An Implementation of LF with Coercive Subtyping & Universes [*]

Paul Callaghan and Zhaohui Luo
Department of Computer Science, University of Durham,
South Road, Durham DH1 3LE, U.K.
`http://www.dur.ac.uk/CARG`
{`P.C.Callaghan, Zhaohui.Luo`}`@durham.ac.uk`

Abstract.
We present 'Plastic', an implementation of LF with Coercive Subtyping, and focus on its implementation of Universes. LF is a variant of Martin-Löf's logical framework, with explicitly typed λ-abstractions. We outline the system of LF with its extensions of inductive types and coercions. Plastic is the first implementation of this extended system; we discuss motivations and basic architecture, and give examples of its use.

LF is used to specify type theories. The theory UTT includes a hierarchy of universes which is specified in Tarski style. We outline the theory of these universes and explain how they are implemented in Plastic. Of particular interest is the relationship between universes and inductive types, and the relationship between universes and coercive subtyping.

We claim that the combination of Tarski-style universes together with coercive subtyping provides an ideal formulation of universes which is both semantically clear and practical to use.

Keywords: type theory, universes, logical framework, implementation

1. Introduction

Conventionally, a type theory is implemented directly and furnished with sophisticated machinery to make that theory easier to use. This often leads to limited reuse and some duplication of effort in implementation (and in meta-theory) when a new type theory is required. A different approach is adopted in [24], of defining an object type theory UTT via a logical framework LF, resulting in a more precise and semantically clearer version of the theory ECC (implemented in Lego [35]).

The logical framework approach has other benefits. It is a vehicle for recent work in Durham on Coercive Subtyping [25], where definitional

[*] This work is supported by the project on Subtyping, Inheritance, and Reuse, funded by UK EPSRC (GR/K79130, see `http://www.dur.ac.uk/CARG/sir.html`) and by the project on Mathematical Vernacular funded by the Leverhulme Trust (`http://www.dur.ac.uk/CARG/mv.html`).

equality in the framework is an ideal way of expressing the abbreviational nature of coercive subtyping. Object theories specified in LF can take advantage of subtyping automatically, without having to justify the theory in the new setting. One also has the flexibility of being able to specify precisely the type theory needed for a particular task, rather than attempting to stay within a subset of a general theory.

Universes are an important part of UTT. In [24] they are specified using the style à la Tarski, using explicit lifting operators to represent cumulativity in universes. The terminology is due to Martin-Löf [30], and contrasted against the style à la Russell, based on the overloading of common term operators, which was used in ECC. Russell-style is practically more convenient but semantically less clear, and such universes are not compatible with the elimination rules for inductive types (e.g., for Σ), where the subject reduction property fails to hold[1].

Plastic is the first implementation of LF with its various extensions (as presented in [24] and [25]), and consequently the first implementation of coercive subtyping via a logical framework. We plan to use it as a tool in future research in coercive subtyping (e.g. to understand the effects of coercions in non-trivial case studies, and to improve the efficiency of implementation). More practically, it will be used to implement specific object theories; a particular interest is the construction of domain-specific reasoning tools which are based on type theory but will not require great expertise in type theory to use.

After describing the current state of Plastic, we consider how it implements universes in the Tarski style. Of particular interest is the relationship between universes and inductive types, and the relationship between universes and coercive subtyping. Coercions are shown to play a significant part by providing some of the convenience of the Russell style but within the preferable Tarski style.

Section 2 introduces LF, highlighting the important aspects of the basic system and presenting the extensions of inductive types, universes, and coercive subtyping. Section 3 presents Plastic, detailing the main components of the system, with examples of its use. The final section focuses on the implementation of universes in the Tarski style in Plastic; a key issue is whether the practical convenience of Russell-style universes can be regained with the semantically-oriented (but less convenient) Tarski-style universes together with coercive subtyping. The main contributions of this paper are: an implementation of a hierarchy of Tarski-style universes inside LF; its integration with

[1] The problem occurs because the formulation of the elimination rule has not taken into the account that, in the presence of subtyping, a super-type also contains canonical objects of its subtypes. There does not appear to be an easy way to fix this problem for Russell-style universes. For more information, see [25], p. 124.

the implementation of inductive types; and use of coercive subtyping to make working with Tarski-style universes more convenient.

1.1. Universes in Type Theory

Universes increase the proof theoretic strength of the theory, for example allowing constructive formalisations of category theory. There are also many important practical applications, such as representing structured specifications of programs or abstract mathematical structures. A notion of abstract theory [23, 24] for reasoning about such structures is expressible *internally* to a type theory (in ECC and UTT this requires four universe levels).

Early presentations (e.g. [30, 32]) give just one universe. In these references, the presentation uses the Tarski style (terminology due to Martin-Löf), in which types in universes are represented by codes (i.e., names) and a decoding function maps such names to appropriate types. This is contrasted against Russell-style, where codes and the types they represent are identified.

A standard example of universe use is to prove that the constructors of inductive types are distinct. For the boolean type, this is $true \neq_{Bool} false$; see section 4.5.2 for the proof in Plastic. Such properties cannot be proved without universes [37].

The theory ECC [24] contains a hierarchy of universes *Prop*, $Type_0$, $Type_1$, ... presented in the Russell style. The universe *Prop* of propositions is impredicative, allowing quantification over the totality of propositions, in contrast to the predicative universes $Type_i$ which are intended for data types. The Calculus of (Inductive) Constructions (as implemented in recent versions of Coq [8]) contains a similar hierarchy (there are some technical differences but they are not relevant here).

The theory UTT is derived from ECC, and also contains a hierarchy of universes. But UTT is specified by means of a logical framework [24], and its universes are introduced in the Tarski style. Luo claims that use of a logical framework allows a clearer and more precise presentation of the complete type theory (see section 2 for further discussion). For example, Tarski-style universes avoid the failure of subject reduction mentioned above.

In standard use of type theory, e.g. formalisation of mathematics or of programs, the above universe schemes are usually sufficient. Rarely can a use be found for universes above the fourth level in the hierarchy. But it is possible to define more powerful universes, to obtain type theories of greater proof-theoretic strength [33]. An alternative to universes is to use induction recursion [14]. This extended notion of recursion involves large elimination and is therefore powerful enough to define universes internally to the type theory.

1.2. Related Work

This section surveys the implementations of universes in other proof assistants, and then compares these to Plastic in more general terms.

ECC and its Russell-style universes are implemented in Lego [35], together with a form of universe polymorphism [17]; the facility of "typical ambiguity" allows one to omit explicit universe level information and the implementation ensures that an consistent set of levels is used. It allows the flexibility of $Type : Type$, but without the inconsistency. The universes are integrated with Lego's inductive types; the universes are closed under the (inductive) type formers as the type is defined, and no further action is required. Coq [8] also implements Russell-style universes, but with a few differences from Lego (these differences are not relevant to this paper, so we avoid discussion).

One may define Tarski-style universes internally in Agda's type theory [10, 9] because it allows sufficiently powerful recursion. If several universes are required then they must all be defined manually, and there does not appear to be support for automatically closing such universes over arbitrary inductive types. If a cumulative hierarchy is required, then one must also provide appropriate lifting operators and use them explicitly in terms.

Plastic's implementation of UTT provides a hierarchy of universes in the Tarski style, and codes for inductive types in all universes (above the lowest legal universe for that type). Furthermore, Plastic allows one to declare the universe lifting functions as implicit coercions, so one may use a name in $Type_i$ as if it was in some higher universe $Type_{i+j}$.

In more general terms, Plastic is significantly different from Lego, Coq, or Agda. Firstly, it is not intended to be used directly by expert users but as the underlying layer for other systems. LF is a deliberately restricted language; to obtain a flexible type theory, one must specify it as an object theory and then use the language of that object theory. Plastic does provide some of the functionality of other systems to aid specifications of object theories, such as tactic-based proof, meta-variables and a form of argument synthesis. An experienced user of Plastic can translate developments in ECC/Lego to UTT without much difficulty. But Plastic is not a 'finished' system: our current research considers ways to provide a good user interface for object type theories.

