ΠΣ: Dependent Types without the Sugar

Thorsten Altenkirch[1], Nils Anders Danielsson[1],
Andres Löh[2], and Nicolas Oury[3]

[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 languages. However, these features complicate the metatheoretical study and are a potential source of bugs.

To address these issues we introduce ΠΣ, a dependently typed core language. It is small enough for metatheoretical study and the type checker 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 ΠΣ is suitable as a core language for dependently typed programming.

1 Introduction

Dependent types offer programmers a flexible path towards formally verified programs and, at the same time, opportunities for increased productivity through new ways of structuring programs (Altenkirch et al. 2005). Dependently typed programming languages like Agda (Norell 2007) are gaining in popularity, and dependently typed *programming* is also becoming more popular in the Coq community (Coq Development Team 2009), for instance through the use of 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 Ωmega (Sheard 2005), ATS (Cui et al. 2005), and the Strathclyde Haskell Enhancement (McBride 2009).

Dependently typed languages tend to offer a number of high-level features for reducing the complexity of programming in such a rich type discipline, and at the same time improve the readability of the code. These features include:

Datatype definitions A convenient syntax for defining 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 indices can often be inferred using unification, which means that the programmer does not have to read or write them. This provides an alternative to polymorphic type inference à 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 some of 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 can 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 designed in such a way that we can easily translate the high-level features mentioned above; 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 independent certificates of totality (and the related notions of positivity and stratification); such certificates could be produced manually, or through the use of a termination checker.

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

- Dependent function types (Π-types) and dependent product types (Σ-types).
- A (very) impredicative universe of types with $\mathsf{Type} : \mathsf{Type}$.
- 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 ΠΣ in Haskell.[5]

Before we continue, note that we have not yet developed the metatheory of ΠΣ formally, so we would not be surprised if there were some problems with the presentation given here. We plan to establish important metatheoretic properties such as soundness (well-typed programs do not get stuck) for a suitably modified version of ΠΣ in a later paper. The main purpose of this paper is to introduce the general ideas underlying the language, and to start a discussion about them.

Outline of the Paper

We introduce ΠΣ 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 have 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 ΠΣ 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 ΠΣ'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 core language $\mathrm{F_C}$ (Sulzmann et al. 2007) provides support for GADTs, among other things.

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, Ulf Norell, Simon Peyton Jones and Stephanie Weirich for discussions related to the ΠΣ project. We would also like to thank Darin Morrison, who has contributed to the implementation of ΠΣ, and members of the Functional Programming Laboratory in Nottingham, who have given feedback on our work.

2 ΠΣ by Example

In this section we first briefly introduce the syntax of ΠΣ (Sect. 2.1). The rest of the section demonstrates how to encode a number of high-level features from dependently typed programming languages in ΠΣ: (co)datatypes

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

(Sects. 2.2 and 2.3), equality (Sect. 2.4), families of datatypes (Sect. 2.5), and finally induction-recursion (Sect. 2.6).

2.1 Syntax Overview

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

$$\begin{array}{lll}
\text{Terms} & t, u, \sigma, \tau ::= & \mathsf{let}\ \Gamma\ \mathsf{in}\ t \mid x \mid \mathsf{Type} \mid (x : \sigma) \rightarrow \tau \mid \lambda x \rightarrow t \mid t\ u \\
 & \mid & (x : \sigma) * \tau \mid (t, u) \mid \mathsf{split}\ t\ \mathsf{with}\ (x, y) \rightarrow t \\
 & \mid & \{\overline{l}\} \mid {}'l \mid \mathsf{case}\ t\ \mathsf{of}\ \{\overline{l \rightarrow u}^{|}\} \\
 & \mid & \uparrow\sigma \mid [t] \mid !t \mid \mathsf{Rec}\ \sigma \mid \mathsf{fold}\ t \mid \mathsf{unfold}\ t\ \mathsf{as}\ x \rightarrow u \\
\text{Contexts}\ \Gamma, \Delta & ::= & \varepsilon \mid \Gamma; x : \sigma \mid \Gamma; x = t
\end{array}$$

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 write $\overline{a}^s$ 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 $\mathsf{Type} : \mathsf{Type}$.

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, u). The eliminator $\mathsf{split}\ t\ \mathsf{with}\ (x, y) \rightarrow u$ deconstructs the scrutinee t, binding its components to x and y, and then evaluates u.

Enumerations Enumerations are finite sets, written $\{\overline{l}\}$. 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 $\mathsf{case}\ t\ \mathsf{of}\ \{\overline{l \rightarrow u}^{|}\}$ 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. To enable the definition of recursive types we introduce a type former Rec which turns a suspended type (i.e., an inhabitant of $\uparrow\mathsf{Type}$) into a type. Rec comes together with a constructor fold and an eliminator unfold.

