \end{aligned}$$

Again we use boxing to stop the infinite unfolding of the recursive call (note that type checking a dependently typed program can involve evaluation under binders). Note that we have to box both branches of the case to satisfy the type checker—they both have type $\uparrow Nat$. Once the variable *lm* gets instantiated with a concrete label the case reduces and the box in the matching case branch gets forced by the '!'. As a consequence, computations like $2 + 1$, i. e., $add \ (suc \ (suc \ zero)) \ (suc \ zero)$, evaluate correctly—in this case to $('suc, ('suc, ('suc, ('zero, 'unit))))$.

2.3 Codata

The type *Nat* is an eager datatype, corresponding to an inductive definition. In particular, it does not make sense to write the following definition:

$$omega : Nat = ('suc, omega);$$

Here the recursive occurrence is not guarded by a box and hence *omega* will simply diverge if evaluation to normal form is attempted. To define a lazy or coinductive type like the type of streams we have to use lifting ($\uparrow \dots$) explicitly in the type definition:

$$Stream : \mathbf{Type} \rightarrow \mathbf{Type} = \lambda A \rightarrow A * [\uparrow(Stream \ A)];$$

We can now define programs by *corecursion*. As an example we define *from*, a function that creates streams of increasing numbers:

$$\begin{aligned} from &: Nat \rightarrow Stream \ Nat; \\ from &= \lambda n \rightarrow (n, [from \ ('suc, n)]); \end{aligned}$$

The type system forces us to protect the recursive occurrence with a box. Evaluation of *from zero* terminates with $(zero, \mathbf{let} \ n : Nat = zero \ \mathbf{in} \ [from \ ('suc, n)])$.

The use of lifting to indicate corecursive occurrences allows a large flexibility in defining datatypes. In particular, it facilitates the definition of mixed inductive/coinductive types such as the type of stream processors (Hancock et al. 2009), a concrete representation of functions on streams:

$$\begin{aligned}
SP &: \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \mathbf{Type}; \\
SP &= \lambda a b \rightarrow (l : \{ get\ put \}) * \mathbf{case}\ l \mathbf{of} \{ get \rightarrow [a \rightarrow SP\ a\ b] \\
&\quad | put \rightarrow [b * \uparrow(SP\ a\ b)]];
\end{aligned}$$

The basic idea of stream processors is that we can only perform a finite amount of *gets* before issuing the next of infinitely many *puts*. As an example, we define the identity stream processor corecursively:

$$idsp : (A : \mathbf{Type}) \rightarrow SP\ A\ A = \lambda A \rightarrow ('get, \lambda a \rightarrow ('put, (a, [idsp\ A])));$$

We can use mixed recursion/corecursion to define the semantics of stream processors in terms of functions on streams:

$$\begin{aligned}
eval &: (A\ B : \mathbf{Type}) \rightarrow SP\ A\ B \rightarrow Stream\ A \rightarrow Stream\ B; \\
eval &= \lambda A\ B\ sp\ as \rightarrow \mathbf{split}\ sp \mathbf{with}\ (lsp, asp) \rightarrow !\mathbf{case}\ lsp \mathbf{of} \\
&\quad \{ get \rightarrow \mathbf{split}\ as \mathbf{with}\ (a, as') \rightarrow [eval\ A\ B\ (asp\ a)\ (!as')] \\
&\quad | put \rightarrow \mathbf{split}\ asp \mathbf{with}\ (b, sp') \rightarrow [(b, [eval\ A\ B\ (!sp'\ as)]]];
\end{aligned}$$

Inspired by $\Pi\Sigma$ the latest version of Agda also supports definitions using mixed induction and coinduction, using basically the same mechanism (but in a total setting). Applications of such mixed definitions are explored in more detail by Danielsson and Altenkirch (2009).

Current high-level languages such as Agda and Coq control the evaluation of recursive definitions using the evaluation context with different mechanisms for defining datatypes and values. In contrast, $\Pi\Sigma$ handles recursion uniformly, at the cost of additional annotations stating where to lift, box and force. For a core language this seems to be a price worth paying, though. We can still recover syntactical conveniences as part of the translation of high level features.

2.4 Equality

$\Pi\Sigma$ does not come with a propositional equality like the one provided by inductive families in Agda, and currently such an equality cannot, in general, be defined. This omission is intentional. In the future we plan to implement an extensional equality, similar to that described by Altenkirch et al. (2007), by recursion over the structure of types. However, for types with decidable equality a (substitutive) equality can be defined in $\Pi\Sigma$ as it is. As an example, we consider natural numbers again (cf. Sect. 2.2); see also Sect. 2.6 for a generic approach.

