IJCAR 2004

Second International Joint Conference on Automated Reasoning

University College Cork, Cork, Ireland

Workshop Programme

Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)

Carsten Schürmann (Chair)

WS 4 – July 5

Preface

These are the Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004). The workshop was held in Cork, Ireland on July 5, 2004 in conjunction with IJCAR 2004.

Logical frameworks and meta-languages form a common substrate for representing, implementing, and reasoning about a wide variety of deductive systems of interest in logic and computer science. Their design and implementation has been the focus of considerable research over the last two decades, using competing and sometimes incompatible basic principles. This workshop brings together designers, implementors, and practitioners to discuss all aspects of logical frameworks.

The papers in this workshop proceedings were selected but not formally refereed by the following program committee.

Thierry Coquand	Chalmers University
Amy Felty	University of Ottawa
Christoph Kreitz	University of Potsdam
Jose Meseguer	University of Illinois
Dale Miller	INRIA
Frank Pfenning	Carnegie Mellon University
Randy Pollack	Edinburgh University
Carsten Schürmann (Chair)	Yale University

A formally referreed selection of the presented papers will be published in Elsevier's ENTCS Series (Electronic Notes in Theoretical Computer Science).

Carsten Schürmann
New Haven, June 2004

Table of Contents

Weak Normalization for the Simply-Typed Lambda-Calculus in Twelf

Andreas Abel [1]

Department of Computer Science, Chalmers University of Technology
Rännvägen 6, SWE-41296 Göteborg, Sweden

Abstract

Weak normalization for the simply-typed λ-calculus is proven in Twelf, an implementation of the Edinburgh Logical Framework. Since due to proof-theoretical restrictions Twelf Tait's computability method does not seem to be directly usable, a combinatorical proof is adapted and formalized instead.

Key words: Edinburgh Logical Framework, HOAS, Mechanized Proof, Normalization, Twelf

1 Introduction

Twelf is an implementation of the Edinburgh Logical Framework which supports reasoning in full higher-order abstract syntax (HOAS); therefore it is an ideal candidate for reasoning comfortably about properties of prototypical programming languages with binding. Previous work has focused on properties like subject reduction, confluence, compiler correctness. Even cut elimination for various sequent calculi has been proven successfully. But until recently, there were no formalized proofs of normalization [2] in Twelf. The reason might be that normalization is typically proven by Tait's method, which cannot be applied directly in Twelf. This work explains why Tait's method is at least not directly applicable and provides a combinatorical proof for the simply-typed lambda-calculus.

[1] Research supported by the *Graduiertenkolleg Logik in der Informatik* of the Deutsche Forschungsgemeinschaft, the thematic networks *TYPES* (IST-1999-29001) and *Applied Semantics II* (IST-2001-38957) of the European Union and the project *CoVer* of the Swedish Foundation of Strategic Research.

[2] There have been normalization proofs in logical frameworks with inductive definitions, for instance, Altenkirch's proof of strong normalization for System F in LEGO [2]. Since HOAS is not available in a framework like LEGO, he represents terms using de Bruijn indices.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

$$
\begin{array}{lll}
\mathbb{K} & ::= \mathsf{type} & \text{kind of types} \\
& \mid \{\mathbb{X}{:}\mathbb{A}\}\mathbb{K} & \text{dependent function kind} \\
\mathbb{A} & ::= \mathbb{F}\ \mathbb{M}_1 \ldots \mathbb{M}_n & \text{base type} \\
& \mid \{\mathbb{X}{:}\mathbb{A}\}\mathbb{A} & \text{dependent function type} \\
& \mid \mathbb{A} \to \mathbb{A} & \text{non-dependent function type} \\
\mathbb{M} & ::= \mathbb{C} & \text{term constant} \\
& \mid \mathbb{X} & \text{term variable} \\
& \mid [\mathbb{X}{:}\mathbb{A}]\mathbb{M} & \text{term abstraction} \\
& \mid \mathbb{M}\,\mathbb{M} & \text{term application}
\end{array}
$$

Fig. 1. Syntactic classes of LF.

2 Twelf

The Edinburgh Logical Framework (LF [3]) [6,7] is a dependently-typed lambda-calculus with type families and $\beta\eta$-equality, but neither polymorphism, inductive data types nor recursion. Expressions are divided into three syntactic classes: kinds, types and terms, generated by the grammar in Fig. 1. Herein, the meta variable $\mathbb{X}$ ranges over a countably infinte set of variable identifiers, while $\mathbb{F}$ resp. $\mathbb{C}$ range over type-family resp. term constants provided in a signature Σ. Note that neither a type nor a kind can depend on a type; consequently, abstraction is missing on the type level [10, p. 1124].

Syntax.

$$
\begin{array}{llll}
r, s, t, u & ::= & x \mid \lambda x.t \mid r\ s & \text{untyped terms} \\
A, B, C & ::= & * \mid A \to B & \text{simple types} \\
\Gamma & ::= & \diamond \mid \Gamma,\, x{:}A & \text{contexts}
\end{array}
$$

Type assignment $\Gamma \vdash t : A$.

