Structured Specifications with Hiding in the Edinburgh Logical Framework LF

Guided Research Thesis

Stefania Dumbrava

Supervisors: Michael Kohlhase, Florian Rabe
Jacobs University Bremen

20. May 2010

Abstract

Structured specifications are a common technique for achieving data encapsulation via modularity, both in computer science and in mathematics. Such specifications are constructed from basic ones through high-level operations, among which the most important ones are union, instantiation of parametric specifications and hiding. Our setting is that of logical frameworks, which are tools for specifying deductive systems. Specifically, we work in the proof-theoretical **LF** language, based on dependent type theory and for which a module system has recently been developed. The **LF** module system is based on signature morphisms and provides the structuring primitives of union and instantiation. We aim at extending the language such that it also supports hiding. The problem is particularly challenging, since hiding is a difficult operation to realize, given that it doesn't allow for as good of a specification decomposition as do other structuring operations. Moreover, existing languages with hiding have a model-theoretic semantics, while we need to introduce the operation at a proof-theoretic level. We propose an extension to the **LF** language with hiding and give its semantics, formally defining this structuring concept and the induced application of partial morphisms. We evaluate the resulting language against a series of benchmark cases, discussing the newly achieved expressivity, possible applications and future improvements. Even though our definitions and results are based on **LF**, they can easily be extended to other type theories.

1 Introduction

Given the present computing ubiquity, our dependence on software has increased dramatically, making it necessary to ensure the reliability and robustness of computer systems. To this extent, **formal methods** have been introduced, in order to systematically and rigorously study software development. More specifically, such methods are based on mathematical techniques and can be applied, on the one hand, to a **specification** language, in order to describe a system's behavior, and, on the other hand, to a **verification** process, in order to validate the given specification against reality. Automated verification requires machine readability and, thus, the use of **formal specifications**. Also, since the intricacy of certain systems is reflected by their specification, dealing with such complex cases, commands the use of **structured specifications**, which allow for scalability. We focus on the Edinburgh Logical Framework **LF**, which is a meta-language for specifying logics and for which a module system has been developed. In this setting, we aim to extend **LF**, with the hiding operation.

This thesis is organized as follows. In Section 2, we give an overview of formal specification languages, such as OBJ, CASL, and Development Graphs. Next, in Section 3, we introduce the concept of logical frameworks through the Edinburgh Logical Framework **LF**. In Section 4, we present the proposed extended LF grammar and partial signature morphisms. Our main result can be summarized as follows: partial morphisms preserve typing, whenever defined. In Section 5 we analyze a series of benchmarks, discuss their solutions and the extent to which they are supported by our current setup. We summarize our results, outline possible applications, future improvements and present our conclusions in Section 6.

2 Related Work

2.1 Formal Specifications

The initial step in using formal methods in system implementation development is designing the system's specification, which provides a description of the problem that needs to be solved. While there are numerous informal and semi-formal means of writing specifications, such as using natural language, notations, diagrams or languages like UML [Bur97] or OCL [WK99], their poor semantics makes reasoning and inference based on them unfeasible. Consequently, formal specification languages have been introduced to provide a higher level of abstraction and accuracy, with strong underlying semantics.

A formal specification is a mathematical description of a program's properties and can be designed using various methods based on logic, e.g the specification language Z [Spi88] is based on Zermelo-Fraenkel set theory and First Order Logic. Among these methods are the model-oriented approach, better suited for the description of state based systems and the algebraic approach, better suited for abstract datatype specifications.

The model-oriented approach treats the system as a mathematical model characterized by a state space and certain operations, which describe the system's behavior. For example, typical VDM-SL (Vienna Development Method Specification Language) [Daw91] or Z specifications have a state space consisting of a set of variables, whose value changes correspond to certain events, reflecting the system's behavior.

The algebraic approach specifies systems using methods from abstract algebra and category theory. Algebraic specification languages are procedural and facilitate stand-alone specifications. To this extent, these languages are part of the set-theory based group of module systems, which are systems constructed using self-contained components. In the case of such specification languages, the behavior of the system is described by a set of so called characterizing operations whose meaning is given by a set of (equational) axioms formalizing the relations between them.

In writing large specifications, it is more convenient to design them using a structural approach of combining and modifying smaller specifications. This supports modular decomposition, enabling distributed development, i.e a divide-and-conquer approach to defining a system's components. This is needed due to scalability reasons, given that a complex system usually contains numerous functions and axioms, which become hard to manage when defined using a simple basic specification.

Such basic specifications are of the form: $Sp = \langle \Gamma, Ax \rangle$, where Γ is a signature, i.e set of symbols and Ax is a set of axioms over the signature, i.e a set formulae from $\mathit{wff}(\Gamma, Var)$. Its semantics is given by the class of all models (possible implementations) that satisfy the axioms. Structured specifications are obtained from basic specifications via structuring operations such as union, extension, translation and hiding.

OBJ

The pioneer example of an algebraic specification language is the OBJ [GWM+93] family of languages, developed in the 1970's. It is based on the Clear programming language, which structures specifications through methods independent of the underlying syntax, i.e of the institution in which the specifications are expressed. Thus, the system allows for modular specifications, which are generic over formalisms. This feature was extended by OBJ and led to the relativization of algebraic specifications over any logic [DFI+].

CASL

A recent algebraic specification formalisms is given by the Common Algebraic Specification Language (CASL) [CoF04]. It has been developed as part of the Common Framework Initiative (CoFI), in order to unify the multitude of existing algebraic specification languages and to define a standard. CASL consists of several layers of modules called specifications, including basic or unstructured specifications, which can be combined through various mechanisms,

among which are unions, extensions, translations, hiding and reductions, in order to obtain structured specifications. Specifications can thus be *united*, *extended* with further signature items, their models may be *restricted* to initial models and their signatures may be *translated* to use different symbols through signature morphisms or may be partially *hidden*.

Development Graphs

The development graph language is a modular language used to encode structured specifications in various phases of program development ([MAH06], [AHMS99]). Its modules are theories and represent nodes in the graph, such that the leaves correspond to basic specifications, which do not use other theories and inner nodes correspond to structured specifications, which use existing theories to derive new ones. The links in the graph define how theories are used inside other theories and are of two types: **definitional links**, representing module imports and **theorem links**, representing proof obligations.

Hiding is done via hiding definitional links, which are similar to **global** definition links – directed links that import the whole subgraph below a node – with the addition that one can hide symbols of the signature. The hiding operation from a node N to a node M, $M \xrightarrow[h]{\sigma} N$ is denoted by a signature morphism of the form $\sigma : \Sigma^N \rightarrow \Sigma^M$ that goes against the direction of the link.