Using $Bool : \mathbf{Type} = \{ true\ false \}$, we first implement a decision procedure for equality of natural numbers (omitted here due to lack of space):

$$eqNat : Nat \rightarrow Nat \rightarrow Bool;$$

We can lift a *Boolean* to the type level using the Truth predicate T , and then use that to define equality:

```

Empty : Type           = { };
T      : Bool → Type   = λb → case b of { true → Unit | false → Empty };
EqNat : Nat → Nat → Type = λm n → T (eqNat m n);

```

Using recursion we now implement a *proof*⁶ of reflexivity:

```

reflNat : (n : Nat) → EqNat n n;
reflNat = λn → split n with (ln, n') → !case ln of { zero → ['unit]
                                                    | suc  → [reflNat n'] };

```

Here we are using *dependent elimination*, i.e., the typing of the branches for **split** and **case** exploit the constraint that the scrutinized term is equal to the corresponding pattern. Currently, this is only used if the scrutinee is a variable. This seems sufficient to encode dependent pattern matching.

To complete the definition of equality we have to show that *EqNat* is substitutive. This can also be done by recursion over the natural numbers (we omit the definition for reasons of space):

```

substNat : (P : Nat → Type) → (m n : Nat) → EqNat m n → P m → P n;

```

Using *substNat* and *reflNat* it is straightforward to show that *EqNat* is a congruence. For instance, transitivity can be proved as follows:

```

transNat : (i j k : Nat) → EqNat i j → EqNat j k → EqNat i k;
transNat = λi j k p q → substNat (λx → EqNat i x) j k q p;

```

The approach outlined above is limited to types with decidable equality. While we can define an equality for *Stream Nat* (corresponding to the extensional equality of streams, or bisimulation), we cannot derive a substitution principle. The same applies to function types, where we can define an extensional equality but not prove it to be substitutive.

2.5 Families

Dependent datatypes, or families, are the workhorse of dependently typed languages like Agda. As an example, consider the definition of vectors (lists indexed by their length) in Agda:

```

data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _ :: _ : {n : ℕ} → A → Vec A n → Vec A (suc n)

```

Using another family, the family of finite sets, we can define a total lookup function for vectors. This function, unlike its counterpart for lists, will never raise a runtime error:

⁶ Because termination is not checked, *reflNat* is not a formal proof. However, in this case it is easy to see that the definition is total.

```

data Fin :  $\mathbb{N} \rightarrow \text{Set}$  where
  zero : { n :  $\mathbb{N}$  }  $\rightarrow$  Fin (suc n)
  suc : { n :  $\mathbb{N}$  }  $\rightarrow$  Fin n  $\rightarrow$  Fin (suc n)

lookup :  $\forall \{ A \ n \} \rightarrow$  Fin n  $\rightarrow$  Vec A n  $\rightarrow$  A
lookup zero (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs

```

How can we encode these families and the total lookup function in $\Pi\Sigma$? One possibility is to use recursion over the natural numbers:

```

Vec : Type  $\rightarrow$  Nat  $\rightarrow$  Type;
Vec =  $\lambda A \ n \rightarrow$  split n with (nl, nr)  $\rightarrow$  case nl of { zero  $\rightarrow$  Unit
| suc  $\rightarrow$  A * [Vec A nr]};

Fin : Nat  $\rightarrow$  Type;
Fin =  $\lambda n \rightarrow$  split n with (nl, nr)  $\rightarrow$ 
  case nl of { zero  $\rightarrow$  { }
| suc  $\rightarrow$  (l : { zero suc }) * case l of { zero  $\rightarrow$  Unit
| suc  $\rightarrow$  [Fin nr]}];

```

Given these types it is straightforward to define *lookup* by recursion over the natural numbers:

```

lookup : (A : Type)  $\rightarrow$  (n : Nat)  $\rightarrow$  Fin n  $\rightarrow$  Vec A n  $\rightarrow$  A;

```

The recursive encoding of families has a number of shortcomings. It does not reflect the nature of the Agda definitions, which are not using recursion over the indices. Indeed, there are types which cannot easily be encoded this way, for example the simply typed λ -terms indexed by contexts and types. Another issue with the recursive definition is that all programs seem to *depend* on the indices, which seems to prohibit some of the runtime optimisations suggested by Brady et al. (2004).