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash x : A}\ \texttt{of\_var}$$

$$\frac{\Gamma,\, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ \texttt{of\_lam} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}\ \texttt{of\_app}$$

Weak head reduction $t \longrightarrow_{\mathsf{w}} t'$.

$$\frac{}{(\lambda x.t)\ s \longrightarrow_{\mathsf{w}} [s/x]t}\ \texttt{beta} \qquad \frac{r \longrightarrow_{\mathsf{w}} r'}{r\ s \longrightarrow_{\mathsf{w}} r'\ s}\ \texttt{appl}$$

Fig. 2. Simply-typed λ-calculus and weak head reduction.

The framework comes with judgements for typing, $\mathbb{M} : \mathbb{A}$, kinding, $\mathbb{A} : \mathbb{K}$,

[3] This is not to be confused with Martin-Löf's framework for dependent type theory, which is also abbreviated by LF.

and wellformedness of kinds, $\mathbb{K}$ *kind*, plus $\beta\eta$-equality on for terms, types, and kinds [7]. An object theory can be described in the framework by providing a suitable signature Σ which adds kinded type family constants $\mathbb{F} : \mathbb{K}$ and typed term constants $\mathbb{C} : \mathbb{A}$.

Twelf [11] is an implementation of LF whose most fundamental task is to check typing (and kinding) of a user given signature Σ, usually provided as a set of ASCII files. Symbols reserved for the framework are the following.

```
: . ( ) [ ] { } -> type
```

All others can be used to denote entities in the object theories. In the remainder of this section, we show how to represent the simply-typed λ-calculus with weak head reduction, as specified in Fig. 2, in Twelf.

2.1 Representation of Syntactic Objects

Untyped lambda terms t can be represented by one type family constant `tm` and two term constants:

```
tm     : type.
lam    : (tm -> tm) -> tm.
app    : tm -> tm -> tm.
```

The lack of a construct for variables is due to the use of HOAS: object variables are represented by variables of the framework, e. g., in the code for the twice function:

```
twice = lam [f:tm] lam [x:tm] app f (app f x).
```

A more detailed explanation of higher-order encodings has been given by Schürmann [14, p. 20ff]. Simple types A can be generated from a nullary constant `*` for some base type and a binary constant `=>`, used infix, for function type formation.

```
ty     : type.
*      : ty.
=>     : ty -> ty -> ty.
```

2.2 Representation of Judgements and Relations

Type assignment for untyped terms, $\Gamma \vdash t : A$, can be represented by two constants as well: one for function introduction and one for function elimination. Note that in Twelf syntax, the types of new constants may contain free variables (captial letters), which are regarded as universally quantified on the outside.

```
of     : tm -> ty -> type.
of_lam : ({x:tm} x of A -> (T x) of B)
           -> (lam [x:tm] T x) of (A => B).
of_app : R of (A => B) -> S of A -> (app R S) of B.
```

Again, there is no separate rule for the typing of variables, instead it is part of the rule for abstraction. The premise of rule `of_lam` is to be read as:

> Consider a temporary extension of the signature by a fresh constant `x:tm` and assume `x of A`. Then `(T x) of B` holds.

This adds a *dynamical* typing rule `x of A` for each new variable `x` instead of inserting a typing hypothesis $x : A$ into the typing context Γ. Hence, we do not explicitely encode Γ, but let the framework handle the typing hypotheses.

Similar to the typing relation, we can represent weak head reduction $t \longrightarrow_{\mathsf{w}} t'$, which eliminates the head (resp. key) redex in term t but does not step under a binding.

```
-->w  : tm -> tm -> type.
beta  : app (lam T) S -->w T S.
appl  : R -->w R' -> app R S -->w app R' S.
```

One advantage of HOAS is that substitution does not have to be defined, but can be inherited from the framework. Since in rule `beta`, term `T : tm -> tm` is λ-function, substitution $[u/y]t$ is simply expressed as application `T U`.

Lemma 2.1 *If* $t \longrightarrow_{\mathsf{w}} t'$ *then* $[u/y]t \longrightarrow_{\mathsf{w}} [u/y]t'$.

Proof. By induction on the derivation of $t \longrightarrow_{\mathsf{w}} t'$.

- Case $(\lambda x.t)\, s \longrightarrow_{\mathsf{w}} [s/x]t$. W.l.o.g. $x \neq y$ and x not free in u. Then,

$$\begin{array}{rcl} [u/y]((\lambda x.t)\, s) & = & (\lambda x.[u/y]t)\ [u/y]s \\ & \longrightarrow_{\mathsf{w}} & [[u/y]s/x][u/y]t \ = [u/y][s/x]t. \end{array}$$

- Case $r\, s \longrightarrow_{\mathsf{w}} r'\, s$ with $r \longrightarrow_{\mathsf{w}} r'$. By ind. hyp., $[u/y]r \longrightarrow_{\mathsf{w}} [u/y]r'$. Hence,

$$\begin{array}{rcl} [u/y](r\, s) & = & ([u/y]r)\ ([u/y]s) \\ & \longrightarrow_{\mathsf{w}} & ([u/y]r')\ ([u/y]s) = [u/y](r'\, s) \end{array}$$

□

Fig. 3. Weak head reduction is closed under substitution.

