Implementation Techniques for Inductive Types in Plastic*

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. In the context of Plastic, a proof assistant for a variant of Martin-Löf's Logical Framework LF with explicitly typed λ-abstractions, we outline the technique used for implementing inductive types from their declarations. This form of inductive types gives rise to a problem of non-linear pattern matching; we propose this match can be ignored in well-typed terms, and outline a proof of this. The paper then explains how the inductive types are realised inside the reduction mechanisms of Plastic, and briefly considers optimisations for inductive types.
Key words: type theory, inductive types, LF, implementation.

1 Introduction

This paper considers implementation techniques for a particular approach to inductive types in constructive type theory. The inductive types considered are those given in Chapter 9 of [15], in which Luo presents a variant of Martin-Löf's Logical Framework LF which has explicitly typed λ-abstractions, and a schema for inductive types within this LF which is based on strictly positive operators. The proof assistant 'Plastic' implements this variant of LF with inductive types, together with extensions for Universes and for Coercive Subtyping [4, 2].

The theory and implementation of inductive types is not new. Theoretical investigations can be found in papers such as [7, 8, 14]. Various forms of inductive types, with differing properties, have been implemented in several type theory proof assistants in recent years, e.g. Lego [23], Coq [5], ALF [18], and Agda [6]. Inductive types have also been implemented in Isabelle [21]; whilst this is not a type theory-based system, its approach of implementing object logics through a framework system is very relevant to our aims for LF/Plastic. To our knowledge, the issues concerning actual implementation of inductive types in a type theory proof assistant, as opposed to the meta-theory, have not been considered significantly in the literature to date.

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

The contribution of this paper is to explain in some detail our technique for transforming the syntax of an inductive type declaration to new constants and computation rules, and then their incorporation in to the reduction mechanism of Plastic. The context of LF and Plastic has several differences from the implementations mentioned above (including Lego, which implements a form of Luo's schema directly in ECC); thus, some novel issues are raised.

The inductive types implemented are weaker than those in Coq, for example, as they are restricted to strictly positive recursion, and are manipulated via elimination operators. By contrast, Coq allows more powerful forms of inductive type, and users can manipulate them by 'case' constructs and explicit recursion, subject to certain constraints which ensure termination of recursion. We do not attempt to justify our form of inductive types against others, or to undertake a comparison of the various approaches; this paper will concentrate on *implementing* a single approach.

We also consider a problem of non-linear pattern matching which arises in the computation rules for parametrized inductive types. This is a problem that affects several other implementations of inductive types, and one that has not (to our knowledge) been addressed seriously in the literature. We propose that the non-linearity can be safely ignored since it is irrelevant in a well-typed term, and outline a proof of this.

Inductive types are central to proper use of LF. LF is a framework theory: one should not use LF directly, but should use it to define a particular 'object' type theory and use that. For example, LF does not have a notion of Sigma type (i.e. dependent pair) but Sigma types can be introduced as an inductive type. Widespread use, especially for key parts of object type theories like Pi- and Sigma- types, means efficient operation is an issue in our approach. We discuss ways to improve performance at the end of the paper, many of which have been implemented and are showing good results.

1.1 Implementation Context.

The techniques presented in this paper have been developed and implemented in the system Plastic. Plastic is an implementation of the type theory presented in Chapter 9 of the monograph [15], with extensions for inductive families, coercive subtyping, and universes. There are several motivations for implementing this variant of LF (henceforth LF; Martin-Löf's LF will be named explicitly), including:

- A tool for research on subtyping. Recent work on Coercive Subtyping by Luo, Soloviev, and Jones [16, 13] analyses subtyping as a mechanism for *abbreviation* that is built directly in to the (meta-level) LF. Object type theories specified in LF (such as UTT) then automatically inherit subtyping. Plastic is currently being used in case studies of coercion use, and to experiment with algorithms for implementing coercions [2].
- To use LF seriously as a framework for implementing object theories. One specifies a customised object type theory in LF (e.g. UTT or a subset), and

then uses just the type theory specified. Users of the object theory should not be aware of the underlying framework, and can concentrate on their specific problem domain. (This is a type theory counterpart of Isabelle [12], where object logics are implemented in terms of an underlying framework.)
- A platform for building useful applications. We plan to use type theory technology to build applications for non-experts, that require little or no knowledge of type theory to use successfully. An example we are considering is a tool for verification of concurrent programs.
- A simple type theory implementation for use with applications presenting a Mathematical Vernacular interface [3, 17].

Plastic is implemented in the non-strict functional language Haskell [11], and is best used with Aspinall's Proof General interface for `xemacs` [1]. For more information on Plastic, see the URL `www.dur.ac.uk/CARG/plastic.html` or the paper [4]. (Binaries together with a few libraries are available by `ftp`. Contact the first author for details.)

2 LF and Inductive Types

2.1 Basic Details of LF.

LF is a framework theory, so provides only the basic components of a type theory plus mechanisms which support definitions of more flexible 'object' type theories. In contrast to (e.g.) ECC, it is a simpler theory and places more constraints upon what can be done directly. (See Sect. 3.10 for an example.) The following is a brief overview of aspects of LF relevant to the remainder of the paper.

The LF Kind System. Figure 1 gives the rules for forming kinds in LF. Kinds are framework-level types, as distinguished from the object level types that a user would be concerned with. There is a distinguished kind **Type**, which signifies the universe of object types. A value of this kind is lifted to kind level by the *El* operator, as per the second rule, to form an "*El*-kind". Given a type A (written $A : \mathbf{Type}$), a value t in type A is a member of the *El*-kind of A, written $t : El(A)$. Finally, dependent products are the kinds of functions (the third rule). Note that dependent products are not directly equivalent to Pi types of ECC; LF provides the η-rule for dependent products, and dependent products cannot be defined inductively (see Sect. 3.10 for a discussion).