2.2 The Hiding Operation

The concept of information hiding was developed by one of the pioneers of Software Engineering, David Parnas, and is acclaimed as one of the key principles of the discipline [Par72]. Information hiding is used to mask irrelevant (uncommon/unshared) data information details from a given level of implementation at a lower one. Thus, it allows for a better structuring of information distribution, serving as a major method for achieving *data encapsulation* via *modularity*. Also, the operation allows for reusability of proofs and handling of untranslatable theory fragments.

Moreover, hiding is not only important in software engineering, but also in algebraic specifications, where, as seen above, it is used as a building operation. In this case, hiding is also motivated on theoretical grounds, due to the limitations of algebraic specifications. Specifically, it has been proven that some Σ-algebras cannot be specified as initial Σ-algebras over a finite Σ-equation set ([Maj77]), but that all computable Σ-algebras can be specified as restrictions of an initial $\Sigma^{'}$-algebra over a finite $\Sigma^{'}$-equation set, for some finite $\Sigma^{'} \supset \Sigma$ ([BT95],[MG85],[Ros04]).

As a motivational example, we refer to the method used for specifying the natural numbers in the first order framework CASL. We specify the natural numbers **HNat**, through the Peano axioms, which belong to First Order Logic, except for the fifth one, the induction axiom, which uses quantification over predicates and is thus only specifiable in Higher Order Logic. In order to be able

to still specify the naturals in CASL (i.e. in First Order Logic), one would be forced to hide the higher order induction axiom, i.e hiding along **HOL2FOL**.[1]

Higher Order Nat (HNat)

hide HOL2FOL

First Order "Nat" (FNat)

spec HNat =
sort $\mathbb{N}$;
ops $0 : \mathbb{N}$;
$\mathsf{succ} : \mathbb{N} \to \mathbb{N}$;

axioms $\forall x.\neg\, \mathsf{succ}(x) \doteq x$
$\forall x.\neg\, \mathsf{succ}(x) \doteq 0$
$\forall x \forall y.\neg(x \doteq y) \Rightarrow \neg(\mathsf{succ}(x) \doteq \mathsf{succ}(y))$
$\forall^2 \mathsf{P}.\ (\mathsf{P}(0) \wedge \forall n : \mathsf{HNat}.\mathsf{P}(n) \Rightarrow \mathsf{P}(\mathsf{succ}(n))) \Rightarrow \forall n : \mathsf{HNat}.\mathsf{P}(n)$

3 Logical Frameworks: LF

Logical frameworks are a tool for specifying logical systems. They consist of a meta-langage and a description of both the class of logics to be represented and of the mechanisms used to this extent. The motivation behind developing such logical frameworks can be identified as two-fold.

At a theoretical level, logical frameworks provide a formal basis for describing logical reasoning. This attempt to define logic is an especially valuable insight, given the foundational crisis in mathematics, which has placed logic at the basis of research in both mathematics and computer science.

At a practical level, such logical frameworks constitute a suitable environment for developing reliable and powerful formal verification tools, since they allow for logic-independent proof development. One such example is Z-in-Isabelle [SKS02, KB95], which is an embedding of the specification language Z and a deductive system for Z in the generic theorem prover Isabelle.

Logical frameworks can be **set-theoretical**, based on Tarski's view of consequence and characterizing logics model theoretically or **type-theoretical** based on the Curry-Howard isomorphism, characterizing logics proof theoretically. The former are exemplified by institutions [GB92, GR01] and General Logics [Mes89], while the latter by Automath [dB70], Isabelle [Pau94] and the Edinburgh Logical Framework **LF** ([HHP93]). An overview is given in [HR09].

The latter example, **LF**, is a corner of the λ-cube obtained by adding dependent types to the typed λ-calculus. The resulting $\lambda\Pi$-calculus consists of simply typed terms, types and kinded type families. Since **LF** is a type theory, objects s and types S are related via the typing judgment $s : S$. Also, following the Curry-Howard isomorphism, **LF** represents all object language judgements as types and proofs as terms.

Recently, a module system for the Twelf implementation of **LF** was developed [RS09]. An outline of the syntax of the module system is given below,

[1] Still, to properly represent the naturals in FOL, one needs a weakened version of induction. One approach, discussed in Section 4, is the axiomatization of inductive higher-order theorems, that are still representable in FOL

omitting details which are irrelevant for the purposes of this paper. The module system has modules Γ, whose primitive concepts are **signatures** and **signature morphisms**. Σ is the body of a signature and contains constant and structure declarations. Signature morphisms define mappings between signatures and are distinguishable into *structures*, which copy and instantiate a signature into another and *views* which provide translations between signatures. Drawing a parallel to Development Graphs (see Section 2), we can consider structures as definitional theory morphisms and views as postulated theory morphisms. Also, notice that we merge the categories of kinds, types and objects into that of terms.

$$
\begin{array}{lll}
\text{Signature body:} & \Sigma & ::= \ . \mid \Sigma, c : E \mid \Sigma, c : E = e \\
 & & \quad \mid \Sigma, \ \%\mathbf{struct}\ s : S = \{\ \sigma\ \}. \\
\text{Assignments:} & \sigma & ::= \ . \mid \sigma, c := E. \\
\text{Module:} & \Gamma & ::= \ . \mid \Gamma, T = \{\ \Sigma\ \} \mid \Gamma, v : T \to T = \{\ \sigma\ \}. \\
\text{Expressions:} & E & ::= c \mid x \mid \text{type} \mid E\ E \mid \lambda x : E.e \mid \Pi x : E.E.
\end{array}
$$

Important operations in **LF** are function type and term constructions. Thus, given types S and S', the type $S \to S'$ is the type of functions from S to S' and, if $t : S'$ with free variable $x : S$, then $\lambda_{x:S}t$ is the function of type $S \to S'$, which returns for $s : S$, the substitution **s** for x in t: $t[x/s]$. **LF** also allows for operations such as signature translations, union, views. As stated previously, this paper sets to investigate solutions to extending **LF** with hiding.

The semantics of the modular **LF** is given by elaboration to non-modular syntax, i.e through flattening. To illustrate this, consider the example of signature definitions given below:

```
Normal:
%sig S = {a:type. b:type = a.}. %% Symbolically equal elements
%sig T = {%struct s:S = {}.}.   %% Include in envelope sig.

Elaborates to:
%sig T = {s.a:type. s.b:type = s.a.}. %% Expand to primitives
```

4 Hiding in LF

Even though **LF** is not a logic, given that it does not have formulas or consequence relation between them, it nevertheless has a semantics, as does every type theory. Analogous to logics, the semantics of **LF** can be defined from both a proof-theoretical and a model-theoretical perspective, the type/proof-theoretical semantics being the one primarily used.

Depending on which approach to defining semantics we chose, introducing hiding can become a challenging task. More precisely, in the model-theoretic case, the semantics is given by the models and, hence, removing (hiding) declarations does not change the structure of the theory, but only how it refers to such declarations. However, the proof-theoretic case is problematic, since there are no models and the semantics is given by the typing well-formedness. Hence, we cannot apply hiding directly, since the operation would change the objects themselves, hiding entire expressions and not just their names, as it would in the previous case.