Let A let expression's first argument Γ is a context, i.e., a sequence of declarations $x : \sigma$ and (possibly recursive) definitions $x = t$. Definitions and declarations may occur in any order in a let context, subject to the following constraints:

- Before a variable can be defined, it must first be declared.
- Every declared variable must be defined exactly once in the same context (subject to shadowing, i.e., $x : \sigma; x = t; x : \tau; x = u$ is fine).
- Every declaration and definition has to type check with respect to the previous declarations and definitions. Note that the order matters and that we cannot always shift all declarations before all definitions, because type checking a declaration may depend on the definition of a mutually defined variable (see Sect. 2.6).

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 $\sigma \to \tau$, and a non-dependent product as $\sigma * \tau$ (both $\to$ and $*$ associate to the right). Several variables may be bound at once in λ abstractions ($\lambda x_1\ x_2 \ldots x_n \to t$), function types ($(x_1\ x_2 \ldots x_n : \sigma) \to \tau$), and product types ($(x_1\ x_2 \ldots x_n : \sigma) * \tau$). We can also combine a declaration and a subsequent definition: instead of $x : \sigma; x = t$ we write $x : \sigma = t$. Finally we write $\mathsf{unfold}\ t$ as a shorthand for $\mathsf{unfold}\ t\ \mathsf{as}\ x \to x$.

2.2 Datatypes

$\Pi\Sigma$ does not have a builtin mechanism to define datatypes. Instead, we rely on its more primitive features—finite types, $\varSigma$-types, recursion and lifting—to model datatypes.

As a simple example, consider the declaration of (Peano) natural numbers and addition. We represent *Nat* as a recursively defined $\varSigma$-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:

$$\mathit{Nat} : \mathsf{Type} = (l : \{\mathit{zero}\ \mathit{suc}\}) * \mathsf{case}\ l\ \mathsf{of}\ \{\mathit{zero} \to \mathit{Unit} \mid \mathit{suc} \to \mathsf{Rec}\ [\mathit{Nat}]\};$$

In the case of *zero* we use a one element type *Unit* which is defined by $\mathit{Unit} : \mathsf{Type} = \{\mathit{unit}\}$. The recursive occurrence of *Nat* is placed inside a box (i.e., $[\mathit{Nat}]$ rather than *Nat*). Boxing prevents infinite unfolding during evaluation. Evaluation is performed while testing type equality, and boxing is essential to keep the type checker from diverging. Note also that we need to use Rec because $[\mathit{Nat}]$ has type $\Uparrow\mathsf{Type}$ but we expect an element of Type here.

Using the above representation we can derive the constructors $\mathit{zero} : \mathit{Nat} = ({'}\mathit{zero}, {'}\mathit{unit})$ and $\mathit{suc} : \mathit{Nat} \to \mathit{Nat} = \lambda i \to ({'}\mathit{suc}, \mathsf{fold}\ i)$. (We use fold to construct an element of $\mathsf{Rec}\ [\mathit{Nat}]$.) Addition can then be defined as follows:

$$\begin{array}{l}
\mathit{add} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat};\\
\mathit{add} = \lambda m\ n \to \mathsf{split}\ m\ \mathsf{with}\ (m_l, m_r) \to\\
\qquad\qquad\quad !\ \mathsf{case}\ m_l\ \mathsf{of}\ \{\ \mathit{zero} \to [n]\\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{suc}\ \ \to [\mathit{suc}\ (\mathit{add}\ (\mathsf{unfold}\ m_r)\ n)]\};
\end{array}$$

Here we use *dependent elimination*, i.e., the typing rules for split and case exploit the constraint that the scrutinized term is equal to the corresponding pattern. In the *zero* branch m_r has type *Unit*, and in the *suc* branch it has type $\mathsf{Rec}\ [\mathit{Nat}]$. In the latter case we use unfold to get a term of type *Nat*.

Note that, yet again, we use boxing to stop the infinite unfolding of the recursive call (type checking a dependently typed program can involve evaluation under binders). We have to box *both* branches of the case to satisfy the type checker—they both have type $\Uparrow\mathit{Nat}$. Once the variable m_l 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., $\mathit{add}\ (\mathit{suc}\ (\mathit{suc}\ \mathit{zero}))\ (\mathit{suc}\ \mathit{zero})$, evaluate correctly—in this case to $({'}\mathit{suc}, \mathsf{fold}\ ({'}\mathit{suc}, \mathsf{fold}\ ({'}\mathit{suc}, \mathsf{fold}\ ({'}\mathit{zero}, {'}\mathit{unit}))))$.

