Object languages in a type-theoretic meta-framework

Paul Callaghan, Zhaohui Luo, and Jianmin Pang

Computer Assisted Reasoning Group
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, Jianmin.Pang}@durham.ac.uk

Abstract. This paper concerns techniques for providing a convenient syntax for object languages implemented via a type-theoretic Logical Framework, and reports on work in progress. We first motivate the need for a type-theoretic logical framework. Firstly, we take the logical framework seriously as a meta-language for implementing object languages (including object type theories). Another reason is the goal of building domain-specific reasoning tools which are implemented using type theory technology but do not require great expertise in type theory to use productively. We then present several examples of bi-directional translations between an encoding in the framework language and a more convenient syntax. The paper ends by discussing several techniques for implementing the translations and properties that we may require for the translation. Coercive subtyping is shown to help in the translation.

1 Introduction

This paper concerns techniques for providing a convenient syntax for object languages implemented via a type-theoretic Logical Framework, and reports on work in progress. It is relevant to the topic of proof presentation because it seeks to support communication between the internal language of the framework and non-expert users of applications built upon the framework (which can require forms of proof). Coercive subtyping is a central part of this work, and it may be useful in other approaches to reasoning tools, especially in the treatment of overloading of notations.

The framework (henceforth LF) is a variant of Martin-Löf's Logical Framework. Martin-Löf proposed this framework in 1986 (see [20] for details) as a *meta-language* to present his earlier intensional type theory [18]. LF provides the basic ingredients of a type theory: types, objects in types, families of types, and families of objects of types, plus a definitional mechanism. The components of Martin-Löf's intensional type theory can be introduced using this LF. We use the form of LF from Luo's monograph [12] and subsequent publications, featuring type annotations on all abstraction forms, a mechanism for introducing inductive types, and an extension for coercive subtyping [13].

Being a meta-language, LF is a deliberately restricted language. To use it for specific applications, the components of an object language must be introduced as new constants and computation rules, typically by defining suitable inductive types. For example, the theory UTT (related to Lego's ECC [16, 23]) can be specified by introducing some inductive types together with a hierarchy of universes [12]. It is possible to reproduce examples from Lego in UTT, but working with the encoding is less convenient than working directly in Lego. Techniques for making such encodings easier to work with are the subject of this paper.

But why should we want to use a meta-language, when several type theory proof assistants already exist and provide a good basis for advanced use of type theory? Why do we make further difficulties by working within LF? There are several reasons, both theoretical and practical, including (our comments refer primarily to Coq [7] and Lego [23]):

- Theoretical clarity. Some current systems do not provide a clear distinction between families of types and Π-types (see section 2.1 for discussion). This can lead to misunderstandings in use of the type theory implemented in the system. An LF-based presentation is a more precise specification of a type theory [12].
- Modularity. Current systems are large and complex: as well as implementing a type theory, they also implement mechanisms to support work with that specific type theory. Extending the theory typically means non-trivial changes in many areas of the system, and consequently this is a risk in software engineering terms. In an LF-based system, the type theory is changed by just modifying the specification.
- Genericity. Current systems are tied to a particular theory (or close variants of it). It is difficult to use a system to work with a different theory. Plus, if one wants to use a subset of a theory, it is not easy to enforce use of just that subset. In LF, one can specify precisely the type theory needed for a particular application.

For our current work, there is a further reason: *Applications*. A research goal in Durham is to build domain-specific reasoning tools which use type theory as the base logic, but which do not require expertise in type theory to use. Current proof assistants are oriented towards helping expert type theorists build proofs quickly, and they are very successful in this, but they do not appear suitable for use in less specialist settings.

We envisage the following scenario. An implementation of LF provides the core reasoning functionality. A type theory expert encodes (in LF) the type theory needed for an application domain, and gives a mapping between LF syntax and a concrete syntax for the application. A non-expert can then use the resulting system by working in the object language, whilst being exposed to the minimum amount of detail about the underlying framework. In this process, the provision of a convenient object syntax is a very important component, and the one we investigate in this paper.