2. The logical framework LF

This section describes the type theory LF as introduced in Chapter 9 of the monograph [24] (henceforth abbreviated as C&R), and extended with Coercive Subtyping [25]. LF is a version of Martin-Löf's logical framework [30]. The main difference is that Luo's LF has type labels on λ-abstractions (i.e., $[x : K]k$ rather than $[x]k$). There are also small syntactic differences, several reflecting the influence of Edinburgh LF [16]. The extra type labels ensure that type checking is decidable for this LF, whereas for Martin-Löf's LF it is only decidable for a subset of terms[2].

Theoretically, there are several reasons to be interested in LF. Luo introduces LF as a meta-language for specifying a type theory (UTT in this case), because it allows a clearer and more satisfactory presentation: specifically, one can make clear the distinction between an object language (to be used for programming and reasoning) and the meta-level mechanisms which are used to define it. There are also several practical reasons for interest in LF, which we discuss in section 3. We postpone discussion of universes to section 4.

2.1. Rules of LF

Figure 1 shows the rules of basic LF (some admissible rules for substitution have been omitted). Important features include:

- 'Types' in the framework are called 'kinds' in order to avoid confusion with types in an object (i.e., user-specified) type theory.

- There is a special kind **Type** which represents the conceptual universe of types of the object type theory. Given a type A : **Type**, there is a kind $El(A)$ corresponding to the collection of objects of type A. We refer to such kinds as *El-kinds*.

- In addition to the usual judgement form $\Gamma \vdash k : K$, LF has a form to represent definitional equality, e.g. $\Gamma \vdash k = k' : K$, which says k and k' are equal objects of kind K. Definitional equality is a framework notion, and is a suitable mechanism for representing abbreviations, such as definitions and coercive subtyping.

[2] It is claimed (e.g. [3]) that in practice this is not a hindrance. The system ALF [29] implements a form of Martin-Löf's LF and provides decidable type checking subject to certain restrictions. However, implementations cannot freely expand definitions since the result may not type-check (the implementation does not have the property we call "definitional transparency").

Contexts and assumptions

$$\frac{}{\langle\rangle\ \mathbf{valid}} \qquad \frac{\Gamma \vdash K\ \mathbf{kind} \quad x \notin FV(\Gamma)}{\Gamma, x : K\ \mathbf{valid}} \qquad \frac{\Gamma, x : K, \Gamma'\ \mathbf{valid}}{\Gamma, x : K, \Gamma' \vdash x : K}$$

General equality rules

$$\frac{\Gamma \vdash K\ \mathbf{kind}}{\Gamma \vdash K = K} \quad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \quad \frac{\Gamma \vdash K = K' \quad \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$\frac{\Gamma \vdash k : K}{\Gamma \vdash k = k : K} \quad \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k : K} \quad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' : K}$$

Equality typing rules

$$\frac{\Gamma \vdash k : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k : K'} \quad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k = k' : K'}$$

The kind Type

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash \mathbf{Type}\ \mathbf{kind}} \qquad \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash El(A)\ \mathbf{kind}} \qquad \frac{\Gamma \vdash A = B : \mathbf{Type}}{\Gamma \vdash El(A) = El(B)}$$

Dependent product kinds

$$(DP) \quad \frac{\Gamma \vdash K\ \mathbf{kind} \quad \Gamma, x : K \vdash K'\ \mathbf{kind}}{\Gamma \vdash (x : K)K'\ \mathbf{kind}}$$

$$(DP\ Eq) \quad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x : K_1 \vdash K_1' = K_2'}{\Gamma \vdash (x : K_1)K_1' = (x : K_2)K_2'}$$

$$(abs) \quad \frac{\Gamma, x : K \vdash k : K'}{\Gamma \vdash [x : K]k : (x : K)K'}$$

$$(\xi) \quad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x : K_1 \vdash k_1 = k_2 : K}{\Gamma \vdash [x : K_1]k_1 = [x : K_2]k_2 : (x : K_1)K}$$

$$(app) \quad \frac{\Gamma \vdash f : (x : K)K' \quad \Gamma \vdash k : K}{\Gamma \vdash f(k) : [k/x]K'}$$

$$(app\ Eq) \quad \frac{\Gamma \vdash f = f' : (x : K)K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$$

$$(\beta) \quad \frac{\Gamma, x : K \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma \vdash ([x : K]k')(k) = [k/x]k' : [k/x]K'}$$

$$(\eta) \quad \frac{\Gamma \vdash f : (x : K)K' \quad x \notin FV(f)}{\Gamma \vdash [x : K]f(x) = f : (x : K)K'}$$

Figure 1. The inference rules of LF.

- Functions in LF are called *functional operations*; a subset of these are λ-abstractions, written $[x : K]k$. The kinds of functional operations are called *dependent products*, written $(x : K)K'$. Reduction of functional operations applied to objects in their domain corresponds to β-reduction.

- Dependent products are the Π-types of the framework, and should not be confused with Π-types in an object language like ECC or UTT[3]. A significant difference is the η rule for functional operations (which have dependent product kind). There is no equivalent rule for Π-types, although we can prove propositional equality for Π-types through use of the η rule [25] (p. 108).

- Object theories are specified by declaring new constants and then extending the definitional equality to specify the behaviour of the constants. More precisely, the former corresponds to a new rule:

  $$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash k : K}$$

  and, for a kind K which is either **Type** or of the form $El(A)$ (but not a dependent product), one can assert computation rules by writing $k = k' : K$, which introduces the following rule for definitional equality:

  $$\frac{\Gamma \vdash k : K \quad \Gamma \vdash k' : K}{\Gamma \vdash k = k' : K}$$

For convenience, we allow several abbreviations of notation. We usually omit the lifting operator El: this can be done safely since it will always be clear where it must be inserted. Dependent products $(x : K)K'$ where the bound variable x does not occur free in K' are abbreviated[4] to $K \to K'$. Finally, since we do not deal significantly with object type theories (i.e., our discussion concerns LF and extensions which allow definition of object theories, rather than object theories themselves), we allow the imprecision of calling kinds 'types'.

2.2. Inductive Types and Inductive Families

A class of types that may be safely added to LF are inductive types, as specified by the schema in C&R (p. 177) (based on earlier work

[3] ECC/Lego does not distinguish these, which can sometimes lead to confusion.

[4] Earlier papers used the notation $(A)B$ for this. But $(A)B$ can also represent an application of term A applied to term B, and requires type information to disambiguate. We avoid this unnecessary complication.

[12, 13, 22]). The addition is safe because it does not break useful meta-theoretic properties such as strong normalisation and subject reduction: see [15] for an investigation of the meta-theory. The schema identifies a class of inductive types which recurse through strictly positive operators, and specifies how the elimination operators and computation rules are formed for each type. An alternative explanation of the schema (with details of how to implement it) may be found in [7]. Elimination over these inductive types is small; it cannot be used to define kinds. To define a function value (e.g. of kind $Nat \to Nat \to Bool$) by induction on the first Nat, one must use an elimination family $[n : Nat]\Pi(Nat, [m : Nat]Bool)$.

For example, a type Nat of natural numbers can be defined as $Nat =_{\mathrm{df}} \mathcal{M}[\bar{\Theta}]$, where $\bar{\Theta}$ represents the kinds of the constructors – in this case, kinds X and $X \to X$ where X is a placeholder for the name of the inductive type. The associated introduction operators are $zero =_{\mathrm{df}} \iota_1[\bar{\Theta}] : Nat$ and $succ =_{\mathrm{df}} \iota_2[\bar{\Theta}] : Nat \to Nat$. The elimination operator and computation rules are:

$$\begin{aligned}
\mathbf{E}_{Nat} \;&=_{\mathrm{df}}\; \mathbf{E}[\bar{\Theta}] \\
&:\; (C : Nat \to \mathbf{Type})(c : C(zero)) \\
&\qquad (f : (x : Nat)\ C(x) \to C(succ(x)))(n : Nat)C(n),
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}_{Nat}(C, c, f, zero) &= c \\
\mathbf{E}_{Nat}(C, c, f, succ(x)) &= f(x, \mathbf{E}_{Nat}(C, c, f, x)).
\end{aligned}$$

Inductive families are a generalisation of inductive types, where a *family* of types is inductively defined. An extended schema for these is given at the end of C&R. A standard example is the family of vectors: the type name contains the length of the vector, so an empty vector has type $Vec(zero)$ and so on.