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:

$$\mathit{omega} : \mathit{Nat} = ({}'\mathit{suc}, \mathsf{fold}\ \mathit{omega});$$

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:

$$\mathit{Stream} : \mathsf{Type} \to \mathsf{Type} = \lambda A \to A * \mathsf{Rec}\ [\uparrow(\mathit{Stream}\ A)];$$

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

$$\begin{array}{l}
\mathit{from} : \mathit{Nat} \to \mathit{Stream}\ \mathit{Nat};\\
\mathit{from} = \lambda n \to (n, \mathsf{fold}\ [\mathit{from}\ (\mathit{suc}\ n)]);
\end{array}$$

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

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:

$$\begin{array}{l}
\mathit{SP} : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type};\\
\mathit{SP} = \lambda A\ B \to (l : \{\mathit{get}\ \mathit{put}\}) * \mathsf{case}\ l\ \mathsf{of}\ \begin{array}[t]{l}\{\ \mathit{get} \to A \to \mathsf{Rec}\ [\mathit{SP}\ A\ B]\\ \mid \mathit{put} \to B * \mathsf{Rec}\ [\uparrow(\mathit{SP}\ A\ B)]\};\end{array}
\end{array}$$

The basic idea of stream processors is that we can only perform a finite number of *get*s before issuing the next of infinitely many *put*s. As an example we define the identity stream processor corecursively:

$$\begin{array}{l}
\mathit{idsp} : (A : \mathsf{Type}) \to \mathit{SP}\ A\ A;\\
\mathit{idsp} = \lambda A \to ({}'\mathit{get}, (\lambda a \to \mathsf{fold}\ ({}'\mathit{put}, (a, \mathsf{fold}\ [\mathit{idsp}\ A]))));
\end{array}$$

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

$$\begin{array}{l}
\mathit{eval} : (A\ B : \mathsf{Type}) \to \mathit{SP}\ A\ B \to \mathit{Stream}\ A \to \mathit{Stream}\ B;\\
\mathit{eval} = \lambda A\ B\ \mathit{sp}\ \mathit{aas} \to \mathsf{split}\ \mathit{sp}\ \mathsf{with}\ (\mathit{sp}_l, \mathit{sp}_r) \to \mathsf{!case}\ \mathit{sp}_l\ \mathsf{of}\\
\quad \{\ \mathit{get} \to \mathsf{split}\ \mathit{aas}\ \mathsf{with}\ (a, \mathit{aas}') \to [(\mathit{eval}\ A\ B\ (\mathsf{unfold}\ (\mathit{sp}_r\ a))\ (!(\mathsf{unfold}\ \mathit{aas}')))]\\
\quad \mid \mathit{put} \to \mathsf{split}\ \mathit{sp}_r\ \mathsf{with}\ (b, \mathit{sp}') \quad \to [(b, \mathsf{fold}\ [\mathit{eval}\ A\ B\ (!(\mathsf{unfold}\ \mathit{sp}'))\ \mathit{aas}])]\};
\end{array}$$

Inspired by ΠΣ 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).

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, ΠΣ 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

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

$$\mathit{eqNat} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Bool};$$

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

$$\begin{array}{lll}
\mathit{Empty} & : \mathsf{Type} & = \{\,\};\\
T & : \mathit{Bool} \to \mathsf{Type} & = \lambda b \to \mathsf{case}\ b\ \mathsf{of}\ \{\mathit{true} \to \mathit{Unit} \mid \mathit{false} \to \mathit{Empty}\};\\
\mathit{EqNat} & : \mathit{Nat} \to \mathit{Nat} \to \mathsf{Type} & = \lambda m\ n \to T\ (\mathit{eqNat}\ m\ n);
\end{array}$$

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

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{reflNat} : (n : \mathit{Nat}) \to \mathit{EqNat}\ n\ n;}\\
\mathit{reflNat} = \lambda n \to \mathsf{split}\ n\ \mathsf{with}\ (n_l, n_r) \to\ !\mathsf{case}\ n_l\ \mathsf{of} & \{\ \mathit{zero} \to [{}'\mathit{unit}]\\
 & \mid \mathit{suc}\ \ \to [\mathit{reflNat}\ (\mathsf{unfold}\ n_r)]\,\};
\end{array}$$

Note the use of dependent elimination to encode dependent pattern matching here. Currently dependent elimination is only permitted if the scrutinee is a variable (as is the case for n and n_l here; see Sect. 5), but the current design lacks subject reduction for open terms (see Sect. 6), so we may reconsider this design choice.