4.1 Syntax

Grammar We aim at extending the initial **LF** grammar with the minimum non-invasive number of primitives, such that we can preserve previous results. We manage to do so by only modifying the assignments with the additional hide c construct, which allows for a higher degree of generality, since the objects we work with, given in the signature, remain unchanged.

$$\begin{array}{lll} \text{Signature:} & \Sigma & ::= \ . \mid \Sigma, c : E \mid \Sigma, c : E = E \\ \text{Assignments:} & \sigma & ::= \ . \mid \sigma, c := E \mid \sigma, \ \text{hide } c \\ \text{Expressions:} & E & ::= c \mid x \mid \text{type} \mid E\ E \mid \lambda x : E.E \mid \Pi x : E.E \end{array}$$

Strict Morphism Application Syntactically, we add hiding in **LF** by allowing morphisms to be partial. Hence, instead of an assignment $c := E$, a morphism may contain hide c, in which case we write $\sigma(c) = \bot$. Then, the semantics of σ is the homomorphic extension to a partial mapping.

Hence, assuming a signature morphism σ, we define $\overline{\sigma(-)}^{L}$ as:

$$\begin{aligned}
&\overline{\sigma}^{L}_{aux}(type) = type \\
&\overline{\sigma}^{L}_{aux}(\lambda x : E.F) = \lambda x : \overline{\sigma}^{L}_{aux}(E).\overline{\sigma}^{L}_{aux}(F) \\
&\overline{\sigma}^{L}_{aux}(\Pi x : E.F) = \Pi x : \overline{\sigma}^{L}_{aux}(E).\overline{\sigma}^{L}_{aux}(F) \\
&\overline{\sigma}^{L}_{aux}(E\ F) = \overline{\sigma}^{L}_{aux}(E)\ \overline{\sigma}^{L}_{aux}(F) \\
&\overline{\sigma}^{L}_{aux}(x : E) = x : \overline{\sigma}^{L}_{aux}(E) \\
&\overline{\sigma}^{L}_{aux}(c) = \begin{cases} E & \text{if } c := E \text{ in } \sigma \\ \bot & \text{if hide } c \text{ in } \sigma \end{cases} \\
&\overline{\sigma}^{L}_{aux}(.) = . \\
&\overline{\sigma}^{L}_{aux}(\Gamma, x : E) = \overline{\sigma}^{L}(\Gamma), x : \overline{\sigma}^{L}(E). \\
&\overline{\sigma}^{L}(E) = \begin{cases} \overline{\sigma}^{L}_{aux}(E) & \text{if } \bot \notin \overline{\sigma}^{L}_{aux}(E) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

4.2 Typing

Inference System We keep the original **LF** judgement system for signature morphisms, contexts, substitutions and terms, An extension is necessary to the morphism judgements, in order to incorporate the hide c constructor and the $\bot$ constants, allowing for complete and constraint hiding, as discussed in Section 5. In particular, *decmap* and *defmapf* are the same as their counterparts in the **LF** judgement system, as they govern the well-defined cases.

$$\frac{}{\vdash \cdot}\ sigempty \qquad \frac{\vdash \Sigma}{\vdash \cdot : \cdot \to \Sigma}\ mapempty$$

$$\frac{\vdash \Sigma \quad c \notin \Sigma \quad \cdot \vdash_\Sigma E \in \{type, kind\}}{\vdash \Sigma, c : E}\ dec \qquad \frac{\vdash \Sigma \quad c \notin \Sigma \quad \vdash_\Sigma E_2 : E_1}{\vdash \Sigma, c : E_1 = E_2}\ def$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \cdot \vdash_{\Sigma'} E' : \sigma(E) \neq \bot}{\vdash \underline{\sigma, c := E'} : \underline{\Sigma, c : E} \to \Sigma'}\ decmap \qquad \frac{\vdash \sigma : \Sigma \to \Sigma' \quad \sigma(E) \neq \bot \quad \sigma(E') \neq \bot}{\vdash \underline{\sigma, c := \sigma(E')} : \underline{\Sigma, c : E = E'} \to \Sigma'}\ defmapforced$$

$$\frac{\sigma(E) \neq \bot \quad \vdash \sigma : \Sigma \to \Sigma'}{\vdash \underline{\sigma, \text{hide } c} : \underline{\Sigma, c : E} \to \Sigma'}\ dechide \qquad \frac{\vdash \sigma : \Sigma \to \Sigma'}{\vdash \underline{\sigma, \text{hide } c} : \underline{\Sigma, c : E = E'} \to \Sigma'}\ defhide$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \sigma(E) = \bot}{\vdash \underline{\sigma, \text{hide } c} : \underline{\Sigma, c : E} \to \Sigma'}\ dechideforced \qquad \frac{\vdash \sigma : \Sigma \to \Sigma' \quad \sigma(E) = \bot}{\vdash \underline{\sigma, c := \sigma(E')} : \underline{\Sigma, c : E = E'} \to \Sigma'}\ defhideforced$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \sigma(E) \neq \bot = \sigma(E') \quad \cdot \vdash_{\Sigma'} E'' : \sigma(E)}{\vdash \underline{\sigma, c := E''} : \underline{\Sigma, c : E = E'} \to \Sigma'}\ defmap$$

$$\frac{\vdash \Sigma}{\vdash_\Sigma \cdot}\ conempty \qquad \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma E : type}{\vdash_\Sigma \Gamma, x : E}\ condec$$

$$\frac{\vdash_\Sigma \Gamma'}{\vdash_\Sigma \cdot : \cdot \to \Gamma'}\ subsempty \qquad \frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \quad \Gamma' \vdash_\Sigma t : \overline{\gamma}(E)}{\vdash_\Sigma \underline{\gamma, x/t} : \underline{\Gamma, x : E} \to \Gamma'}\ subsdec$$

$$\frac{c : E\ [= E'\ ] \in \Sigma \quad \vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma c : E}\ termcon \qquad \frac{x : E \in \Gamma \quad \vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma x : E}\ termvar$$

The base cases are given by *sigempty* and *mapempty*. The *dec* and *def* rules specify the cases for extending a signature with a constant with, respectively without, a definition. Also, *decmap* and *defmapforced* handle the cases for mapping a constant with, respectively without, a definition, in the normal setting, in which hiding does not occur. Next, *dechide* and *defhide* give the rules for hiding a constant with, respectively without, a definition. Similarly, *dechideforced* and *defhideforced* treat the same cases with the added strictness restriction that the type of the constants is hidden. The most interesting rule is *defmap*, as it postulates that a constant with a non-hidden type can be mapped to a target expression, even if the definition of the constant is hidden.

Note that we are omitting the rules for typing non-atomic terms and the rules for equality of terms.