As an alternative we can use equality explicitly to define Agda-style families:⁷

```

Vec : Type  $\rightarrow$  Nat  $\rightarrow$  Type;
Vec =  $\lambda A \ n \rightarrow$  (l : { nil cons }) *
  case l of { nil  $\rightarrow$  EqNat zero n
| cons  $\rightarrow$  [(n' : Nat) * A * Vec A n' * EqNat (suc n') n]};

Fin : Nat  $\rightarrow$  Type;
Fin =  $\lambda n \rightarrow$  (l : { zero suc }) *
  case l of { zero  $\rightarrow$  (n' : Nat) * EqNat (suc n') n
| suc  $\rightarrow$  [(n' : Nat) * Fin n' * EqNat (suc n') n]};

```

Terms corresponding to the Agda constructors, e. g., $- :: -$, are definable:

```

cons : (A : Type)  $\rightarrow$  (n : Nat)  $\rightarrow$  A  $\rightarrow$  Vec A n  $\rightarrow$  Vec A (suc n);
cons =  $\lambda A \ n \ a \ as \rightarrow$  ('cons, (n, (a, (as, reflNat (suc n))))));

```

⁷ The categorically minded reader may notice that we are exploiting the fact that the left adjoint to instantiation is definable using dependent product and equality types—this is also used in the construction of indexed containers (Altenkirch and Morris 2009).

For reasons of space we omit the implementation of *lookup* based on these definitions—it has to make the equational reasoning which is behind the Agda pattern matching explicit (Goguen et al. 2006).

2.6 Universes

In Sect. 2.4 we remarked that we can define equality for types with a decidable equality on a case by case basis. However, we can do better using the fact that reflection can be represented within a language like $\Pi\Sigma$. Which types have a decidable equality? Clearly, if we only use enumerations, dependent products and (well-behaved) recursion, equality is decidable. We can reflect the syntax and semantics of this subset of $\Pi\Sigma$'s type system as a type.

We exploit the fact that $\Pi\Sigma$ allows very flexible mutually recursive definitions to encode induction-recursion (Dybjer and Setzer 2006). We define a universe U of type codes together with a decoding function El . We start by declaring both:

$$\begin{aligned} U &: \mathbf{Type}; \\ El &: U \rightarrow \mathbf{Type}; \end{aligned}$$

We can then define U using the fact that we know the type (but not the definition) of El :

$$U = (l : \{ enum \ sigma \ box \}) * \mathbf{case} \ l \ \mathbf{of} \ \begin{cases} enum & \rightarrow Nat \\ sigma & \rightarrow [(a : U) * (El \ a \rightarrow U)] \\ box & \rightarrow [\uparrow U]; \end{cases}$$

Note that we define enumerations *up to isomorphism*—we only keep track of the number of constructors. El is defined by exploiting that we know the types of U and El , and also the definition of U :

$$El = \lambda a \rightarrow \mathbf{split} \ a \ \mathbf{with} \ (a_l, a_r) \rightarrow \begin{cases} \mathbf{!case} \ a_l \ \mathbf{of} \ \begin{cases} enum & \rightarrow [Fin \ a_r] \\ sigma & \rightarrow [\mathbf{split} \ a_r \ \mathbf{with} \ (b, c) \rightarrow (x : El \ b) * (El \ (c \ x))] \\ box & \rightarrow [[El \ (!a_r)]]; \end{cases} \end{cases}$$

Note that, unlike in a simply typed framework, we cannot arbitrarily change the order of the definitions above—the definition of U is required to type check El . $\Pi\Sigma$ supports any kind of mutually recursive definition, with declarations and definitions appearing in any order, as long as each item type checks with respect to the previous items.

It is straightforward to translate type definitions into elements of U . For instance, Nat can be represented as follows:

$$\begin{aligned} nat : U = (&'sigma, (('enum, suc \ (suc \ zero)), \\ &(\lambda i \rightarrow \mathbf{split} \ i \ \mathbf{with} \ (i_l, i_r) \rightarrow \mathbf{!case} \ i_l \ \mathbf{of} \ \begin{cases} zero & \rightarrow [('enum, suc \ zero)] \\ suc & \rightarrow [('box, [nat])]); \end{cases} \end{aligned}$$

We can now define a generic decidable equality $eq : (a : U) \rightarrow El \ a \rightarrow El \ a \rightarrow Bool$ by recursion over U (omitted here). Note that the encoding can also be used for *families* with decidable equality; Fin can for instance be encoded as an element of $El \ nat \rightarrow U$.