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash \mathbf{Type\ kind}} \qquad \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash El(A)\ \mathbf{kind}} \qquad \frac{\Gamma \vdash K\ \mathbf{kind} \quad \Gamma, x{:}K \vdash K'\ \mathbf{kind}}{\Gamma \vdash (x{:}K)K'\ \mathbf{kind}}$$

Fig. 1. Kind formation in LF

In the paper, we use some notational conveniences: El 'operators' are often omitted, since type checking can insert them without ambiguity, and write $(A)B$ or $A \to B$ for dependent products where the bound variable does not occur free in B. Since we do not discuss object-level type theories (where the users' types occur), we allow the imprecision of calling kinds 'types'.

Extending LF**.** To specify an object theory with LF, new constants must be declared and the definitional equality extended to show the behaviour of those constants. The constructors and elimination operator of an inductive type are an example of the former, and the latter is exemplified by the computation rules which describe the elimination process. Note that definitional equality is a central concept of LF, and is expressed as a judgement form.

Inductive Types in LF**.** The Coq system (and others) provide a 'case' construct for indicating computations over inductive types. LF instead provides *elimination operators* for this purpose. We briefly explain how these are used. Generically, uses of them have this form: $E\ C\ \overline{f}\ z$. Firstly, elimination over one inductive type produces a value of the same type or of another type. The result type is encoded by C, e.g. $[\_ : List]Nat$ – a transformation from $List$ to Nat, as would happen when taking the length of a list. One name for the C argument is the *elimination family*[1]. Note that elimination families have kind (X)**Type**, signifying that elimination can only produce values in a **Type**, and not in an arbitrary kind. This means that we cannot inductively define a dependent product (see Sect. 3.10). Next, for each constructor there is a 'function' f_i that indicates what to do when that constructor is matched; this has parameters of the constructor arguments and the results of elimination over some of those arguments. In the list length example, nil maps to $zero$ and for $cons$, the function $[\_ : A][xs' {:} Nat]succ\ xs'$ adds one to the length of the list tail. The final z is the object to be eliminated.

2.2 The Schema for Inductive Types.

Chapter 9 of [15] contains a pattern which captures a well-behaved subset of inductive types and shows how to generate an elimination operator and computation rules for them. It is defined in terms of a language LF_Θ, which prescribes definitional equalities between (identical) constants of the same inductive type[2].

The following schema only covers inductive types. Inductive families may be handled by an extension of the schema (see Sect. 9.3.5 of [15] for more information), and parametrized inductive types are obtained by abstracting over the final results. The symbol X is a placeholder for the name of the inductive type

[1] McBride [pers. comm.] suggests the alternative term 'motive' for them.

[2] The details of LF_Θ are not relevant to the implementation, since we use concrete names throughout, and constants of the same name are convertible (and names of constants must be unique in the context).

being introduced. The definitions are w.r.t. a context Γ, but these details have been omitted for reasons of space.

Relevant Definitions. First, we say that a kind is *small* if it is either of the form $El(A)$ or of the form $(x{:}K_1)K_2$ with K_1 and K_2 small kinds. A dependent product is a *Strictly Positive Operator* w.r.t. X if it has the form $(x_1{:}K_1)...(x_m{:}K_m)X$, with $m \geq 0$ and K_i being small kinds in which X does not occur free. An *inductive schema* Θ w.r.t. (the placeholder) X is of one of the following forms:

1. $\Theta \equiv X$, or
2. $\Theta \equiv (x{:}K)\Theta_0$, where K is a small kind and Θ_0 is an inductive schema, or
3. $\Theta \equiv (\Phi)\Theta_0$, where Θ_0 is an inductive schema and Φ is a strictly positive operator w.r.t. X.

An inductive type $\mathcal{M}[\bar{\Theta}]$ is generated by a sequence of inductive schemata $\bar{\Theta} \equiv \Theta_1, ..., \Theta_n$ (w.r.t. the same placeholder X which becomes bound in $\mathcal{M}[\bar{\Theta}]$). Associated with the inductive type are its introduction operators $\iota_i[\bar{\Theta}]$ ($i = 1, ..., n$) and an elimination operator $\mathbf{E}[\bar{\Theta}]$ with appropriate computation rules.

The Elimination Operator. Let $\bar{\Theta} \equiv \langle\Theta_1, ..., \Theta_n\rangle$ ($n \in \omega$) be a sequence of inductive schemata in Γ. Define auxiliaries $\Theta^\circ[A, C, z]$ and $\Phi^\natural[A, C, f, z]$ (latter of kind $\Phi^\circ[A, C, z]$[3]) as follows:

1. Let $\Theta \equiv (x_1{:}M_1)...(x_m{:}M_m)X$ be an inductive schema and $\langle\Phi_{i_1}, ..., \Phi_{i_k}\rangle$ the subsequence of $\langle M_1, ..., M_m\rangle$, which consists of the strictly positive operators. Then, for $A : \mathbf{Type}$, $C{:}(A)\mathbf{Type}$ and $z : \Theta[A]$:

$$\begin{aligned}\Theta^\circ[A, C, z] \;\; =_{df} \;\; & (x_1{:}M_1[A])...(x_m{:}M_m[A]) \\ & (\Phi^\circ_{i_1}[A, C, x_{i_1}])...(\Phi^\circ_{i_k}[A, C, x_{i_k}]) \; C(z(x_1, ..., x_m)).\end{aligned}$$

2. Let $\Phi \equiv (x_1{:}K_1)...(x_m{:}K_m)X$ be a strictly positive operator w.r.t. X. Then, for $A : \mathbf{Type}$, $C : (A)\mathbf{Type}$, $f : (x{:}A)C(x)$ and $z : \Phi[A]$:

$$\Phi^\natural[A, C, f, z] =_{df} [x_1{:}K_1]...[x_m{:}K_m]f(z(x_1, ..., x_m)).$$