Type Preservation

Lemma. Given $\Gamma \vdash_\Sigma E : F$ and $\vdash \sigma : \Sigma \to \Sigma'$, such that $\overline{\sigma}^L(E) \neq \bot \neq \overline{\sigma}^L(F)$ and $\bot \notin \overline{\sigma}^L_{aux}(E)$, $\bot \notin \overline{\sigma}^L_{aux}(F)$, the following hold:

- Type preservation: $\overline{\sigma}^L(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(E) : \overline{\sigma}^L(F)$.
- Equality preservation: $\overline{\sigma}^L(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(E) = \overline{\sigma}^L(F)$.

Proof: We prove type preservation. Equality preservation follows similarly.

We proceed by induction on the derivation of $\Gamma \vdash E : F$.

- 1.1 **constants**

  Given $\Gamma \vdash_\Sigma c : E$ and $\vdash \sigma : \Sigma \to \Sigma'$, we aim to prove $\overline{\sigma}^L(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(c) : \overline{\sigma}^L(E)$, where $\overline{\sigma}^L(c) \neq \bot \neq \overline{\sigma}^L(E)$.

  We can either have $c : E$ in Σ or $c : E = E'$ in Σ (by *termcon*).

  In the first case, $c : E$ in Σ, since $\overline{\sigma}^L(c) \neq \bot$, it follows that hide c is not in σ. Hence, only the *decmap* rule may have been applied to derive $\vdash \sigma : \Sigma \to \Sigma'$, where $c := E'$ in σ. As a result, we get that $\cdot \vdash_{\Sigma'} E' : \overline{\sigma}^L(E)$, which is the same as $\cdot \vdash_{\Sigma'} \overline{\sigma}^L(c) : \overline{\sigma}^L(E)$. From this, we can conclude by *weakening*, that: $\overline{\sigma}^L(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(c) : \overline{\sigma}^L(E)$.

  In the second case, $c : E = E'$ in Σ. Since $\overline{\sigma}^L(E) \neq \bot \Rightarrow \overline{\sigma}^L(c) \neq \bot$ and hide c is not in σ. Therefore, *defhideforced* was not applied to derive $\vdash \sigma : \Sigma \to \Sigma'$. If $\overline{\sigma}^L(E') = \bot$, then, according to *defmap*, $c := E''$ in σ, we have that $E'' : \sigma(E)$ and, as above, we can conclude that $\overline{\sigma}^L(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(c) : \overline{\sigma}^L(E)$.

  If $\overline{\sigma}^L(E') \neq \bot$, then *defmapforced*, $c := \sigma(E')$ in σ, must have been applied. From *def* we get $\vdash_\Sigma E' : E$ and hence that $\overline{\sigma}^L(E') : \overline{\sigma}^L(E)$. Applying *weakening* and the induction hypothesis we obtain what had to be proven.

- 1.2 **variables**

  Given $\Gamma \vdash_\Sigma x : F$, by LF, we have that $x : E$ in σ and according to the morphism application rules, $x : \sigma(E)$ in $\sigma(\Gamma)$. Since $\overline{\sigma}^L(E) \neq \bot$, by LF, we have that $\vdash_{\Sigma'} x : \sigma(E)$

- 1.3 **./types**

  Given $\sigma : \Sigma \to \Sigma'$, we know that: $\overline{\sigma}^L_{aux}(\cdot) = \cdot$, $\overline{\sigma}^L_{aux}(\text{type}) = \text{type}$, which trivially implies type preservation, since $\overline{\sigma}^L_{aux}$ is the identity morphism in these cases.

- 2.1 **λ-abstraction**

  Assume $\Gamma \vdash_\Sigma \lambda x : E.E' : \Pi x : E.E''$, where we know that $\overline{\sigma}^L(\lambda x : E.E') \neq \bot$, $\overline{\sigma}^L(\Pi x : E.E'') \neq \bot$.

By LF, we have that $\Gamma, x : E \vdash_\Sigma E' : E''$. If $\bot \notin \overline{\sigma}^L_{aux}(\lambda x : E.E')$, $\bot \notin \overline{\sigma}^L_{aux}(\Pi x : E.E'')$, then also $\bot \notin \overline{\sigma}^L_{aux}(E)$, $\bot \notin \overline{\sigma}^L_{aux}(E')$, $\bot \notin \overline{\sigma}^L_{aux}(E'')$. Thus, by the induction hypothesis, $\overline{\sigma}^L(\Gamma), x : \overline{\sigma}^L(E) \vdash_{\Sigma'} \overline{\sigma}^L(E') : \overline{\sigma}^L(E'')$ and, by λ-abstraction, we have that: $\overline{\sigma}^L(\Gamma) \vdash \lambda x : \overline{\sigma}^L(E) : \overline{\sigma}^L(E') : \Pi x : \overline{\sigma}^L(E) : \overline{\sigma}^L(E'')$.

- 2.2 **Π-abstraction**

  Analogous to case 2.1.

- 2.3 **term application**

  We know that: $\Gamma \vdash_\Sigma E : \Pi x : A.B$, $\Gamma \vdash_\Sigma E' : A$. Assuming we have: $\bot \notin \overline{\sigma}^L(EE')$, $\bot \notin \overline{\sigma}^L(B[x/E'])$, we want to prove that $\overline{\sigma}^L(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(EE') : \overline{\sigma}^L(B[x/E'])$. From $\bot \notin \overline{\sigma}^L(EE')$, we have $\bot \notin \overline{\sigma}^L(E)$ and $\bot \notin \overline{\sigma}^L(E')$. To show that $\sigma(\Gamma) \vdash_\Sigma \sigma(E) : \sigma(\Pi x : A.B)$ and $\sigma(\Gamma) \vdash_\Sigma \sigma(E') : \sigma(A)$, in order to apply the induction hypothesis, we need to make sure that $\bot \notin \overline{\sigma}^L(E)$, $\bot \notin \overline{\sigma}^L(E')$, $\bot \notin \overline{\sigma}^L(A)$, $\bot \notin \overline{\sigma}^L(\Pi x : A.B)$. We have ensured the first two conditions previously, so only the latter remain to be proven. Since $\bot \notin \overline{\sigma}^L(x)$, $\bot \notin \overline{\sigma}^L(E)$, then $\bot \notin \overline{\sigma}^L(B)$. Hence, using this result, if we are able to show $\bot \notin \overline{\sigma}^L(A)$, we can conclude $\bot \notin \overline{\sigma}^L(\Pi x : A.B)$.

  - $\bot \notin \overline{\sigma}^L(A)$, then we can safely apply the induction hypothesis to obtain the desired result $\sigma(\Gamma) \vdash_{\Sigma'} \overline{\sigma}^L(EE') : \overline{\sigma}^L(B[x/E'])$.
  - $\bot \in \overline{\sigma}^L(A)$: Handled by the below **Theorem**.