2.3 Representation of Theorems and Proofs

Fig. 3 shows the first lemma of our object theory. How do we represent it? Twelf's internal logic is constructive, therefore the lemma must be interpreted constructively: Given a derivation $\mathcal{P}$ of $t \longrightarrow_{\mathsf{w}} t'$ and a term u, we can construct a derivation $\mathcal{P}'$ of $[u/y]t \longrightarrow_{\mathsf{w}} [u/y]t'$. In type theories with inductive types and recursion, like Agda, Coq [8] and LEGO [13], the lemma would be

represented as a recursive function of the dependent type

$$\Pi t : tm.\, \Pi t' : tm.\, \Pi \mathcal{P} : t \longrightarrow_{\mathtt{w}} t'.\, \Pi u : tm.\, \Pi y : var.\, [u/y]t \longrightarrow_{\mathtt{w}} [u/y]t'.$$

In Twelf, however, with no recursive functions at hand, the lemma is represented as a relation between input and output derivations, and, thus, via the propositions-as-types paradigm, as just another type family.

```
subst_red : {U:tm} ({y:tm} T y -->w T' y)
                          -> T U -->w T' U -> type.
%mode subst_red +U +P -P'.
```

The `%mode` statement marks the first two arguments of type family `subst_red` as inputs (+) and the third as output (−). Thus, the lemma is a functional relation, and its proof is a logic program with two clauses, one for each case in the proof.

```
subst_red_beta: subst_red U ([y] beta) beta.
subst_red_appl: subst_red U ([y] appl (P y)) (appl P')
                <- subst_red U P P'.
%terminates P (subst_red _ P _).
```

The base case of the induction is given by the constant `subst_red_beta`, and the step case, which appeals to the induction hypothesis, by `subst_red_appl`. The *types* of these constants are the actual program and correspond to PROLOG clauses. Note that in the second type a reversed arrow "`<-`", which resembles PROLOG's "`:-`", has been used to encourage an operational reading:

> Substitution in a derivation whose last rule is `appl`, and the remainder, `P`, may mention `y`, results in a derivation `P'` extended by an application of rule `appl`. Herein, `P'` is constructed from `P` recursively.

Since it is a logic program, we can even "execute" the lemma. Execution in Twelf is search: Given a type with free variables, find an inhabitant of the type and solutions for the free variables. For example:

```
P : {y} app (app (lam [x] x) y) y -->w app y y
  = [y] appl beta.
%define P' = X
%solve K : subst_red (lam [z] z) P X.
```

This defines a 2-rule derivation `P` which witnesses that $(\lambda x.x)\, y\, y \longrightarrow_{\mathtt{w}} y\, y$. The `%solve` statement asks Twelf for a derivation `P'` which arises from `P` by substituting $\lambda z.z$ for y, according to the lemma. The answer is:

```
P' : app (app (lam [x] x) (lam [z] z)) (lam [z] z)
       -->w app (lam [z] z) (lam [z] z)
   = appl beta.
K  : subst_red (lam [z] z) P (appl beta)
   = subst_red_appl subst_red_beta.
```

Since the value of `P'` equals `P`, the shape of the derivation has not changed, only its result: the type of `P'`. The value of `K` gives an execution trace of logic program `subst_red`: First, clause `subst_red_appl` has fired, then clause `subst_red_beta` has concluded the search.

2.4 External Properties: Termination and Coverage

A logic program in Twelf corresponds to a partial function from inputs to outputs as specified by the mode declaration. Since only total functions correspond to valid inductive proofs we must ensure that the defined function *terminates* on all inputs and *covers* all possible cases. Both properties cannot be shown within the framework, e. g., we cannot give a proof that `subst_red` is terminating. Instead, totality of a function needs external reasoning and can be ensured by built-in tactics.

Brigitte Pientka [12] contributed a termination checker which is invoked by the `%terminates` pragma. In our case, the second argument `P` decreases structurally in each recursive call. Case coverage is ensured by an algorithm by Pfenning and Schürmann [15]. Both termination and coverage checking are necessarily undecidable. For the proof developed in the remainder of this article, we found the implemented termination checker powerful enough to pass our code, whereas the coverage checker could not "see" that indeed all cases are handled. Thus, coverage had to be established manually, but for lack of space we will not detail on it.