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):

$$\mathit{substNat} : (P : \mathit{Nat} \to \mathsf{Type}) \to (m\ n : \mathit{Nat}) \to \mathit{EqNat}\ m\ n \to P\ m \to P\ n;$$

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

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

```
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 a non-boolean equality *eqStreamNat* : *Stream Nat* → *Stream Nat* → Type (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:

```
data Fin : ℕ → Set where
  zero : {n : ℕ}          → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
lookup : ∀ {A n} → Fin n → Vec A n → 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 ΠΣ? One possibility is to use recursion over the natural numbers:

```
Vec : Type → Nat → Type;
Vec = λA n → split n with (n_l, n_r) → ! case n_l of
        { zero → [Unit]
        | suc  → [A * Vec A (unfold n_r)]};
Fin : Nat → Type;
Fin = λn → split n with (n_l, n_r) → case n_l of
        { zero → { }
        | suc →  (l : {zero suc}) * ! case l of { zero → [Unit]
                                                 | suc  → [Fin (unfold n_r)]}};
```

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

```
lookup : (A : Type) → (n : Nat) → Fin n → Vec A n → A;
```

However, these recursive encodings do not reflect the nature of the definitions in Agda, which are not using recursion over the indices. There are types which cannot easily be encoded this way, for example the simply typed λ-terms indexed by contexts and types. To define Agda-style families we can use explicit equalities instead:

$$\begin{array}{l}
\mathit{Vec} : \mathsf{Type} \rightarrow \mathit{Nat} \rightarrow \mathsf{Type};\\
\mathit{Vec} = \lambda A\ n \rightarrow (l : \{\, \mathit{nil}\ \mathit{cons}\,\}) *\\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \{\, \mathit{nil} \quad\ \rightarrow \mathit{EqNat}\ \mathit{zero}\ n\\
\qquad\qquad\qquad\ \mid \mathit{cons} \rightarrow (n' : \mathit{Nat}) * A * (\mathsf{Rec}\ [\,\mathit{Vec}\ A\ n'\,]) * \mathit{EqNat}\ (\mathit{suc}\ n')\ n\,\};\\
\mathit{Fin} : \mathit{Nat} \rightarrow \mathsf{Type};\\
\mathit{Fin} = \lambda n \rightarrow (l : \{\, \mathit{zero}\ \mathit{suc}\,\}) *\\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \{\, \mathit{zero} \rightarrow (n' : \mathit{Nat}) * \mathit{EqNat}\ (\mathit{suc}\ n')\ n\\
\qquad\qquad\qquad\ \mid \mathit{suc} \quad \rightarrow (n' : \mathit{Nat}) * (\mathsf{Rec}\ [\,\mathit{Fin}\ n'\,]) * \mathit{EqNat}\ (\mathit{suc}\ n')\ n\,\};
\end{array}$$

Terms corresponding to the Agda constructors, for instance _::_, are definable:

$$\begin{array}{l}
\mathit{cons} : (A : \mathsf{Type}) \rightarrow (n : \mathit{Nat}) \rightarrow A \rightarrow \mathit{Vec}\ A\ n \rightarrow \mathit{Vec}\ A\ (\mathit{suc}\ n);\\
\mathit{cons} = \lambda A\ n\ a\ v \rightarrow ('\mathit{cons}, (n, (a, (\mathsf{fold}\ v, \mathit{reflNat}\ (\mathit{suc}\ n)))));
\end{array}$$

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 datatype-generic programming via 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, then 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:

$$U : \mathsf{Type}; \mathit{El} : U \rightarrow \mathsf{Type};$$

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

$$\begin{array}{l}
U = (l : \{\, \mathit{enum}\ \mathit{sigma}\ \mathit{box}\,\}) * \mathsf{case}\ l\ \mathsf{of}\ \{\, \mathit{enum} \quad \rightarrow \mathit{Nat}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{sigma} \rightarrow \mathsf{Rec}\ [(a : U) * (\mathit{El}\ a \rightarrow U)]\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{box} \quad\ \rightarrow \mathsf{Rec}\ [\uparrow U\,]\,\};
\end{array}$$

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:

$$\begin{array}{l}
El = \lambda u \to \textsf{split}\ u\ \textsf{with}\ (u_l, u_r) \to \textsf{!}\ \textsf{case}\ u_l\ \textsf{of} \\
\qquad \{\ enum \ \ \to [Fin\ u_r] \\
\qquad \ \mid sigma \to [\textsf{unfold}\ u_r\ \textsf{as}\ u_r' \to \textsf{split}\ u_r'\ \textsf{with}\ (b, c) \to (x : El\ b) * El\ (c\ x)] \\
\qquad \ \mid box \quad\ \to [\textsf{unfold}\ u_r\ \textsf{as}\ u_r' \to \textsf{Rec}\ [El\ (!u_r')]]\};
\end{array}$$