Related work. The Isabelle system [21, 11] provides a framework of higher order logic, in which other logics may be encoded. It allows users to specify a concrete syntax for their logic, by supplying rewrite rules (supplemented by ML code procedures in certain cases). NuPRL and MetaPRL [6, 19] have a similar facility, allowing users to give "display forms" for values being manipulated, enabling suppression of unnecessary detail. Pollack's argument synthesis [22] also helps to suppress detail by allowing inferable subterms to remain implicit.

There is much to learn from such work, both from the handling of object language syntax and from the many interesting and significant applications which use those systems. But we underline a key difference: our work makes essential use of a type theoretic meta-language. As demonstrated by our examples, this setting raises a group of problems not covered by all of the above approaches, in particular the amount of type information used in an encoding and the overloading of certain notations. A possible question is: to implement the translation, do we need knowledge about the types of the terms being translated? On the basis of our examples, we believe that some type information is required for encodings of several components of an object language, thus preventing use of simple rewrite rules.

Organisation. We first present more details about LF and the way we intend to use it in domain-specific applications. The third section considers the problem of translating between a Lego-style syntax and an encoding of an object language in LF syntax. Although it does not demonstrate use of a domain-specific language, it is a useful intermediate step to achieve because the problems we expect to face in application-oriented languages do occur in such a language. The final section considers techniques for implementing the translations, focusing on the benefits that coercive subtyping brings.

2 The framework LF and an implementation

2.1 The framework LF

The version of LF we use is presented in [12]. It is a variant of Martin-Löf's Logical Framework [20], which was proposed in 1986 as a *meta-language* to present Martin-Löf's earlier intensional type theory [18]. The main difference is the requirement for type annotations on all abstraction forms.

LF provides the four basic concepts required in a type theory: types, objects in types, families of types, and objects of these families of types[1], plus a definition mechanism. It makes a firm distinction between the *types* required for a specific application, and *kinds*, which are part of the framework. Objects of the kind **Type** are types in an object language, and objects of a type T are objects of the

[1] LF as described in [12] actually allows *families of families of types*, a more general concept than families of types. Arguably, this generality is not required; indeed, there seems little call for it in object languages. Luo [14] studies a framework PAL^+ where families of families are not allowed.

kind $El(T)$. One may form dependent product kinds $(x : K_1)K_2$ and abstractions whose kinds are dependent products $[x : K_1]y : (x : K_1)K_2$, where K_1 is a kind, and under assumption $x : K_1$, K_2 is a kind and $y : K_2$. Where x does not occur free in K_2, we write $(x : K_1)K_2$ as $K_1 \to K_2$. Relevant rules are shown in figure 1 (some rules for substitution and equality are omitted).

The kind Type

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

Dependent product kinds

$$\frac{\Gamma \vdash K\textbf{ kind} \quad \Gamma, x : K \vdash K'\textbf{ kind}}{\Gamma \vdash (x : K)K'\textbf{ kind}} \qquad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x : K_1 \vdash K_1' = K_2'}{\Gamma \vdash (x : K_1)K_1' = (x : K_2)K_2'}$$

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

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

$$(\beta)\ \frac{\Gamma, x : K \vdash k'\!: K' \quad \Gamma \vdash k\!: K}{\Gamma \vdash ([x : K]k')(k) = [k/x]k'\!: [k/x]K'} \qquad (\eta)\ \frac{\Gamma \vdash f\!: (x : K)K' \quad x \notin FV(f)}{\Gamma \vdash [x : K]f(x) = f\!: (x : K)K'}$$

Fig. 1. The inference rules of LF (abridged).