Then, $\bar{\Theta}$ generates an elimination operator $\mathbf{E}[\bar{\Theta}]$ of the following type:

$$\begin{aligned}\mathbf{E}[\bar{\Theta}] : \; & (C{:}(\mathcal{M}[\bar{\Theta}])\mathbf{Type}) \\ & (f_1{:}\Theta^\circ_1[\mathcal{M}[\bar{\Theta}], C, \iota_1[\bar{\Theta}]]) \; ... \; (f_n{:}\Theta^\circ_n[\mathcal{M}[\bar{\Theta}], C, \iota_n[\bar{\Theta}]]) \\ & (z{:}\mathcal{M}[\bar{\Theta}])C(z)\end{aligned}$$

[3] Notice that Φ can be regarded as an inductive schema with no strictly positive operators, and hence $\Phi^\circ[A, C, z]$ is defined.

and generates the following n computation rules for $i = 1, ..., n$:

$$
\begin{aligned}
& \mathbf{E}[\bar{\Theta}](C, \bar{f}, \iota_i[\bar{\Theta}](\bar{x})) \\
= \; & f_i(\bar{x},\ \Phi_{i_1}^{\natural}[\mathcal{M}[\bar{\Theta}], C, \mathbf{E}[\bar{\Theta}](C, \bar{f}), x_{i_1}], ..., \Phi_{i_k}^{\natural}[\mathcal{M}[\bar{\Theta}], C, \mathbf{E}[\bar{\Theta}](C, \bar{f}), x_{i_k}]) \\
: \; & C(\iota_i[\bar{\Theta}](\bar{x}))
\end{aligned}
$$

where Θ_i has form $(x_1{:}M_1)...(x_{m_i}{:}M_{m_i})X$, and $\langle \Phi_{i_1}, ..., \Phi_{i_k} \rangle$ is the subsequence of $\langle M_1, ..., M_{m_i} \rangle$ that consists of the strictly positive operators, and $\bar{f}$ stands for $f_1, ..., f_n$ and $\bar{x}$ for $x_1, ..., x_{m_i}$.

An Example. A type Nat of natural numbers can be defined as $Nat =_{\mathrm{df}} \mathcal{M}[\bar{\Theta}]$, where $\bar{\Theta} \equiv X, (X)X$. The introduction operators are $0 =_{\mathrm{df}} \iota_1[\bar{\Theta}] : Nat$ and $succ =_{\mathrm{df}} \iota_2[\bar{\Theta}] : (Nat)Nat$. The elimination operator and computation rules are:

$$
\begin{aligned}
\mathbf{E}_{Nat} : \; & (C{:}(Nat)\mathbf{Type})(c{:}C(0)) \\
& (f{:}(x{:}Nat)(C(x))C(succ(x)))(z{:}Nat)C(z)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{E}_{Nat}(C, c, f, 0) &= c \\
\mathbf{E}_{Nat}(C, c, f, succ(x)) &= f(x, \mathbf{E}_{Nat}(C, c, f, x))
\end{aligned}
$$

3 Basic Implementation of Inductive Types

This section outlines the process of transforming declarations to constants and computation rules. The presentation follows the structure of Sect. 2.2, showing how each stage is realised in Plastic as Haskell code; the correspondence is quite close. (See Sect. 3.3 for a brief explanation of Haskell notation.) The presentation mirrors the simple architecture of the implementation: first, *analysis* collects useful information in addition to checking the constraints, then *synthesis* rearranges this collected information. Finally, we consider extensions of the basic scheme, namely parametrisation, inductive families, and mutually recursive types.

3.1 Running Example.

We take the inductive type of lists (with a fixed element type A) as a concrete example. In the notation of Sect. 2.2, this is introduced by

$$
\begin{aligned}
List_A &=_{\mathrm{df}} \mathcal{M}[X, (A)(X)X] \\
nil_A &=_{\mathrm{df}} \iota_1[X, (A)(X)X] : List_A \\
cons_A &=_{\mathrm{df}} \iota_2[X, (A)(X)X] : A \to List_A \to List_A
\end{aligned}
$$

(The elimination operator and computation rules will be presented later.) The form of declaration expected by Plastic is shown below. This declaration is the starting point of the process. On the notation used in Plastic: LF dependent products are shown as `(x:A)B`, and if the bound variable does not occur free in B, then as `A -> B`, where `->` associates to the right. Comments begin with `--`. Function application binds tighter than infix operators (like `->`), hence we omit unnecessary parentheses. Instead of the placeholder X, we write directly the name of the type being introduced.

```
[A:Type];                                            -- hypothesis
Inductive [List:Type]                                -- type name
  Constructors [nil  : List]                         -- constructors
               [cons : (x:El A) (xs : List) List];
```

Note that the declaration depends on a type `A`. We could write this as a explicitly parametrized type (see Sect. 3.7 for more details). Or we could have chosen a simple inductive family (see Sect. 3.8), such as Vectors – lists which record their length as the family argument, e.g. `vnil : Vec A zero`. But the techniques to be explained do not change significantly when used on these extended forms, so we prefer the simpler version.

3.2 Which Syntax?

Plastic uses two representations for terms: a concrete syntax for all input – including object theory syntax, and an abstract syntax for internal representation, which is specialised towards efficient manipulation of LF terms. Thus we have a choice of input to the inductive types mechanism: concrete or (possibly type-checked) abstract. At present, we operate on the concrete form of the declaration. This is possible because the transformations can be implemented entirely as simple syntax manipulations of a sub-language of LF terms with a few additional occurs-checks. Switching to type-checked abstract syntax terms may have advantages, e.g. allowing *El* to be omitted safely, but it will have disadvantages, such as handling the de Bruijn indices. Note that everything arising from the inductive declaration is thoroughly type-checked at the end, so there is no particular advantage in starting with type-checked declarations.

3.3 Haskell Preliminaries.

This section gives a brief survey of the Haskell notation used in this paper. Fragments of Haskell code are quoted by prefixing each line with symbol `>`.