2.3. Coercive Subtyping

LF is extended with a notion of Coercive Subtyping in [25], with additional studies in [20, 38, 28]. Coercive Subtyping is viewed as an abbreviational mechanism of the meta-language (LF), and not part of a particular object type theory. The machinery of subtyping is expressed as a fundamental part of LF and object type theories can make use of it by virtue of their definition in LF. Expressing subtyping as an extension of LF also allows us to study various forms of subtyping in a general framework, e.g. Bailey's notions of argument coercion, Π-coercion, and kind coercion [2] are instantiations of the underlying mechanism[5].

[5] Bailey implemented several forms of coercion in Lego for use in a large-scale development [2]. See [36] for a comparable development in the Coq system.

A coercion is a function $c : K \to K'$, which lifts an object of kind K to kind K'. The meaning of coercion use is expressed via the *coercive definition rule*:

$$\frac{f : (x : K)K' \quad k_0 : K_0 \quad K_0 <_c K}{f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

This says that term $f(k_0)$ abbreviates and is definitionally equal to a term where the coercion is made explicit, namely $f(c(k_0))$, when a coercion c exists to lift object k_0 to the kind expected by the functional operation f. Notice that coercions are only used in a context where the expected kind is known, i.e. where we know both K and K_0. A concrete example: if f expects a vector but k_0 is a list, and there was a coercion which converts lists to vectors, then term $f(k_0)$ would be well typed and definitionally equal to the explicitly coerced form[6]. There are other rules in the extended system $LF_<$, such as for contravariant sub-kinding on dependent products.

Coercive subtyping generalises notions of subset-based subtyping (e.g. even numbers are a subtype of natural numbers by virtue of being a subset), and of record-based inheritance (e.g. groups are a subtype of monoids by virtue of a group being a monoid with extra structure). It also allows novel forms of subtyping, e.g. in Natural Language semantics, where coercions have provided elegant ways of analysing various phenomena that are problematic in other approaches [27, 26].

But the extra power does not come without complication. When using coercions, we require a property of *coherence*: the coercions declared must be mutually compatible. In any sizeable example, several coercions may be valid at a point, and we must be sure that all possibilities lead to the same result (allowing us then to choose any of them). So-called "coherence checking" of a set of coercions is in principle undecidable, and thus an interesting question for implementations. A major problem is with parametrized coercions, which are essential for realistic (i.e., large-scale) use; e.g. first projection on arbitrarily parametrized sigma types. The coherence checking in [2] and [36] is decidable because both use a restricted form of coercion. One possibility for workable coherence checking is a concept we call "dynamic checking", where we only check coherence of the *subset* of actual coercions (i.e., fully instantiated) applicable in a given term. This technique is being investigated in Plastic [5].

Several meta-theoretic results have been proved about coercive subtyping for type theories such as UTT, subject to coherence. A key

[6] This is a *dependent coercion* of kind $(l : List_A)Vec_A(length_A(l))$. The range depends on the argument: the size of the vector is the length of the list.

result is "coercion completion": for every derivation in the type theory extended with subtyping, one can always insert coercions correctly ("coercion completion") to obtain a derivation in the original type theory. A related result is conservativity: every judgement that is derivable in the extended type theory and that does not use coercions, can be derived in the original type theory [38]. Elimination of transitivity in *sub-kinding* has also been established; although elimination of transitivity in sub-typing has not yet been proved.

The correct notion of reduction in $LF_<$ is *typed reduction*. Notice that a term $f(x)$ with an implicit coercion is not a $\beta\eta$-redex because it cannot satisfy the typing premisses of the β rule or the η rule. For reduction to occur, the coercion must be made explicit via the coercive definition rule, resulting in a term which does satisfy the appropriate premisses. Consequently, adding coercions does not add new reductions to the system: it merely extends the class of well-typed terms. This underlines the point that coercions are just abbreviations. On this basis, it is conjectured that strong normalisation properties can be transferred from an object theory specified in LF to the object theory extended with coercions [25]. Observe that the Church-Rosser property does not hold if untyped $\beta\eta$ reduction is used (consider the term $[x : \mathit{Even}]([y : \mathit{Nat}]y)(x)$, where $\mathit{Even} < \mathit{Nat}$). Unfortunately, typed reduction is prohibitively expensive. But coercion completion allows us to convert a term in $LF_<$ to a term in LF by inserting appropriate coercions (see section 3.2), so untyped reduction is safe when all coercions are inserted. Plastic uses untyped reduction.

There are many applications of subtyping, e.g. making type theory more convenient to use by allowing abbreviation. The classic example is the use of coercions between parts of algebraic structure, where one may supply a value representing a group where (e.g.) a set is required: one just 'forgets' the additional structure. Coercions are also useful with inductive data types: in [25], Luo outlines how simple coercions may be lifted functorially over inductive types, e.g. coercing a List of element type A to a List of element type B given a coercion $c : A \to B$. We are also finding interesting applications through regular use in Plastic: the simple coercion from function spaces (i.e., non-dependent Π-types, see section 4.5.1) to the (dependent) Π-types is proving very useful, enabling one to use the most appropriate form in each case. Subtyping also has applications with universes, as we discuss in section 4.4 with concrete examples.

3. Plastic: an Implementation of LF

Plastic is an implementation of LF with inductive types, universes, and coercive subtyping. It is a proof assistant in the style of Lego[7] and Coq [35, 8]. As noted in the introduction, there are several motivations for a new implementation, including: first implementation of LF, as a tool for coercive subtyping research, and as a tool to investigate the possibility of building customised reasoning tools on top of type theory. The various motivations give rise to a collection of design goals: to keep the core of the system simple, to allow easy embedding of the core in customised reasoning tools, and to facilitate adaptation to different object type theories. Given the importance of inductive types to object theories, we also require good performance from the inductive types.

Plastic[8] is implemented in the non-strict functional language Haskell [18], and can be run via several implementations, e.g. the GHC compiler [34]. It is best used with Aspinall's Proof General interface for `xemacs` [1]. More information on Plastic can be found in [5, 6, 7]. Plastic currently uses a simple script-based model of interaction: Lego and Coq use this model, whereas ALF [29] and its relatives (Alfa, Agda [9], etc.) are based on direct term manipulation.

3.1. Overview of Core System

3.1.1. *Notation in the Concrete Syntax*

Plastic concrete syntax is shown in `Courier` font. The mapping between LF syntax and Plastic syntax is given below. Note that Plastic uses curried forms of function application and such applications take precedence over any operators, hence fewer parentheses are required.

LF form	Plastic syntax	Explanation
$(x : K)K'$	`(x:K)K'`	dependent products
$K \to K'$	`K -> K'`	non-dependent products
$[x : K]k$	`[x:K]k`	λ-abstractions
$f(a, b)$	`f a b`	function application

[7] The name 'Plastic' reflects the influence of Lego, whilst recalling that Plastic represents a meta-level version of Lego.

[8] Binaries together with a few libraries are available by ftp. Contact the first author for details.

3.1.2. *Abstract Syntax*

This internal representation contains standard constructs for application, binders, and names, plus it represents bound variables with de Bruijn indices. There are also special constructs representing cases of computation where certain optimisations are possible, e.g. with inductive types [7]. This abstract syntax is equipped with standard algorithms for reduction (via substitution) and other utilities. We do not use any form of explicit substitution; the laziness of Haskell permits a straightforward implementation.