3 A Formalized Proof of Weak Normalization

In this section, we present a combinatorial proof of weak normalization for the simply-typed lambda-calculus. It is similar to the textbook proof in Girard, Lafont and Taylor [4, Ch. 4], but we avoid reasoning with numbers altogether. In fact, we follow closely the very syntactical presentation of Joachimski and Matthes [9], which has also been implemented in Isabelle/Isar by Nipkow and Berghofer [3]. The main obstacle to a direct formalization in Twelf is the use of a vector notation for terms by Joachimski and Matthes, which allows them to reason on a high level in some cases. In this section, we will see a "de-vectorized" version of their proof which can be outlined as follows:

(i) Define an inductive relation $t \Downarrow A$.

(ii) Prove that for every term $t : A$ the relation $t \Downarrow A$ holds.

(iii) Show that every term in the relation is weakly normalizing.

3.1 Inductive Characterization of Weak Normalization

Inductive characterizations of normalization go back to Goguen [5] and van Raamsdonk and Severi [16,17]. We introduce a relation $\Gamma \vdash t \Downarrow A$ which stipulates that t is weakly normalizing of type A, and an auxiliary relation

$\Gamma \vdash t \downarrow^x A$ which additionally claims that $t = x\,\boldsymbol{s}$ for some sequence of terms $\boldsymbol{s}$, i.e., t is neutral and head-redex free.

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash x \downarrow^x A} \qquad \frac{\Gamma \vdash r \downarrow^x A \to B \qquad \Gamma \vdash s \Downarrow A}{\Gamma \vdash r\,s \downarrow^x B}\ \texttt{wne\_app} \qquad \frac{\Gamma \vdash r \downarrow^x A}{\Gamma \vdash r \Downarrow A}\ \texttt{wn\_ne}$$

$$\frac{\Gamma, x{:}A \vdash t \Downarrow B}{\Gamma \vdash \lambda x.t \Downarrow A \to B}\ \texttt{wn\_lam} \qquad \frac{r \longrightarrow_{\mathsf{w}} r' \qquad \Gamma \vdash r' \Downarrow A}{\Gamma \vdash r \Downarrow A}\ \texttt{wn\_exp}$$

The Twelf representation is similar to the typing relation: Again, Γ and the hypothesis rule are indirectly represented in rule `wn_lam`.

```
wne     : tm -> ty -> tm -> type.
wn      : tm -> ty -> type.

wne_app : wne R (A => B) X -> wn S A -> wne (app R S) B X.
wn_ne   : {X:tm} wne R A X -> wn R A.
wn_lam  : ({x:tm} wne x A x -> wn (T x) B)
            -> wn (lam T) (A => B).
wn_exp  : R -->w R' -> wn R' A -> wn R A.
```

3.2 Closure under Application and Substitution

To show that each typed term $t : A$ is in the relation $t \Downarrow A$, we will proceed by induction on the typing derivation. Difficult is the case for an application of the form $(\lambda x.r)\,s$. It can only be shown to be in the relation by rule `wn_exp`, which requires us to prove that $[s/x]r$ is in the relation. If x is head variable of r, substitution might create new redexes. In this case, however, we can argue that the type of r is a smaller type than the one of s. These preliminary thoughts lead to the following lemma.

Lemma 3.1 (Application and Substitution) *Let $\mathcal{D} :: \Gamma \vdash s \Downarrow A$.*

(i) *If $\mathcal{E} :: \Gamma \vdash r \Downarrow A \to C$ then $\Gamma \vdash r\,s \Downarrow C$.*

(ii) *If $\mathcal{E} :: \Gamma, x{:}A \vdash t \Downarrow C$, then $\Gamma \vdash [s/x]t \Downarrow C$.*

(iii) *If $\mathcal{E} :: \Gamma, x{:}A \vdash t \downarrow^x C$, then $\Gamma \vdash [s/x]t \Downarrow C$ and C is a part of A.*

(iv) *If $\mathcal{E} :: \Gamma, x{:}A \vdash t \downarrow^y C$ with $x \neq y$, then $\Gamma \vdash [s/x]t \downarrow^y C$.*

In Twelf, the lemma is represented by four type families. The invariant that C is a subexpression of A will be expressed via a `%reduces` statement later, which makes is necessary to make type C an explicit argument to type family `subst_x`.

```
app_wn  : {A:ty} wn S A ->
                    wn R (A => C) -> wn (app R S) C -> type.

subst_wn: {A:ty} wn S A ->
 ({x:tm} wne x A x -> wn  (T x) C  ) -> wn  (T S) C   -> type.
```

```
subst_x : {A:ty} wn S A -> {C:ty}
 ({x:tm} wne x A x -> wne (T x) C x) -> wn  (T S) C   -> type.

subst_y : {A:ty} wn S A ->
 ({x:tm} wne x A x -> wne (T x) C Y) -> wne (T S) C Y -> type.

%mode app_wn   +A +D    +E -F.
%mode subst_wn +A +D    +E -F.
%mode subst_x  +A +D +C +E -F.
%mode subst_y  +A +D    +E -F.
```

Proof of Lemma 3.1 Simultaneously by main induction on type A and side induction on the derivation $\mathcal{E}$.