- Tuples are written `('a',"b")` etc., with their types written `(Char,String)` etc.. The function `fst` extracts the first component of a pair (2-ary tuple).
- Lists are written `['a','b','c']`, with type `[Char]`. Cons is represented by the infix symbol `:`, Nil is written `[]`, and List append is `++`.
- List comprehensions are a convenient shorthand for mapping, filtering, and concatenation of lists. The following definitions are equivalent:

```
> foo1 ns = [ x * y | x <- ns, y <- ns, x+y < 10 ]
> foo2 ns
>  = let f1 x = map (*x) (filter (\y -> x + y < 10) ns)
>     in concat (map f1 ns)
```

– Functions can be used as infix operators by quoting them in backticks, e.g. `id `map` [1,2,3]`. Unless declared otherwise, these operators are left associative with highest precedence (only exceeded by function application).
– A double hyphen `--` marks a comment which extends to the end of the line.
– Algebraic data-types are introduced with `data`, and may be pattern matched. The following introduces a type of binary trees with integers at the nodes:

```
> data Tree = Empty | Tree Int Tree Tree
```

3.4 Auxiliary Definitions.

For the purposes of this paper, the concrete syntax of Plastic is as follows:

```
> type Var = String              -- type synonym
> data Term = Nm Var             -- name
>           | El Term            -- El-term
>           | Ap Term Term       -- application
>           | FO Var Term Term   -- lambda abstraction
>           | DP Var Term Term   -- dependent product
```

The `Var` type is shown as a string here for simplicity; in Plastic, it is a more complex type which distinguishes conventional names from symbolic operators, infix names, and 'underscores' (or nameless bound variables). The following functions assist in building trees of applications, and in building bound terms.

```
> ap_ts :: Term -> [Term] -> Term
> ap_ts = foldl Ap
>                -- left nested applications, terms at leaves
> ap_nms :: Name -> [Name] -> Term
> ap_nms f as = Nm f `ap_ts` map Nm as
>                -- left nested applications, names at leaves

> un_bs :: [(Var,Term)] -> Term -> Term
> un_bs ts t = foldr (\(v,ty) -> DP v ty) t ts
>                -- dependent products, from list of bindings

> un_bs_FO :: [(Var,Term)] -> Term -> Term
> un_bs_FO ts t = foldr (\(v,ty) -> FO v ty) t ts
>                -- lambda abstractions, from list of bindings
```

3.5 Analysis of Declarations.

An example of the concrete syntax for declarations was given in Sect. 3.1. The first check is on the inductive type name: for simple types, the declaration must have the form [*name* : **Type**], a reminder that we are inductively defining an element of the kind **Type**. Secondly, a check is made that none of the constructor names occur in the kinds of the constructors; this cannot occur in the $\bar{\Theta}$ notation, but with explicit names, the user is able to write it, hence the need to check. Then, each constructor is analysed in terms of the following three concepts, starting with the inductive schema level and working downwards.

Small Kinds. A small kind is a kind that does not contain the kind **Type**, so it is either an *El*-kind or a dependent product whose domain and range are small kinds. The purpose of requiring small kinds is to avoid paradox.

Implementation. Checking the smallness of a kind is implemented by recursion on the structure of the term, and returns a Boolean value. Notice that we can't define kind-valued functions in LF, so there is no need to check that an expression reduces to a dependent product etc. This allows us to use a straightforward syntactic check. There is no attempt here to check if the El-kinds are well-formed – e.g. to prohibit $(El(El\ X))$ – this will be done by later type checking.

The schema calls for small kinds which do not contain free occurrences of the placeholder X; this is not implemented here, but is checked as part of the tests for strictly positive operators and inductive schemata as an explicit occurs-check.

Strictly Positive Operators. A strictly positive operator (SPO) is a kind of form $(x_1{:}K_1)...(x_m{:}K_m)X$, with $m \geq 0$ and K_i being small kinds in which X, the name of the inductive type, does not occur free.

Implementation. The test for a SPO is a simple structural recursion plus an occurs-check on the small kinds; as above, we do not need to consider reduction, since LF does not contain kind-valued functions. The occurs check is also a simple structural recursion; since we are working in concrete syntax, this is just a tree traversal which looks for a specific name in a non-binder position and ignores subtrees where the name is bound.

We retain information about the dependent product bindings for later use, and keep it in the following data structure; it is the series of bindings represented as a list of pairs (x_i, K_i) of bound variable name and its kind:

```
> data SPO = SPO [(Var, Term)]
```

Inductive Schemata. This is the top level of a constructor's kind, indicating that the constructor may take non-recursive arguments (as small kinds) and recursive arguments (strictly positive operators), and return a value in the type X. An *inductive schema* Θ w.r.t. a placeholder X has one of the following forms:

1. $\Theta \equiv X$
2. $\Theta \equiv (x{:}K)\Theta_0$, where K is a small kind that does not contain X, and Θ_0 is an inductive schema w.r.t. X
3. $\Theta \equiv (\Phi)\Theta_0$, where Θ_0 is an inductive schema and Φ is a strictly positive operator, both w.r.t. X.

Implementation. Checking for an inductive schema is also implemented as structural recursion with occurs-checks. The main difference is the information collected; we represent all bindings as above, plus the information from the subset which are strictly positive operators:

```
> data ISch = ISch [(Var, Term)] [(Var,SPO)]
```

This is a two-field algebraic data type. The second field is derivable from the first (the first represents *all* bindings), so we could regenerate this information later, but storing it in this form is convenient.

Case (3) above indicates non-dependence (i.e., no bound variable), but the data structure explicitly requires one. This is deliberate; we require users to give explicit and unique names to all bindings in their declarations of inductive types. The generated constants will then use names selected by the user and meaningful to them, rather than automatically generated names (`x1`, `x2`, etc.).