LF is extended by declaring new constants and computation rules involving those constants, subject to certain restrictions. For example, inductive types can be added by declaring a type name, constructors, an elimination operator, and computation rules for elimination. Luo's book gives a schema which allows the addition of a large class of inductive types. Strong normalisation has been proved for this class of inductive types [9]. Introduction of inductive types is thus the preferred method of extending LF.

We emphasise that the notion of dependent product is one in the meta-framework. It is different from that of Π-type as found in some object type theories. The dependent product provides parameterisation mechanisms used to define a type theory, while a Π-type is an inductively defined construct in an object language, representing the *type* of dependent functions. An important difference is that there is a notion of elimination for Π-types but not for dependent products. For example, an object of dependent product kind

$$(A : \textbf{Type})(B : (x : A)\textbf{Type})\textbf{Type}$$

is a family of types parameterised by a type A and a family of types B indexed by objects of type A. Representing such a family by means of an "internal"

Π-type, as done in systems like Lego, is inappropriate and leads to possible misunderstandings in use of type theory.

In later work, Luo proposed an extension of LF which provides an abbreviational mechanism of coercive subtyping [13]. Coercive subtyping allows an application of a functional operation $f : (x : B)C$ to an object $y : A$ to be well-formed if a coercion c from type A to type B is known (written $A <_c B$). Thus $f(y)$ is a well-typed abbreviation of $f(c(y))$. Placing coercive subtyping in the framework allows any theory specified in the framework to use it. Coercive subtyping has many applications, such as simplifying the use of structured objects (e.g. algebraic structures in [2]), overloading of names, aiding use of Tarski-style universes [5], and in natural language semantics [15]. It can also aid the translations described in this paper, by providing an internal way of implementing certain mappings.

2.2 Using theories in LF

One example theory is the Unifying Theory of dependent Types (UTT) [12]. UTT makes use of the meta-framework LF to provide a more satisfactory definition of the concepts appearing in the theory ECC (which is implemented in Lego [23]). ECC combines Coquand and Huet's Calculus of Constructions [8] and Martin-Löf's type theory with universes [18]. UTT extends ECC with a collection of standard inductive types, but uses a Tarski style of definition to introduce an impredicative universe *Prop* of propositions and a hierarchy of (predicative) universes $Type_0, Type_1 \ldots$, as opposed to ECC's Russell style of definition. (Please consult chapter 9 of [12] for further discussion.)

Using this encoding, it is possible to reconstruct developments originally done in ECC/Lego. But there are often several ways to encode a concept. This is natural, because the LF presentation separates notions which are (incorrectly) conflated in Lego and similar systems. The choice of encoding can also restrict options for proof later. For example, one can code a binary relation as a framework-level family (i.e., a dependent product), but it is impossible to define directly such binary relations by elimination of some inductive type[2]. Such issues are relevant to the handling of object syntax, because it places limits on what we should expect to translate. (In terms of figure 2, it places a constraint on the set LF_T.)

2.3 Plastic: an implementation of LF

LF has been implemented in the form of a proof assistant: Plastic [5, 4][3] Plastic's interface is based on ideas from Lego [16] and Coq [7], providing tactic-based

[2] Strong normalisation has only been proved for small elimination of inductive types. This restricts elimination to producing values in types, not values in dependent products. Thus we need to introduce a new inductive type for binary relations, or to use inductively defined Π-types.

[3] Executables for Solaris and Linux, together with the current libraries, are available by ftp. Contact the first author for details.

proof. (This can be contrasted against the syntax editing style of the ALF group of systems [17].) Plastic's libraries contain most of UTT (from [12]), plus some developments using that type theory.

Plastic's proof facilities are rudimentary, the main tactic being refinement (based on Lego's mechanism). With practice, it is possible to complete large developments. The largest use (to date) is a translation of Pollack's formalisation of records [24] from Lego to Plastic. The resulting Plastic script was around double the size of the Lego script, but did not require any change in the basic proof technique. Much of the translation involved understanding a Lego idiom and establishing a corresponding idiom in Plastic. This translation has provided much useful information for the work described below.

