ΠΣ: A Core Language for Dependently Typed Programming

Thorsten Altenkirch Nicolas Oury

University of Nottingham
{txa,npo}@cs.nott.ac.uk

Abstract

We introduce ΠΣ, a core language for dependently typed programming. Our intention is that ΠΣ should play the role extensions of System F are playing for conventional functional languages with polymorphism, like Haskell. The core language incorporates mutual dependent recursive definitions, **Type** : **Type**, Π- and Σ-types, finite sets of labels and explicit constraints. We show that standard constructions in dependently typed programming can be easily encoded in our language. We address some important issues: having an equality checker which unfolds recursion only when needed, avoiding looping when typechecking *sensible* programs; the simplification of type checking for eliminators like case by using equational constraints, allowing the flexible use of case expressions within dependently typed programming and the representation of dependent datatypes using explicit constraints.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory

General Terms Dependent types, core language

Keywords Type theory, Type systems

1. Introduction

Functional programmers are beginning to discover the power of dependent types (Aspinall and Hofmann 2005) — this is witnessed by a recent surge of languages and systems. Some of them like Agda (Norell 2007) and Epigram (McBride and McKinna 2004b) are off-springs of implementations of Type Theory, others like Ω (Sheard 2006) and Ynot (Nanevski et al. 2006) are based on functional programming, and a 3rd class of systems like ATS (Cui et al. 2005) and Delphin (Poswolsky and Schürmann 2007) are inspired by the Logical Framework. Clearly there is the need for consolidation in this area to enable people to share programs, ideas and tools. The present paper attempts to contribute to this goal, by proposing a very minimal core language, which, we hope, could play a role for dependently typed programming as System F and $F^{\leqslant}$ (F-sub) have played for ordinary functional programming.

We dub our language ΠΣ so that its name already comprises two of it's most important type constructors Π-types (dependent function types) and Σ-types (dependent product types). Unlike most systems based on Type Theory but like Augustsson's early proposal Cayenne (Augustsson 1998), our language has mutual, general recursion and hence is, without further restrictions, unsuitable as a logical system. It is, after all, intended to be primarily a core language for programming, not for reasoning. Apart from Σ- and Π-types our language has only a few more, namely:

Type : **Type** This is the simplest choice, the most general form of impredicative polymorphism possible. It is avoided in systems used for reasoning because it destroys logical consistency. We don't care because we have lost consistency already by allowing recursion.

Finite types A finite type is given by a collection of labels, e.g. $\{\mathsf{true},\mathsf{false}\}$ to define the type of Booleans. Our labels are a special class and can be reused, opening the scope for a hereditary definition of subtyping.

Lifting We are using boxes to control the unfolding of recursion. On the level of types this is reflected by a lifting operator on types $(-)_\perp$ which enables us to distinguish lazy and eager datatypes. This is essential since both have different behaviours for symbolic evaluation.

Constrained types We allow types to contain first order equational constraints which are automatically handled by a constraint solver complete for the first order part of our language. This way we can encode most inductive families and dependent pattern matching straightforwardly.

We haven't included equality types, because most uses of equality types can be captured using constrained types - we will discuss possible extensions in the conclusions. Our proposal doesn't contain any features related to implicit syntax and elaboration, which are essential to make dependent types accessible to the programmer. However, this aspect is clearly a part of the translation of a high level language into our core language and hence we will leave it out.

Related work

We have already mentioned Augustsson's Cayenne, so what are the differences? First of all, since not intended as a core language, Cayenne has already a number of high level features such as record types, datatypes, pattern matching and implicit syntax. Unlike ΠΣ's **Type** : **Type** Cayenne implements a hierarchy of universes to support phase distinction. [1] However, the main innovations of ΠΣ over Cayenne are in the following areas:

Controlled unfolding of recursion Like for Cayenne, ΠΣ's type checking problem is undecidable — a fate shared by any language with *full blown dependency* and recursion. In practice, this doesn't matter as long as the type checker only diverges due to the presence of non-terminating terms within types, because those programs should be rejected anyway. However, a Cayenne compiler may loop on sensible programs, and used a

[1] Augustsson himself expressed doubts whether this was a good idea.

heuristic approach to decide which recursive definitions to unfold. We replace this by a much more principled approach based on boxed areas in types and contexts.

Local case expressions Cayenne only allows case-expressions over variables, while other current systems introduce other restrictions, e.g. Epigram only supports top-level pattern matching, while Coq's *Calculus of Inductive Constructions* [CIC] (The Coq Development Team 2008) requires an explicit **using**-clause in many situations. These restrictions are a reflection of the difficulty to implement dependently typed eliminators: in each branch you have to be able to exploit the fact that the scrutinee of the case equals the pattern. We offer a novel solution to this problem by automatically exploiting equational constraints generated by the eliminators for finite types and Σ-types.

Moreover Cayenne has equality types which can be used for explicit equality reasoning, while ΠΣ uses constraints which while less general, considerably reduces the amount of noise generated by equational reasoning.

Implementation

A prototypical typechecker and evaluator for ΠΣ has been implemented in Haskell and is available from our web pages. The examples in this paper have been typechecked with this checker. We discuss some aspects of the implementation in section 4.

A web interface to the implementation can be reached through: `http://www.cs.nott.ac.uk/~npo`.

P	::	ε	empty program
		$x : t.P$	declaration
		$x = t.P$	definition
t, u, σ, τ	::	x	variables
		Type	type of all types
		$(x : \sigma) \to \tau$	Π-types
		$\lambda x \to t$	λ-abstraction
		$t\,u$	application
		$x : \sigma; \tau$	Σ-types
		(t, u)	pairs
		split $(x, y) = t$ **in** u	Σ-elimination
		$\{L_1, L_2, \ldots, L_n\}$	finite types
		L	label
		case t **of** $\{L_1 \to t_1 \mid L_2 \to t_2 \mid \cdots \mid L_n \to t_n\}$	case expressions
		#	impossible
		$(t \equiv u) \Rightarrow \sigma$	constrained type
		$x : \sigma \mid t \equiv u$	explicit constraint
		let P **in** t	letrec
		$\sigma_\perp$	lifting
		$[t]$	box
		$!t$	box opener

Figure 1: Syntax

2. Using ΠΣ

Since our language is a core language it is not primarily intended to be used by humans. However, at least for small examples it is quite possible to program directly in ΠΣ and doing so is a good way to introduce its features and to show how high level features would be mapped into core language constructs.

2.1 Overview

We start with a quick overview of the language — the syntax is given in figure 1 — we abuse notation by identifying metavariables and syntactic classes. We assume as given an infinite supply of labels L and variables x. We distinguish them by using roman font for labels (e.g. true) and italic for variables (e.g. x). We define the syntax of programs P and terms t, u and types σ, τ. While there is no formal difference between terms and types on the level of syntax we use different metavariables to suggest the intended use. Programs P are sequences of declarations $(x : \sigma)$ and, possibly recursive, definitions $x = t$. Every name has to be declared before it is used, and every declared name has to be eventually defined.

Our syntax for Π-types $(x : \sigma) \to \tau$ is the same as used in Agda and Cayenne, we also adopt the usual conventions that $\lambda x \vec{y} \to t :\equiv \lambda x \to \lambda \vec{y} \to t$ and that $(x\ \vec{y} : \sigma) \to \tau :\equiv (x : \sigma) \to (\vec{y} : \sigma) \to \tau$. Non dependent function types arise when x does not appear in the codomain type, in which case we write $\sigma \to \tau$ for $(x : \sigma) \to \tau$.

We write Σ-types as $x : \sigma; \tau$, brackets are not compulsory. The combinator ; is right associative, i.e. we read $x : \sigma; y : \tau; \rho$ as $x : \sigma; (y : \tau; \rho)$. Elements of Σ-types are written as tuples (t, u), and corresponding to the right associativity of ; we adopt the convention that $(t, u, v) :\equiv (t, (u, v))$. Tuples can be deconstructed using **split** which corresponds to a special case of pattern matching using **let** in languages like Haskell. We shall use split-patterns for n-ary tuples, which is easily derivable from the binary case. As for Π-types we omit the binder if the variable isn't used, i.e. $\sigma; \tau$ is a type of non-dependent pairs.

Finite types, i.e. set of labels are written using set-theoretic notation, e.g. $Bool = \{\text{true}, \text{false}\}$ after declaring $Bool : \mathbf{Type}$, and we have that $\text{true} : Bool$ and $\text{false} : Bool$. We analyze elements of finite sets using **case**, e.g. given $b : Bool$ we write **if** b **then** t **else** u as **case** $b\{\text{true} \to t \mid \text{false} \to u\}$, but note that our typing rule is stronger than the conventional one because we add constraints when analyzing t, u. Since our constraints may become inconsistent, i.e. we may be able to derive $\text{true} \equiv \text{false}$, we have a special constant # for *impossible*, which has any type in an inconsistent context.