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 (subject to the requirement that there is exactly one definition per declaration), 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:

$$\begin{array}{l}
nat : U = ('sigma, \textsf{fold}\ (('enum, suc\ (suc\ zero)), (\lambda i \to \\
\qquad\qquad \textsf{split}\ i\ \textsf{with}\ (i_l, i_r) \to \textsf{!}\ \textsf{case}\ i_l\ \textsf{of}\ \{\ zero \to [('enum, suc\ zero)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid suc \ \ \to [('box, \textsf{fold}\ [nat])]\})));
\end{array}$$

We can now define a generic decidable equality $eq{:}(a{:}U) \to El\ a \to El\ a \to Bool$ 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 $El\ nat \to U$.

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.

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

$$\frac{}{\Delta; x = t \vdash x = t} \qquad \frac{\Delta \vdash x = t \quad x \not\equiv y}{\Delta; y : \sigma \vdash x = t} \qquad \frac{\Delta \vdash x = t \quad x \not\equiv y}{\Delta; y = u \vdash x = t}$$

Values (v), weak head normal forms (w) and neutral values (n) are defined as follows:

$$\begin{array}{l}
v\ ::= w \mid n \\
w ::= \textsf{Type} \mid (x : \sigma) \to \tau \mid \lambda x \to t \mid (x : \sigma) * \tau \mid (t, u) \mid \{\bar{l}\} \mid {}'l \mid {\uparrow}\sigma \mid [t] \mid \textsf{Rec}\ \sigma \mid \textsf{fold}\ t \\
n\ ::= x \mid n\ t \mid \textsf{split}\ n\ \textsf{with}\ (x, y) \to t \mid \textsf{case}\ n\ \textsf{of}\ \{\overline{l \to t}^{\,|}\} \mid \textsf{!}n \mid \textsf{unfold}\ n\ \textsf{as}\ x \to t
\end{array}$$

We specify β-reduction using a big-step semantics; $\Delta \vdash t \rightsquigarrow v$ means that t β-reduces to v in the context Δ. To avoid cluttering the rules with renamings we give simplified rules in which we assume that variables are suitably fresh; $x \# \Delta$ means that x is not declared in Δ. Reduction and equality do not keep track of all type information, so we allow declarations without a type signature, denoted by $x{:}$, and use the abbreviation $x := t$ for $x{:}; x = t$. We also overload ; for concatenation of contexts:

$$\frac{}{\varDelta \vdash w \rightsquigarrow w} \qquad \frac{\varDelta \vdash t \rightsquigarrow (t_0, t_1) \quad x, y \mathrel{\#} \varDelta \quad \varDelta \vdash \mathsf{let}\ x := t_0; y := t_1\ \mathsf{in}\ u \rightsquigarrow v}{\varDelta \vdash \mathsf{split}\ t\ \mathsf{with}\ (x, y) \to u \rightsquigarrow v}$$

$$\frac{\varDelta \vdash x = t \quad \varDelta \vdash t \rightsquigarrow v}{\varDelta \vdash x \rightsquigarrow v} \qquad \frac{\varDelta \vdash t \rightsquigarrow \lambda x \to t' \quad x \mathrel{\#} \varDelta \quad \varDelta \vdash \mathsf{let}\ x := u\ \mathsf{in}\ t' \rightsquigarrow v}{\varDelta \vdash t\ u \rightsquigarrow v}$$

$$\frac{\varDelta \vdash t \rightsquigarrow {}'l_i \quad \varDelta \vdash u_i \rightsquigarrow v}{\varDelta \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\overline{l_i \to u_i}^{\mid}\} \rightsquigarrow v} \qquad \frac{\varDelta \vdash t \rightsquigarrow [u] \quad \varDelta \vdash u \rightsquigarrow v}{\varDelta \vdash {!t} \rightsquigarrow v}$$

$$\frac{\varDelta \vdash t \rightsquigarrow \mathsf{fold}\ t' \quad x \mathrel{\#} \varDelta \quad \varDelta \vdash \mathsf{let}\ x := t'\ \mathsf{in}\ u \rightsquigarrow v}{\varDelta \vdash \mathsf{unfold}\ t\ \mathsf{as}\ x \to u \rightsquigarrow v} \qquad \frac{\varDelta; \varGamma \vdash t \rightsquigarrow v \quad \varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ v \mapsto v'}{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ t \rightsquigarrow v'}$$