(i) Show $\Gamma \vdash r\, s \Downarrow C$. If the last rule of $\mathcal{E}$ was `wn_ne`, hence, r is neutral, then $r\, s$ is also neutral by rule `wne_app`, thus, it is in the relation. If the last rule was `wn_exp`, we can apply the side ind. hyp. The interesting case is $r = \lambda x.t$ and

$$\frac{\Gamma, x{:}A \vdash t \Downarrow C}{\Gamma \vdash \lambda x.t \Downarrow A \to C}\ \texttt{wn\_lam}.$$

Here, we proceed by side ind. hyp. ii.

```
app_wn_ne   : app_wn A D (wn_ne X E) (wn_ne X (wne_app E D)).
app_wn_exp  : app_wn A D (wn_exp P E) (wn_exp (appl P) F)
               <- app_wn A D E F.
app_wn_lam  : app_wn A D (wn_lam E) (wn_exp beta F)
               <- subst_wn A D E F.
```

(ii) Show $\Gamma \vdash [s/x]t \Downarrow C$ for $\Gamma, x{:}A \vdash t \Downarrow C$. If t is not neutral, we conclude by ind. hyp. and possibly Lemma 2.1. Otherwise, we distinguish on the head variable of t: is it x, then we proceed by side ind. hyp. iii, otherwise by side ind. hyp. iv.

```
subst_wn_lam: subst_wn A D
 ([x][dx] wn_lam ([y][dy] E y dy x dx)) (wn_lam F)
               <- {y}{dy} subst_wn A D (E y dy) (F y dy).

subst_wn_exp: subst_wn A (D : wn S A)
 ([x][dx] wn_exp (P x) (E x dx)) (wn_exp P' E')
               <- subst_wn A D E E'
               <- subst_red S P P'.

subst_wn_x  : subst_wn A D
 ([x][dx] (wn_ne x (E x dx) : wn (T x) C)) F
               <- subst_x A D C E F.

subst_wn_y  : subst_wn A D
```

```
  ([x][dx] wn_ne Y (E x dx)) (wn_ne Y F)
                <- subst_y A D E F.
```

(iii) Show $\Gamma \vdash [s/x]t \Downarrow C$ for $\Gamma' \vdash t \downarrow^x C$ with $\Gamma' := \Gamma, x : A$. In case $t = x$, the type C is trivially a part of $A = C$ and we conclude by assumption $\Gamma \vdash s \Downarrow C$. Otherwise, $t = r\,u$ and the last rule in $\mathcal{E}$ was

$$\frac{\Gamma' \vdash r \downarrow^x B \to C \qquad \Gamma' \vdash u \Downarrow B}{\Gamma' \vdash r\,u \downarrow^x C}\ \texttt{wne\_app}.$$

By side ind. hyp. iii we know that $B \to C$ is a part of A and $\Gamma \vdash r' \Downarrow B \to C$ where $r' := [s/x]r$. Similarly $\Gamma \vdash u' \Downarrow B$ for $u' := [s/x]u$ by side ind. hyp. ii. Since B is a *strict* part of A, we can apply the main ind. hyp. i and obtain $\Gamma \vdash r'\,u' \Downarrow C$.

```
subst_x_x   : subst_x A D A ([x][dx]dx) D.
subst_x_app : subst_x A D C ([x][dx] wne_app
                                (E x dx)
                                (F x dx : wn (U x) B)) EF
                <- subst_x A D (B => C) E E'
                <- subst_wn A D F F'
                <- app_wn B F' E' EF.
%reduces C <= A (subst_x A D C E F).
```

The `%reduces` declaration states that the type expression C is a subexpression of A. Twelf checks that this invariant is preserved in all possibilities of introducing `subst_x A D C E F`. In case `subst_x_x` it holds because `C` is instantiated to `A`. In case `subst_x_app` it follows from the ind. hyp. which states that already `B => C` is a subexpression of `A`.

(iv) Show $\Gamma \vdash [s/x]t \downarrow^y C$ for $\Gamma, x : A \vdash t \downarrow^y C$. There a two cases. $t = y$, which holds immediately, and $t = r\,u$, which follows from side ind. hyp.s ii and iv. In our Twelf representation, we cannot distinguish variable y from any other term, so we widen the first case to cover all t such that x is not free in t. This is expressed by letting `E` not refer to `x` or `dx`.

```
subst_y_y   : subst_y A D ([x][dx] E) E.
subst_y_app : subst_y A D ([x][dx] wne_app (E x dx) (F x dx))
                                   (wne_app E' F')
                <- subst_y A D E E'
                <- subst_wn A D F F'.
```

□

To justify the appeals to the ind. hyp.s we invoke the Twelf termination checker with the following termination order.

```
%terminates {(Ax Ay As Aa) (Ex Ey Es Ea)}
  (subst_x  Ax _ _ Ex _)
  (subst_y  Ay _   Ey _)
  (subst_wn As _   Es _)
```

```
(app_wn   Aa _   Ea _).
```

It expresses that the four type families are mutually recursive and terminate w. r. t. the lexicographic order on pairs $(A, \mathcal{E})$ of types A and typing derivations $\mathcal{E}$. This corresponds on a main induction on A and a side induction on $\mathcal{E}$. To verify termination, Twelf makes use of the `%reduces` declaration.