We can also use explicit constraints, we write $(t \equiv p) \Rightarrow \sigma$ to denote elements of σ under the assumption that $t \equiv p$. Here p is a left-linear pattern built from (unconstrained) variables, tupling and labels. Dually we write $x : \sigma \mid t \equiv p$ for elements of σ such that the constraint $t \equiv p$ is satisfied . Clearly, in the presence of equality types $\Rightarrow$ could be encoded using $\to$ and $\mid$ using ;. However, the advantage of using $\Rightarrow$ and $\mid$ is that the constraints are used *silently* without any noise in the term. The restriction in the form of constraints is required to maintain (partial) decidability, it is subject of further investigations to what degree they can be liberalized.

Programs can be local to a term (or type) using **let**. We control the unfolding of recursive definitions using boxes $[t]$. Definitions from outside a box are not expanded inside. We write $\sigma_\perp$ for the type of boxes containing elements of σ. Boxes can be opened using ! which unfreezes the recursive definitions inside a box. We also use a variant of **case** written **case** which boxes all its branches – the precise definition will be given later.

Our implementation uses an ASCCIfication of the syntax presented here, the details are available from the webpage.

2.2 Algebraic datatypes

The reader may have already noticed that we do not introduce any constructs for algebraic datatypes. Indeed, this is not necessary, because they can be encoded. We start with labeled sums, e.g. types like Haskell's *Either* can be represented as follows:

$$Either : \mathbf{Type} \to \mathbf{Type} \to \mathbf{Type}$$
$$Either = \lambda A\ B \to l : \{\text{left}, \text{right}\};$$
$$\mathbf{case}\ l\ \mathbf{of}$$

$$\begin{array}{l}\{\ \text{left}\ \ \to A\\ \mid \text{right} \to B\}\end{array}$$

The idea is that we represent elements of *Either A B* as a Σ type whose first component is a label, i.e. an element of $\{\text{left}, \text{right}\}$ and the 2nd component is, **depending** on the first either A or B. Combining this technique with recursion we can define recursive types like the natural numbers, using the defined *Unit*-type:

$$\begin{array}{l}\mathit{Unit} : \mathbf{Type}\\ \mathit{Unit} = \{\text{unit}\}\\ \\ \mathit{Nat} : \mathbf{Type}\\ \mathit{Nat} = l : \{\text{zero}, \text{succ}\};\\ \qquad \underline{\mathbf{case}}\ l\ \mathbf{of}\\ \qquad\quad \{\ \text{zero} \to \mathit{Unit}\\ \qquad\quad \mid \text{succ} \to \mathit{Nat}\}\end{array}$$

E.g. the number 3 is written as $(\text{succ}, \text{succ}, \text{succ}, \text{zero}, \text{unit}) : \mathit{Nat}$. Using recursion again we can define addition recursively, simulating pattern matching by combining **split** and **case**:

$$\begin{array}{l}\mathit{add} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\ \mathit{add} = \lambda m\ n \to \mathbf{split}\ (l, m') = m\\ \qquad\qquad \mathbf{in}\ \underline{\mathbf{case}}\ l\ \mathbf{of}\\ \qquad\qquad\quad \{\ \text{zero} \to n\\ \qquad\qquad\quad \mid \text{succ} \to (\text{succ}, \mathit{add}\ m'\ n)\}\end{array}$$

In the succ-branch of the **case**, we have to exploit the constraint that $l \equiv \text{succ}$ to be able to deduce that $m' : \mathit{Nat}$.

We notice that both *Nat* and *add* are recursive definitions, we don't make any difference between recursion for programs and types. However, the type checker has to symbolically evaluate recursive definitions. How do we stop it to unfold *Nat* indefinitely? Indeed, the same question arises in the case of *add*, which as we will see soon, can also appear in types. The answer turns out to be the same in both instances: we do not unfold recursion which goes across **case**. We will refine this restriction later using a notion of boxes, and **case** will turn out to be just an instance of this general approach.

Parametric types like lists and polymorphic functions like append are easily implemented by abstracting over **Type**:

$$\begin{array}{l}\mathit{List} : \mathbf{Type} \to \mathbf{Type}\\ \mathit{List} = \lambda A \to l : \{\text{nil}, \text{cons}\};\\ \qquad\qquad \underline{\mathbf{case}}\ l\ \mathbf{of}\\ \qquad\qquad\quad \{\ \text{nil}\ \ \to \mathit{Unit}\\ \qquad\qquad\quad \mid \text{cons} \to (A; \mathit{List}\ A)\}\\ \mathit{append} : (A : \mathbf{Type}) \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A\\ \mathit{append} = \lambda A\ \mathit{as}\ \mathit{bs} \to\\ \qquad\quad \mathbf{split}\ (l, \mathit{aas}) = \mathit{as}\\ \qquad\quad \mathbf{in}\ \underline{\mathbf{case}}\ l\ \mathbf{of}\\ \qquad\qquad\quad \{\ \text{nil}\ \ \to \mathit{bs}\\ \qquad\qquad\quad \mid \text{cons} \to \mathbf{split}\ (a, \mathit{as}') = \mathit{aas}\\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\text{cons}, a, \mathit{append}\ A\ \mathit{as}'\ \mathit{bs})\}\end{array}$$

Similarly to the the *add* function, in the cons-branch of the **case**, we have to exploit the constraint $l \equiv \text{cons}$ in order to check that *aas* is in a Σ type. Indeed, the type of *aas* is a case expression on l and knowing $l \equiv \text{cons}$ allows to reduce it to $(A; \mathit{List}\ A)$.

In our core language instantiation of a polymorphic function like append is always explicit, e.g. we have to write *append Nat ms ns* to apply append to lists of natural numbers. It is the task of the elaborator, which translates high-level language expressions into the core language to infer implicit arguments like the type A for *append*. Unlike in conventional languages this inference is not limited to the type of types but also applies to other types such as the indices of vectors introduced in section 2.3.

n-ary addition

So far we have only emulated features already present in conventional languages like Haskell. To give a flavour of the power of dependency consider the following problem: we want to implement n-ary addition *nadd*, which is terminated by 0, e.g. *nadd* 1 0 and *nadd* 1 2 3 0 are both expressions of type *Nat*. We start by introducing the type of *nadd* which we call *NAdd*:

$$\begin{array}{l}\mathit{NAdd} : \mathbf{Type}\\ \mathit{NAdd} = (n : \mathit{Nat}) \to \mathbf{split}\ (nl, \_) = n\\ \qquad\qquad\qquad\quad \mathbf{in}\ \underline{\mathbf{case}}\ nl\ \mathbf{of}\\ \qquad\qquad\qquad\qquad \{\ \text{zero} \to \mathit{Nat}\\ \qquad\qquad\qquad\qquad \mid \text{succ} \to \mathit{NAdd}\}\end{array}$$

NAdd is a Π-type: its domain is always *Nat* but its codomain depends on the input: if it was 0 it is *Nat* but otherwise it is recursively defined to be *NAdd*. We define *nadd* using a local recursive definition, defining an auxiliary function *nadd*$'$ which keeps track of the sum using an accumulator:

$$\begin{array}{l}\mathit{nadd} : \mathit{NAdd}\\ \mathit{nadd} = \mathbf{let}\ \mathit{nadd}' : \mathit{Nat} \to \mathit{NAdd}\\ \qquad\qquad \mathit{nadd}' = \lambda m\ n \to \mathbf{split}\ (nl, \_) = n\\ \qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \underline{\mathbf{case}}\ nl\ \mathbf{of}\\ \qquad\qquad\qquad\qquad\qquad \{\text{zero}\ \ \to m\\ \qquad\qquad\qquad\qquad\qquad \mid \text{succ} \to \mathit{nadd}'\ (\mathit{add}\ m\ n)\}\\ \qquad\quad \mathbf{in}\ \mathit{nadd}'\ (\text{zero}, \text{unit})\end{array}$$

While *nadd* is an artificial example, it shows that dependent types can be used to represent flexible data formats. This can be used to define a combinators for a flexible data format description language, see (Oury and Swierstra 2008).