3 Object syntax and framework syntax

This section explains the problems faced by way of several examples. Techniques for implementing the translations required are considered in section 4. The examples also supply further information about how LF can be used, although space does not permit a thorough explanation. (For more information, please refer to [12, 3].)

We consider the problem of translating between an ECC-like language and the LF encoding of UTT, using the syntax of Lego as a model for the former. That is, we would like to use Lego-style syntax to work with the encoding of UTT in LF, rather than having to use the encoding directly. Whilst Lego-style syntax hardly represents a demanding test of our approach, this level is a good intermediate stage for our work. We expect that the problems arising in this setting will appear when we attempt to handle real, application-oriented languages, particularly the issues of dealing with the additional type information required in an encoding, and of overloading of notations. The examples below are chosen to display typical instances of these problems.

After introducing some preliminary notations and concepts, we begin with examples from the internal logic of UTT, then consider argument synthesis and universes. Solutions to these problems will be considered in section 4.

3.1 Preliminaries

For clarity, object-level (i.e. Lego-style) terms are given in the LaTeX style for mathematical expressions, whilst LF terms are given in `courier` font using Plastic's concrete syntax. The scope of a binder extends as far to the right as possible. Extra parentheses may be added to avoid possible misunderstandings. We also do one additional translation for LF syntax, of omitting the operator El which mediates between types and kinds (if $A : \mathbf{Type}$, then $El(A)$ is a kind).

***Prop* – an impredicative universe of propositions.** UTT contains an impredicative universe *Prop* of propositions, specified in the Tarski style. It allows new propositions to be formed by quantifying over some arbitrary type, including the type of propositions. There is a decoder `Prf` which maps propositions to their corresponding objects in the LF kind **Type**, a "type" constructor `FA` (i.e. "for all") representing universal quantification over types to form new propositions, and its corresponding "value" constructor `LL`, which builds proofs of quantifications over propositions. *Prop* is introduced by declaring the following constants and asserting the computation rule for an elimination operator `E_FA`.

```
Prop : Type
Prf  : Prop -> Type
FA   : (A:Type)(A -> Prop) Prop
LL   : (A:Type)(P:A -> Prop) ((x:A)Prf (P x)) -> Prf (FA A P)

E_FA : (A:Type) (P:A -> Prop)
       (C_FA:Prf (FA A P) -> Prop)
       ((g:(x:A)Prf (P x)) Prf (C_FA (LL A P g)))
        -> (z:Prf (FA A P))Prf (C_FA z)
-- E_FA A P C f (LL A P g) = f g
```

Using `E_FA`, one may define `App`, which produces a proof of a proposition `P a` for an object `a` given a proof of the universal quantification.

```
-- App : (A:Type)(P:A -> Prop)Prf (FA A P) -> (a:A)Prf (P a)
```

Some inductive types. *Π*-types and Sigma types can be introduced by declaring the following inductive types. First introduce a type `Pi` parametrised by a type `A` and a family of types `B`, to represent As noted above, there are differences between this object language type and the framework's dependent products.

```
Inductive    [A : Type] [B : A -> Type] [Pi : Type]
Constructors [ La : (f:(x:A) B x) Pi]
```

This gives a type constructor `Pi` with a (value) constructor `La`.

```
-- Pi : (A : Type)(B : A -> Type)Type
-- La : (A : Type)(B : A -> Type) ((x:A)B x) Pi A B
```

Using the elimination operator computed from the declaration, we can define application of such function types.

```
-- ap : (A:Type)(B:A->Type)(f:Pi A B) (x:A)B x
```

Dependent pairs, or Sigma types, are introduced with this inductive type declaration. The (value) constructor is `sigma`. Left and right projections can be defined from the elimination operator arising from the inductive type.

```
Inductive    [A : Type] [B : A -> Type] [Sigma : Type]
Constructors [sigma:(x:A)(y:B x)Sigma]

-- pi1 : (A:Type)(B:A->Type)Sigma A B -> A
-- pi2 : (A:Type)(B:A->Type)(s:Sigma A B) B (pi1 A B s)
```