3.1.3. *Type checking and Conversion*

Plastic uses several techniques from "The Constructive Engine" [19]. Since functional operations are typed, and there are no complications such as universes in the core LF, type synthesis is performed by a straightforward compositional algorithm and type checking of a term M is performed by synthesising the type of M and testing convertibility with the claimed type. The convertibility test also follows [19], except for η conversion which does not occur in the Constructive Engine. We use the technique due to Coquand [11].

3.1.4. *Parsing and Pretty-printing*

The concrete syntax is simple, containing infix operators, term juxtaposition, binding forms, and names. The parser is implemented using a `yacc`-like tool for Haskell. Plastic allows several forms of syntactic sugar. The pretty-printer makes use of a library of pretty-printing combinators, and implements several compressions of information, such as flattening of application terms and nested binders. Users may control some aspects of pretty-printing, such as how to display implicit coercions. The resulting layout is of high quality and easy to read.

3.1.5. *Context Mechanisms*

Plastic currently maintains a simple linear context which contains assumptions ('hypotheses'), global definitions, and declarations of inductive types. The main operation on the context is *Cut*, as in the LF rules; it does a type-safe replacement of a hypothesis by a term, expanding (where legal) any global definitions whose name is lost from scope.

3.1.6. *Meta-variables*

Plastic provides a basic form of meta-variable to fill in information which is inferrable with simple unification techniques. (For more powerful systems of meta-variables, see e.g. [31].) The functionality is implemented as a 'preprocessor' for the core system, and the separation allows us to keep the core simple. Inside the core, meta-variables are treated as hypotheses. The only core operation possible on them is cutting with a term of convertible type. The core cannot distinguish meta-variables from hypotheses, and there is no special treatment of them inside the core.

New meta-variables may be added to the context at any time by claiming a name of a given type, e.g.

```
> Claim some_name : A -> A -> B
```

Fresh meta-variables can also appear in a term as either unnamed (symbol `?`) or named (symbol `?foo` – `foo` is a name chosen by the user); appropriate meta-variables are added to the context if they are not solved by constraints within the term. These meta-variables are first order in the sense that one cannot abstract over them. Under a binder, introducing a meta-variable that cannot be solved is an error, for example the term $[x : K]?f(x)$ (which needs higher order unification to solve).

The type inference algorithm in the preprocessor works by propagating type constraints downwards when available (e.g. "this subterm should have kind K"), and uses a form of the conversion test which calculates (first-order) unifiers. This technique (independently developed) appears similar to that of [36], where one of its uses is to calculate suitable arguments for parametrized coercions. Saïbi also studies the theoretical properties of his algorithms.

3.1.7. *Development of Proofs*

Lego and Coq implement a goal-directed proof state which controls which subgoals the user must prove next. This style is ideal for expert users and helps them to work efficiently. Plastic uses a more flexible model, where the 'user' can work on any unsolved meta-variable in the current (linear) context and need not completely solve a meta-variable before attempting another. In earlier work, we identified this flexibility as being useful to applications like Mathematical Vernacular [6], and potentially more convenient if the user was in fact another program.

Meta-variables are the central notion: proof is the process of developing instantiations for them. Proofs start with a `Claim` for the kind representing that which is to be proved. The proof commands may act on that claimed meta-variable, which in turn could introduce further meta-variables (i.e., 'sub-goals'). The main commands are explained below; an example of their use appears in section 4.5. Each of them takes an optional target meta-variable to act on, which defaults to the most recent meta-variable (this gives an approximation to Lego's goal-directed proof mode):

Refine – this is modelled on the Lego command, with similar behaviour: it computes a term to instantiate a meta-variable M of known type. It is implemented in terms of the meta-variable preprocessor. *Cut* is applied to M using the computed term t, and any new meta-variables arising ('sub-goals') are inserted in the context immediately before M. A subset of the new meta-variables may appear in t. Meta-variables which occur in the type of M may also be solved indirectly when the type is unified with the type of the instantiation t.

Intros – when used on a meta-variable `M : (x:K)K'`, the context appearing after and including `M` is replaced with a hypothesis `x : K` and a new meta-variable `M' : K'`. This creates a branch in the context (the original context is hidden, awaiting a solution for M – there is no notion of branch switching as in [31]). In the new context, the user must supply a term of kind `K'`.

Return – marks closure of an `Intros`, namely all meta-variables introduced by (and since) the `Intros` have been solved. The action is to *abstract* the solution for `M'` by `x : K` and cut the result into the context existing prior to the corresponding `Intros` command.

3.1.8. *Inductive Types*

Plastic implements both inductive types and inductive families, each of which may be parametrized. The syntax for these is based on Lego and recursion is provided through elimination operators rather than pattern-matching definitions[9] [31] (see examples in section 4.5). Details of the basic implementation and optimisations can be found in [7].

3.1.9. *Comments on Implementation*

We are using a non-strict language, Haskell [18], whose implementations offer various degrees of lazy evaluation. This allows us to write algorithms in a natural way without being too concerned about the run-time implications. For example, the implementation of substitution is very close to a paper presentation of the algorithm with de Bruijn indices. Laziness means that work is only done when required; thus in a large term, the work of substitution is done incrementally, or not at all if the subterm is never used in a computation. (In effect, this defers responsibility for some aspects of evaluation to the compiler.) Contrast this against a strict language. Typically, one needs to implement mechanisms to prevent substitutions being done throughout a large term,

[9] Pattern matching is much more convenient to use. McBride is investigating how his techniques may be adapted to the setting of LF.

e.g. using explicit substitutions or an abstract machine. Such measures do make the calculus or the implementation more complex.

Non-strict languages do have some disadvantages, primarily that laziness adds overheads to the resource use of a program (time and space). People have believed these to be too prohibitive for serious use in the past. However, recent advances in compiler technology, e.g. as implemented in GHC [34] have significantly reduced this obstacle, so Haskell is becoming a reasonable choice for many applications. As evidence, the computation speed of Plastic compares favourably with that of Coq (the fastest of contemporary type theory proof assistants, and written in `ocaml`, a dialect of ML) on small examples (reduction of large Church numerals (e.g. 10^7) and sorting lists of numbers). But note that we cannot make firm conclusions about performance until Plastic is tested on large developments of the size successfully handled by Coq.

3.2. Coercive Subtyping

Plastic is being used to test ideas on coercive subtyping, in particular based on recent work in Durham [25, 20, 38, 28]. The forms of coercion implemented include parametrized and dependent coercions, non-dependent subkinding (e.g. allowing contravariant subtyping of functions at the framework level), and the lifting of coercions over inductive types [4]. There is also support for the use of coercions with universes, as discussed in section 4.4 below. The implementation is based on coercion insertion during type checking, as justified by the coercion completion results [38], and uses the meta-variable mechanism to calculate appropriate parameters. However, this must not introduce unsolved meta-variables: the missing information must be fully inferrable by unification.

Forms of coercive subtyping have been implemented in Lego by Bailey [2] and in Coq by Saïbi [36], but these have required various restrictions to remain tractable. Both implementations are based around the concept of a finite graph of coercions. Another restriction is use of *syntactic matching*: a potential coercion only matches if the types involved are syntactically equal. This is in contrast to matching based on convertibility, which is much more general, although more expensive to use. Plastic uses convertibility as the matching criteria.

Plastic can handle more coercions than Lego or Coq, in particular the lifting of coercions over inductive types, but our coherence checking (see section 2.3) is fairly primitive. In general, coherence checking is undecidable, hence the restrictions in Lego and Coq. We expect that a mixture of theoretical and practical restrictions may be necessary to retain feasibility of coercion use; for more discussion, see [5].