- 2.4 **context**

  Given $\vdash_\Sigma \Gamma$, $\overline{\sigma}^L : \Sigma \to \Sigma'$, we have $\overline{\sigma}^L(\Gamma, x : E) = \overline{\sigma}^L(\Gamma), x : \overline{\sigma}^L(E)$. By induction hypothesis, $\overline{\sigma}^L$ is type preserving on lengths of derivation less or equal to k and we also have that $x : \overline{\sigma}^L(E)$ is equivalent to the base case 1.2, hence type-preserving, q.e.d.

Morphism Application with Normalization In order to define the morphism application with normalization, let us consider the following extended signature and morphism:

$\tilde{\Sigma'} = \Sigma',\ \bot_c : \sigma(A)$, for every $c : A$ in Σ, with hide c in σ, in the order of Σ

$\tilde{\sigma} : \Sigma \to \tilde{\Sigma'} = \sigma,\ c := \bot_c \backslash \{\text{hide } c\}$, if hide c in σ

Hence, $\tilde{\sigma}$ is defined to be the same as σ, but replacing the hide constructors with well-typed $\bot$ symbols, extending the target signature respectively.

Thus, the resulting morphism application is similar to that of the strict morphism application:

$\overline{\sigma}_{aux}(type) = type$

$\overline{\sigma}_{aux}(\lambda x : E.F) = \lambda x : \overline{\sigma}_{aux}(E).\overline{\sigma}_{aux}(F)$

$\overline{\sigma}_{aux}(\Pi x : E.F) = \Pi x : \overline{\sigma}_{aux}(E).\overline{\sigma}_{aux}(F)$

$\overline{\sigma}_{aux}(E\ F) = \overline{\sigma}_{aux}(E)\ \overline{\sigma}_{aux}(F)$

$\overline{\sigma}_{aux}(x : E) = x : \overline{\sigma}_{aux}(E)$

$$\overline{\sigma}_{aux}(c) = \begin{cases} E & \text{if } c := E \text{ in } \tilde{\sigma} \\ \bot_c & \text{if hide } c \text{ in } \tilde{\sigma} \end{cases}$$

$\overline{\sigma}_{aux}(.) = .$

$\overline{\sigma}_{aux}(\Gamma, x : E) = \overline{\sigma}(\Gamma), x : \overline{\sigma}(E).$

$$\overline{\sigma}(E) = \begin{cases} \overline{\sigma}_{aux}(E'E'')^{\beta\eta} & \text{if } E = E'E'', \bot \notin \overline{\sigma}_{aux}(E), \bot \in \overline{\sigma}_{aux}(A), E'' : A, \\ & \bot \notin \overline{\sigma}_{aux}(E)^{\beta\eta}, \bot \notin \overline{\sigma}_{aux}(F)^{\beta\eta}, E : F \\ \overline{\sigma}_{aux}(E) & \bot \notin \overline{\sigma}_{aux}(E), E : E' \\ \overline{\sigma}_{aux}(E)^{\beta\eta} & \text{otherwise }, \bot \notin \overline{\sigma}_{aux}(E)^{\beta\eta}, \bot \notin \overline{\sigma}_{aux}(F)^{\beta\eta}, E : F \\ \bot & \text{otherwise} \end{cases}$$

Note that we assign to $\bot$ the union of all defined typed $\bot$ symbols, i.e. $\bot = \bigcup_{T:\text{type}\in\Sigma'} \bot_T$

In the morphism application for expressions, we start with a particular rule, that for term application, motivated as a supplement to the case of the **Lemma** considering λ and Π abstraction with $\bot$ in the type of the bound variable.

The imposed conditions for the morphism are reasonable, because they combine the lemma conditions: $\bot \notin \overline{\sigma}_{aux}(E), \bot \in \overline{\sigma}_{aux}(A)$, where $E'' : A$ and the $\beta\eta$ normalization conditions: $\bot \notin \overline{\sigma}_{aux}(E)^{\beta\eta}, \bot \notin \overline{\sigma}_{aux}(F)^{\beta\eta}$, where $E : F$

Theorem. Given $\Gamma \vdash_\Sigma E : F$ and $\vdash \sigma : \Sigma \to \Sigma'$, such that $\overline{\sigma}(E) \neq \bot \neq \overline{\sigma}(F)$, the following hold:

- Type preservation: $\overline{\sigma}(\Gamma) \vdash_{\Sigma'} \overline{\sigma}(E) : \overline{\sigma}(F)$.
- Equality preservation: $\overline{\sigma}(\Gamma) \vdash_{\Sigma'} \overline{\sigma}(E) = \overline{\sigma}(F)$.

Proof: We prove type preservation. Equality preservation follows similarly.

Given $\vdash_\Sigma E : F$, such that $\overline{\sigma}(E) \neq \bot \neq \overline{\sigma}(F)$, we analyze the cases:

- If $\bot \notin \overline{\sigma}_{aux}(E)$ and $\bot \notin \overline{\sigma}_{aux}(F)$, then we have the following:
  - $E \neq E'E''$, then $\overline{\sigma}_{aux} = \overline{\sigma}^L_{aux}$ and according to the above proven lemma, we can conclude that: $\overline{\sigma}(\Gamma) \vdash_{\Sigma'} \overline{\sigma}(E) : \overline{\sigma}(F)$.

– $E = E'E''$, where $E' : \Pi x : A.B$, $E'' : A$.

If $\bot \notin \overline{\sigma}_{aux}(A)$, then the result holds by the above proven lemma.

Let us now consider the case that $\bot \in \overline{\sigma}_{aux}(A)$. We know from the premises that $\bot \notin \overline{\sigma}_{aux}(E)$, hence $\bot \notin \overline{\sigma}_{aux}(E'E'')$. According to the morphism application, $\overline{\sigma}_{aux}(E'E'') = \overline{\sigma}_{aux}(E')\overline{\sigma}_{aux}(E'')$, and, from the previous result, we get: $\bot \notin \overline{\sigma}_{aux}(E')$ and $\bot \notin \overline{\sigma}_{aux}(E'')$ (1). Since $E' : \Pi x : A.B$, $E'' : A$, then $E'E'' : B[x/E'']$. From this and the premises that $\bot \notin \overline{\sigma}_{aux}(F)$, we have that $\bot \notin \overline{\sigma}_{aux}(B[x/E''])$ (2). From (1), (2) and the fact that $\bot \notin \overline{\sigma}_{aux}(x) = x$, we get: $\bot \notin \overline{\sigma}_{aux}(B)$. From $\overline{\sigma}_{aux}(E') : \Pi x : \bot_A.\overline{\sigma}_{aux}(B)$ and $\overline{\sigma}_{aux}(E'') : \bot_A$, by β-reduction, we have that: $\overline{\sigma}_{aux}(E'E'')^{\beta} = (\overline{\sigma}_{aux}(E')\overline{\sigma}_{aux}(E''))^{\beta} : \overline{\sigma}_{aux}(B)[x/\overline{\sigma}_{aux}(E'')]$.