For reference, the elimination operator computed has the following kind and computation rule.

```
-- E_Sigma : (A:Type) (B:A -> Type)
             (C_Sigma:Sigma A B -> Type)
             ((x:A)(y:B x)C_Sigma (sigma A B x y))
             -> (z:Sigma A B)C_Sigma z
-- E_Sigma A B C f (sigma A B x y) = f x y
```

3.2 Propositions in the internal logic

The Lego term $tautology =_{\mathrm{df}} \{P : Prop\}P \to P$ is a type representing the proposition: for all propositions P, P implies P. The term *tautology* in the LF encoding of UTT (expressed in Plastic concrete syntax) is this: `FA Prop [P:Prop] FA (Prf P) [_:Prf P]P`. The Lego syntax is clearly more preferable.

Next, consider an inhabitant of *tautology*. In Lego syntax, the canonical inhabitant is $[P : Prop][p : P]p$. The LF abstraction term `[P:Prop][p:Prf P]p` is well-typed (it has kind `(P:Prop)(p:Prf P)Prop`), but this term does not inhabit the type required. (Note that the decoder `Prf` must be applied to `P` because the type annotation on bound variables must be a kind, and remember that we are omitting the *El* operator.) We must use the (value) constructor `LL` applied to an internal functional operation. The correct term is:

```
LL Prop ([P:Prop]FA (Prf P) ([_:Prf P]P))
                 ([P:Prop]LL (Prf P) ([_:Prf P]P) ([p:Prf P]p))
```

Some points to note. Firstly, our motivation for providing a better syntax should be clear. Secondly, although the LF terms look complex, there is a direct correspondence between the Lego version and the LF version. More importantly, in principle no information is gained or lost. Once one understands the Lego term, writing the LF term is straightforward, albeit tedious.

3.3 Argument synthesis

The above terms can be simplified by use of argument synthesis [22], where pieces of a term which are inferable (by unification) are not required in the input and are suppressed in the output. Plastic does not implement argument synthesis directly, because the original intention was to handle it via the translation techniques discussed in this paper. One reason for this was to avoid complicating the core system by something that can effectively be done externally.

As a first approximation, we rely on meta-variables together with the refinement proof mechanism to fill in the obvious components. The inferable terms are those which may be solved by unification. For example, when `FA` is applied to a metavariable `?y` and a functional operation `f : (x:T)Prop`, the metavariable can be replaced by term `T` because in the type of `FA`, the first argument and the domain of the second argument must be convertible. Replacing subterms by fresh meta-variables where they are inferable from the surrounding context, the terms can be abbreviated to `FA ? [P:Prop] FA ? [_:Prf P]P` and `LL ? ? [P:Prop] LL ? ? [p:Prf P]p`, which makes the correspondence between Lego-style syntax and the LF version slightly clearer[4].

We could implement argument synthesis by the translation suppressing a pre-specified subset of arguments in an application term on output, and on input replacing the suppressed arguments by fresh meta-variables. The specifier of a type theory would encode this information as part of the syntax of the object language's constants. A notation for imposing a choice for the implicit arguments would also be required, similar to Lego's "bar" notation. This scheme definitely works: we've been doing it by hand since meta-variables were implemented in Plastic, and it is a necessary part of building large terms. With argument synthesis, terms above can then be simplified to: `FA [P:Prop] FA [_:Prf P]P` and `LL [P:Prop] LL [p:Prf P]p`.

3.4 Universes

Section 3.2 mentioned the universe *Prop*. Similar issues arise for the predicative universes $Type_0, Type_1, \ldots$ in UTT. These universes are specified in the Tarski style, as opposed to the Russell style used in ECC. Although semantically clearer, Tarski style universes are less easy to use. We would like to allow use of Tarski style universes to be as convenient as the Russell style universes in Lego. Coercive subtyping can be used to remedy some of the problems by allowing universe names to be used in any higher universe via implicit coercions (see [5] for more details).