2.3 Dependent datatypes

Vectors, i.e. lists with a fixed length are a well known example of dependently typed programming (McBride 2004). There are two ways to define vectors and other dependent datatypes (also called families): by recursion over the index or by using dependent (focused) constructors. Cayenne and previous version of Agda used the first approach while languages like Epigram, Coq's CIC and recent versions of Agda support the latter approach which is more flexible and better in separating constraints on data from the actual data. In $\Pi\Sigma$ we can represent both approaches. We can define vectors by recursion over length:

$$\begin{array}{l}\mathit{Vec}_1 : \mathbf{Type} \to \mathit{Nat} \to \mathbf{Type}\\ \mathit{Vec}_1 = \lambda A\ n \to \mathbf{split}\ (l, n') = n\\ \qquad\qquad\quad \mathbf{in}\ \underline{\mathbf{case}}\ l\ \mathbf{of}\\ \qquad\qquad\qquad\quad \{\ \text{zero} \to \mathit{Unit}\\ \qquad\qquad\qquad\quad \mid \text{succ} \to (A; \mathit{Vec}_1\ A\ n')\}\end{array}$$

We can now implement a variant of *append* which makes it explicit what happens with the length of lists when appending:

$$\begin{array}{l}\mathit{vappend}_1 : (A : \mathbf{Type}) \to (m, n : \mathit{Nat}) \to \mathit{Vec}_1\ A\ m \to \mathit{Vec}_1\ A\ n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Vec}_1\ A\ (\mathit{add}\ m\ n)\\ \mathit{vappend}_1 = \lambda A\ m\ n\ \mathit{as}\ \mathit{bs} \to\\ \qquad\quad \mathbf{split}\ (lm, m') = m\\ \qquad\quad \mathbf{in}\ \underline{\mathbf{case}}\ lm\ \mathbf{of}\\ \qquad\qquad\quad \{\ \text{zero} \to \mathit{bs}\\ \qquad\qquad\quad \mid \text{succ} \to \mathbf{split}\ (a, \mathit{as}') = \mathit{as}\\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (a, \mathit{vappend}_1\ A\ \mathit{as}'\ \mathit{bs})\}\end{array}$$

We can also give a total implementation of *vtail*:

$$vtail_1 : (A : \mathbf{Type}) \to (n : Nat) \to Vec_1\ A\ (\mathrm{succ}, n) \to Vec_1\ A\ n$$
$$vtail_1 = \lambda A\ n\ as \to \mathbf{split}\ (a, as') = as\ \mathbf{in}\ as$$

The alternative is to use constraints to express directly the idea that the family *Vec* is generated by the dependent constructors, here presented in high level Epigram-like syntax

$$\frac{}{\mathrm{nil} : Vec\ A\ \mathrm{zero}} \qquad \frac{a : A; as : Vec\ A\ n}{\mathrm{cons}\ a\ as : Vec\ A\ (\mathrm{succ}\ n)}$$

We can encode this in ΠΣ by applying *Henry Ford's principle* that *you can have the car in any colour as long as it is black* and write:

$$\begin{array}{l} Vec_2 : \mathbf{Type} \to Nat \to \mathbf{Type} \\ Vec_2 = \lambda A\ n \to l : \{\mathrm{nil}, \mathrm{cons}\}; \\ \qquad \underline{\mathbf{case}}\ l\ \mathbf{of} \\ \qquad\quad \{\ \mathrm{nil} \quad \to Unit \mid n \equiv (\mathrm{zero}, \mathrm{unit}) \\ \qquad\quad \mid \mathrm{cons} \to m : Nat; A; Vec_2\ A\ m \\ \qquad\qquad\qquad \mid n \equiv (\mathrm{succ}, m)\ \} \end{array}$$

vappend no longer has to analyze the indices first:

$$\begin{array}{l} vappend_2 : (A : \mathbf{Type}) \to (m, n : Nat) \to Vec_2\ A\ m \to Vec_2\ A\ n \\ \qquad\qquad\qquad\qquad\qquad\qquad \to Vec_2\ A\ (add\ m\ n) \\ vappend_2 = \lambda A\ m\ n\ as\ bs \to \\ \qquad \mathbf{split}\ (l, as') = as \\ \qquad \mathbf{in}\ \underline{\mathbf{case}}\ l\ \mathbf{of} \\ \qquad\quad \{\mathrm{nil} \quad \to \mathbf{case}\ as'\ \mathbf{of}\ \{Unit \to bs\} \\ \qquad\quad \mid \mathrm{cons} \to \mathbf{split}\ (m, a, cs) = as' \\ \qquad\qquad\quad \mathbf{in}\ (\mathrm{cons}, (\mathrm{succ}, add\ m\ n), \\ \qquad\qquad\qquad\qquad a, vappend_2\ A\ m\ n\ cs\ bs)\ \} \end{array}$$

In the nil-branch, the **case** analysis on as' seems unnecessary. In fact, the constraint $l \equiv \mathrm{nil}$, that has just been added, allows to simplify the type of as' from a **case** expression on l to $Unit \mid n \equiv (\mathrm{zero}, \mathrm{unit})$. The constraint $n \equiv (\mathrm{zero}, \mathrm{unit})$ is necessary to ensure that *bs* is of type $Vec_2\ A\ (add\ m\ n)$. Indeed, this constraint allows to reduce $add\ m\ n$ to n. The **case** expression in the nil branch is used to help the type checker to realise that the type of as' has changed. Hence, it can use the newly inferred constraint.

To implement *vtail* we use the special constant # which has any type in a context with inconsistent constraints, i.e. where two different labels are equated:

$$\begin{array}{l} vtail_2 : (A : \mathbf{Type}) \to (n : Nat) \to Vec_2\ A\ (\mathrm{succ}, n) \to Vec_2\ A\ n \\ vtail_2 = \lambda A\ n\ as \to \mathbf{split}\ (l, as') = as \\ \quad \mathbf{in}\ \underline{\mathbf{case}}\ l\ \mathbf{of} \\ \qquad \{\mathrm{nil} \quad \to \mathbf{case}\ as'\ \mathbf{of}\ \{Unit \to \#\} \\ \qquad \mid \mathrm{cons} \to \mathbf{split}\ (m, a, bs) = as' \\ \qquad\quad \mathbf{in}\ bs\} \end{array}$$

The later variant suggests that a clever compiler doesn't have to store the indices at run-time, see (Brady 2005) where this is investigated in a total setting. Another reason for preferring the approach using constraints is that not every dependent datatype has a straightforward representation using recursion over the indices like vectors. A good example is the type of simply typed λ-terms *Lam*, which are indexed over simple types *Ty* and contexts *Con*, which are just lists of types:

$$\begin{array}{l} Ty : \mathbf{Type} \\ Ty = l : \{\mathrm{base}, \mathrm{arr}\}; \\ \quad \mathbf{case}\ l\ \mathbf{of}\ \{\mathrm{base} \to Unit \\ \qquad\qquad \mid \mathrm{arr} \to (Ty; Ty)\ \} \\ Con : \mathbf{Type} \\ Con = List\ Ty \end{array}$$

Our goal is to represent typed λ-terms using de Bruijn indices using the following constructors written in a high level syntax ala Epigram:

$$\frac{G : Con \quad A : Ty}{\mathrm{zero} : Lam\ (\mathrm{cons}\ A\ G)\ A} \qquad \frac{t : Lam\ G\ A}{\mathrm{succ}\ t : Lam\ (\mathrm{cons}\ B\ G)\ A}$$

$$\frac{t : Lam\ (\mathrm{cons}\ A\ G)\ B}{\mathrm{lam}\ t : Lam\ G\ (\mathrm{arr}\ A\ B)} \qquad \frac{t : Lam\ G\ (\mathrm{arr}\ A\ B); u : Lam\ G\ A}{\mathrm{app}\ t\ u : Lam\ G\ B}$$

In ΠΣ we can encode this type using constraints:

$$\begin{array}{l} Lam : Con \to Ty \to \mathbf{Type} \\ Lam = \lambda G\ A \to \\ \qquad l : \{\mathrm{zero}, \mathrm{succ}, \mathrm{lam}, \mathrm{app}\}; \\ \qquad \underline{\mathbf{case}}\ l\ \mathbf{of} \\ \qquad\quad \{\mathrm{zero} \ \to G' : Con \mid G \equiv (\mathrm{cons}, A, G') \\ \qquad\quad \mid \mathrm{succ} \to G' : Con; B : \mathrm{type}\ ; Lam\ G'\ A \\ \qquad\qquad\qquad \mid G \equiv (\mathrm{cons}, B, G') \\ \qquad\quad \mid \mathrm{lam} \ \to B, C : Ty; Lam\ (\mathrm{cons}, B, G))\ C \\ \qquad\qquad\qquad \mid A \equiv (\mathrm{arr}, B, C) \\ \qquad\quad \mid \mathrm{app} \ \to B : Ty; Lam\ G\ (\mathrm{arr}, B, A); Lam\ G\ B\}) \end{array}$$

Given *Lam* we can implement well-typed substitution following (Altenkirch and Reus 1999): Here a substitution is a function from variables to terms, where variables are a subtype of terms:

$$\begin{array}{l} Var : Con \to Ty \to \mathbf{Type} \\ Var = \lambda G\ A \to \\ \qquad l : \{\mathrm{zero}, \mathrm{succ}, \mathrm{lam}, \mathrm{app}\}; \\ \qquad \underline{\mathbf{case}}\ l\ \mathbf{of} \\ \qquad\quad \{\mathrm{zero} \ \to G' : Con \mid G \equiv (\mathrm{cons}, A, G') \\ \qquad\quad \mid \mathrm{succ} \to G' : Con; B : \mathrm{type}\ ; Lam\ G'\ A \\ \qquad\qquad\qquad \mid G \equiv (\mathrm{cons}, B, G')\ \} \\ Subst : Con \to Con \to \mathbf{Type} \\ Subst = \lambda G\ D \to (A : Ty) \to Var\ D\ A \to Lam\ G\ A \end{array}$$

Substitution can now be implemented by recursion over terms:

$$\begin{array}{l} subst : (G\ D : Con) \to Subst\ G\ D \to (A : Ty) \\ \qquad\qquad\qquad\qquad\qquad \to Term\ D\ A \to Term\ G\ A \end{array}$$

using some auxiliary functions.