3 β -reduction

This section gives an operational semantics for $\Pi\Sigma$ by inductively defining the notion of (weak) β -reduction. Instead of defining substitution, we use local definitions. In this sense $\Pi\Sigma$ is an explicit substitution calculus. Reduction and equality make no use of type signatures, so we allow declarations without a type signature, denoted by x ., and use the abbreviation $x := t$ for x .; $x = t$.

We start by defining $\Delta \vdash x = t$, which is defined if the definition $x = t$ is visible in Δ . The rules are straightforward:

$$\frac{}{\Delta; x = t \vdash x = t} \quad \frac{\Delta \vdash x = t \quad x \neq y}{\Delta; y : A \vdash x = t} \quad \frac{\Delta \vdash x = t \quad x \neq y}{\Delta; y = u \vdash x = t}$$

We specify β -reduction using a big-step semantics $\Delta \vdash t \rightsquigarrow v$ which means that t β -reduces to v in the context Δ . We define values (v) together with weak-head normal forms (w) and neutral terms (n) as follows:

$$\begin{aligned} v &::= \mathbf{let} \Gamma \mathbf{in} w \mid n \\ w &::= \mathbf{Type} \mid (x : \sigma) \rightarrow \tau \mid \lambda x \rightarrow t \mid (x : \sigma) * \tau \mid (t, t) \mid \{\bar{l}\} \mid 'l \mid \uparrow \sigma \mid [t] \\ n &::= x \mid n t \mid \mathbf{split} \ n \ \mathbf{with} \ (x, x) \rightarrow t \mid \mathbf{case} \ n \ \mathbf{of} \ \{\bar{l} \rightarrow t^l\} \mid !n \end{aligned}$$

Note that v is not closed under \mathbf{let} , but this is easy to fix by defining that $\mathbf{let} \Delta \mathbf{in} (\mathbf{let} \Gamma \mathbf{in} w)$ is equal to $\mathbf{let} \Delta; \Gamma \mathbf{in} w$. For neutral terms we apply \mathbf{let} to every subterm, e.g. $\mathbf{let} \Delta \mathbf{in} n t = (\mathbf{let} \Delta \mathbf{in} n) (\mathbf{let} \Delta \mathbf{in} t)$.⁸ β -reduction is defined inductively:

$$\begin{aligned} &\frac{}{\Delta \vdash w \rightsquigarrow \mathbf{let} \ \varepsilon \ \mathbf{in} \ w} \quad \frac{\Delta; \Gamma \vdash u \rightsquigarrow v}{\Delta \vdash \mathbf{let} \ \Gamma \ \mathbf{in} \ u \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ v} \quad \frac{\Delta \vdash x = t \quad \Delta \vdash t \rightsquigarrow v}{\Delta \vdash x \rightsquigarrow v} \\ &\frac{\Delta \vdash t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ \lambda x \rightarrow t' \quad \Delta; \Gamma; x := \mathbf{let} \ \Delta \ \mathbf{in} \ u \vdash t' \rightsquigarrow v}{\Delta \vdash t u \rightsquigarrow \mathbf{let} \ \Gamma; x := \mathbf{let} \ \Delta \ \mathbf{in} \ u \ \mathbf{in} \ v} \\ &\frac{\Delta \vdash t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ (t_0, t_1) \quad \Delta; y := \mathbf{let} \ \Gamma \ \mathbf{in} \ t_0; z := \mathbf{let} \ \Delta; \Gamma \ \mathbf{in} \ t_1 \vdash u \rightsquigarrow v}{\Delta \vdash \mathbf{split} \ t \ \mathbf{with} \ (y, z) \rightarrow u \rightsquigarrow \mathbf{let} \ y := \mathbf{let} \ \Gamma \ \mathbf{in} \ t_0, z := \mathbf{let} \ \Delta; \Gamma \ \mathbf{in} \ t_1 \ \mathbf{in} \ v} \\ &\frac{\Delta \vdash t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ l_i \quad \Delta \vdash u_i \rightsquigarrow v}{\Delta \vdash \mathbf{case} \ t \ \mathbf{of} \ \{\bar{l}_i \rightarrow u_i\} \rightsquigarrow v} \quad \frac{\Delta \vdash t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ [u] \quad \Delta; \Gamma \vdash u \rightsquigarrow v}{\Delta \vdash !t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ v} \end{aligned}$$