By **LF**, we know $\vdash_{\tilde{\Sigma}'} \overline{\sigma}_{aux}(E'E'') : \overline{\sigma}_{aux}(B[x/E''])$, since $\tilde{\Sigma}'$ contains well-typed $\bot$ symbols guaranteeing the type preservation of the morphism translation. Since have shown above that $\bot$ does not appear neither in $\overline{\sigma}_{aux}(E'E'')$ nor in $\overline{\sigma}_{aux}(B[x/E''])$, we have that $\bot$ is not in the $\beta\eta$ normal form. Given that $\beta\eta$-reduction cannot introduce new base types, it follows that $\bot \notin \overline{\sigma}_{aux}(E'E'')^{\beta\eta}$. By the type-preservation property of the $\beta\eta$-normalization, we have that $\vdash_{\tilde{\Sigma}'} \overline{\sigma}_{aux}(E'E'')^{\beta\eta} : \overline{\sigma}_{aux}(B[x/E''])^{\beta\eta}$. From this we get, by *weakening*, that $\vdash_{\Sigma'} \overline{\sigma}_{aux}(E'E'') : \overline{\sigma}_{aux}(B[x/E''])$ and from the definition of $\overline{\sigma}$, we obtain that $\vdash_{\Sigma'} \overline{\sigma}(E'E'') : \overline{\sigma}(B[x/E''])$ and can conclude by *weakening* that $\overline{\sigma}(\Gamma) \vdash_{\Sigma'} \overline{\sigma}(E'E'') : \overline{\sigma}(B[x/E''])$.

Note that whenever the lemma refers to the theorem, the case to which it refers to terminates via $\beta\eta$ normal form. Inversely, whenever the theorem references the lemma, it refers to some of the recursively expandable cases that do not immediately refer back to the theorem. While a back reference is possible, when recursively tracing the induction hypothesis, it will be on a different expression and, as shown above, such reference would then terminate in the theorem.

- If $\bot \in$ in $\overline{\sigma}_{aux}(E)$ or $\bot \in \overline{\sigma}_{aux}(F)$, then by the definition of $\overline{\sigma}$ and the assumption, we have that there is no $\bot$ in $\overline{\sigma}_{aux}(E)^{\beta,\eta}$ or $\overline{\sigma}_{aux}(F)^{\beta,\eta}$, by an analogous argument to the one above.

5 Discussion and Future Work

In the following, we present a set of benchmarks, which reveal desirable properties of the extension of **LF** with the hiding operation. From them, we have identified three desirable language features. The first, as seen in the translation from the typed to the untyped lambda calculus, is **complete hiding**, which stands for eliminating unnecessary information, i.e removing expressions. Next, we will show instances, such as in signature definitions and the first order axiomatization of the naturals, in which **constraint hiding** would be useful,

allowing to keep certain structures, by undefining their constraints. Finally, the encoding of the Curry-Howard isomorphism reveals a situation in which **contextual hiding**, enabling the contextual uncovering of a hidden term, is also necessary.

Complete hiding can be realized by hiding terms with hidden sub-constructs, unless the latter are $\beta\eta$-reducible. Constraint hiding can be obtained by removing the definition of constructs, when they depend on hidden theories, but not on their type. While these two cases are generally solvable, the last is challenging, since the context is determined by non-trivial patter-matching, which is difficult to implement, without relying too much on the $\bot$ symbols.

However these distinct features rarely occur isolated, as we also discovered in our target examples. We will begin by exemplifying such interaction in the context of the natural numbers.

5.1 Set to Type Theory

A revealing example demonstrating the power and utility of complete hiding is the morphism between set-theoretical and type-theoretical natural numbers. Let us consider a definitional introduction of the natural numbers in a set theory S, represented as the **LF** signature `SNat` below. We define zero as the empty set and the successor function as the one uniting the current set with the set containing it, i.e $0 : \texttt{set} = \emptyset$. From these we can derive the traditional Peano axiomatization of the natural numbers as Peano theorems and use them in the future. The analogous construction of the type theoretical natural numbers, also represented by an **LF** signature, namely `TNat`, is given axiomatically, by introducing two undefined constants, `0` and `succ`, and the Peano axioms.

%sig SNat = {
 set : type . $\emptyset$: set . 0 : set = $\emptyset$.
 $\cup$: set $\to$ set $\to$ set . {} : set $\to$ set .
 succ : set $\to$ set = $\lambda n.n \cup \{n\}$.
 Peano Theorems .
}

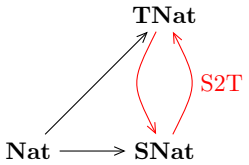
%sig TNat = {
 TTNat : type . 0 : TTNat .
 succ : TTNat $\to$ TTNat .
 Peano Axioms .
}

A translation from the type-theoretical naturals `TNat` to the set-theoretical naturals `SNat` can be given as an **LF** signature morphism, but we need hiding to give a translation from `SNat` to `TNat`. When mapping `SNat` to `TNat`, we hide all the set theoretical symbols, i.e $\sigma(\text{set}) = \sigma(\emptyset) = \sigma(\cup) = \bot$, and assign to the arithmetical symbols `0` and `succ` their syntactic equivalent in type theory. **LF** then automatically hides the set theoretical definitions of terms in `SNat` based on the hide declarations. It is important to observe that such an induced invocation of constraint hiding, done automatically by **LF** is a powerful consequence of the *defmap* judgement, which allows us to seamlessly integrate complete and constraint hiding in some cases. This directly relates the two theories effectively turning the Peano theorems of `SNat` into the Peano axioms of `TNat`. Thus, we manage to hide the low-level symbols, but recover the high-level ones that build

on top of them. Note that the types and definitions of the latter depend on the former. Of course, this idea breaks a number of natural assumptions about morphisms, for example, σ does not preserve equality anymore: We have $0 = \emptyset$ in S, but $\sigma(\emptyset) = \bot$ and $\sigma(0) = 0$. But this is exactly what we set out for: 0 can be interpreted in T, but $\emptyset$ cannot.