2.4 Local case expressions

Type checking case expressions in a dependently typed language is much harder than in a conventional one. The reason is that branches cannot be checked uniformly, the fact that we are in a given branch may have to be taken into account. In Cayenne this lead to the restriction that case expressions are only allowed over variables. Indeed, all programs we have presented so far fall into this category. However, there are many programs which do use case over a non-trivial expression. An interesting example is the following implementation of the *filter*-function which returns a list of elements together with evidence that the filter predicate returned true:

$$\begin{array}{l} filter : (A : \mathbf{Type}) \to (P : A \to Bool) \to List\ A \\ \qquad\qquad\qquad\qquad\qquad\qquad \to List\ (a : A \mid P\ a \equiv \mathrm{true}) \\ filter = \lambda A\ P\ as \to \\ \qquad \mathbf{split}\ (l, as') = as' \\ \qquad \mathbf{in}\ \underline{\mathbf{case}}\ l\ \mathbf{of} \\ \qquad\quad \{\ \mathrm{nil} \quad \to (\mathrm{nil}, \mathrm{unit}) \\ \qquad\quad \mid \mathrm{cons} \to \\ \qquad\qquad \mathbf{split}\ (a, bs) = as' \end{array}$$

$$
\begin{array}{l}
\textbf{in case } P\ a \textbf{ of} \\
\quad \{ \text{false} \to \mathit{filter}\ A\ P\ bs \\
\quad \mid \text{true} \ \to (\text{cons}, a, \\
\qquad\qquad \mathit{filter}\ A\ P\ bs) \}\}
\end{array}
$$

In ΠΣ the program above type-checks, because the type checker is able to exploit the equational constraint that $P\ a \equiv \text{true}$ to verify that $a : (a : A \mid P\ a \equiv \text{true})$ in the right hand side of that case expression.

Epigram and Agda offer local pattern matching using a *with*-rule. However, this feature involves a translation to a definition of a local function and is quite complex to implement. Indeed, in Epigram 1 the *with* rule was never implemented and Agda only acquired it recently (Norell 2007). We hope that our approach opens up a new and principled way to deal with this issue.

2.5 Lazy datatypes and boxes

In section 2.2 we have linked the control of unfolding recursion to the use of **case**. However, it is clear that this approach won't work for lazy types like streams. E.g. the definition of of a stream starting from a given number doesn't use any case-analysis at all. Indeed, as already indicated **case** is only syntactic sugar for a common pattern of controlling recursion, which is also applicable to lazy types.

In ΠΣ we have to differentiate between lazy and eager datatypes. This distinction is similar to the distinction between coinductive and inductive types, which is present in total languages like Coq's CIC. Here we are not motivated by totality but that lazy and eager datatypes behave differently w.r.t. symbolic evaluation. In case of an eager list we can evaluate under constructors but we should stop evaluating inside case-expressions. For lazy lists we should stop evaluating under a constructor, but there is no point stopping inside case-expressions.

We introduce lazy datatypes by guarding recursive occurrences with a special type constructor $\sigma_\perp$, elements of $\sigma_\perp$ are constructed by *boxes* $[t] : \sigma_\perp$ where $t : \sigma$. Boxes stop the unfolding of recursive definitions from outside the box.

Using $(-)_\perp$ we can define lazy lists:

$$
\begin{array}{l}
\mathit{LList} : \textbf{Type} \to \textbf{Type} \\
\mathit{LList} = \lambda A \to l : \{ \text{nil}, \text{cons} \}; \\
\quad \underline{\textbf{case}}\ l\ \textbf{of} \\
\qquad \{ \text{nil} \ \ \to \mathit{Unit} \\
\qquad \mid \text{cons} \to (A; (\mathit{LList}\ A)_\perp) \})
\end{array}
$$

The *from* function which recursively constructs an infinite list of increasing numbers can be defined using a box, which prevents the infinite unfolding of this program:

$$
\begin{array}{l}
\mathit{from} : \mathit{Nat} \to \mathit{LList}\ \mathit{Nat} \\
\mathit{from} = \lambda n \to (\text{cons}, n, [\mathit{from}\ (\text{succ}, n)])
\end{array}
$$

However, sometimes we have to open a box to perform a computation with the term inside. For this purpose we introduce a box opener $!t : \sigma$, given $t : \sigma_\perp$. Computationally, $![t]$ reduces to t. Using ! we can define the map operation for lazy lists:

$$
\begin{array}{l}
\mathit{lmap} : (A\ B : \textbf{Type}) \to (A \to B) \to \mathit{LList}\ A \to \mathit{LList}\ B \\
\mathit{lmap} = \lambda A\ B\ f\ \mathit{as} \to \\
\qquad \textbf{split}\ (l, \mathit{as}') = \mathit{as} \\
\qquad \textbf{in case}\ l\ \textbf{of} \\
\qquad\quad \{ \text{nil} \ \ \to (\text{nil}, \text{unit}) \\
\qquad\quad \mid \text{cons} \to \textbf{split}\ (a, bs) = \mathit{as} \\
\qquad\qquad\quad \textbf{in}\ (\text{cons}, f\ a, [\mathit{lmap}\ f\ (!bs)]\}
\end{array}
$$

Using ! we control the unfolding of the infinite list which only gets unfolded on demand.

Finally, we can define **case** in terms of the primitive **case** and the box primitives, i.e.

$$\underline{\textbf{case}}\ t\ \textbf{of} \{ L_1 \to u_1 \mid L_2 \to u_2 \mid \cdots \mid L_n \to u_n$$

is defined as

$$!\textbf{case}\ t\ \textbf{of} \{ L_1 \to [u_1] \mid L_2 \to [u_2] \mid \cdots \mid L_n \to [u_n]$$

case has the same derived typing rule as **case** but behaves differently computationally: recursive calls inside branches are not unfolded, but once the case expression can be reduced, the box will disappear.

For example, if we evaluate $\mathit{add}\ (\text{succ}, \text{zero}, \text{unit})\ (\text{zero}, \text{unit})$, the first **case** expression reduces to its succ-branch. This results in $![(\text{succ}, \mathit{add}\ (\text{zero}, \text{unit})\ (\text{zero}, \text{unit}))]$. ! opens the box, and the recursive call to *add* can be unfolded.

2.6 Induction-recursion

Dybjer and Setzer have developed the concept of *induction-recursion* (Dybjer and Setzer 1999) which justifies and formalizes constructions in Type Theory which mutually define a type by induction and a function by structural recursion. A standard example is the definition of a *universe* defining inductively a type of codes $U : \textbf{Type}$ mutually with a function $\mathit{El} : U \to \textbf{Type}$. E.g. in a partial theory we may want to reflect a universe which reflects its own code. We give a high level specification:

$$\frac{}{u : U} \qquad \frac{a : U; b : \mathit{El}\ a \to U}{\text{pi}\ a\ b : U}$$

$$
\begin{array}{ll}
\mathit{El}\ u & = U \\
\mathit{El}\ (\text{pi}\ a\ b) & = (x : \mathit{El}\ a) \to \mathit{El}\ (b\ x)
\end{array}
$$

It is relatively straightforward to encode this definition in ΠΣ since inductive datatypes and recursive functions are realized using the same mechanism. Hence, the definition of the universe sketched above, boils down to a straightforward inductive-recursive definition in ΠΣ:

$$
\begin{array}{l}
U : \textbf{Type} \\
\mathit{El} : U \to \textbf{Type} \\
U = (l : \{ \text{type}\ , \text{pi} \}; \\
\qquad \underline{\textbf{case}}\ \ l\ \textbf{of} \\
\qquad\quad \{ \text{type} \to \mathit{Unit} \\
\qquad\quad \mid \text{pi} \ \ \to (a : U; \mathit{El}\ a \to U) \}) \\
\mathit{El} = \lambda a \to \textbf{split}\ (l, a') = a \\
\qquad\quad \textbf{in}\ \underline{\textbf{case}}\ l\ \textbf{of} \\
\qquad\qquad \{ \text{type} \to U \\
\qquad\qquad \mid \text{pi} \ \ \to \textbf{split}\ (\mathit{dom}, \mathit{cod}) = a' \\
\qquad\qquad\quad \textbf{in}\ (x : \mathit{El}\ \mathit{dom}) \to \mathit{El}\ (\mathit{cod}\ x)
\end{array}
$$

We are exploiting ΠΣ's general approach to mutual recursion, we need to declare *El* before defining *U*. Unlike in non-dependent programming the order of the mutual definitions matters: we have to define *U* before *El* because *El* only typechecks when *U* is already defined. On the other hand the definition of *U* doesn't rely on the definition of *El*, otherwise we would be in an infinite regress.