The let rule uses the auxiliary relation $\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ v \mapsto v'$, which pushes lets inside constructors. We only give a representative selection of the rules for this relation. Note that it maps neutral terms to neutral terms:

$$\frac{x \mathrel{\#} \varDelta; \varGamma}{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ \lambda x \to t \mapsto \lambda x \to \mathsf{let}\ \varGamma\ \mathsf{in}\ t} \qquad \frac{}{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ [t] \mapsto [\mathsf{let}\ \varGamma\ \mathsf{in}\ t]}$$

$$\frac{\varDelta; \varGamma \nvdash x = t}{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ x \mapsto x} \qquad \frac{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ n \mapsto n'}{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ n\ t \mapsto n'\ (\mathsf{let}\ \varGamma\ \mathsf{in}\ t)}$$

$$\frac{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ n \mapsto n' \quad x \mathrel{\#} \varDelta; \varGamma}{\varDelta \vdash \mathsf{let}\ \varGamma\ \mathsf{in}\ \mathsf{unfold}\ n\ \mathsf{as}\ x \to t \mapsto \mathsf{unfold}\ n'\ \mathsf{as}\ x \to \mathsf{let}\ \varGamma\ \mathsf{in}\ t}$$

Finally we give some of the rules for computations which are stuck:

$$\frac{\varDelta \nvdash x = t}{\varDelta \vdash x \rightsquigarrow x} \qquad \frac{\varDelta \vdash t \rightsquigarrow n}{\varDelta \vdash t\ u \rightsquigarrow n\ u} \qquad \frac{\varDelta \vdash t \rightsquigarrow n}{\varDelta \vdash \mathsf{unfold}\ t\ \mathsf{as}\ x \to u \rightsquigarrow \mathsf{unfold}\ n\ \mathsf{as}\ x \to u}$$

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:[7]

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

However, we still want the ordinary α-equality

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

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

[7] In this particular example the definition is not actually recursive, but this is irrelevant, because we do not distinguish between recursive and non-recursive definitions.

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

We also want to allow weakening:

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

We achieve the goals outlined above by specifying that two expressions of the form $\mathsf{let}\ \Gamma\ \mathsf{in}\ t$ and $\mathsf{let}\ \Gamma'\ \mathsf{in}\ t'$ 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 some relevant partial bijection). In the implementation we construct this identification lazily: if we are forced to identify two let-bound variables, we replace the definitions of these variables with a single, fresh (undefined) variable and check whether the original definitions are equal. This way we do not unfold recursive definitions more than once.

We specify partial bijections using $\varphi ::= \varepsilon \mid \varphi; (\iota, o)$, where $\iota, o ::= x \mid -$. Here $(x, -)$ is used when the variable x is ignored, i.e., not a member of the partial bijection. Lookup is specified as follows:

$$\frac{}{\varphi; (x, y) \vdash x \sim y} \qquad \frac{\varphi \vdash x \sim y \quad x \not\equiv \iota \quad y \not\equiv o}{\varphi; (\iota, o) \vdash x \sim y}$$

We specify α- and β-equality at the same time, indexing the rules on the metavariable $\kappa \in \{\alpha, \beta\}$, because all but one rule is shared between the two equalities. 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:

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

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 most of these rules—the following are typical examples:

$$\frac{\begin{array}{l}\varphi : \Delta \sim \Delta' \vdash t \equiv_\kappa t' \\ \varphi : \Delta \sim \Delta' \vdash u \equiv_\kappa u'\end{array}}{\varphi : \Delta \sim \Delta' \vdash t\ u \equiv_\kappa t'\ u'} \qquad \frac{\varphi; (x, x') : (\Delta; x{:}) \sim (\Delta'; x'{:}) \vdash t \equiv_\kappa t'}{\varphi : \Delta \sim \Delta' \vdash \lambda x \rightarrow t \equiv_\kappa \lambda x' \rightarrow t'}$$

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

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

Finally we have some rules for let expressions. Empty lets can be added, and contexts can be merged (we omit the symmetric cases):

$$\frac{\varphi : \Delta \sim \Delta' \vdash \mathsf{let}\ \varepsilon\ \mathsf{in}\ t \equiv_\kappa t'}{\varphi : \Delta \sim \Delta' \vdash t \equiv_\kappa t'} \qquad \frac{\varphi : \Delta \sim \Delta' \vdash \mathsf{let}\ \Gamma; \Gamma'\ \mathsf{in}\ t \equiv_\kappa t'}{\varphi : \Delta \sim \Delta' \vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \Gamma'\ \mathsf{in}\ t \equiv_\kappa t'}$$

There is also a congruence rule. 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"). Note that ; is overloaded for concatenation:

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

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:

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

Alternatively, we can ignore a declaration:

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

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:

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

The definition of an ignored variable has to be ignored as well:

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

Definitions and declarations can also be reordered if they do not depend on each other. For reasons of space we omit the details.