3.3 Soundness of Inductive Characterization

To complete our proof of weak normalization, we need to show that for each term t in the relation $t \Downarrow A$ or $t \downarrow^x A$ there exists a normal form v such that $t \longrightarrow^* v$. After formulating full reduction $\longrightarrow$ with the usual closure properties, the proof is a simple induction on the derivation $\mathcal{E} :: t \Downarrow A$ resp. $\mathcal{E} :: t \downarrow^x A$. For lack of space we exclude the details, an implementation of the proof is available online [1].

4 On Proof-Theoretical Limitations of Twelf

Having successfully completed the proof of weak normalization we are interested whether it could be extended to strong normalization and stronger object theories, like Gödel's T. In this section, we touch these questions, but our answers are speculative and preliminary.

Joachimski and Matthes [9] extend their proof to Gödel's T, using the infinitary ω-rule to state when a recursive function over natural numbers is weakly normalizing. Their proof is not directly transferable since only finitary rules can be represented in Twelf.

For the same reason, Tait's proof cannot be formalized in Twelf. Its key part is the definition

$$\frac{\forall s.\ \ s \Downarrow A \ \Rightarrow\ r\, s \Downarrow B}{r \Downarrow A \to B}$$

with an infinitary premise. Its literal translation into Twelf

```
wn_arr : ({S:tm} wn S A -> wn (app R S) B) -> wn R (A => B)
```

means something else, namely "if for a fresh term `S` for which we assume `wn S A` it holds that `wn (app R S) B`, then `wn R (A => B)`". Translating this back into mathematical language, we obtain the rule

$$\frac{x \Downarrow A \ \Rightarrow\ r\, x \Downarrow B}{r \Downarrow A \to B} \text{ for a fresh variable } x.$$

Since variables x are weakly normalizing anyway, we can simplify the premise further to $r\, x \Downarrow B$, obtaining clearly something we did not want in the first place.

Due to the interpretation of quantification in Twelf, infinitary rules cannot be represented, which also obstructs the definition of the predicate *strongly*

normalizing sn by the inductive rule

$$\frac{\forall t'.\ t \longrightarrow t' \Rightarrow \mathsf{sn}\, t'}{\mathsf{sn}\, t},$$

expressing that the set of strongly normalizing terms is the accessible part of the reduction relation.

Concluding, one might say that normalization of Gödel's T and proofs of strong normalization are at least difficult to express in Twelf. To see whether they are feasible at all, a detailed proof-theoretic analysis of Twelf would be required.

5 Conclusion and Related Work

We have presented a formalization of Joachimski and Matthes' version of an elementary proof of weak normalization of the simply-typed λ-calculus in Twelf. We further have outlined some problems with direct proofs of strong normalization and Tait style proofs.

In the 1990s, Filinski has investigated feasibility of logical relation proofs in the Edinburgh LF, but his findings remained unpublished. According to Pfenning, a possible way is to first define a logic in LF, and then within this logic investigate normalization of λ-calculi. This path is taken in the Isabelle system whose framework is similar to LF but only simply-typed instead of dependently typed. On top of core Isabelle, higher-order logic (HOL) is implemented which serves as the meta language in which, in turn, object theories are considered. Rich tactics for HOL make up for the loss of framework mechanism due to the extra indirection level. In Twelf, one could follow this path as well, with the drawback that the built-in facilities like termination checker and automated prover [14] would be rendered inapplicable.

Independently of the author, Watkins and Crary have formalized a normalization algorithm and proof in Twelf, namely for Watkins' concurrent logical framework. It is said to follow the principle of our Lemma 3.1, namely showing that canonical forms (=normal forms) are closed under eliminations.

Acknowledgments.

The author likes to thank Ralph Matthes, Frank Pfenning, Brigitte Pientka, Carsten Schürmann and Kevin Watkins for discussions on the topic in the years 2000–2004. He is indebted to Thierry Coquand for comments on the draft of this paper.

References

[1] Abel, A., *A Twelf proof of weak normalization for the simply-typed λ-calculus*, Twelf code, available on the author's homepage (2004).

[2] Altenkirch, T., *A formalization of the strong normalization proof for System F in LEGO*, in: M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, TLCA'93*, Lecture Notes in Computer Science **664** (1993), pp. 13–28.

[3] Berghofer, S., "Proofs, Programs and Executable Specifications in Higher-Order Logic," Ph.D. thesis, Technische Universität München (2003).