We are not claiming that we capture induction-recursion in the sense of (Dybjer and Setzer 1999) since we make no attempt to give criteria when these definitions are admissible in a total theory.

3. Defining ΠΣ

We formally introduce ΠΣ's typing and equality judgments, which are defined inductively by the derivation rules given below.

$$
\begin{array}{lcll}
\Gamma & :: & & \text{empty context} \\
& | & \Gamma, x : \sigma & \text{type assumptions} \\
& | & \Gamma, x = t & \text{definition} \\
& | & \Gamma, t \equiv u & \text{constraint} \\
& | & [\Gamma] & \text{box}
\end{array}
$$

Figure 2: Syntax of contexts

Contexts (see figure 2) are sequences of type assumptions, definitions and constraints; parts of a context may be boxed which we use to express that recursive definitions inside this part are not visible outside. We don't specify the lookup of assumptions in detail, but use the suggestive notation $x : \sigma \in \Gamma$, $t \equiv u \in \Gamma$, and $x = t \in \Gamma$ where the lookup of definitions does not look inside boxes. We write $\mathtt{Def}\,\Gamma = \{x \mid x = t \in \Gamma\}$ for the set of variables defined in a program p and $\mathtt{Decl}\,\Gamma = \{x \mid x : \sigma \in \Gamma\}$ for the set of variables declared in p. We write $\mathrm{Free}\,\Gamma$ for the set of variables which are defined but not declared and which do not appear in any constraints.

We use the following judgments:

$\vdash \Gamma$
: Γ is a well-formed context,

$\Gamma \vdash t : \sigma$
: t is a term with type σ in context Γ. We maintain the invariant that this entails $\vdash \Gamma$ and $\Gamma \vdash \sigma : \mathbf{Type}$.

$\Gamma \vdash t \equiv u$
: t and u are convertible terms. We will only introduce β-rules hence equality is actually independent of the typing. However, equality depends on equational assumptions in the context. To save space and avoid boredom we omit the rules stating that $\equiv$ is a congruence.

$\Gamma \vdash p\,\mathbf{Pat}$
: p is a pattern in Γ. This notion is dependent on the context because we require that all pattern variables are free, i.e. undefined and unconstrained.

While our syntax makes no difference between terms and types, we use the letters t, u for objects which occur at term level and σ, τ for syntactic objects occurring at type level. We write $t[x \leftarrow u]$ for the capture-avoiding substitution of u for x in t, all syntax should be interpreted up to α-equivalence.

3.1 System U

We start with the core type theory containing only Π-types and **Type** : **Type**, see figure 3. This is the initial pure type system and is basically equivalent to Girard's system U. Cardelli has investigated programming in this system using impredicative encodings (Cardelli 1986). It is also possible to derive a looping combinator approximating recursion by encoding Girard's paradox. However, these constructions have only theoretical interest and are not relevant for programming, since they lead to inefficient code and are cumbersome to use. However, **Type** : **Type** is useful to represent higher order polymorphic definitions and reflective metaprogramming.

3.2 Recursive definitions and boxes

In figure 4 we present the rules for local definitions and boxes. We only allow the definition of a variable if it has been declared but not yet defined. We require that all variables declared in a let-expression are also defined. We use the notation $\Gamma.p$ for appending a program to the context — this is possible because programs are a particular collection of context extensions.

$$
\frac{}{\vdash \emptyset} \qquad \frac{\Gamma \vdash \tau : \mathbf{Type}}{\vdash \Gamma, x : \tau}
$$

$$
\frac{\vdash \Gamma \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma t : \sigma \quad \Gamma \vdash \tau : \mathbf{Type} \quad \Gamma \vdash \sigma \equiv \tau}{\Gamma \vdash t : \tau}
$$

$$
\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Type} : \mathbf{Type}}
$$

$$
\frac{\Gamma, x : \sigma \vdash \tau : \mathbf{Type}}{\Gamma \vdash (x : \sigma) \to \tau : \mathbf{Type}} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x \to t : (x : \sigma) \to \tau}
$$

$$
\frac{\Gamma \vdash t : (x : \sigma) \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\,u : \tau[x \leftarrow u]} \qquad \frac{\Gamma \vdash (\lambda x \to t)\,u : \sigma}{\Gamma \vdash (\lambda x \to t)\,u \equiv t[x = u]}
$$

Figure 3: System U

The unfolding of recursive definitions is controlled by boxes: when we enter a box we box the current context thus freezing all recursive definitions from outside the box. Boxes can be opened using !, this is needed in order to unfold recursive definition during a computation. The type constructor $(-)_\perp$ introduces the type of boxed expressions, and can be used explicitly to introduce lazy types.

Other implementations of dependent types such as Coq, Epigram or Agda have to address the issue of unfolding recursion as well. In Coq recursion is only unfolded when it is in an evaluation context. This is only possible because in Coq pattern matching is primitive and there is an explicit distinction between inductive and coinductive types. Agda also has primitive pattern matching and only implements a call-by-value discipline. Epigram on the other hand translates recursive definitions into a core calculus with a recursion combinator which only unfolds recursive definitions in an evaluation context.

The approach we use seems to be closer to the practice in functional programming in reducing recursion to recursive definitions. However, in a dependently typed language we have the additional problem that symbolic evaluation has to terminate. Instead of fixing the evaluation contexts in advance, we provide a flexible solution by using boxes.

Also our approach is very general in allowing us to model different kinds of recursion, e.g. recursive types and recursive values using the same construct. As we have already shown this naturally captures inductive-recursive definitions of universes (section 2.6).

3.3 Σ-types and finite types

Instead of adding primitives to represent datatypes, we encode them using Σ-types and finite types. Σ-types are also useful to represent dependent records and finite sets can be used to encode fields in a record. We present the rules for Σ-types in figure 6 and for finite types in figure 7. While in both cases the formation and introduction rules are straightforward, the remaining rules require some discussion.

The eliminators for a theory with dependent types are much more complex then elimination for simple types. As an example let's consider the type of booleans, in our syntax $\mathit{Bool} = \{\mathrm{false}, \mathrm{true}\}$. The simply typed eliminator is **case**, or if-then-else in this instance:

$$
\frac{\Gamma \vdash t : \mathrm{Bool} \quad \Gamma \vdash u_0, u_1 : \sigma}{\Gamma \vdash \mathbf{case}\,t\,\mathbf{of}\,\{\mathrm{false} \to u_0 \mid \mathrm{true} \to u_1\} : \sigma}
$$

$$\frac{x:\tau\in\Gamma \qquad x\notin \mathtt{Def}\{\Gamma\} \qquad \Gamma\vdash t:\tau}{\vdash \Gamma, x=t} \qquad \frac{x=t\in\Gamma}{\Gamma\vdash x\equiv t}$$

$$\frac{\Gamma.P\vdash t:\tau \qquad \mathtt{Def}\,P=\mathtt{Decl}\,P}{\Gamma\vdash \mathbf{let}\,P\,\mathbf{in}\,t:\mathbf{let}\,P\,\mathbf{in}\,\tau}$$

$$\frac{\Gamma.p\vdash u\equiv v}{\Gamma\vdash \mathbf{let}\,p\,\mathbf{in}\,u\equiv \mathbf{let}\,p\,\mathbf{in}\,v} \qquad \frac{\Gamma.p\vdash u\equiv v \quad \Gamma\vdash v:\sigma}{\Gamma\vdash \mathbf{let}\,p\,\mathbf{in}\,u\equiv v}$$

$$\frac{\Gamma\vdash t:\tau}{\Gamma\vdash [t]:\tau_\perp} \qquad \frac{\Gamma\vdash t:\tau_\perp}{\Gamma\vdash !t:\tau}$$

$$\frac{[\Gamma]\vdash u\equiv v}{\Gamma\vdash [u]\equiv[v]} \qquad \frac{\Gamma\vdash u:\sigma}{\Gamma\vdash ![u]\equiv u}$$

Figure 4: Letrec and boxes

This rule is not sufficient for a system with dependent types because we need to exploit the fact that we are either in the if-branch or in the else branch. Traditionally, e.g. in Coq, this is fixed by providing elimination with a motive:

$$\frac{\Gamma\vdash t:\mathrm{Bool} \quad \Gamma\vdash \sigma:\mathrm{Bool}\to\mathbf{Type} \qquad \begin{array}{l}\Gamma\vdash u_0:\sigma\,\mathrm{false}\\ \Gamma\vdash u_1:\sigma\,\mathrm{true}\end{array}}{\Gamma\vdash \mathbf{case}\,t\,\mathbf{of}\,\{\mathrm{false}\to u_0\mid \mathrm{true}\to u_1\}\,\mathbf{using}\,\sigma:\sigma\,t}$$

This is inconvenient for writing programs because each time we use the eliminator we have to provide the motive σ explicitly. Epigram addresses this problem by generating the motive from high level tactics to generate pattern matching. However, this can be quite complex generating considerable overhead in the generated code. Also it is not trivial to generalize this approach to local case expressions as we have already discussed.