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

5 The Type System

We define a bidirectional type system. There are three main, mutually inductive judgements: $\Gamma \vdash \Delta$, which means that Δ is well-formed with respect to Γ; $\Gamma \vdash t \Leftarrow \sigma$, 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. We also use $\Gamma \vdash x : \sigma$, which means that $x : \sigma$ is visible in Γ:

$$\frac{}{\Gamma; x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash x : \sigma \quad x \not\equiv y}{\Gamma; y : \tau \vdash x : \sigma} \qquad \frac{\Gamma \vdash x : \sigma}{\Gamma; y = t \vdash x : \sigma}$$

Context formation is specified as follows:

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

There is one type checking rule which applies to all terms: the *conversion rule*. This rule changes the direction: if we want to check whether a term has type τ, we can first infer the type σ and then verify that σ and τ are convertible:

$$\frac{\Gamma \vdash \tau \Leftarrow \mathsf{Type} \quad \Gamma \vdash t \Rightarrow \sigma \quad \Gamma \vdash \sigma \equiv_\beta \tau}{\Gamma \vdash t \Leftarrow \tau}$$

The remaining rules apply to specific term formers. We have chosen to simplify some of the rules to avoid the use of renamings. This means that the type system, as given, is not closed under α-equality. The rules below use two new metavariables: $\diamond$, which stands for the quantifiers $\rightarrow$ and $*$; and $\Leftrightarrow$, which can be instantiated with either $\Rightarrow$ or $\Leftarrow$. We also use the notation $\Gamma \vdash t \Rightarrow^\beta \sigma$, which stands for $\Gamma \vdash t \Rightarrow \sigma'$ and $\Gamma \vdash \sigma' \rightsquigarrow \sigma$. This is used when we match against a particular type constructor:

$$\frac{\begin{array}{l} \Gamma \vdash \Delta \quad \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash \rho \equiv_\alpha \mathsf{let}\ \Delta\ \mathsf{in}\ \sigma \quad \Gamma; \Delta \vdash t \Leftarrow \sigma \end{array}}{\Gamma \vdash \mathsf{let}\ \Delta\ \mathsf{in}\ t \Leftarrow \rho} \qquad \frac{\Gamma \vdash \Delta \quad \Gamma; \Delta \vdash t \Rightarrow \sigma}{\Gamma \vdash \mathsf{let}\ \Delta\ \mathsf{in}\ t \Rightarrow \mathsf{let}\ \Delta\ \mathsf{in}\ \sigma}$$

$$\frac{\varepsilon \vdash \Gamma \quad \Gamma \vdash x : \sigma}{\Gamma \vdash x \Rightarrow \sigma} \qquad \frac{\varepsilon \vdash \Gamma}{\Gamma \vdash \mathsf{Type} \Leftrightarrow \mathsf{Type}}$$

$$\frac{\begin{array}{c} \Gamma \vdash \sigma \Leftarrow \mathsf{Type} \\ \Gamma, x : \sigma \vdash \tau \Leftarrow \mathsf{Type} \end{array}}{\Gamma \vdash (x : \sigma) \diamond \tau \Rightarrow \mathsf{Type}} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash \rho \rightsquigarrow (x : \sigma) \rightarrow \tau \\ \Gamma, x : \sigma \vdash t \Leftarrow \tau \end{array}}{\Gamma \vdash \lambda x \rightarrow t \Leftarrow \rho} \qquad \frac{\begin{array}{l} \Gamma \vdash t\ \Rightarrow^\beta (x : \sigma) \rightarrow \tau \\ \Gamma \vdash u \Leftarrow \sigma \quad x \,\#\, \Gamma \end{array}}{\Gamma \vdash t\ u \Rightarrow \mathsf{let}\ x : \sigma = u\ \mathsf{in}\ \tau}$$

$$\frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \quad \Gamma \vdash \rho \rightsquigarrow (x : \sigma) * \tau \\ \Gamma \vdash t \Leftarrow\ \sigma \quad x \,\#\, \Gamma \\ \Gamma \vdash u \Leftarrow \mathsf{let}\ x : \sigma = t\ \mathsf{in}\ \tau \end{array}}{\Gamma \vdash (t, u) \Leftarrow \rho} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \quad x, y \,\#\, \Gamma \\ \Gamma \vdash t \Rightarrow^\beta (x : \sigma) * \tau \quad \Gamma \vdash t \rightsquigarrow z \\ \Gamma; x : \sigma; y : \tau; z = (x, y) \vdash u \Leftarrow \rho \end{array}}{\Gamma \vdash \mathsf{split}\ t\ \mathsf{with}\ (x, y) \rightarrow u \Leftarrow \rho}$$