Once such a morphism σ is defined, the homomorphic extension of σ can yield for a theorem $p : F$ of S a T-theorem $\sigma(p) : \sigma(F)$. This works if σ is defined for all constants in p and F. Firstly, $\sigma(F)$ is undefined if the formula F cannot be expressed in S, e.g., if it is the S-theorem $0 \in 1$; but then we are typically not interested in translating it to T. Secondly, $\sigma(p)$ is undefined, if the proof p refers to axioms or rules of S for which σ is undefined; since σ is defined for all Peano theorems, it is defined for all proofs that refer only to them.

```
%view S2T : SNat → TNat = {
  hide set . hide ∅ . hide ∪ . hide {} .
  0 := TNat.0 .
  succ : TNat.succ .
  Peano Theorems := TNat.Peano Axioms .
} .
```



5.2 Typed to Untyped Lambda Calculus

Our next example is given by the translation from typed lambda calculus to untyped lambda calculus. Typed lambda calculus has a type `tp`, for base types, a function type constructor `fun`, a dependent type `tm`, for terms of a certain type, as well as the `app` and `lam` constructors, representing term application and lambda abstraction. The typed term application takes a term of function type A to B, a term of type A and applies them, producing a term of type B. Lambda abstraction can be considered an inverse of the term application operation, taking a term of A going to a term of B in **LF** and returning a typed lambda calculus term of the function type A to B. The untyped lambda calculus has no types, hence there are no typing constructors. However, it still has the term, application and lambda abstraction constructors, but defined in an untyped way. Thus, `tm` becomes atomic, instead of depending on a type and, respectively, `app` and `lam` only depend on `tm` and are untyped. When translating from the typed to the untyped lambda calculus, one has to hide the `tp` and `fun` constructors, as well as the type dependence of `tm`. The term translation is realized via hiding the type argument of the type family in typed lambda calculus, which makes it equivalent to the untyped one, i.e a term of type `x`, `tm x`, becomes a generic term `tm`, as x is completely hidden. The translation of the application and lambda abstraction constructors is straightforward once we have translation of terms, since the typed and untyped definitions are the same modulo types.

Currently, we have allowed for hiding `tp` and `fun` and require an extension to the judgement system that permits the explicit hiding of lambda bound vari-

ables. While the general case is not supported, any instance of a type `A : tp` will be automatically resolved by **LF**, when in a term application with `tm`, namely: `tm A : type := tm : type`.

```
Typed λ-calculus:
tp  : type.                    %% represent types by tp
fun : tp → tp → tp.            %% function type constructor
tm  : tp → type.               %% term constructor
app : tm (fun A B) → tm A → tm B.   %% term application
lam : (tm A → tm B) → tm (fun A B). %% lambda abstraction

Untyped λ-calculus:
tm  : type.              %% No λ-types, so Twelf primitive.
app : tm → tm → tm.                 %% term application
lam : (tm → tm) → tm.               %% lambda abstraction

Possible translation:
hide tp.                 %% Hide base types
hide fun.                 %% Hide function types
tm  := [hide a] tm.          %% Effect: tm a → tm.
app := [f:tm][a:tm] app f a.
lam := [f:tm → tm] lam f.
```

5.3 Curry-Howard translation from Cat to Prop

Our last example is given by the representation of the categorical formulation of the Curry-Howard isomorphism, which interprets formulas as objects and proofs as morphisms. We encode the category `Cat` as a signature having, as seen in the first example, a base type constructor `tp` and a type-dependent term constructor `tm`. We represent the the categorical concept of an object as a base type, using the `tp` constructor. Next, the morphism constructor is encoded as taking one object term (its domain, `tm obj`), another object term (its codomain) and returning the categorical type of this class of morphisms. As in every category, we have a class of identity morphisms, consisting of morphism terms going from an object to itself and the `comp` morphism composition law, which, taking a morphism term from A to B and a morphism term from B to C, returns a morphism term from A to C. The signature of propositional logic has only two relevant constructs, namely the type of formulas `o` and the constructor `ded` of deductions/proofs. When translating from `Cat` to `PL`, one has to hide the constructs which do not have a counterpart of equivalent granularity in `PL`, according to the Curry-Howard isomorphism, namely: `tm`, `tp`, `obj` and `mor`, which are too fine granular. However, according to the Curry-Howard isomorphism, an object term `(tm obj)` is equivalent to a formula via the translation and, even though `tm` and `obj` are hidden, their application needs to be visible. Analogously, this is also the case when one views morphism as proofs, which requires the application of `tm` and `mor`, `(tm mor)` to be visible.

For this case we have the hiding mechanism, but we still need to extend the judgement system to the case that maps expressions o expressions, as opposed to mapping constants to expressions. In general, we can observe that this problem can be solved by manually adding to the signature the constants whose definitions we want to map. Then, an explicit mapping of those new constants

in the morphism is sufficient. Since this procedure of introducing constant abbreviations for arbitrary expressions is generic and at the heart of LF, we can claim that mapping from expressions to expressions has the same properties as the mapping from constants to expressions. Also, we can distinguish this situation where we need to manually reveal expressions consisting of hidden terms, from that of automatically revealing expressions consisting of terms whose types are (partially) hidden. For example, hiding the type `tp` still allows for properly mapping `(tm obj)` and `(tm mor)` to `type` in PL.

```
%sig Cat = {
  %include DFOL       %% Dependently-Typed First Order Logic
  tp   :type.         %% Make DFOL sorts Twelf primitives
  tm   :tp→type.      %% Sort-dependent term construction
  obj  :tp.                                   %% Objects
  mor  :tm obj→ tm obj→ tp.                   %% Morphisms
  id   :tm (mor A A).                         %% Identity
  comp:tm (mor B C)→ tm (mor A B)→ tm (mor A C) %% Composition

 %sig PL = {
  o   :type.                          %% Formulas
  ded:o → type.       %% Proofs
 }.
```

5.4 Definition Hiding

Another example of hiding can be given with respect to signature definitions in Twelf. Recall the example given in Section 4. Let us consider a signature S, which contains two types **a** and **b** and also restricts **b** definitionally as equal to **a**, i.e **b** can be though of as a "synonym" of **a**. Taking T to be another signature with a structure **s** of type S, we can exhibit hiding by using an %undefine assignment to mask the symbolic link between **a** and **b** in signature S. This is useful when one explicitly wants to prohibit or shadow the inheritance of certain constructs in object-oriented programming.

```
Normal:
%sig S = {a:type. b:type = a.}. %% Symbolically equal elements
%sig T = {%struct s:S = {}.}.   %% Include in envelope sig.

Elaborates to:
%sig T = {s.a:type. s.b:type = s.a.}. %% Expand to primitives


Hiding a definition:
%sig S = {a:type. b:type = a.}. %% S as before
%sig T = {%struct s:S = {%undefine b.}.}. %% Mask equality

Elaborates to:
%sig T = {s.a:type. s.b:type.}. %% Expand to primitives
```

In this particular example, both `a` and `b` are constants, so the effect of undefining `b` can be achieved only via specifying `hide a` in the import. However, if one wants to preserve both `a` and `b` without their definitions, one would need to introduce new machinery, e.g an %undefine constant. While LF will automatically hide definitions that are partially or completely hidden, i.e have $\perp$ somewhere in their mapping, it is powerless in this case.