The explicit naming means extra checks are required to maintain adherence to the schema. Specifically, the problem is that a bound name with strictly positive operator kind can occur in Θ_0 (this problem is similar to the issue of constructor names occuring in the kinds of the other constructors). That is, expressions of the following form might occur, where Φ is a valid SPO:

$$\Theta = (x : \Phi)(y : K)X$$

The problem is that the explicitly given name x may occur in K. However, it is impossible for the bound name of SPO kind to occur in a well-typed range (i.e., the Θ_0 kind). K itself cannot be x because $\Gamma, x : \Phi \vdash x$ **kind** is not derivable. The only place x can occur is as an argument to some function. But in Γ, there are no (monomorphic) functions which can take a value in the inductive type being defined, because the type name has not been declared yet. Neither could a polymorphic function be used (such as $id{:}(A{:}Type)(a{:}A)A$), since the name of the inductive type would occur in its arguments, and this would be caught by the occurs-checks. Hence, no explicit check has been implemented in Plastic.

Remark. We can view the concepts above as a grammar for valid constructor kinds. The implementation can be regarded as a parser for this language, with the semantic action of representing the structure in a more convenient form.

3.6 Creation of Elimination Operator and Rules.

The first step is to declare the constants (type and constructor names) from the inductive type declaration. Initially these are treated as simple hypotheses.

The type of the elimination operator is then constructed and declared as a hypothesis. Finally, we generate a computation rule for each constructor, and check the validity of its type in the context of the previous declarations. The detailed treatment of computation rules is postponed until Sect. 5. Observe that type checking supplements the prior analysis, implicitly doing some of the tests that are difficult on concrete syntax.

For reference, below is the analysis result for the List example, which is the input to this stage, followed by the output – the type of the elimination operator and its computation rules:

```
> ischs = [ ISch    -- for nil
>               []    -- no bindings
>               []    -- so no SPOs
>         , ISch    -- for cons
>               [("x", El "A"), ("xs", "List")] -- two bindings
>               [("xs", SPO [])] ]            -- 'xs' is a SPO

E_List : (C_List:El List -> Type)      -- elimination family
         El (C_List nil) ->            -- case function f_nil
         (                             -- case function f_cons
           (x:El A)                        -- constr arg
           (xs:El List)                    -- constr arg
           El (C_List xs) ->               -- recursive arg
           El (C_List (cons x xs))) ->     -- result
         (z:El List)                   -- value to be eliminated
         El (C_List z)                 -- overall result

E_List C f_nil f_cons nil              -- comp. rule for nil
  = f_nil

E_List C f_nil f_cons (cons x xs)      -- comp. rule for cons
  = f_cons x xs (E_List C f_nil f_cons xs)
```

Creation of the Elimination Operator. The type of the elimination operator is created by simple transformation of the collected `ISch` information of the constructors. These structures hold the relevant information in a convenient form, so the construction is straightforward.

Strictly Positive Operators. These can be treated as inductive schemata without strictly positive operators and whose binding is non-dependent (hence the nameless variable `underscore`[4]), hence we just use the function for inductive schemata, which we will explain next. The argument `c_name` is the bound name for the elimination family.

[4] We could also use some name derived from the bound name, e.g. prefixed with 'r' as a reminder that this variable results from a recursive call.

```
> spo_type :: Var -> (Var, SPO) -> (Var, Term)
> spo_type c_name (nm, SPO bs)
>  = (underscore, isch_type c_name (nm, ISch bs []))
```

Inductive Schemata. Function `isch_type` produces the type of the function which is applied to a constructor's arguments and the results of recursion via the strictly positive operators. There appears to be no good name for such functions, so we use the term "case function". Informally, the type of the case function is a dependent product, one binding per constructor argument then one binding per recursive call, ending with $C(c\ x_1 ... x_n)$, where C is the elimination family, c the constructor, and x_i the constructor arguments. The bindings are constructed from the `ISch` value; the normal bindings are unchanged, and bindings for the SPOs are created with `spo_type`. The result type comes from applying the elimination family to the constructor applied to its arguments. We build an LF term by changing the list of bindings to a nested dependent product with function `un_bs` (see Sect. 3.4). The overall result is a binding, where the bound variable is the constructor name prefixed by `f_`.

```
> isch_type :: Var -> (Var, ISch) -> (Var, Term)
> isch_type c_name (nm, ISch bs spos)
>  = (f_name nm,
>     un_bs (bs ++ map (spo_type c_name) spos) c_result)
>    where
>     c_result = Ap c_name (nm `ap_nms` map fst bs) -- C(c(x1-xn))
>     f_name n = "f_" ++ n
```

Inductive Schemata. The main function `mk_elim` takes the name of the inductive type and the list of `ISch` values, each paired with the constructor name. Again, we assemble the bindings (first the elimination family, then a case function for each constructor, and then the target z[5] of elimination), and use `un_bs` to produce a dependent product term. The variable name binding the elimination family is the type name prefixed with `C_`. Notice that the elimination family has kind $(\_ : X)\mathbf{Type}$. The significance of this is explored in Sect. 3.10.

```
> mk_elim :: String -> [(Var, ISch)] -> Term
> mk_elim type_name is
>  = un_bs (    [(c_nm, c_type)]
>           ++ [isch_type c_nm i | i <- is ]
>           ++ [("z", type_name)]
>          ) (Ap c_nm "z")
>    where
>      c_nm   = "C_" ++ type_name
>      c_type = DP underscore type_name Type
```

[5] The name z is actually hard-coded in Plastic, and there is a chance that a parameter of the inductive type can clash with this fixed name. We make no attempt to handle clashes of names, requiring the user to choose distinct names, and also to avoid z.