Notice that, in contrast to previous implementations, we cannot pre-calculate finite directed acyclic graphs of coercions as the closure of all declared coercions. Consider the following rule, which allows the lifting of arbitrary coercions over lists:

$$\frac{\Gamma \vdash A <_c B}{\Gamma \vdash \mathit{List}\ A <_{map_{A,B}(c)} \mathit{List}\ B}$$

This pattern can be repeated infinitely (i.e., A and B can be lists themselves), so we cannot build a finite graph containing all instances of this rule and the resulting compositions with other coercions. Instead, we must synthesise such coercions as they are required. The universe lifting coercions below are a similar case.

4. Universes in LF and Plastic

4.1. Basic Theory

The type theory UTT, which is specified as an object theory in LF, contains a hierarchy of predicative universes Type_i with an impredicative universe *Prop* at its base. This hierarchy is specified in the Tarski style of universes (C&R p. 182). The cumulativity is represented by means of explicit lifting operators, where codes from a universe are injected into the next universe. This formulation is named "Universes as uniform constructions" in [33]. *El* is explicit here, to reinforce the distinction between types and their respective *El*-kinds.

– There is an impredicative universe *Prop* of propositions[10]. **Prf** maps propositions to types; it is the decoding function for *Prop*.

$$\mathit{Prop} \ : \ \mathbf{Type}$$

[10] *Prop* is introduced by declaring the constants above plus the following:

$$\begin{aligned}
\forall \ &: \ (A:\mathbf{Type})(A \to \mathit{Prop}) \to \mathit{Prop} \\
\Lambda \ &: \ (A:\mathbf{Type})(P:A \to \mathit{Prop})((x:A)\mathbf{Prf}(P(x))) \to \mathbf{Prf}(\forall(A,P)) \\
\mathbf{E}_\forall \ &: \ (A:\mathbf{Type})(P:A \to \mathit{Prop})(R:\mathbf{Prf}(\forall(A,P)) \to \mathit{Prop}) \\
&\quad ((g:(x:A)\mathbf{Prf}(P(x)))\mathbf{Prf}(R(\Lambda(A,P,g)))) \to \\
&\quad (z:\mathbf{Prf}(\forall(A,P)))\mathbf{Prf}(R(z))
\end{aligned}$$

and by asserting the following computation rule:

$$\mathbf{E}_\forall(A,P,R,f,\Lambda(A,P,g)) = f(g) \ : \ \mathbf{Prf}(R(\Lambda(A,P,g))).$$

Notice that *Prop* is defined, and can be used, independently of universes; it is the basis of second-order logic in UTT.

$$\mathbf{Prf} \ : \ \mathit{Prop} \to \mathbf{Type}$$

- There are (predicative) universes of types, whose objects are (the names of) types in that universe.

$$\mathit{Type}_i : \mathbf{Type} \qquad (i \in \omega)$$

- For each universe Type_i, there is a 'decode' function which maps names in Type_i to a type in **Type**.

$$\mathbf{T}_i : \mathit{El}(\mathit{Type}_i) \to \mathbf{Type}$$

- *Prop* has a name in Type_0, which decodes to *Prop*. (*Prop* is the lowest universe in the hierarchy.)

$$\mathit{prop} : \mathit{El}(\mathit{Type}_0)$$

$$\mathbf{T}_0(\mathit{prop}) = \mathit{Prop} : \mathbf{Type}$$

- There is a lifting function from *Prop* to Type_0; names in *Prop* decode to the types of their proofs.

$$\mathbf{t}_0 : \mathit{El}(\mathit{Prop}) \to \mathit{El}(\mathit{Type}_0)$$

$$\mathbf{T}_0(\mathbf{t}_0(P)) = \mathbf{Prf}(P) : \mathbf{Type}$$

- Each universe has a name in the next universe in the hierarchy; such names decode to the appropriate universe.

$$\mathit{type}_i : \mathit{El}(\mathit{Type}_{i+1})$$

$$\mathbf{T}_{i+1}(\mathit{type}_i) = \mathit{Type}_i : \mathbf{Type}$$

- Every type with a name in Type_i has a name in Type_{i+1} by virtue of lifting function $\mathbf{t}_{i+1}$; a lifted name decodes to the same result as when unlifted.

$$\mathbf{t}_{i+1} : \mathit{El}(\mathit{Type}_i) \to \mathit{El}(\mathit{Type}_{i+1})$$

$$\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a) : \mathbf{Type}$$

4.2. Implementation

Lego and Coq implement their universe hierarchies directly, as part of their abstract syntax and main algorithms. We prefer not to do this in Plastic: it would orient the underlying core system towards a particular object type theory, and it may hinder experimentation with different formulations of universes. Rather than being restricted to 'pre-set' hierarchies, we aim to allow users to declare the universe structures appropriate to their domain of interest. Thus, the design must implement the Tarski style of universe in a way that has small impact on the core, and is not tied to the details of the UTT hierarchy.

The key step is to realise that the universe names and decoding functions have an inductive behaviour, namely the decoding functions are elimination operators and the names are constructors. This observation allows us to implement the reduction behaviours via the underlying reduction mechanism used for inductive types (described in [7]).

Thus, the 'constructors' are internally represented as terms which produce the required behaviour when the appropriate elimination operator is applied. For example, the constructor $type_i$ will decode to $Type_i$ when decoded with $\mathbf{T}_{i+1}$, as specified in the rules. The decoding of the lifting functions will do a form of recursion, calling the decode function for the lower universe. The type system ensures that these constructors are only used with appropriate eliminators, e.g. applying $\mathbf{T}_j$ to $type_i$ where $j \neq i + 1$ (i.e., an incompatible use) will be caught as a type error. This use of type information applies to all other names related to universes.

The next issue is that the set of names encoding infinite universes is infinite, so the implementation cannot rely on adding each new name to the standard context, as is done for conventional entities. Our solution is to store a 'generator' for names in the context, and concrete occurrences are generated as needed. For example, the name of universe $type_3$, which is written in Plastic[11] as `type^3`, is synthesised by applying the generator for the $type_i$ names to the index 3. The appropriate generator is called each time a non-standard identifier occurs in the concrete syntax. The generator will also calculate the type of the name (and the type *itself* may contain universe names, thus may need to use other generators); this type can then be used as a conventional type in the type-checking algorithms. Finally, the 'base constructors' *prop* and $\mathbf{t}_0$ are implemented as concrete names (i.e., not as generators), with appropriate behaviour under decoding with $\mathbf{T}_0$.

This generator scheme provides an external indexing of the universe constants. Another possibility is an internal indexing, e.g. where $Type_i$ etc. are replaced by a family $Type : (i : Nat)\mathbf{Type}$, but this does allow quantification over universe levels which is not intended in the original theory.

[11] Subscripts are not possible in ASCII, so we must find a simpler representation for names like $type_3$. The current representation is for our convenience; the obvious alternative (e.g. `type(3)`) would require non-trivial changes to the parser.

4.3. Universes and Inductive Types

4.3.1. *Theory of Names in Universes*

C&R (p. 183) outlines the relationship between inductive types and universes. Intuitively, an inductive type $\mathcal{M}$ has a name μ_i in Type_i only if *all* types A (i.e., $A : \mathbf{Type}$) occurring in the kinds of its constructors also have names in Type_i. Such names of $\mathcal{M}$ have behaviour $\mathbf{T}_i(\mu_i) = \mathcal{M} : \mathbf{Type}$. There is a minimal possible universe for $\mathcal{M}$, which is the lowest universe that all types A have names in; $\mathcal{M}$ has a name in this universe and all higher universes. This restriction is required to avoid paradox, and corresponds to the restrictions in the earlier theory ECC, e.g. where Π-types can only be formed for types at the same universe level. Notice that an inductive type can never have a name in *Prop*, reflecting the intended distinction between logical propositions and data types.