Our approach is to modify the simply typed rule by adding constraints to the context, hence avoiding the need of motives in the core language and simplifying the translation of pattern matching and in particular local case expressions. Instantiated to the case of *Bool* we obtain the following typing rule:

$$\frac{\Gamma\vdash t:\mathrm{Bool} \qquad \begin{array}{l}\Gamma,t\equiv\mathrm{false}\vdash u_0:\sigma\\ \Gamma,t\equiv\mathrm{true}\vdash u_1:\sigma\end{array}}{\Gamma\vdash \mathbf{case}\,t\,\mathbf{of}\,\{\mathrm{false}\to u_0\mid\mathrm{true}\to u_1\}:\sigma}$$

Our case expression can be used in the same flexible way as the simply typed case-expression and we do not need to provide a motive. The price we have to pay is that we have to decide definitional equality with constraints. This is not as hard as it sounds because we never have to instantiate any variables, our definitional equality is not closed under substitution. We would loose this property if we would allow constraints between λ-abstractions, hence this is ruled out by our definition. We will discuss in section 4 how our constraints can be implemented.

In figure 5 we present the rules for allowing constraints in the context. We restrict the left hand sides of constraints to be first-order left-linear patterns. We exploit the constraint mechanism in the elimination rules for Σ-types (**split**) and finite types (**case**). We also add rules reflecting the fact that constructors are injective. In the case of finite types this leads to the possibility of equationally inconsistent contexts. In such a context the special constant # has any type.

3.4 Explicit constraints

In the previous section we have been using constraints to simplify the typing of elimination constants. However, the same mechanism can be put to a more general use: we can add explicit constraints

$$\frac{x\in\mathrm{Free}\,\Gamma}{\Gamma\vdash x\,\mathbf{Pat}} \qquad \frac{}{\Gamma\vdash L\,\mathbf{Pat}}$$

$$\frac{\Gamma\vdash p,p'\,\mathbf{Pat} \quad \mathrm{FV}\,p\cap\mathrm{FV}\,p'=\emptyset}{\Gamma\vdash (p,p')\,\mathbf{Pat}}$$

$$\frac{\Gamma\vdash t,p:\sigma \quad \Gamma\vdash p\,\mathbf{Pat} \quad \mathrm{FV}\,t\cap\mathrm{FV}\,p=\emptyset}{\vdash \Gamma, t\equiv p} \qquad \frac{t\equiv p\in\Gamma}{\Gamma\vdash t\equiv p}$$

Figure 5: Constraints

$$\frac{\Gamma,x:\sigma\vdash\tau:\mathbf{Type}}{\Gamma\vdash(x:\sigma;\tau):\mathbf{Type}} \qquad \frac{\Gamma\vdash u:\sigma \qquad \Gamma\vdash v:\tau[x\leftarrow u]}{\Gamma\vdash(u,v):(x:\sigma;\tau)}$$

$$\frac{\Gamma\vdash u:(x:\sigma;\tau) \qquad \Gamma,u\equiv(x,y)\vdash v:\rho}{\Gamma\vdash \mathbf{split}\,(x,y)=u\,\mathbf{in}\,v:\rho}$$

$$\frac{\Gamma\vdash t\equiv t' \quad \Gamma,t\equiv(x,y)\vdash u\equiv u'}{\Gamma\vdash \mathbf{split}\,(x,y)=t\,\mathbf{in}\,u\equiv\mathbf{split}\,(x,y)=t'\,\mathbf{in}\,u'}$$

$$\frac{\Gamma\vdash \mathbf{split}\,(x,y)=(t,u)\,\mathbf{in}v:\sigma}{\Gamma\vdash \mathbf{split}\,(x,y)=(t,u)\,\mathbf{in}\,v\equiv v[x\leftarrow t,y\leftarrow u]}$$

$$\frac{\Gamma\vdash(t,u)\equiv(t',u')}{\Gamma\vdash t\equiv t' \quad \Gamma\vdash u\equiv u'}$$

Figure 6: Σ-types

$$\frac{\vdash\Gamma}{\Gamma\vdash\{L_0,\ldots,L_{n-1}\}:\mathbf{Type}} \qquad \frac{\vdash\Gamma \quad 0\leqslant i<n}{\Gamma\vdash L_i:\{L_0,\ldots,L_{n-1}\}}$$

$$\frac{(\Gamma,s\equiv l_i\vdash u_i:\sigma)_{0\leqslant i<n} \quad \Gamma\vdash s:\{L_0\ldots L_{n-1}\}}{\Gamma\vdash \mathbf{case}\,s\,\mathbf{of}\,\{L_i\to u_i\}_{0\leqslant i<n}:\sigma}$$

$$\frac{\Gamma\vdash s\equiv s' \quad (\Gamma,s\equiv L_i\vdash u_i\equiv v_i)_{0\leqslant i<n}}{\Gamma\vdash \mathbf{case}\,s\,\mathbf{of}\,\{L_i\to u_i\}_{0\leqslant i<n}\equiv\mathbf{case}\,s'\,\mathbf{of}\,\{L_i\to v_i\}_{0\leqslant i<n}}$$

$$\frac{\Gamma\vdash\mathbf{case}\,L_k\mathbf{of}\{L_i\to u_i\}_{0\leqslant i<n}:\sigma}{\Gamma\vdash\mathbf{case}\,L_k\mathbf{of}\{L_i\to u_i\}_{0\leqslant i<n}\equiv u_k}$$

$$\frac{\Gamma\vdash L_i\equiv L_j \quad i\neq j \quad \Gamma\vdash\sigma:\mathbf{Type}}{\Gamma\vdash\#:\sigma}$$

Figure 7: Finite types

of the form $t\equiv p$ where p is a pattern, i.e. made up from variables, labels and pairs. We have two ways to introduce constraints: we write $(t\equiv p)\Rightarrow\sigma$ to express that σ is typable assuming that the constraint $t\equiv p$ holds, and $x:\sigma\mid t\equiv p$ to indicate that we require the constraint $t\equiv p$ to hold to construct an element of the type. The rules are given in figure 8.

4. Implementing $\Pi\Sigma$

The aim of this work is to provide a system powerful enough to be used as a target for most high level languages with dependent types, while remaining small enough to be easy to study and to implement. We have written a simple proof–of–concept implementa-

$$\frac{\Gamma, t \equiv p \vdash \sigma : \mathbf{Type}}{\Gamma \vdash (t \equiv p) \Rightarrow \sigma : \mathbf{Type}} \qquad \frac{\Gamma, t \equiv p \vdash s : \sigma}{\Gamma \vdash s : (t \equiv p) \Rightarrow \sigma}$$

$$\frac{\Gamma \vdash s : (t \equiv p) \Rightarrow \sigma : \mathbf{Type} \quad \Gamma \vdash t \equiv p}{\Gamma \vdash s : \sigma}$$

$$\frac{\vdash \Gamma, x : \sigma, t \equiv p}{\Gamma \vdash \{x : \sigma \mid t \equiv p\} : \mathbf{Type}} \qquad \frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[x \leftarrow s] \equiv p[x \leftarrow s]}{\Gamma \vdash s : (x : \sigma \mid t \equiv p)}$$

$$\frac{\Gamma \vdash s : (x : \sigma \mid t \equiv p)}{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[x \leftarrow s] \equiv p[x \leftarrow s]}$$

Figure 8: Explicit constraints

tion of ΠΣ. For this implementation, we used the Haskell language (Peyton Jones 2003) and the *ghc* compiler.

The goal of this prototype implementation is not to be efficient or user-friendly, but to demonstrate how to implement ΠΣ in principle. This implementation has been used to typecheck and evaluate the examples presented in this paper.

In this section, we present the key points of this implementation, while trying not to overwhelm the reader with technical details. We concentrate on three points: the general structure of the implementation, the idea we use to support box controlled general conversion, and the way we handle *constraints* during equality checking.

4.1 Structure of the prototype

This implementation is based on a *locally nameless bidirectional type checker*. Giving the details of such a type checker is beyond the scope of this work, see (Coquand 1996; Löh et al. 2008; Chapman et al. 2006). We simply recall here the main ideas underlying such an implementation.