But other problems remain. One issue is whether we can retain ECC-style overloading of the Π-type notation $\{x : A\}B$. The ECC concept it represents maps to several concepts in UTT, namely `FA` with `Prop` or (names of) Π-types with the predicative universes $Type_i$. A further complication is that that *all* types in ECC are a member of some type universe in the hierarchy, which is not true for UTT because the objects in universes are just *codes* for types in the kind **Type**, and not the types themselves. We are also considering how to incorporate the flexibility of universe polymorphism [10] into the system.

4 Implementing the translations

Plastic implements some simple syntax translations already. They are very useful for reducing the amount of information in type expressions so that they are more readable. For example, `FA A [x:A]B` can be entered as, and shown as, `{x:A}B` (but note that this prevents use of the notation with Π-types, because

[4] Plastic does not implement higher order unification, which the example above may appear to require for the second argument to `LL`. For example, we have to synthesise a term P given the information that $P(x)$ converts with some term T. In these cases, the required information can be synthesised from the type of the argument following it, so Plastic implements it as a special case in the metavariable processing.

the underlying types are different). There is another syntax for working with inductive types whose type constructors have two arguments, the second of which is an object of a family of types; for example, `Sigma A [x:A]B` can be written as `Sigma[x:A]B` because the first argument is inferable from the domain of the second argument. These translations are implemented in two separate areas of code, one during parsing, the other during pretty-printing. They are simple translations, and do not require any type information. But this solution is not ideal.

Intuitively, we want to specify a mapping which can be used in *both* directions, and which can be specified in a simple language, and for which properties consistent with the other components of the reasoning tool can be established, such as soundness. Since we are working in a type theory setting, it would be ideal if type information can be used to support this. We'd also like the process to be simple, ideally without the complication of needing detailed type information about the terms being translated.

There are several issues to consider *outside* of a particular implementation. Do we need to think of a text-to-text translation scheme? A related point is: does every system need to do its own pretty-printing or parsing? Arguably not, because most systems have similar functionality in both areas. This functionality can be done by a standard component which communicates with a specific proof system via some protocol. In the course of work on Proof General, Aspinall has been investigating such protocols based on XML [1]. So ideally, our translations should be based directly on trees.

We now discuss relevant technologies.

4.1 Ranta's GF

Ranta's GF ("Grammatical Framework") [25] can be used to link two or more concrete syntaxes via a common abstract syntax. It is a dependent type theory with extensions for handling concrete syntax. Given an abstract syntax and concrete syntaxes, it can derive parsing and pretty-printing algorithms for the concrete syntaxes to the abstract syntax and vice versa, thus allows the concrete syntax of one language to be translated to the concrete syntax of another language. For example, it can be used to translate fragments of English to Swedish (and vice versa), or to translate terms in Haskell to Natural Language.

For our problem, it provides a good way to specify the translations declaratively, in a reversible form. Jianmin Pang has developed GF grammars to translate a subset of Lego-style terms to their UTT encoding in LF. However, GF is currently oriented towards converting between strings, whereas we argue above that converting between trees is more appropriate for our problem. One consequence of using strings is that the grammar must contain details of precedence and associativity, a detail which we would prefer to avoid whilst specifying the translations.

Another problem is our plans for argument synthesis. As discussed above, output terms will be generated with fresh meta-variables, and the corresponding parts of input terms should be ignored. This technique is not symmetric: slightly different grammars are needed between input and output. Handling this may require additional complexity in the grammar, e.g. parametrizing the grammar by a mode, or it may require changes to GF.

In conclusion, some aspects of GF are ideal for our problem, but some features of our application do not fit well with the original design goals of GF.