Let us look at a similar example, in a mathematical setting, by considering the natural numbers signature `HNat` given below. As we have seen in the previous section, in order to fully axiomatize the natural numbers, we need, apart from other axioms, which we omit, the second order induction axiom, `ind`. Next, let us take a proof of some formula F, which uses the `ind` axiom in its derivation and name it `t`. One way of defining the first order natural numbers, given by the `Nat1` signature, would be completely hiding the `ind` axiom, as seen in the first example, while an alternative approach, given by the `Nat2` signature, would be hiding `ind` and undefining `t`, i.e removing its proof derivation, as seen in the previous example. As a result, we get two possible elaborations. In `Nat1`, hiding `ind`, consequently leads to hiding `t`, since the latter builds on the axiom. Conversely, in `Nat2`, `t` elaborates to an axiom, since once its derivation is undefined, it no longer depends on the axiom `ind`.

```
Defining the naturals HNat and the first order naturals Nat1 and Nat2:

%sig HNat = {
              ind:proof (...).
                 t:proof F = (...ind...).
            }.
%sig FNat1 = {%struct n:HNat = {%hide ind}.}.
%sig FNat2 = {%struct n:HNat = {%hide ind.
                                %undefine t.
                               }.
            }.

Elaborates to:

%sig FNat1 = { }.          %% When F is arbitrary
%sig FNat2 = {t:proof F}.  %% When F is FOL-expressible
```

Hence, although complete hiding is not too restrictive when we are dealing with arbitrary formulas, we see that, in the case that such a formula F is FOL-expressible, the second approach, that of constraint hiding allows us to still be able to keep the result without its proof. In general, while constraint hiding, would help increase the system's flexibility, one might argue on the one hand that hiding a theorem's proof renders the result useless, since it cannot be applied or verified and hence, complete hiding is more adequate. On the other hand, in some cases one is not interested in the details of a theorem's proof, but only in the fact that it is indeed provable and hence, turning such a theorem into an axiom is a viable solution.

If we were to use the approach of an external morphism between HNat and FNat, we would need to manually assign any result of interest in HNat to a predetermined result of interest in FNat, losing any and all benefits in the approach. That is why we do a structural import of HNat into FNat, moving the burden of determining which and in what way the HNat terms should be hidden. By specifying $\forall^2$ and $\exists^2$ as hidden, **LF**, via its judgement system, can automatically filter out any nonexpressible fragments, merging the rest with FNat. For this to work, there is an essential prerequisite that the well-defined fragments of the structural import must clearly be a subset of the importing theory.

6 Conclusion

In the paper we have proposed a solution for the introduction of the hiding structuring operation in the modular system for the Edinburgh Logical Framework **LF**, operation which enables removing or ignoring parts of a theory. To this extent, after introducing the key concepts the paper builds on and giving an overview of the field, we proceeded to investigate the challenges faced when trying to solve the problem. We showed that it is necessary to develop a new language and a new approach, as the existing ones are not applicable in our case. In order to be able to understand what properties the extended language should have, we presented a series of benchmarks: the translation from typed to untyped lambda calculus, the encoding of the natural numbers in the First Order Logic framework CASL and the encoding of the categorical Curry-Howard isomorphism.

We then identified three desirable features, namely **complete hiding**, which eliminates all instances that contain the hidden construct, **constraint hiding**, which keeps structures, but undefines their definitions, transforming theorems into axioms, and **contextual hiding**, which enables the uncovering of hidden terms, depending on the context.

We have introduced and proved the expected behavior of complete hiding, which also granted a related subset of constraint hiding. Our inclusion of the $\beta\eta$ normalizations in the type preservation proof, also grants us a subset of contextual hiding, in the case in which one hides an entire type.

However, the problem of fully expressing and introducing constraint and contextual hiding is largely open and seemingly disjoint from the complete hiding machinery. Since we already motivated the need of fully expressing all three features, we can decisively conclude that this is an important direction of future work. It remains to be decided what the extent of any change to the signatures should be, particularly in the case of the $\perp$ constant artifacts.

This thesis lays the foundations for solving the hiding problem in **LF**, setting a clear cut frame and providing the basis for the remaining extensions. We open the doors to future applications that were previously impossible due to their dependence on **partial morphisms**. The key advantage of our approach is that we manage to introduce partial views that are preserving typing and, hence, can constitute the basis of trusted **partial translations**.

This is particularly relevant, since we are successfully employing the **LF** module system to give an atlas of formal languages, such as logics, type theories and mathematical foundations, as well as the translations between them, in the LATIN project. One intended application of the LATIN multi graph of formal languages is to aid system integration in a verified setting, by explicating the formal languages and the relations underlying the used reasoning systems. We are thus anticipating the solution to various such system integration problems, e.g translating from Mizar to Isabelle and back, typed to untyped translations, as well as higher to first order borrowing of theorems in formal specifications.

Acknowledgements

The author is very grateful to Dr. Florian Rabe and Prof. Dr. Michael Kohlhase for their continuous collaboration, support and guidance throughout the course of this thesis.

References

[AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.

[BT95] J. Bergstra and J. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *J. ACM*, 42(6):1194–1230, 1995.

[Bur97] R. Burkhardt. *UML-Unified Modeling Language*. Addison Wesley, 1997.

[CoF04] CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer, 2004.

[Daw91] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[dB70] N. de Bruijn. The mathematical language AUTOMATH. volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, Berlin, 1970.

[DFI+] R. Diaconescu, K. Futatsugi, M. Ishisone, A. Nakagawa, and T. Sawada. An overview of CafeOBJ.

[GB92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[GR01] J. Goguen and G. Rosu. Institution morphisms. Formal aspects of computing, to appear, 2001.

[GWM+93] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993. also to appear as Technical Report from SRI International.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM: Journal of the ACM*, 40, 1993.

[HR09] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In *Fourth Workshop on Logical and Semantic Frameworks, with Applications*, volume 256 of *Electronic Notes in Theoretical Computer Science*, pages 49–65, 2009.

[KB95] I. Kraan and P. Baumann. Logical frameworks as a basis for verification tools: A case study. In *KBSE*, pages 36–43, 1995.

[MAH06] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1–2):114–145, 2006.

[Maj77] M. Majster. Limits on the "algebraic" specification of abstract data types. *SIGPLAN Notices*, 12(10):37–41, October 1977.

[Mes89] J. Meseguer. General logics. In *Proc. Logic Colloquium '87*. North Holland, 1989.

[MG85] J. Meseguer and J. Goguen. Initiality, induction and computability. In Nivat and Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[Pau94] L. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Ros04] G. Rosu. Behavioral abstraction is hiding information. *Theor. Comput. Sci*, 327(1-2):197–221, 2004.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In *Proceedings of the Workshop on Logical Frameworks Meta-Theory and Practice (LFMTP)*, 2009.

[SKS02] Graeme Smith, Florian Kammüller, and Thomas Santen. Encoding Object-Z in Isabelle/HOL. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 2002.

[Spi88] J.M Spivey. *Introducing Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[WK99] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.