Reduction of open terms may get stuck due to the occurrence of a variable, so we add rules to cover this case. For reasons of space we only include one rule in the paper:

$$\frac{\Delta \vdash t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ n}{\Delta \vdash t u \rightsquigarrow (\mathbf{let} \ \Gamma \ \mathbf{in} \ n) u}$$

For the type system we also need an auxiliary notion $\Delta \vdash t \rightsquigarrow^! v$ which forces a box only if necessary:

$$\frac{\Delta \vdash t \rightsquigarrow v}{\Delta \vdash t \rightsquigarrow^! v} \quad \frac{\Delta \vdash t \rightsquigarrow \mathbf{let} \ \Gamma \ \mathbf{in} \ [u] \quad \Delta; \Gamma \vdash u \rightsquigarrow^! v}{\Delta \vdash t \rightsquigarrow^! \mathbf{let} \ \Gamma \ \mathbf{in} \ v}$$

⁸ We overload ; to mean cons and append.

4 α - and β -equality

As mentioned earlier, $\Pi\Sigma$ uses a structural equality for recursive definitions. This makes it necessary to define a novel notion of α -equality. Let us look at some examples. We have already discussed the use of boxes to stop infinite unfolding of recursive definitions. This is achieved by specifying that inside a box we are only using α -equality. For instance, the following terms are not β -equal, because this would require looking up a definition inside a box:⁹

$$\mathbf{let } x : Bool = 'true \mathbf{ in } [x] \not\equiv_{\beta} ['true].$$

However, we still want the equality

$$\mathbf{let } x : Bool = 'true \mathbf{ in } [x] \equiv_{\alpha} \mathbf{let } y : Bool = 'true \mathbf{ in } [y]$$

to hold, because we can get from one side to the other by consistently renaming bound variables. This means that we have to compare the definitions of variables which we want to identify—while being careful not to expand recursive definitions indefinitely—because clearly

$$\mathbf{let } x : Bool = 'true \mathbf{ in } [x] \not\equiv_{\beta} \mathbf{let } y : Bool = 'false \mathbf{ in } [y].$$

We also want to allow weakening:

$$\mathbf{let } x : Bool = 'true, y : Bool = 'false \mathbf{ in } [x] \equiv_{\alpha} \mathbf{let } z : Bool = 'true \mathbf{ in } [z].$$

We achieve the goals outlined above by specifying that two expressions of the form $\mathbf{let } \Gamma \mathbf{ in } t$ and $\mathbf{let } \Gamma' \mathbf{ in } t'$ are α -equivalent if there is a partial bijection (a bijection on a subset) between the variables defined in Γ and Γ' so that t and t' are α -equal up to this identification of variables; the identified variables, in turn, are required to have β -equal definitions (up to related partial bijections). In the implementation we construct this identification lazily: if we are forced to identify two let-bound variables, we replace all uses of these variables with a single, fresh (undefined) variable and check whether the definitions of the two variables are equal. This way we do not unfold recursive definitions more than once.

We specify partial bijections using $\varphi ::= \overline{(x^?, x^?)}$, where $x^? ::= x \mid -$. Here ‘ $-$ ’ is used to shadow variables which due to weakening are not associated to another variable. Lookup is specified as follows:

$$\frac{}{\varphi; (x, y) \vdash x \sim y} \quad \frac{\varphi \vdash x' \sim y' \quad x' \not\equiv x^? \quad y' \not\equiv y^?}{\varphi; (x^?, y^?) \vdash x' \sim y'}$$

Because all but one rule is shared between the two, we specify both α - and β -equality at the same time, indexing the rules on the meta-variable $\kappa \in \{\alpha, \beta\}$. The judgement $\varphi : \Delta \sim \Delta' \vdash t \equiv_{\kappa} t'$ means that, given a partial bijection φ for the contexts Δ and Δ' , the terms t and t' are κ -equivalent. The difference between α - and β -equality is that the latter is closed under β -reduction:

⁹ In this particular example the definition is not actually recursive, but this does not matter because we do not make a difference between recursive and non-recursive definitions.

$$\frac{\Delta \vdash t \rightsquigarrow v \quad \Delta' \vdash t' \rightsquigarrow v' \quad \varphi : \Delta \sim \Delta' \vdash v \equiv_{\beta} v'}{\varphi : \Delta \sim \Delta' \vdash t \equiv_{\beta} t'}$$