Creation of the Computation Rules. The process is similar to that for the elimination operator type, as it synthesises terms from the set of ISch values. We first consider recursive calls. All of these have the form of a fixed 'prefix' applied to varying arguments, where the prefix is $\mathbf{E}[\bar{\Theta}](C, \bar{f})$, the elimination operator applied to the elimination family and the n case functions. The form of this prefix does not change in recursive calls, so we construct the term once and pass it to the other rule-constructing functions. Function mk_e_prefix constructs the prefix, which is passed as the e_prefix argument of the later functions.

```
> mk_e_prefix :: Var -> Var -> [(Var,ISch)] -> Term
> mk_e_prefix e_name c_name is
>  = e_name `ap_nms` (c_name : map fst is)  -- E(C, f1, ..., fn)
```

Strictly Positive Operators. Recall that SPOs in an inductive schema embody recursion, so we must construct a call via the e_prefix. SPOs also take a number of small kind arguments too. In phi_nat, the SPO arguments are converted to LF's lambda-bindings (un_bs_FO does this, given a list of bindings and a body), and then the e_prefix is applied to the name of the SPO after it is applied to the bound names. Alternatively, if Φ has form $(y_1{:}K_1)...(y_m{:}K_m)X$, then the result is $([y_1{:}K_1]...[y_m{:}K_m](E\ C\ \overline{f})(s\ y_1...y_m))$.

```
> phi_nat :: Term -> (Var, SPO) -> Term
> phi_nat e_prefix (nm, SPO bs)
>  = un_bs_FO bs (Ap e_prefix (nm `ap_nms` map fst bs))
```

Inductive Schemata. The process for ISch is simpler: we just apply the relevant function (named by prefixing the constructor name with f_) to the normal constructor arguments and then to the results of the recursive calls arising from the SPOs. Example output is shown at the start of this section, for the List example.

```
> construct_rule :: Term -> (Var, ISch) -> Term
> construct_rule e_prefix (nm, ISch bs spos)
>  = f_name nm `ap_nms` constr_args `ap_ts` recursive_args
>    where
>      constr_args    = map fst bs
>      recursive_args = map (phi_nat e_prefix) spos
>      f_name n = "f_" ++ n
```

The actual implementation is slightly more complex; it also stores information about the free names in the result of construct_rule, with their types. This is explained further in Sect. 5.

3.7 Parametrized Types.

Declarations record parameters by listing them before the inductive type name, as shown below. One technique for implementing parametrisation is to first process the declaration in the context of assumed parameters, then to *discharge* those parameters. Currently, Plastic does not have a full discharge tactic.

```
Inductive [A:Type] [List:Type]
  Constructors [nil  : List]
               [cons : (x : El A) (xs : List) List];
```

We use a direct method, where parametrisation is treated as an *abbreviation* for the explicit form. It is much easier to insert the extra parameters (or bindings for them, in type expressions) directly in the appropriate places in the terms constructed, and this has some advantages in implementing some of the later optimisations. The technique is straightforward; for example, bindings for the parameters are inserted as the first parameters for elimination operators, and occurrences of the inductive type name are suffixed with the new bound names.

3.8 Inductive Families.

These are a generalisation of inductive types, where the entity being defined is a *family* of types, rather than just the kind **Type**. The family arguments vary according to a constructor and its arguments. A nice example is of Vectors, where the length of the vector is carried as the family argument, as shown below:

```
Inductive
    [A:Type]
    [Vec:(n:El Nat)Type]
  Constructors
    [vnil:Vec(zero)]
    [vcons:(m:El Nat)(x:El A)(xs:Vec(m))Vec(succ(m))];
```

The extended schema is outlined at the end of [15], and inductive families have been implemented in Plastic. To extend the implementation here is straightforward: it just involves storing more information and inserting it appropriately in the constructed terms. Recall that strictly positive operators and inductive schemata were defined in terms of X. Now, we have $X(p_1...p_k)$, and must store the particular family arguments $p_1...p_k$ for each occurrence of X – achieved by extending the data types above and changing the simple checks on X to ones that return the family information. Later, the Synthesis phase can just read off the required information.

3.9 Mutually Recursive Types and Inductive Propositions.

Simple forms of mutually recursive types, e.g. a set $\mathcal{T}$ of mutually defined types whose constructors can recurse over values in any type in $\mathcal{T}$, are possible by extension of the above framework, and a form of this has been implemented in Lego. They have not been implemented in Plastic, but we do not envisage difficulties, and expect the work required to be possible by minimal changes to the above techniques, such as carrying some extra information. We have not considered more complex forms of mutual recursion. Inductive propositions, as implemented in Lego, are expected to be straightforwardly implementable; the main difference is that instead of defining a type in **Type**, one is defining an entity in the universe *Prop* of propositions (where *Prop* : **Type**).

3.10 Inductive Π Types.

As a final example, and to illustrate an important feature of LF's inductive types, we consider the problem of defining equality eq_{Nat} on natural numbers, with kind $(x : Nat)(y : Nat)Bool$. With elimination operators, the usual technique is to do induction over the x to define a 1-ary predicate over Nat. That requires an elimination family $[x{:}Nat]Nat \rightarrow Bool$. But this is not allowed, since $Nat \rightarrow Bool$ is a kind and not a **Type**. We must introduce an inductive type which represents the required function, hence the inductive type `Pi`:

```
Inductive [A:Type][B:(x:A)Type] [Pi:Type]
   Constructors [La : (f : (y:A) B y) Pi];
[ap = -- (defined by elimination)
    : (A:Type)(B:A -> Type) Pi A B -> (x:A) B x ];
```

We can now refine the subgoal `?i : Nat -> Bool` with `ap`, to give a new subgoal `?j : Pi Nat ([y:Nat]Bool)` which we can solve with elimination, using the family $([x{:}Nat]\Pi(Nat, [y{:}Nat]Bool))$.

Plastic does implement forms of sugaring for Π and related type constructors (e.g. $\forall$ in second order logic), plus we could also use a simpler, non-dependent form of the Π-type. But the added complication is an intended consequence of LF, and reinforces that LF is not meant to be used directly. How to implement a user-friendly object level theory is a significant part of our current research.

4 A Property of Computation Rules

4.1 The Problem.

Readers may have realised that parametrisation leads to duplication of parameters in the computation rules, e.g. the rule for `nil`, where the 'A' appears twice.

```
E_List A C f_nil f_cons (nil A) = f_nil
       ^---------------------^
```

At first sight, it appears that elimination of inductive types requires non-linear pattern matching. One consequence is to complicate the reduction process, since we have to perform a (framework-level) conversion test before allowing the reduction. So, when testing the term `(E_List A t1 t2 t3 (nil B))` for reducibility, we would have to test convertibility of A and B. Pollack [pers. comm.] noticed this problem in his work on Lego: he relies on implicit syntax in this case, shifting the problem of pattern matching to solvability of meta-variables.