[4] Girard, J.-Y., Y. Lafont and P. Taylor, "Proofs and Types," Cambridge Tracts in Theoretical Computer Science **7**, Cambridge University Press, 1989.

[5] Goguen, H., *Typed operational semantics*, in: M. Deziani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications (TLCA 1995), Proceedings*, Lecture Notes in Computer Science **902** (1995), pp. 186–200.

[6] Harper, R., F. Honsell and G. Plotkin, *A Framework for Defining Logics*, Journal of the Association of Computing Machinery **40** (1993), pp. 143–184.

[7] Harper, R. and F. Pfenning, *On equivalence and canonical forms in the LF type theory*, ACM Transactions on Computational Logic (2004), (To appear).

[8] INRIA, "The Coq Proof Assistant Reference Manual," Version 8.0 edition (2004), http://coq.inria.fr/doc/main.html.

[9] Joachimski, F. and R. Matthes, *Short proofs of normalization*, Archive of Mathematical Logic **42** (2003), pp. 59–87.

[10] Pfenning, F., *Logical frameworks*, , **2** (2001), pp. 1063–1147.

[11] Pfenning, F. and C. Schürmann, *System description: Twelf - a meta-logical framework for deductive systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence **1632** (1999), pp. 202–206.

[12] Pientka, B., *Termination and reduction checking for higher-order logic programs*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001*, Lecture Notes in Artificial Intelligence **2083** (2001), pp. 401–415.

[13] Pollack, R., "The Theory of LEGO," Ph.D. thesis, University of Edinburgh (1994).

[14] Schürmann, C., "Automating the Meta-Theory of Deductive Systems," Ph.D. thesis, Carnegie-Mellon University (2000).

[15] Schürmann, C. and F. Pfenning, *A coverage checking algorithm for LF*, in: D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, Lecture Notes in Computer Science **2758** (2003), pp. 120–135.

[16] van Raamsdonk, F. and P. Severi, *On normalisation*, Technical Report CS-R9545, CWI (1995).

[17] van Raamsdonk, F., P. Severi, M. H. Sørensen and H. Xi, *Perpetual reductions in lambda calculus*, Information and Computation **149** (1999), pp. 173–225.

Redundancy Elimination for LF

Jason Reed [1]

Carnegie Mellon University
Pittsburgh, Pennsylvania
jcreed@cs.cmu.edu

Abstract

We present a type system extending the dependent type theory LF, whose terms are more amenable to compact representation. This is achieved by carefully omitting certain subterms which are redundant in the sense that they can be recovered from the types of other subterms. This system is capable of omitting more redundant information than previous work in the same vein, because of its uniform treatment of higher-order and first-order terms. Moreover the 'recipe' for reconstruction of omitted information is encoded directly into annotations on the types in a signature. This brings to light connections between bidirectional (synthesis vs. checking) typing algorithms of the object language on the one hand, and the bidirectional flow of information in the ambient encoding language. The resulting system is a compromise seeking to retain both the effectiveness of full unification-based term reconstruction such as is found in implementation practice, and the logical simplicity of pure LF.

Key words: Proof Compression, Dependent Type Theory, Bidirectional Type Checking

1 Introduction

The use of logical frameworks in domains such as proof-carrying code [Nec97] makes the efficiency of proof representation and manipulation a nontrivial issue. Proofs of safety for realistic programs can be, if naïvely represented, unfeasibly large. Necula and Lee [NL98] developed one technique which addressed this issue. They give a way of representing proof terms in the logical framework LF [HHP93] in a more efficient way, by rewriting them with whole subtrees of the proofs erased. They then describe an algorithm which recovers these omitted parts, using typing information found in other parts of the proof.

[1] This work was supported by NSF Grant CCR 0306313 "Efficient Logical Frameworks".

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Their experimental results are good: proofs so represented tend to have size roughly $O(\sqrt{n})$ of the originals, with similar improvements in checking time.

To get a flavor of how omission works, consider the following example of encoding a natural deduction proof theory in LF. We have a signature

$$o : \mathsf{type} \qquad pf : o \to \mathsf{type}$$

$$\supset : o \to o \to o \qquad \wedge : o \to o \to o$$

where o is declared as the type of propositions, pf is the type family of proofs, indexed by proposition, and $\supset$ and $\wedge$ are the familiar logical connectives. Take one of the two natural deduction elimination rules for $\wedge$:

$$\frac{A \wedge B}{A} \wedge E_1$$

In LF it becomes

$$ande1 : \Pi a{:}o.\Pi b{:}o.pf\ (\wedge\ a\ b) \to pf\ a$$

Consider a use of this proof rule, $ande1\ a\ b\ d$. Here d must be a derivation of $a \wedge b$, and this larger proof $ande1\ a\ b\ d$ is a proof of a. This is excessively verbose, in a sense: knowing what type d is supposed to have (that is, $pf(\wedge\ a\ b)$) reveals what a and b must be. We would like to just write $ande1\ d$. It is not at all obvious, however, that the object d itself uniquely determines its type. This is a central issue, and we return to it below.