The remaining rules apply to both equalities. Variables are equal if identified by the partial bijection:

$$\frac{\varphi \vdash x \sim y}{\varphi : \Delta \sim \Delta' \vdash x \equiv_{\kappa} y}$$

A congruence rule is included for each term former. For reasons of space we omit some of these rules—the following are typical examples:

$$\frac{\varphi : \Delta \sim \Delta' \vdash t \equiv_{\kappa} t' \quad \varphi : \Delta \sim \Delta' \vdash u \equiv_{\kappa} u'}{\varphi : \Delta \sim \Delta' \vdash t u \equiv_{\kappa} t' u'} \quad \frac{\varphi; (x, x') : (\Delta; x) \sim (\Delta'; x') \vdash t \equiv_{\kappa} t'}{\varphi : \Delta \sim \Delta' \vdash \lambda x \rightarrow t \equiv_{\kappa} \lambda x' \rightarrow t'}$$

As noted above, the congruence rule for boxes only allows α -equality in the premise:

$$\frac{\varphi : \Delta \sim \Delta' \vdash t \equiv_{\alpha} t'}{\varphi : \Delta \sim \Delta' \vdash [t] \equiv_{\kappa} [t']}$$

The last κ -rule shown is the congruence rule for **let**:

$$\frac{\varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma' \quad \varphi; \psi : (\Delta; \Gamma) \sim (\Delta'; \Gamma') \vdash t \equiv_{\kappa} t'}{\varphi : \Delta \sim \Delta' \vdash \mathbf{let} \Gamma \mathbf{in} t \equiv_{\kappa} \mathbf{let} \Gamma' \mathbf{in} t'}$$

This rule uses an auxiliary judgement $\varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma'$ which extends a partial bijection over a pair of contexts (ψ can be seen as the rule's “output”).

The rules for $\varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma'$ allow us to choose which variables get identified and which are ignored. The base case is $\varphi : \Delta \sim \Delta' \vdash \varepsilon : \varepsilon \sim \varepsilon$. We can extend a partial bijection by identifying two variables under the condition that the associated types are β -equal:

$$\frac{\varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma' \quad \varphi; \psi : (\Delta; \Gamma) \sim (\Delta'; \Gamma') \vdash A \equiv_{\beta} A'}{\varphi : \Delta \sim \Delta' \vdash (\psi; (x, x')) : (\Gamma; x : A) \sim (\Gamma'; x' : A')}$$

Alternatively, we can ignore a declaration:

$$\frac{\varphi : \Delta \sim \Delta' \vdash \psi; (x, -) : \Gamma \sim \Gamma'}{\varphi : \Delta \sim \Delta' \vdash \psi : (\Gamma; x : A) \sim \Gamma'}$$

There is a symmetric rule for ignoring a declaration on the right, which we omit for reasons of space. To check whether two definitions are equal we compare the terms using β -equality. Note that this takes place before the definitions are added to the context:

$$\frac{\varphi \vdash x \sim x' \quad \varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma' \quad \varphi; \psi : (\Delta; \Gamma) \sim (\Delta'; \Gamma') \vdash t \equiv_{\beta} t'}{\varphi : \Delta \sim \Delta' \vdash \psi : (\Gamma; x = t) \sim (\Gamma'; x' = t')}$$

If we have previously decided to ignore a variable we have to ignore its definition as well:

$$\frac{\varphi \vdash x \sim - \quad \varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma'}{\varphi : \Delta \sim \Delta' \vdash \psi : (\Gamma; x = t) \sim \Gamma'}$$

As before, there is a symmetric rule which we omit here.

In the typing rules in Sect. 5 we only use β -equality with respect to the identity partial bijection, so we define $\Delta \vdash t \equiv_{\beta} u$ to mean $\text{id}_{\Delta} : \Delta \sim \Delta \vdash t \equiv_{\beta} u$.

5 The type system

We define a bidirectional type system. There are two judgements, $\Gamma \vdash t \Leftarrow \sigma$ which means that we can check that t has type σ , and $\Gamma \vdash t \Rightarrow \sigma$ which means that we can infer that t has type σ . In some rules we use \Leftarrow as a metavariable that can be instantiated with either \Rightarrow or \Leftarrow . The elimination rules have to force the type of the scrutinee to be able to handle recursive types. Hence we define a derived judgment $\Gamma \vdash t \Rightarrow^! \sigma$ which means $\Gamma \vdash t \Rightarrow \sigma'$ and $\Gamma \vdash \sigma' \rightsquigarrow^! \sigma$. To handle local definitions we also introduce a judgement $\Gamma \vdash \Delta$ which means that Δ is well-formed with respect to Γ .