The approach used in [30, 32] is more explicit, although it considers only a single universe. It is formulated in the Tarski style, with universe U and decoding function T. A name $\hat{\pi}$ constructs codes for $\Pi(A, B)$ in universe U (using the notation of this paper):

$$\frac{a : U \quad b(x) : U[x : T(a)]}{\hat{\pi}(a, b) : U}$$

The computation rule is $T(\hat{\pi}(a, b)) = \Pi(T(a), [x : T(a)]T(b(x)))$. This pattern may be generalised to any type former, including inductive types, and adapted to hierarchies of universes by providing a set of names $\hat{\pi}_i$ for each universe Type_i (replacing U by Type_i and T by $\mathbf{T}_i$).

4.3.2. *Names in Practice*

How are universes used, and specifically, how should one use universes in LF? There is little experience to draw upon because the implementations of Russell-style universes successfully hide much of the detail of universes. We identify three ‘modes’.

- Firstly, one can work with names of types inside specific universes. Typically this occurs when a higher level of proof strength is needed, e.g. proving distinctness of constructors, which cannot be proved without universes [37].

- Secondly, one could be working with values of a type which has a name in some universe. A key example of this is proof of propositions in UTT, using the universe *Prop* (with decoding function **Prf**). The names P in *Prop* are the propositions being proved, and proof aims to construct an inhabitant of $\mathbf{Prf}(P)$.

– Thirdly, the use of universes with inductive types. Because elimination is small, we cannot define a value of a kind which contains **Type**. Instead, induction must be done over a type which contains $Type_i$ (where i is restricted by the universe constraints on the terms involved). For example, to define $n : Nat \vdash U : \mathbf{Type}$, one must refine U by some $\mathbf{T}_i$ and use induction on n with elimination family $C : [n : Nat]\, Type_i$.

Furthermore, in ECC there is no concept of framework type (i.e., kind) and *every* type is a name in some universe. To use UTT in the style of ECC (e.g. to provide an interface that is familiar to Lego users), one would be forced to use only the names of types in universes.

Luo [24] (p. 183) mentions the issue of uniqueness of names (i.e. requiring $\mathbf{t}_{i+1}(\mu_i) = \mu_{i+1} : Type_{i+1}$), but suggests that it is not essential. Note that names in universe $Type_i$ of the same $\mathcal{M}$ are extensionally equal under the decoding function ($\mathbf{T}_i$) because the names all decode to $\mathcal{M}$ (i.e. $\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(\mu_i)) = \mathbf{T}_{i+1}(\mu_{i+1}) = \mathcal{M}$). That uniqueness is not essential implies that the structure of names is never used, only the fact of their existence and their decoding behaviour. The theory does not even require that the names μ_i are unique in the same universe, so we would have names $\mu_i, \mu_i', \ldots$ as names for $\mathcal{M}$ in $Type_i$. These facts are significant for implementation since it allows a simpler approach to names. We also get a form of equivalence by declaring the universe lifting functions as coercions (section 4.4), allowing one to treat a name in $Type_i$ as if it was in a higher universe $Type_{i+j}$, without needing to lift it explicitly.

4.3.3. *Implementation of Names*

Of the two formulations above, the second is easier to understand. The main difference occurs with the treatment of parametrized inductive types. Parametrization is not part of the official theory, but is a convenience feature implemented in most systems. Without it, use of inductive types would be very cumbersome, since a new inductive type would be needed for each 'instantiation'. Luo's formulation only considers *instances* of parametrized types (i.e. $\Pi(A, B)$ for suitable A, B), and it does not specify how the names μ_i should be generalised with respect to the parameters.

Plastic currently provides two mechanisms. The first mechanism constructs names for (parametrized) inductive types at any universe level from the lowest valid level and upwards. Validity is governed by the conditions for avoiding paradox, as explained above. The technique basically introduces names μ_i (which are functional operations for parametrized types) for the LF-level name $\mathcal{M}$ where:

– the kind of μ_i is obtained by replacing each occurrence of **Type** in the kind of $\mathcal{M}$ with Type_i, and then applying the decode function $\mathbf{T}_i$ to each occurrence of a bound variable whose kind was replaced;

– μ_i itself is a constructor for the relevant decoder $\mathbf{T}_i$; the action of decoding must also decode the arguments that were converted to universe names.

Space prevents a thorough explanation, so we present an example of the result on Π-types, where Π has kind:

$$\Pi : (A : \mathbf{Type})(B : A \to \mathbf{Type})\mathbf{Type}$$

then name π_i of Π in Type_i is:

$$\begin{aligned} \pi_i &: (a : \mathit{Type}_i)(b : \mathbf{T}_i\ a \to \mathit{Type}_i)\,\mathit{Type}_i \\ \mathbf{T}_i(\pi_i\ a\ b) &= \Pi\ (\mathbf{T}_i\ a)\ ([x : \mathbf{T}_i\ a]\mathbf{T}_i(b\ x)) \end{aligned}$$

Specific names $\pi_0, \pi_1, \ldots$ are produced using the dynamic generator technique used for `Type^2` etc., providing names `pi^0`, `pi^1`, ...on demand. The resulting functionality is a generalisation of the way $\hat{\pi}$ is defined in [30, 32].

The second mechanism implemented relies on the observation that unique names are not essential, and thus that specific names for a type in a universe are not important. Names for a type in some universe are generated *dynamically* when a name is needed as the argument to a specific decode function, subject to universe circularity constraints. This generation is done through via unification and meta-variables. Given a term $t : A$ and a specific universe Type_i, it essentially provides a solution to the equation $\mathbf{T}_i\ ?n = A : \mathbf{Type}$. Metavariable n will be replaced by a constant in Type_i whose decoding behaviour is given by this equation. No solution is given if i is lower than the minimum valid universe for A. Whilst sometimes convenient to use, this method is not as general as the previous one because one may not abstract over the constant produced. At present, only the first method is used in the Plastic libraries.

4.4. Coercions and Universes

Coercions have an important application with universes: the lifting functions $\mathbf{t}_i$ can be declared as coercions, hence providing automatic lifting of universe names to higher universes [25]:

$$\frac{\Gamma\ \ \mathbf{valid}}{\Gamma \vdash \mathit{Prop} <_{\mathbf{t}_0} \mathit{Type}_0 : \mathbf{Type}} \qquad \frac{\Gamma\ \ \mathbf{valid}}{\Gamma \vdash \mathit{Type}_i <_{\mathbf{t}_{i+1}} \mathit{Type}_{i+1} : \mathbf{Type}}$$

Plastic does not implement this rule schema directly; instead, the user must declare the lifting functions $\mathbf{t}_0 \ldots \mathbf{t}_i$ as coercions, where i is the highest universe needed for a particular proof. (In principle, implementing the full rule directly is possible, and may be desirable given the exponential increase of possible compositions of the lifting functions.) Such coercions have been useful in our experiments so far.

4.5. Examples in Plastic

The main example is to prove distinctness of constructors for the *Bool* type, i.e. $\mathit{false} \neq_{\mathit{Bool}} \mathit{true}$, which is `(NOT (Eq Bool false true))` in Plastic syntax. We begin with auxiliary definitions.