But, what happens if the conversion fails? Since each constructor has just one possible elimination, then there is no valid alternative apart from failure (i.e., $\perp$). This doesn't fit in a language with strong normalisation. We observe: given the way the type of the elimination operators are constructed, it is impossible to construct an expression where conversion would fail, because that expression would be ill-typed.

So, we claim the following: type checking guarantees that the conversion will succeed, because type checking must have done the required conversions anyway. This allows the implementation to ignore the second occurrence of the repeated arguments and treat the pattern as linear.

4.2 Outline of Proof.

We consider the case of parametrized inductive types (i.e., not an inductive family), in LF extended with the scheme for inductive types. It makes use of three results from [10], Uniqueness of Kinds and the equivalence between conversion and judgemental equality, plus the Church-Rosser property for conversion.

$$\text{(Uniqueness of Kinds)} \quad \frac{\Gamma \vdash M : K \quad \Gamma \vdash M : K'}{\Gamma \vdash K = K'}$$

$$\text{(Conv-Eq)} \qquad \Gamma \vdash M = M' : K \Longleftrightarrow M \simeq M'$$

Let X be a parametrized inductive type; Let X be a parametrized inductive type; a constructor of X has kind of general form (1), and the elimination operator has kind of general form (2), where $\overline{R}$ are the types of the parameters to the inductive type, $\overline{B}$ are the types of the arguments to constructor c, and $\overline{F}$ are the types of the case functions of the elimination operator.

$$c : (\overline{r} \ : \overline{R})(\overline{b} \ : \overline{B})X \ \overline{r} \tag{1}$$

$$E_X : (\overline{r} \ : \overline{R})(C \ : (z \ : X \ \overline{r}) \ \mathbf{Type})(\overline{f} \ : \overline{F})(z{:}X \ \overline{r}) \ C \ z \tag{2}$$

Eliminations have form $(E_X \ \overline{p} \ C \ \overline{f})$ applied to a term $(c \ \overline{q} \ \overline{a})$, where c is a constructor of X. The parameters are represented by $\overline{p}$ and $\overline{q}$, both containing m terms. We want to show that if the elimination is well-typed then $\overline{p}$ is convertible with $\overline{q}$, which allows us to avoid convertibility checks in pattern matching.

$$(E_X \ \overline{p} \ C \ \overline{f}) : (z : X \ \overline{p}) \ C \ z \tag{3}$$

$$(c \ \overline{q} \ \overline{a}) : X \ \overline{q} \tag{4}$$

Since the elimination is well typed, the term $(c \ \overline{q} \ \overline{a})$ of type $(X \ \overline{q})$ also admits the type $(X \ \overline{p})$, since this is the type required for the target of elimination:

$$c \ \overline{q} \ \overline{a} \ : X \ \overline{p}$$

$$c \ \overline{q} \ \overline{a} \ : X \ \overline{q}$$

By the Uniqueness of Kinds theorem stated above, this implies that $(X \ \overline{p})$ and $(X \ \overline{q})$ are judgementally equal. By Conv-Eq, we have the following:

$$X \ \overline{p} \simeq X \ \overline{q}$$

X is a constant, hence both terms above are irreducible. Using the Church-Rosser property for $\simeq$, the above implies the required result, that $\overline{p} \simeq \overline{q}$.

The case for inductive families is more complex, since unlike simple parameters which appear unmodified in the types, the family arguments may be modified. For example, in Vectors and the case for *vcons*, the pattern matching on n is dependent on the family argument matching (*succ n*). We believe the above proof can be extended to the case of parametrized inductive families.

```
E_V A C f_vnil f_vcons (succ n) (vcons A n a l)
              ==> f_vcons n a l (E_V A C f_vnil f_vcons n l)
```

5 Technical Details and Optimisations

5.1 Internal Representation and Mechanism.

All of the computation rule equations have the form of an elimination operator applied to several arguments, ending with a constructor that itself may have several arguments. These are rewritten as LF λ-abstraction terms which take the non-constructor arguments of the eliminator plus the arguments of the constructor (with the repeated arguments ignored, as per the proposition of Sect. 4), and return the right-hand side of the rule. These are called the "reduction" functions.

For example, consider the list constructor `cons`, parametrized by type A. The reduction function `f` is shown in Haskell notation (the LF version with explicit type labels is much longer). Notice that the second `A` has become an underscore (nameless variable).

```
E_List A C f_nil f_cons (cons A x xs)       -- computation
 = f_cons x xs (E_List A C f_nil f_cons xs) -- rule

f = \ A C f_nil f_cons _ x xs ->
             f_cons x xs (E_List A C f_nil f_cons xs)
```

Iota reduction is only attempted if beta-reduction and delta-reduction (definition unfolding) are not applicable. An expression $(M\ N_1...N_n)$ is an iota redex if M is an eliminator of arity n, and N_n is a constructor application $(c\ \bar{x})$ (under reduction to iota-delta-beta whnf). The reduction function of the constructor is then applied to $N_1...N_{n-1}$ plus the arguments $\bar{x}$ of the constructor and reduced to beta whnf; this whnf is the result of the reduction. The purpose of the beta reductions is to substitute the parameters throughout the reduction function body, leaving us with the right-hand side of the computation rule. It does not do any of the computation specified by the case functions.

To avoid repeated context lookup, the internal representation of an elimination operator contains its arity, and constructors contain their reduction functions. Notice that the constructors may have occurrences of their eliminator in their reduction functions, and the type of the eliminator will contain the constructors. Thus, care must be taken when adding the new constants in this form to the context so that the internal representations of all occurrences of the symbols are correctly created.

5.2 Optimisations.

Several aspects of iota reduction allow nice optimisations. In addition to well-known techniques like performing as many simultaneous beta reductions as possible (which avoids repeated adjustment of de Bruijn indices), we are getting excellent results with the techniques listed below.