Let us now give the typing rules. The rules for looking up variable declarations ($\Gamma \vdash x \Rightarrow \sigma$) have the same form as the ones for looking up definitions ($\Gamma \vdash x = t$), so we do not repeat them here. The other structural rules are defined as follows:

$$\frac{\Gamma \vdash \Delta \quad \Gamma; \Delta \vdash t \Leftarrow \sigma}{\Gamma \vdash \mathbf{let} \ \Delta \ \mathbf{in} \ t \Leftarrow \mathbf{let} \ \Delta \ \mathbf{in} \ \sigma} \quad \frac{\Gamma \vdash t \Rightarrow \sigma \quad \Gamma \vdash \sigma \equiv_{\beta} \tau}{\Gamma \vdash t \Leftarrow \tau}$$

Note that the latter rule (the *conversion rule*) changes direction: if we want to check whether a term has type τ , we first infer σ and then verify that σ and τ are convertible.

In the following rules the meta-variable \diamond , which stands for the quantifiers ‘ \rightarrow ’ and ‘ $*$ ’, is used:

$$\frac{\Gamma \vdash \sigma \Leftarrow \mathbf{Type} \quad \Gamma, x : \sigma \vdash \tau \Leftarrow \mathbf{Type}}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Type} \quad \Gamma \vdash (x : \sigma) \diamond \tau \Rightarrow \mathbf{Type}}$$

$$\frac{\Gamma, x : \sigma \vdash t \Leftarrow \tau \quad \Gamma \vdash t \Rightarrow^! (x : \sigma) \rightarrow \tau \quad \Gamma \vdash t \Leftarrow \sigma}{\Gamma \vdash \lambda x \rightarrow t \Leftarrow (x : \sigma) \rightarrow \tau \quad \Gamma \vdash t u \Rightarrow \mathbf{let} \ x : \sigma = u \ \mathbf{in} \ \tau \quad \Gamma \vdash u \Leftarrow \mathbf{let} \ x : \sigma = t \ \mathbf{in} \ \tau} \quad \frac{\Gamma \vdash t \Leftarrow \sigma \quad \Gamma \vdash t \Leftarrow \uparrow \sigma}{\Gamma \vdash [t] \Leftarrow \uparrow \sigma} \quad \frac{\Gamma \vdash t \Leftarrow \sigma \quad \Gamma \vdash !t \Leftarrow \sigma}{\Gamma \vdash \{\bar{l}\} \Rightarrow \mathbf{Type}} \quad \frac{m \in \bar{l}}{\Gamma \vdash m \Leftarrow \{\bar{l}\}}$$

$$\frac{\Gamma \vdash \rho \Leftarrow \mathbf{Type} \quad \Gamma \vdash z \Rightarrow^! (x : \sigma) * \tau}{\Gamma; x : \sigma; y : \tau; z = (x, y) \vdash u \Leftarrow \rho} \quad \frac{\Gamma \vdash \rho \Leftarrow \mathbf{Type} \quad \Gamma \vdash x \Rightarrow^! \{\bar{l}\}}{(\Gamma, x = 'l_i \vdash u_i \Leftarrow \rho)_i} \quad \frac{\Gamma \vdash \mathbf{split} \ z \ \mathbf{with} \ (x, y) \rightarrow u \Leftarrow \rho}{\Gamma \vdash \mathbf{case} \ x \ \mathbf{of} \ \{\bar{l} \rightarrow u\} \Leftarrow \rho}$$

The last two rules are the *dependent* elimination rules for products and labels. The non-dependent variants, which allow terms as scrutinees but whose premises do not get the benefit of equality constraints, are omitted. Note that the indexed premise in the case rule must hold for each of the branches. Note also that the case rule has a side-condition: the branches must have distinct labels.¹⁰

The following rules allow boxed types to be used as types. This can be exploited in the definition of recursive types. For instance, in the definition of *Nat* in Sect. 2.2 we made use of the fact that one of the types of $[Nat]$ is **Type**.