4.2 Coercive Subtyping

Whilst working on the examples in GF, we realised that coercive subtyping could be used in several places to simplify our translation. This is to be expected, since coercive subtyping is primarily a mechanism for *abbreviation*. Specific cases where coercive subtyping is useful are discussed below. But consider first the implications for our problem.

LF with coercive subtyping incorporates a powerful mechanism for abbreviation. It allows us to specify some of the translation *internally*, and in a type-safe way. By handling some aspects of translation inside the system, we reduce the number of cases in translation that we formerly had expected to handle externally. There is now the possibility that the number of cases in translation that cannot be handled by appropriate coercions is small enough to be implemented inside the proof assistant. (Our former concern on this point assumed that we did have to consider many cases.)

This point can be considered another way: coercions allow us to reduce the gap between the object language and the framework language, by allowing significant parts of the object language to be abbreviated inside the framework language. In terms of figure 2, coercions allow us to reduce the subset LF_T which is the target of the translation. Terms in the framework language which contain implicit coercions are able to represent some terms in the object language almost verbatim. There is a stronger sense in which the object language is *embedded* in the framework language.

This idea appeared very recently in our work, and needs much more refinement. Preliminary results are briefly described next; space does not permit a thorough description of the results. All of the examples described work with the current version of Plastic. In addition to the basic coercions given, we also need several "coercion rules" [5] which synthesise coercions over data structures given coercions on data within the structures. This allows propagation of coercions over structured values, e.g. to handle the rightwards nesting of binding forms. The following coercion allows subtyping between Π-types if there is a (parametrized) coercion between the ranges (B and B') of the Π-types.

$$\frac{A : \mathbf{Type} \quad B : A \to \mathbf{Type} \quad B' : A \to \mathbf{Type} \quad x : A \vdash B(x) < B'(x)}{\Pi(A, B) < \Pi(A, B')}$$

- Overloading of binding constructors. In examples before, the constructor `LL` was used to create values in types built with the type (name) constructor `FA`. For similar type constructors, e.g. Π-types and function spaces (as inductive types), and the names in universes of such types, it appeared that a separate constructor was needed in each case.

However, we can define and use coercions between a generic function type and a specific function-like type. For example, with a parametrized coercion between Π-types and (proofs of) universal quantifications (plus relevant coercion rules, as mentioned above)

$$A : \mathbf{Type},\ B : A \to Prop \vdash \Pi(A, [x : A]\mathbf{Prf}(B(x))) < \mathbf{Prf}(\mathrm{FA}(A, B))$$

we can initially translate the ECC term $[P : Prop][p : P]p$ to the value in the Π-type shown below, and then rely on coercions to convert it to the correct propositional type:

```
La ? ? ([P:Prop]La ? ? ([p:P] p))
  : Pi Prop ([P:Prop] Pi P       ([p:P]Prop))
    -- principle type
  : Prf(FA Prop ([P:Prop] FA (Prf P) ([_:Prf P]P)))
    -- by coercions
```

The consequence of this is that any Π-like binding form can be translated to a generic type, and then coerced to the specific type required, rather than giving similar translations for each case - which might require type knowledge to select during translation. This is a significant reduction of difficulty of translation.

- Overloading of tuple notation. A similar technique can be used to solve the problem of giving types to pairs. For example, the expression (a, b) in Lego can have either a dependent pair (i.e. Sigma) type or a non-dependent Pair type. Unless a specific type signature is given, Lego defaults to the latter case. But with coercions, we can translate the tuple values to a generic type and coerce appropriately. (This example was discussed in [13], but it was incomplete.) By adding some coercion rules [5], we can use the technique on arbitrarily complex tuples.
- Argument synthesis: parametric coercions can be used to encode certain cases of argument synthesis, by transferring the implicit parameters of a function to parameters of coercions acting upon the non-implicit arguments of the function. We have demonstrated this technique for certain functions on inductive Π and Sigma types. (However, this does force the use of universes.) This technique is an alternative to the earlier suggestions made concerning suppression of information during translation.