This typechecker separates *syntactical terms* from *semantical values*, corresponding to evaluated terms. The user types in *terms*. Once they have been type checked, those terms can be evaluated into the *Value* type. This type only contains representations of normal forms. It is used internally to represent *types* and for *equality checking*. After evaluation, the values are *quoted* back to a *term* in normal form, which can be printed.

All the functions manipulating terms are *locally nameless* (McBride and McKinna 2004a). Variables in the context are *named* but variables inside the term are designated using *De Bruijn* indices. When a function working on a term goes under a binder, it creates a new *reference*, *i.e.* a fresh name in the context, together with the type of the newly bound variable. Then, it substitutes this new reference to the variable with index 0 in the term, accordingly adjusting the other indices. This way of working with open terms induces two major benefits: using *De Bruijn* variables in closed terms eases the implementation of substitution, and naming local variables inside a term allows to avoid the complexity of index arithmetic.

We separate the terms in two distinct syntactical categories:

- An *inferable* term is a term whose type can be *inferred*. It includes all type constructors, variables and application.
- A *checkable* term is a term which has to be *checked* against a given type. It contains all the remaining constructs.

This separation is a slight restriction with respect to ΠΣ. Indeed, in some situations, the type of a **split**, a **let** or a **case** expression could be inferred. Anyway, this restriction is not problematic in practice: the presence of the type declarations in the **let** bindings gives enough information to the type checker. Indeed, we could have given a more precise account of the typechecker by differentiating between inferable and checkable in the typing rules (Pierce and Turner 1998). However, we refrain from doing this here for the sake of readability and brevity.

When an *inferable* t is *checked* against a given type τ, we have to check that τ is a valid type for t. To do so, we infer the type τ' of t. Then we check that τ and τ' are two variations of the same type, using an *equality check*.

This equality check compares structurally two *Value*s. It uses a *higher-order representation* of the binding constructs: function, Σ–types, Π–types, **split** and **let**. This use of this *higher-order representation* simplifies the treatment of closures: instead of explicitly working with a tuple of the value and its environment, the implementation relies on Haskell closures to represent ΠΣ-closures. Indeed, the only function threading an environment is the *evaluation* function that turns a *term* into a *value*. The representation of *Value*s also relies on Haskell laziness.

This approach leads to a straightforward and short implementation of a basic type checker and moreover – as we shall see – it is quite straightforward to extend with ΠΣ specific features.

4.2 Typechecking with constraints

The first specific feature of ΠΣ is the possibility to extend the equality check with constraints. In this subsection, we sketch how these constraints are used during the conversion in the prototype implementation. This implementation of the constraint system is not aimed at efficiency. The goal is to provide a simple implementation – the code of the part of the implementation handling constraints is less than 200 lines long – easy to understand and eventually to prove correct. Even if we have not formally verified this algorithm yet, we believe it is complete with respect to the typing rules – taking the distinction between inferable and checkable into account. In this section, we explain this algorithm and argue why it is complete.

In order to study the details of our implementation of constraints, we need to describe more precisely the representation of *Value*s. Each constructor of the *Value* type corresponds to a possible head constructors of a normal form. These different head constructor are split into two categories:

- *Canonical values* are values whose head constructors cannot be modified by a further instantiation of a reference to the context. The nature of the term is already known. Their head constructors consist of **Type**, Σ,Π, $\ldots_\perp$ and enumeration types as well as labels, pairs, boxes and functions.
- *Neutrals* corresponds to terms *stuck* on a reference of the context. Their head constructors consist of free variables, application of a neutral value to a term, **split** on a neutral value, opening of a neutral value and **case** expression whose scrutinee is a neutral value.

A constraint is a relation linking two values that are to be equal. The prototype implementation translates each constraint into a rewrite rule.

In order to explain our implementation of constraints enhanced equality checking, we have to stress two important points about the constraints we have to handle.

1. A constraints is rewriting rule of *order 0*. Despite the fact that constraints may contain variables, these variables can not be instantiated when the constraint is used. For example, x does not act as a variable in the rule constraint $x \rightarrow (u, v)$. x is a reference to the exact x of the context, not a variable that could be instantiated during rewriting. No substitution can occur while applying a rewrite rule. Hence, when we look at the system of constraints as a rewriting system, this system is made of *terms without variables*.

2. We only need to extend the conversion with constraints linking a neutral value to a value.

The first point helps us to maintain two invariants:

- The rewrite system is in head normal form. No rule can be applied to head of the left or right hand side of any other rule.
- No rule linking two equal terms is never added to the system. This equality is checked before extending the system. Anyway, such a rule would not extend the equality test.

These two invariants, together with the fact that the rewrite system has no variables, makes the system *confluent* and *weakly normalizing* in absence of recursion. The only cases of non-termination are non-terminating recursive definitions not protected by a box. Without recursion, the system is *weakly normalizing*. We can not hope for more than that, while allowing general recursion.

The second point needs a more precise explanation. Let us study more closely the possible shape of the constraints to show we can always reduce it to a constraint $n \equiv v$, where n is a neutral value.

Every constraint that can be added to the system is of shape $v \equiv p$, where p is a pattern. If p is a variable, then we choose $n = p$. Else, we know that p is either a pair or a label.

Hence, only two cases remain:

- A constraint linking a value v to a label L_i. Either v is a neutral value – and we choose $p = v$ – or it is a canonical value. Such value in an enumeration type is always a label L'. If L is equal to L', then $v \equiv L$ does not bring any new information and the constraint can be discharged. If L and L' are different, then the system of constraints is inconsistent and the only relevant information is this inconsistence.
- A constraint linking a value v to a pair of patterns $(p1, p2)$. Either v is a neutral value – and we choose $p = v$ –, or v is a canonical value. The only canonical values in a Σ type are pairs. Hence v reads $(v1, v2)$. Then $v \equiv (x, y)$ is equivalent to the set of constraints $\{v1 \equiv p1; v2 \equiv p2\}$. We apply the same transformation recursively on both constraints.

The fact that we can always simplify a constraint linking two canonical values is a key property of our implementation. This fact is due to the shape of the constraints generated by our language. With constraints on canonical values in an enumeration or a Σ type, it is always possible to express, by smaller constraints, all the possible consequences of the original constraint. If we include constraints between higher-order constructs – like a constraint linking two functions –, we lose this property: it is impossible to produce a finite set of constraints encoding exactly all the *consequences* of the equality of two λ–terms. Indeed, the consequences of $\lambda x.t \equiv \lambda x.u$ with both terms elements of $(x : \sigma) \to \tau$ are $t[x \leftarrow v] \equiv u[x \leftarrow v]$, for all v :
sigma.

We can now present the implementation of constraints enhanced equality checking. A *system of constraints* is threaded during type checking and equality testing. This system consists in *Maybe* a set of constraints. *Nothing* represents an inconsistent system of constraints, due to a constraint between two different labels. With such a system, the equality test always succeeds and we can check that # is in any type. *Just csts* represents the system generated by the set *csts* of constraints. This set is a list of 0-order rewrite rules from a neutral value to a value.

When we structurally compare two values, we first head-normalize the values with respect to the set of constraints. If a value is neutral, we try to apply each rewrite rule of the system. We then obtain two values, that are in head normal form with respect to the system of constraints. The head constructors of these values are then compared and the equality is recursively called on the sub-components of these values.

This algorithm is far from being optimal. Anyway, the implementation is short and very easy to understand. Moreover, in practice, the number of constraint stays relatively low, as it is proportional to the number of nested **case** and **split** expressions. However, for a full scale implementation, this algorithm could be improved by adding *hash consing* or *congruence closure* (Nieuwenhuis and Oliveras 2005) to the type checker.

4.3 Box controlled recursion

We now turn our attention to the ideas underlying the implementation of $\Pi\Sigma$'s second specific feature: boxes and general recursion. As already mentioned, the prototype reads *terms* from the user. Once, typechecked these terms are *evaluated* to *Value*s, which are used to represent types and for equality testing. One of the benefits of the approach is a simplification: together with the use of higher-order representations, this approach allows to get rid of the threading of an environment during type checking and equality testing. Indeed, evaluation is *separated* from type checking and equality testing. This enhances the readability and the modularity of the implementation.

In this subsection, we show how to maintain this property, while extending the type checker with general recursion controlled by boxes.

Let blocks and general recursion

The main advantage of our implementation choice for general recursion is its locality. Indeed, the only modification with respect to a type checker without such a feature appears exclusively during *evaluation*. Usually, the *evaluation* function takes a term, and an *environment* linking the De Bruijn index of each variable of the term to a value. This environment is extended during the evaluation of an application or a **split** construct: it is used to link the bound variable to its value.

In our setting, general recursion adds a new kind of binders with a specific difficulty: in a block of mutually recursive definitions, the value of a variable referring to such definitions can change depending of where it is *used*.