$$\frac{\varepsilon \vdash \Gamma}{\Gamma \vdash \{\bar{l}\} \Rightarrow \mathsf{Type}} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash \rho \rightsquigarrow \{\bar{l}\} \quad m \in \bar{l} \end{array}}{\Gamma \vdash {}'m \Leftarrow \rho} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash t \Rightarrow^\beta \{\bar{l}\} \quad \Gamma \vdash t \rightsquigarrow x \\ (\Gamma, x = {}'l_i \vdash u_i \Leftarrow \rho)_i \end{array}}{\Gamma \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\overline{l \rightarrow u}\} \Leftarrow \rho}$$

$$\frac{\Gamma \vdash \sigma \Leftarrow \mathsf{Type}}{\Gamma \vdash {\uparrow}\sigma \Rightarrow \mathsf{Type}} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash \rho \rightsquigarrow {\uparrow}\sigma \\ \Gamma \vdash t \Leftarrow \sigma \end{array}}{\Gamma \vdash [t] \Leftarrow \rho} \qquad \frac{\Gamma \vdash t \Rightarrow \sigma}{\Gamma \vdash [t] \Rightarrow {\uparrow}\sigma} \qquad \frac{\Gamma \vdash t \Rightarrow^\beta {\uparrow}\sigma}{\Gamma \vdash {!}t \Rightarrow \sigma} \qquad \frac{\Gamma \vdash t \Leftarrow {\uparrow}\sigma}{\Gamma \vdash {!}t \Leftarrow \sigma}$$

$$\frac{\Gamma \vdash \sigma \Leftarrow {\uparrow}\mathsf{Type}}{\Gamma \vdash \mathsf{Rec}\ \sigma \Rightarrow \mathsf{Type}} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash \rho \rightsquigarrow \mathsf{Rec}\ \sigma \\ \Gamma \vdash t \Leftarrow {!}\sigma \end{array}}{\Gamma \vdash \mathsf{fold}\ t \Leftarrow \rho} \qquad \frac{\Gamma \vdash t \Rightarrow \sigma}{\Gamma \vdash \mathsf{fold}\ t \Rightarrow \mathsf{Rec}\ [\sigma]} \qquad \frac{\begin{array}{l} \Gamma \vdash \rho \Leftarrow \mathsf{Type} \\ \Gamma \vdash t\ \Rightarrow^\beta \mathsf{Rec}\ \sigma \\ \Gamma \vdash t \rightsquigarrow y \quad x \,\#\, \Gamma \\ \Gamma; x{:}{!}\sigma; y = \mathsf{fold}\ x \vdash u \Leftarrow \rho \end{array}}{\Gamma \vdash \mathsf{unfold}\ t\ \mathsf{as}\ x \rightarrow u \Leftarrow \rho}$$

The let rules have a side-condition: the context Δ must contain exactly one definition for every declaration, as specified in Sect. 2.1. The case rule's indexed premise must hold for each of the branches, and there must be exactly one branch for every label in $\{\bar{l}\}$ (recall that $\{\bar{l}\}$ stands for a *set* of labels).

Above we have listed the *dependent* elimination rules for products, labels and Rec (the rules for split, case and unfold). There are also non-dependent variants, which do not require the scrutinee to reduce to a variable, but whose premises do not get the benefit of equality constraints.

6 Conclusions

The definition of ΠΣ 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 definition or declaration 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 ΠΣ 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 reduces to a variable. Unfortunately subject reduction for open terms does not hold in the current design, because a variable may get replaced by a neutral term during reduction. We may allow dependent elimination for arbitrary terms again in a later version of ΠΣ.

Having a small language makes complete reflection feasible, opening the door for generic programming. Another goal is to develop ΠΣ's metatheory formally. The distance between the specification and the implementation seems small enough that we plan to develop a verified version of the type checker in Agda (using the partiality monad). This type checker can then be translated into ΠΣ itself. Using this implementation we hope to be able to formally verify central aspects of (some version of) the language, most importantly type-soundness: β-reduction does not get stuck for closed, well-typed programs.

References

Thorsten Altenkirch and Nicolas Oury. ΠΣ: 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 *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, 2007.

Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, 1998.

James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: A standalone typechecker for ETT. In *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 at `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 *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, 2005.

Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants.* PhD thesis, Université Paris 11, 2008.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, 2007.

Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *The 1998 ACM SIGPLAN Workshop on ML*, 1998.