4.3 Discussion

From our work so far, it appears that working in a type-theoretic framework places additional demands on the translations between object language and framework language. The techniques used in Isabelle are very successful in that setting, but different techniques are needed in our setting to handle aspects like argument synthesis and universes.

We have seen that coercions can do some of the work involved in translation, and do it internally to the system, thus reducing the number of cases we have to implement via external means. It raises the possibility that the number of cases actually required is small enough to be implemented directly inside the proof assistant.

The use of coercions to help with translation needs further study. Also remaining to be studied are inductive families, in particular their elimination operators, whose kinds are not translatable into object languages because they involve complex use of families.

Another interesting question is the effect of *computation* at the framework level. Computation inside the language is a key element of type theory. Referring to figure 2, object language terms will be mapped to a subset LF_T, and we intend that a reverse translation can be achieved from that same set. It is possible that computation on a term in LF_T will result in a term outside of LF_T, and hence produce a term which is not translatable to the object language. This clearly requires further study.

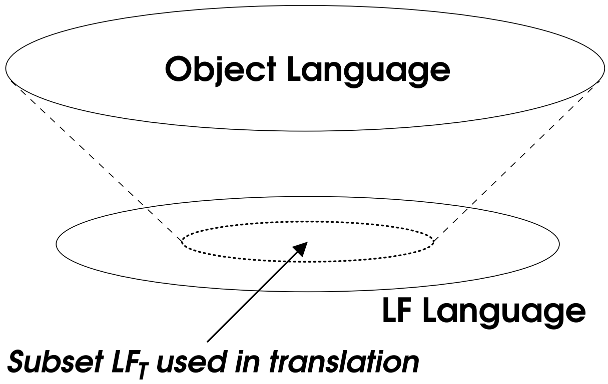

Fig. 2. Mapping between object language and a subset of LF

References

1. David Aspinall. Protocols for interactive e-proof. (available at `http://-zermelo.dcs.ed.ac.uk/~da/drafts/#eproof`), 2000.
2. A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.
3. P. Callaghan. The LF Style Guide: examples of encodings and proofs. (in preparation), 2001.
4. P. Callaghan and Z. Luo. Implementation Techniques for Inductive Types in Plastic. In *Proc. TYPES'99*. Springer-Verlag, 2000.
5. 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).

6. R.L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Pretice-Hall, 1986.
7. Coq. Coq WWW Page. `http://pauillac.inria.fr/coq`.
8. Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.
9. H. Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.
10. R. Harper and R. Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions. *Theoretical Computer Science*, 89(1), 1991.
11. Isabelle. Isabelle WWW Page. `www.cl.cam.ac.uk/Research/HVG/Isabelle/`.
12. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
13. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.
14. Z. Luo. PAL$^+$: a lambda-free logical framework. Proc of Inter Workshop on Logical Frameworks and Meta-languages (LFM'2000), Santa Barbara, California, 2000.
15. 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.
16. Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.
17. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proof and Programs, LNCS*, 1994.
18. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
19. MetaPRL. MetaPRL WWW Page. `http://cvs.metaprl.org:12000/metaprl/`.
20. B. Nordström, K. Petersson, and J. M. Smith. Martin-Löf's Type Theory. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 2000.
21. L. Paulson. Introduction to Isabelle. Technical Report 280, Computer Laboratory, Cambridge University, 1993.
22. R. Pollack. Implicit syntax. In the preliminary Proceedings of the 1st Workshop on Logical Frameworks, 1990.
23. R. Pollack et al. Lego WWW Page. `http://www.dcs.ed.ac.uk/home/lego`.
24. Robert Pollack. Dependently typed records for representing mathematical structure. In Aagard and Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.
25. A. Ranta. Grammatical Framework (GF) Page. `http://www.cs.chalmers.se/~aarne/GF/`.