¹⁰ In the rule for enumeration types the notation $\{\bar{l}\}$ stands for a set, so no side-condition is necessary.

$$\frac{\Gamma \vdash t \Leftarrow \mathbf{Type}}{\Gamma \vdash \uparrow t \Rightarrow \mathbf{Type}} \quad \frac{\Gamma \vdash \sigma \Leftarrow \mathbf{Type}}{\Gamma \vdash [\sigma] \Leftarrow \mathbf{Type}}$$

Finally we specify the rules for context validity:

$$\frac{}{\Gamma \vdash \varepsilon} \quad \frac{\Gamma \vdash \Delta \quad \Gamma; \Delta \vdash \sigma \Leftarrow \mathbf{Type}}{\Gamma \vdash \Delta; x : \sigma} \quad \frac{\Gamma \vdash \Delta \quad \Gamma \vdash x \Rightarrow \sigma \quad \Gamma; \Delta \vdash t \Leftarrow \sigma}{\Gamma \vdash \Delta; x = t}$$

6 Conclusions

The definition of $\Pi\Sigma$ uses several innovations to meet the challenge of providing a concise core language for dependently typed programming. We are able to use one recursion mechanism for the definition of both types and (recursive or corecursive) programs, relying essentially on lifted types and boxes. We also permit arbitrary mutually recursive definitions where the only condition is that, at any point in the program, the current line has to type check with respect to the current context—this captures inductive-recursive definitions. Furthermore all programs and types can be used locally in let-expressions; the top-level does not have special status. To facilitate this flexible use of let-expressions we have introduced a novel notion of α -equality for recursive definitions. As a bonus the use of local definitions makes it unnecessary to define substitution as an operation on terms.

Much remains to be done. We need to demonstrate the usefulness of $\Pi\Sigma$ by using it as a core language for an Agda-like language. As we have seen in Sect. 2 this seems possible, provided we restrict indexing to types with a decidable equality. We plan to go further and realize an extensional equality for higher types based on previous work (Altenkirch et al. 2007).

The current type system is less complicated than that described in a previous draft paper (Altenkirch and Oury 2008). We have simplified the system by restricting dependent elimination to the case where the scrutinee is a variable. However, this may make the translation of local case expressions too difficult, hence we may consider reintroducing more general constraints.

Having a small language makes complete reflection feasible, opening the door for generic programming. Another goal is to develop $\Pi\Sigma$'s metatheory formally. The distance between the specification and the implementation seems small enough that we plan to develop a certified version of the type checker in Agda. This type checker can then be translated into $\Pi\Sigma$ itself.¹¹ Using this implementation we hope to be able to formally verify central aspects of the language, most importantly type-soundness: β -reduction does not get stuck for closed, well-typed programs. We also hope to show that $\Pi\Sigma$ has the subject reduction property, which fails for CCIC in the presence of coinductive types (Giménez 1996).

¹¹ The implementation would use the partiality monad, hence Gödel's theorem is not an issue here.

References

- Thorsten Altenkirch and Peter Morris. Indexed containers. In *LICS 2009*, pages 277–285, 2009.
- Thorsten Altenkirch and Nicolas Oury. $\Pi\Sigma$: A core language for dependently typed programming. Draft, 2008.
- Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Draft, 2005.
- Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV 2007*, pages 57–68, 2007.
- Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP 1998*, pages 239–250, 1998.
- Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *TYPES 2003*, volume 3085 of *LNCS*, pages 115–129, 2004.
- James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: A standalone typechecker for ETT. In Marko van Eekelen, editor, *Trends in Functional Programming Volume 6*. Intellect, 2006.
- The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*, 2009.
- Thierry Coquand. A calculus of definitions. Draft, available from <http://www.cs.chalmers.se/~coquand/def.pdf>, 2008.
- Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. A simple type-theoretic language: Mini-TT. In *From Semantics to Computer Science; Essays in Honour of Gilles Kahn*, pages 139–164. Cambridge University Press, 2009.
- Sa Cui, Kevin Donnelly, and Hongwei Xi. ATS: A language that combines programming with theorem proving. In *FroCoS 2005*, volume 3717 of *LNCS*, pages 310–320, 2005.
- Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. Draft, 2009.
- Peter Dybjer and Anton Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.
- Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996.
- Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning and Computation, Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *LNCS*, pages 521–540. Springer-Verlag, 2006.
- Peter Hancock, Dirk Pattinson, and Neil Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3:9), 2009.
- Conor McBride. the Strathclyde Haskell Enhancement. Available at <http://personal.cis.strath.ac.uk/~conor/pub/she/>, 2009.
- Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.
- Tim Sheard. Putting Curry-Howard to work. In *Haskell 2005*, pages 74–85, 2005.
- Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris 11, 2008.
- Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *ML 1998*, 1998.