4.5.1. *Preliminaries*

We introduce the inductive type `Bool`, with constructors `true` and `false`. For reference, we show the type of `E_Bool`, the elimination operator for `Bool`. `Bool_Type^0` is the (automatically generated) name of `Bool` in universe Type_0, which we abbreviate as `bool`. This name can be used in any higher universe if the necessary lifting functions have been declared as coercions.

```
> Inductive [Bool:Type]
>    Constructors [true:Bool] [false:Bool];
> [ bool  = Bool_Type^0  : El Type^0 ];
>
> -- E_Bool : (C:[b:Bool]Type) C true ->
> --                              C false -> (b:Bool) C b
```

`NOT A` is the type of functions mapping objects in type `A` to objects in the empty type (inductively defined, with no constructors). `NOT` is parametrized by `A`; since in LF we cannot λ-abstract over a dependent product (the term $[A:\mathbf{Type}]A \to \mathit{Empty}$ is not well-typed), we must use a function space `Fn` (the infix operator `=>` is a synonym).

```
> Inductive [Empty:Type] Constructors;
> [ empty = Empty_Type^0 : El Type^0 ];

> Inductive [ A,B : Type ] [ Fn : Type ]
>    Constructors [ La_ : (f:(x:El A)El B) Fn];
> [(=>) = Fn ];

> [ NOT = [A:Type] A => Empty : Type -> Type ];
```

Lastly, we require a substitutive equality; we use Martin-Löf's inductive Equality at the **Type** level, and have proved `Eq_subst` for it. Notice that `Eq` is a parametrized inductive family.

```
> Inductive
>       [A:Type][Eq : (x:El A)(y:El A)Type]
>       Constructors
>       [Eqr : (z:El A)Eq z z];
> [ Eq_subst
>    = ...
>    : (A:Type)(m,n:A)(P:A->Type)Eq ? m n -> P m -> P n];
```

4.5.2. *Distinctness of Constructors*
A simple use of universes is to prove distinctness of constructors [37]. The standard technique is to assume $H : \mathit{false} =_{Bool} \mathit{true}$ and construct an inhabitant for *Empty* from this. A type-valued auxiliary function f is defined by induction on a *Bool*, which maps *true* to *Empty* and false to some non-empty type (e.g. *Bool*). Because of small elimination, it must produce names in Type_0 rather than types in **Type**. Proving inhabitation of *Empty* is equivalent to proving inhabitation of $T_0(f(\mathit{true}))$. We use substitutivity and H to produce a new goal of $T_0(f(\mathit{false}))$, which is trivially true. The Plastic proof is the following:

```
1. > Claim false_not_true : NOT (Eq ? false true);
2. > Refine La_;
3. > Intros H;
4. > [f = E_Bool ([y:Bool]Type^0) empty bool];
5. > Equiv T^0 (f true);
6. > Refine Eq_subst ? ? ? ([b:Bool]T^0 (f b)) H ?object;
7. > object Refine false;
8. > ReturnAll;
```

The first line introduces a new meta-variable, with the given type. In order to use `Intros`, we must first 'unpack' the inductive value by refining with its constructor `La_` to expose the dependent product. Line (4) defines the auxiliary function, mapping `true` to `empty` and `false` to `bool`. Induction on a value is achieved by using the elimination operator with appropriate arguments. Line (5) is a convenience for the user, rewriting the type of the most recent meta-variable if the type is convertible with the given term. `Eq_subst` produces a new goal $T_0(f(\mathit{false}))$, using the equality hypothesis `H`; the subterm `?object` is a named fresh meta-variable, giving a specific name to the entry added to the context. Line 7 targets this specific meta-variable by name. Finally, all open `Intros` (just one here) are closed by `ReturnAll`, effectively discharging the hypotheses. The resulting term is shown below.

```
=> false_not_true
=>    = La_ (Eq Bool false true)
=>         Empty
=>         ([H:El (Eq Bool false true)]
=>          [f=E_Bool ([b:El Bool]Type^0) empty bool]
=>          Eq_subst Bool false true
=>                   ([b:El Bool]T^0 (f b)) H false)
=>    : NOT (Eq Bool false true)
```

4.5.3. *Names of Inductive Types, with Coercions*
This example shows names of the function space `Fn` in conjunction with universe lifting by coercion. Lifting functions $\mathbf{t}_0 \ldots \mathbf{t}_6$ have been declared as coercions; by transitivity, several composite coercions are now available, such as $[x : \mathit{Type}_2]\mathbf{t}_4(\mathbf{t}_3(x)) : \mathit{Type}_2 \to \mathit{Type}_4$. We first define synonyms for `Fn` in Type_0 and Type_2, and a name for *Bool* in Type_0. Definition `e1` is the name of a function `Bool => (Bool => Bool)` in Type_2, as demonstrated by its decoding. It uses a coercion from Type_0 to Type_2. Definition `e2` is a functional operation from a name in Type_5, to its decoding in Type_6 (note the coercion). When reduced to normal form, `e2` yields the expected underlying function.

```
> [bool = Bool_Type^0 : Type^0];
> [fn0 = Fn_Type^0
>      : (A:Type^0)(B:T^0 A -> Type^0) -> Type^0];
> [fn2 = Fn_Type^2
>      : (A:Type^2)(B:T^2 A -> Type^2) -> Type^2];

> [e1 = fn2 bool (fn0 bool bool)];
> Normal T^2 e1;            -- gives Bool => (Bool => Bool)

> [e2 = [t : Type^5]T^6 t : Type^5 -> Type];
> Normal e2 e1;             -- gives Bool => (Bool => Bool)
```

The next example shows more complex aspects of coercive subtyping, using Π types, declared below, whose names in Type_i were introduced in section 4.3.3 (the π_i are written `Pi_Type^i` here). Observe that definition `e3` expects a functional operation involving names in Type_2, but `e4` uses names in Type_0 and Type_1. The application is `e3 e4` is well-typed by virtue of subtyping over *dependent product kinds* [28] (p. 8), by virtue of a coercion on functions[12], and decodes to the expected result.

[12] The term is $[f : \mathbf{T}_1(\mathbf{t}_1(\mathit{bool})) \to \mathit{Type}_0][x : \mathbf{T}_2(\mathbf{t}_2(\mathbf{t}_1(\mathit{bool})))]\mathbf{t}_2(\mathbf{t}_1(f(x)))$.

```
> Inductive [A : Type][B : (x:El A) Type][Pi : Type]
>     Constructors [ La : (f:(x:El A)El (B x)) Pi];
> [pi2 = Pi_Type^2
>      : (A:Type^2)(B:T^2 A -> Type^2) -> Type^2];

> [e3 = pi2 bool         : (T^2 bool -> Type^2)-> Type^2];
> [e4 = [a:T^1 bool]prop : T^1 bool -> Type^0];
> Normal T^2 (e3 e4);      -- gives Pi Bool ([x:Bool]Prop)
```

4.5.4. *Setoids in LF*
Setoids are a type together with an equivalence relation on that type. This can be represented as an (inductively defined) dependently-typed record with constructor $mk\_setoid : (A : \mathbf{Type})EqRel(A) \rightarrow Setoid$, but the schema for inductive types prevents the occurrence of **Type** in the kind of any constructor. We must use universes and supply the name of the intended type in some universe. The following defines the structure. We could avoid use of `T^0` by defining `EqRel` on universe names rather than types; there are several possible representations.

```
> Inductive [Setoid:Type]
>    Constructors [mk: (ca:El Type^0)
>                      (eqr:El (EqRel (T^0 ca)))
>                      Setoid];
```

4.6. Discussion

Readers may have realised that `Setoid` above is only usable with types having names in $Type_0$. This is less attractive than ECC/Lego with the universe polymorphism of Harper and Pollack [17] (henceforth H-P). In Lego, one can use the symbol *Type* and the implementation assigns appropriate universe levels internally each time the definition is used. What can we do for setoids in LF? First note that a small number of universe levels is sufficient in practice, and that the full power of H-P is rarely required. Furthermore, users do not really mind which universe levels are used, as long as cycles in universe use, and hence inconsistencies, are avoided.

With the current form of Plastic, this suggests the following technique: one chooses a suitably high universe level, sufficient to accommodate the constraints of one's application, and names it e.g. define `[TYPE = Type^4]`. This need not be chosen precisely, but extremes are not recommended. Setoids can then be defined in terms of this name. Coercions will provide automatic lifting to the appropriate level.

Should the initial choice be too low, the definition can be changed. New coercions will be inserted appropriately in the relevant proofs. For finer control, one could provide names for the i universes immediately below `TYPE`; this convention is used in the Lego library (`TYPE_minus1` etc.).