Let us illustrate this point with an example. When you first introduce a variable x in a **let** binding block, it has no value. For example, let us have a look at this block:

$x : A$
$y : P\ x$
...

In the declaration of y, x is in normal form. x has been *declared*, but has not been *defined* yet. Later on, in the same block, x is going to be defined and its value will change.

...
$y : P\ x$
$x = t\ y$
$z : P\ x$
...

Now, in the declaration of z, the value of x is $t\ y$, which is a normal form. Hence, the value of x is different from its value two lines earlier in the same program. But it does not stop here: if we now give a definition to y, the value of x will change again.

...
$z : P\ x$
$y = \text{unit}$
$t : P\ x$
...

In the declaration of t, the value of x is now t unit. And this are going to get more and more complex, with the definition of t or even P.

This evolution, in the course of the **let** binding block, of the value of the bound variables makes an implementation using De Bruijn indices quite difficult. Indeed, we cannot represent a definition as a closure containing the context *of the place it was defined* because it would not change in the course of the binding block. So we have to give to the definition the context *of the place it is used*. But the De Bruijn indices in this context does not correspond anymore to those in the definition. A quite complex arithmetic translation on indices would have to be performed.

In our implementation of the *evaluation* function, we extend the usual environment containing the values of the bound variables with a dictionary giving the value of definitions.

When the type checker binds a new *declaration*, a fresh reference of its type is created and subsequently substituted in the term. When the type checker meets a *definition*, the dictionary of definitions is extended with an entry linking the reference corresponding to the define variable and its definition. Each time a variable is evaluated, it is looked up in the current dictionary. If there is a corresponding definition, this definition is used, else the value of the variable is the variable itself.

A dictionary is represented as a list of tuples linking a *reference* to its *definition*. At first, one can think to represent a definition as a *value*. This way, each reference would be linked to the value corresponding to it. Anyway, this representation is not really suitable: as we have seen, the value of a given variable will change depending on the place it is used.

Our choice is to represent a definition as a function from a dictionary to a value.

$$\textbf{data}\ \mathit{Dictionary} = [(\mathit{Reference}, \mathit{Dictionary} \to \mathit{Value})]$$

When looking for the definition of a reference, the evaluation function looks in the dictionary for the definition of the reference and applies for it to the current definition. This process, while remaining fairly simple to understand and to implement, allows to respect the precise semantic of the blocks of mutual definitions.

There is a drawback to this approach: it tends to evaluate the same definitions too often. Even if it is often not very costly as the definitions are mainly functions – which should have been evaluated at each call anyway – this should be improved if we want the prototype to scale to larger programs.

Implementing boxes

Implementing boxes induces a problem similar to the one we have just exposed. The value inside a box depends on the use we made of the box. Indeed, when you test the equality of two boxes you prevent any unfolding of definitions under the box. Whereas when you open a box with the ! construct, unfoldings of definitions have to be enabled.

To solve this problem, we store both values in the *VBox* construct. Hence the type of the *VBox* constructor is:

$$\begin{array}{l}\textbf{data}\ \mathit{Value} = \\ \quad \ldots \\ \quad \mid \mathit{VBox}: \\ \quad\quad \{\mathit{eq\_box}: \mathit{Value}; \\ \quad\quad\quad \mathit{open\_box}: \mathit{Value}\} \\ \quad\quad \ldots\end{array}$$

To evaluate a box, we must creates the two values:

- The one in *eq_box* freezes every definition.
- The one in *open_box* unfolds every definition.

In order to produce this effect, the evaluation in *open_box* is made with the current dictionary of definitions, while the one in *eq_box* is made with an empty dictionary. Hence, the evaluation a definition in the *eq_box* value will return the variable corresponding to the definition.

When testing equality of two boxed values, the equality test uses the $eq-box$ value. To open a box with a !, the evaluation uses the value in *open_box*.

It is interesting to remark that this process constructs a possibly infinite value and relies heavily on laziness for termination. Indeed, the values along a path of *open_box* can correspond to an infinite value. We make use of a *generate* and *prune* paradigm (King and Launchbury 1995) in order to increase modularity. A possibly infinite value is generated. Then, the equality test carefully avoids to follow an infinite path of *open_box* by using *eq_box* to compare two boxed values.

5. Conclusions

With ΠΣ we propose a core language for dependently typed programming which provides a middle ground for the design and implementation of dependently typed programming languages. We have shown that a small core language can encode most constructs which are relevant in dependently typed programming. Our core language can provide an important point of reference both for the front end and the back end. It has the advantage that it is small enough for a metatheoretical analysis, i.e. we would like to establish important properties such as type soundness (i.e. absence of run-time errors) and the correctness of the type checking algorithms formally.

On the front end we need an elaborator which allows the programmer to leave inferable parts of the program implicit. The scope of inferable information in a dependently typed setting is huge it ranges from the usual polymorphic type inference via inferring other indexing expressions to inferring proof objects. On the back end we have to investigate the potential for efficient compilation. One of the most important issues is the ability to omit terms which are only used to ensure the validity of the data, see the discussion on dependent datatypes defined using constraints in section 2.3. To achieve this we have to be able to recognize whether a term is safely total and whether it has to be executed at run-time. Clearly, we have to rely on additional information from the programmer to be able to do a good job here.

We are using boxes to provide a flexible interface to symbolic evaluation. This is an essential ingredient of a type checker for a dependently typed programming language because such a checker has to perform symbolic evaluation. Just by looking at the occurrences of the lifting operator $(-)_\perp$ we can distinguish inductive and coinductive types. This information can be used by a totality checker to automatically recognize whether recursive definitions are structural or guarded. This sort of information is not only relevant if we want to use dependent types for specifications but also to be able to recognize that certain values do not have to be computed at runtime.

With explicit constraints we have proposed an approach which simplifies the compilation of dependently typed pattern matching and covers many uses of equality types, we avoid to clutter up programs with proof terms witnessing equations. We believe that this is a useful feature on the level of the core language because it greatly simplifies the implementation of high level features. However, we haven't covered all the uses of equality types. It is clear that there is no hope to use higher order constraints, such as equalities between λ-abstractions implicitly, because they correspond to universally quantified equations. Hence we are looking for a way to be able to instantiate higher order constraints explicitly with the hope to arrive

at a smooth interpretation of higher order and first order equations. We hope to be able to draw on our experience with the design of *Observational Type Theory* here (Altenkirch et al. 2007)

There are other constructs which have an impact on the core language: Our use of labels suggests a simple subtyping discipline starting from finite types, which can be lifted to other types. There are other potentials for subtyping, e.g. $\sigma \leqslant \sigma_\perp$, which enables us to exploit that eager lists are a subtype of lazy lists. Another exciting idea is a straightforward implementation of reflection, which is simplified by the fact that our core language is quite small. It is easy to define a datatype type reflecting all top-level type constructs, such that $eval : \text{type} \to \mathbf{Type}$ can be implemented. Reflection corresponds to an operation $quote : \mathbf{Type} \to \text{type}$ inverting *eval*.

References

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.

David Aspinall and Martin Hofmann. Dependent types. In *Advanced Topics in Types and Programming Languages*, pages 45 – 86. MIT Press, 2005.

Lennart Augustsson. Cayenne, a language with dependent types. In *Proceedings of the 1998 ICFP*, pages 239–250. ACM, 1998.

Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

Luca Cardelli. A polymorphic lambda-calculus with type:type, 1986. SRC Research Report.

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.

Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

Sa Cui, Kevin Donnelly, and Hongwei Xi. *ATS: A Language That Combines Programming with Theorem Proving*, pages 310–320. Lecture Notes in Computer Science. Springer-Verlag, 2005.

Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculus and Applications*, pages 129–146, 1999.

David J. King and John Launchbury. Structuring depth-first search algorithms in Haskell. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 344–354, New York, NY, 1995.

Andres Löh, Conor McBride, and Wouter Swierstra. Simply easy! (an implementation of a dependently typed lambda calculus). Available on the authors web page, 2008.

Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170, 2004.

Conor McBride and James McKinna. Functional pearl: I am not a number–I am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM, 2004a.

Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004b.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *Proceedings of the 2006 ICFP conference*, 2006.

Robert Nieuwenhuis and Albert Oliveras. *Proof-Producing Congruence Closure*, pages 453–468. Lecture Notes in Computer Science. Springer-Verlag, 2005.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

Nicolas Oury and Wouter Swierstra. The power of Pi. Available on the authors' web page, 2008.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Benjamin C. Pierce and David N. Turner. Local type inference. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 252–265. ACM, 1998.

Adam Poswolsky and Carsten Schürmann. Delphin: A functional programming language with higher-order encodings and dependent types. Available on the Delphin webpage, 2007.

Tim Sheard. Type-level computation using narrowing in Ωmega. In *Proceedings of PLPV '06*, 2006.

The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1*, 2008. URL `http://coq.inria.fr`.