These optimisations were developed by analysing Plastic's performance with the GHC compiler's profiling tools [22], and studying the bottlenecks. This indicated what the expensive or repeatedly performed steps in computation were, and thus showed the *actual* inefficiencies in the algorithms as opposed to those we *believed* were problematic. By far, the major cost is manipulation of de Bruijn indices; several of the optimisations implemented do reduce this significantly, but there is still room for improvement.

- For the elimination expressions on the right hand sides of computation rules (i.e., which perform recursive calls), arity checking is not required. Occurrences of elimination operators in the rules are always fully applied.
- "Partial reflection": instead of using LF-level reduction, part of the work of the reduction functions is done at the Haskell level. Some of the work of beta reduction is done once and kept for certain terms, and the results used whenever the term is. Note that despite this and other optimisations, Plastic still behaves like other proof assistants, i.e. will not do reductions unless they are forced, and only do a single step at a time.
- Closed terms admit a simpler process of substitution. Such terms, when fully applied, make use of an optimised substitution function which avoids certain manipulations of de Bruijn indices.
- The internal reduction functions are a special case of closed terms, allowing a still-simpler substitution to be done.
- Binder prefix memoisation: terms with a binder prefix are associated with a count of the binders, allowing simultaneous reduction and a simplified substitution to be done if the term is fully applied. (This happens regularly in eliminations, consequently this step significantly improves speed.)
- Lastly, some key Haskell functions were carefully rewritten to make best use of non-strictness. This means that such functions now do the least computation possible before choosing the next step of computation. On the other hand, parts of the abstract syntax have been made explicitly strict, which reduces some overheads in their use – for example, there is no advantage in storing de Bruijn indices in a non-strict way.

We have some more optimisations in mind:

- Full reflection for common classes of inductive type. That is, the reduction functions of some types will be implemented by their equivalent implementation language version. Typically, this will be used for common types such

as data structures, enumerations, Pi, Sigma, Nat etc. Such types, especially Pi and Sigma, will be heavily used in LF implementations of object theories.
- Compilation of all terms to more efficient forms. Thus terms may end up with a standard "display form" and an internal form for efficient computation.
- Representing recursive elimination calls in a form which avoids more work in iota reduction. Essentially, most of the arguments in an elimination call do not change, and it is possible to improve the substitution of these.

6 Discussion

We have implemented a subset of inductive families based on recursion through strictly positive operators, which are manipulated by elimination operators, and achieved respectable performance from the implementation with some simple optimisations. Whilst the inductive families implemented lack the generality of Coq's inductive families [9], or the flexibility of McBride's tactics [19], they are adequate for our current uses of Plastic.

We also claim that the technique used to analyse declarations and to construct the elimination operator etc. is clear and easily understood, and will provide a useful alternative view to the mathematical presentation. A property of the schema for inductive types concerning linearity of pattern matching has been identified and documented, with an outline proof of the property.

The work raises some interesting questions for future consideration. Of particular interest is to study the behaviour of strict implementations compared to non-strict implementations on typical proof assistant tasks. Such studies could help understand the computational implications of computation in type theories, and lead to better algorithms. For example, it is generally accepted that de Bruijn indices are a necessary technique, but we do not know of studies which investigate this question in detail. But at present, providing better tools for working with inductive types, such as automatic generation of standard theorems and powerful tactics [19], is of greater importance.

Acknowledgements We would like to thank members of the Durham Computer Assisted Reasoning Group (CARG) for comments on drafts. The first author would also like to thank Zhaohui Luo for his continuing encouragement on Plastic. The presentation has been greatly improved through the suggestions of the referees, one of whom was particularly helpful.

References

1. David Aspinall. Proof General WWW Site. `http://www.dcs.ed.ac.uk/home/-proofgen`.
2. P. Callaghan. Coherence Checking of Coercions in Plastic. In *Proc. Workshop on Subtyping & Dependent Types in Programming*, July 2000.
3. P. Callaghan and Z. Luo. Mathematical vernacular in type theory based proof assistants. *User Interfaces for Theorem Provers (UITP'98), Eindhoven*, 1998.

4. Paul Callaghan and Zhaohui Luo. Plastic: an Implementation of LF with Coercive Subtyping & Universes. *Journal of Automated Reasoning (Special Issue on Logical Frameworks)*, 2000. (to appear).
5. Coq. Coq WWW Page. `http://pauillac.inria.fr/coq`.
6. C. Coquand and Th. Coquand. Structured type theory. In *Proc. Workshop on Logical Frameworks and Meta-languages (LFM'99)*, 1999.
7. Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.
8. P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
9. E. Gimènez. Structural Recursive Definitions in Type Theory. In *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
10. H. Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.
11. Haskell. Haskell WWW Site. `http://www.haskell.org`.
12. Isabelle. Isabelle WWW Site. `www.cl.cam.ac.uk/Research/HVG/Isabelle/`.
13. A. Jones, Z. Luo, and S. Soloviev. 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*, 1998.
14. Z. Luo. A unifying theory of dependent types: the schematic approach. *Proc. of Symp. on Logical Foundations of Computer Science (Logic at Tver'92), LNCS 620*, 1992. Also as LFCS Report ECS-LFCS-92-202, Dept. of Computer Science, University of Edinburgh.
15. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
16. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.
17. Z. Luo and P. Callaghan. Mathematical vernacular and conceptual well-formedness in mathematical language. *Proceedings of the 2nd Inter. Conf. on Logical Aspects of Computational Linguistics, LNCS/LNAI 1582*, 1998.
18. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proof and Programs, LNCS*, 1994.
19. C. McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 2000.
20. B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.
21. L. C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions On page: `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, 1999.
22. S. L Peyton Jones et al. GHC Haskell Compiler WWW Site. `http://-www.haskell.org/ghc`.
23. R. Pollack et al. Lego WWW Page. `http://www.dcs.ed.ac.uk/home/lego`.