Clearly this is not equivalent to H-P, e.g. there is no polymorphism in universe levels. But is a system with Tarski universes and coercive subtyping at a disadvantage compared to Lego? We argue this is not the correct comparison. Instead, one should compare it against just Russell-style universes and regard use of H-P as an *orthogonal* issue, since there is no reason why we cannot implement it in Plastic too. Russell-style universes identify names and types, and contains a form of subtyping (of universes, via cumulativity); both are intended to make its use easier, and do achieve this aim although addition of H-P is desirable for practical use. Tarski universes in isolation are held to be less convenient to work with. But by adding coercive subtyping, we regain much of the convenience, even gaining some of the flexibility of H-P (because we do not need to be precise about all universe levels), plus with Tarski universes there are additional benefits of precision, clarity, and better theoretical properties in conjunction with inductive types. Tarski universes with coercive subtyping seems the better option.

There are several possible extensions to this implementation of universes. It is possible to construct universes with greater proof-theoretic strength [33]. It may be desirable to provide *internally indexed* universes: rather than a set of names $Type_0$, $Type_1, \ldots$ we provide a family $Type : Nat \to \mathbf{Type}$ with decoding function $\mathbf{T} : (n : Nat)Type(n) \to \mathbf{Type}$. Such universes could be indexed by other types, e.g. $Nat + \mathbf{1}$ to represent a hierarchy of universes plus a separate universe. This may support the implementation of multiple universe structures.

We will also study ways to make the universes easier to use. The use of H-P was mentioned above; the algorithms may need modifications to work with user-declared universe structures. Internalising H-P may be beneficial to understanding its effects. We may also allow the decoding functions $\mathbf{T}_i : Type_i \to \mathbf{Type}$ to behave as a special case of coercion, particularly the decoder for the universe `TYPE` (such functions are not currently allowed in the framework of coercive subtyping).

Finally, we will undertake case studies of universe use. McBride's work on programming with dependent types [31], which we wish to adapt to the setting of LF, is an important source of examples. For example, McBride's general treatment of the property of distinctness of constructors for arbitrary inductive families (cf the simple proof in section 4.5.2) makes significant use of universes. Some of this work has already been translated to LF, and is suggesting several fruitful areas for future work.

Acknowledgements

We thank members of CARG – the Computer Assisted Reasoning Group at the University of Durham, for helpful discussions. James McKinna and Conor McBride deserve particular thanks for their contributions. We would also like to thank the anonymous referees for their useful comments.

References

1. Aspinall, D., 'Proof General WWW Site'. `http://www.proofgeneral.org`.
2. Bailey, A.: 1998, 'The Machine-checked Literate Formalisation of Algebra in Type Theory'. Ph.D. thesis, University of Manchester.
3. Barthe, G. and M. H. Sorensen, 'Domain-Free Pure Type Systems'. *J. Functional Programming.* (to appear).
4. Callaghan, P.: 1999, 'Plastic: an implementation of typed LF with coercions'. Talk given in the Annual Conf of TYPES'99.
5. Callaghan, P.: 2000, 'Coherence Checking of Coercions in Plastic'. In: *Proc. Workshop on Subtyping & Dependent Types in Programming.*
6. Callaghan, P. and Z. Luo: 1998, 'Mathematical vernacular in type theory based proof assistants'. *User Interfaces for Theorem Provers (UITP'98), Eindhoven.*
7. Callaghan, P. and Z. Luo: 2000, 'Implementation Techniques for Inductive Types in Plastic'. In: *Proc. TYPES'99.*
8. Coq, 'Coq WWW Page'. `http://pauillac.inria.fr/coq`.
9. Coquand, C., 'Adga WWW Page'. `http://www.cs.chalmers.se/~catarina/-agda/`.
10. Coquand, C. and T. Coquand: 1999, 'Structured Type Theory'. In: *Proc. Workshop on Logical Frameworks and Meta-languages (LFM'99).*
11. Coquand, T.: 1991, 'An algorithm for testing conversion in Type Theory'. In: G. Huet and G. Plotkin (eds.): *Logical Frameworks.*
12. Coquand, T. and C. Paulin-Mohring: 1990, 'Inductively Defined Types'. *Lecture Notes in Computer Science* **417**.
13. Dybjer, P.: 1991, 'Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics'. In: G. Huet and G. Plotkin (eds.): *Logical Frameworks.*
14. Dybjer, P.: 2000, 'A general formulation of simultaneous inductive-recursive definitions in type theory'. *Journal of Symbolic Logic* **65**(2).
15. Goguen, H.: 1994, 'A Typed Operational Semantics for Type Theory'. Ph.D. thesis, University of Edinburgh.
16. Harper, R., F. Honsell, and G. Plotkin: 1987, 'A Framework for Defining Logics'. *Proc. 2nd Ann. Symp. on Logic in Computer Science. IEEE.*
17. Harper, R. and R. Pollack: 1991, 'Type Checking with Universes'. *Theoretical Computer Science* **89**(1), 107–136.
18. Haskell, 'Haskell WWW Site'. `http://www.haskell.org`.
19. Huet, G.: 1989, 'The Constructive Engine'. In: R. Narasimhan (ed.): *A Perspective in Theoretical Computer Science.* World Scientific Publishing. Commemorative Volume for Gift Siromoney.

20. Jones, A., Z. Luo, and S. Soloviev: 1998, 'Some proof-theoretic and algorithmic aspects of coercive subtyping'. *Types for proofs and programs (eds, E. Gimenez and C. Paulin-Mohring), Proc. of the Inter. Conf. TYPES'96, LNCS 1512.*
21. Luo, Z.: 1991, 'Program specification and data refinement in type theory'. *Proc. of the Fourth Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT), LNCS 493.* Also as LFCS report ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University.
22. Luo, Z.: 1992, 'A Unifying Theory of Dependent Types: the schematic approach'. *Proc. of Symp. on Logical Foundations of Computer Science (Logic at Tver'92), LNCS 620.* Also as LFCS Report ECS-LFCS-92-202, Dept. of Computer Science, University of Edinburgh.
23. Luo, Z.: 1993, 'Program specification and data refinement in type theory'. *Mathematical Structures in Computer Science* **3**(3).
24. Luo, Z.: 1994, *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press.
25. Luo, Z.: 1999, 'Coercive subtyping'. *Journal of Logic and Computation* **9**(1), 105–130.
26. Luo, Z. and P. Callaghan: 1998a, 'Coercive subtyping and lexical semantics (extended abstract)'. *LACL'98.*
27. Luo, Z. and P. Callaghan: 1998b, 'Mathematical vernacular and conceptual well-formedness in mathematical language'. *Proceedings of the 2nd Inter. Conf. on Logical Aspects of Computational Linguistics, LNCS/LNAI 1582.*
28. Luo, Z. and S. Soloviev: 1999, 'Dependent coercions'. *The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science* **29**.
29. Magnusson, L. and B. Nordström: 1994, 'The ALF proof editor and its proof engine'. In: *Types for Proof and Programs, LNCS.*
30. Martin-Löf, P.: 1984, *Intuitionistic Type Theory.* Bibliopolis.
31. C. McBride. *Dependently typed functional programs and their proofs.* PhD thesis, University of Edinburgh, 2000.
32. Nordström, B., K. Petersson, and J. Smith: 1990, *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press.
33. Palmgren, E.: 1998, 'On Universes in Type Theory'. In: G. Sambin and J. Smith (eds.): *Twenty Five Years of Constructive Type Theory.* Oxford University Press. Commemorative Volume for Gift Siromoney.
34. Peyton Jones, S. L. et al., 'GHC Haskell Compiler WWW Site'. `http://-www.haskell.org/ghc`.
35. Pollack, R. et al., 'Lego WWW Page'. `http://www.dcs.ed.ac.uk/home/lego`.
36. Saïbi, A.: 1999, 'Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories'. Ph.D. thesis, INRIA, Rocquencourt.
37. Smith, J.: 1988, 'The independence of Peano's fourth axiom from Martin-Löf's type theory without universes'. *Journal of Symbolic Logic* **53**(3).
38. Soloviev, S. and Z. Luo: 2000, 'Coercion completion and conservativity in coercive subtyping'. *Annals of Pure and Applied Logic* (to appear).