Another sort of apparent redundancy appears if we examine the introduction rule for implication. The natural deduction rule is

$$\frac{\begin{array}{c}\overline{A}\\ \vdots \\ B\end{array}}{A \supset B} \supset I$$

The hypothetical derivation of B under the hypothesis A is represented by higher-order abstract syntax [PE98] as a function from $pf\ a$ to $pf\ b$, and the rule is encoded as

$$impi : \Pi a{:}o.\Pi b{:}o.(pf\ a \to pf\ b) \to pf\ (\supset\ a\ b)$$

Here we may notice that if we have a term $impi\ a\ b\ f$, and if we know it, as a whole, is being checked against a certain type, $pf\ (\supset\ a\ b)$ then we can read off what a and b have been. If we knew that the type is going to be provided 'by the environment' somehow, then we can simply write $impi\ f$ instead.

It is this sort of omission of arguments that LF_i obtains its savings from. However, the technique uses a notion of 'reconstruction recipes' external to the

type system to control which arguments are omitted. This work aims to put the basic idea of Necula and Lee on firmer type-theoretical footing, explaining the mechanism of omission in the types themselves. We describe an extension of the LF language, called LF_*, such that the same sort of arguments to type families and constants can be omitted. Our priorities are, in order, (1) making sure that the extension is conservative, (2) making the theory logically well-motivated, (3) making sure that an eventual implementation is simple and easy to trust, and only finally (4) maximizing the number of subterms that can be omitted omitted.

It should be noted that this general idea of 'implicit' syntax is not new: It can be found in the earlier work of Hagiya and Toda [HT94] with LEGO, and Miquel [Miq01] and Luther [Lut01] with the Calculus of Constructions.

However, some approaches (such as [Miq01]) do not treat implicit terms as anything more than a user-interface convenience. Though the front-end reconstructs arguments omitted by the user, and erases them once again when terms are printed, the core of the implementation works with fully explicit terms. The meaning of the implicit calculus is in any event *defined* in terms of the explicit calculus: an implicit term is well-typed if it can be elaborated uniquely into an explicit term. Both [Lut01] and [HT94] agree that it seems "difficult to directly give a foundation to the implicit calculus." That is exactly the aim of this work.

The remainder of the paper is structured as follows. We first present the type theory of LF_*, followed by a description of a decision procedure for the judgments therein. The proof of correctness of this algorithm is sketched. We give a description of a translation from LF to LF_* and argue that it preserves typing and is bijective on terms, so that it witnesses the equivalence of the new language and the old.

2 Type Theory

The two critical questions left unanswered in the introductory example are *when does an object uniquely determine its type?* and *when do we already know, from the surrounding context, what type an object must have?* These are answered by organizing the language and type-checking algorithms of a system so as to support bidirectional type-checking.

The terms are divided into *normal* terms, which can be type-checked if a type is provided as input, *atomic* terms, which can be type-checked in such a way that uniquely determines (one says it *synthesizes*) a type as output if type-checking succeeds. Ordinarily in λ-calculi, we know that functions are normal, and application of a constant or variable to a list (or *spine*) S of arguments is atomic. That is, our grammar of terms looks something like

$$\begin{aligned} \text{terms } M &::= N \mid R \\ \text{normal } N &::= \lambda x.M \end{aligned}$$

$$\text{atomic } R ::= x \cdot S \mid c \cdot S$$
$$\text{spines } S ::= () \mid (M; S)$$

Our reasoning about the example, however, suggests that we may want some constants c — such as *impi* from the example — to require that $c \cdot S$ receive a type as input before type-checking proceeds, so that some omitted arguments in S can be recovered. We divide, therefore, the constants into two halves, the *synthesizable* constants c^+ and the *checkable* constants c^-. Therefore we write $ande1^+$ instead of $ande1$, and $impi^-$ instead of $impi$, for the latter will depend on the 'inherited' type information for reconstruction, where the former does not. In general a spine headed by a c^- constant is a normal term, rather than atomic.

Now we have a further problem, however. The fate of constants such as $ande1$ is in doubt, because they require certain of their arguments to be synthesizing. What if the proof we have in mind of $A \wedge B$ uses $impi^-$ as its last step? There are two conflicting requirements: $ande1^+$ wants to get type information from $impi^-$ to proceed with reconstruction, and vice versa.

We fill this gap by allowing type ascriptions to appear inside spines, so that when an argument does not provide its type, and the constant which it is an argument of requires it, the type can be simply written down in the term. We make a production rule for spine elements

$$E ::= M \mid M^+ \mid *$$

which says that an argument may either be an ordinary term, a term which is adequate when a synthesizing term is required, (see immediately below) or else a placeholder for an omitted argument. Spines are then given by

$$S ::= () \mid (E; S)$$

Terms are now

$$M ::= N \mid R$$
$$N ::= \lambda x.M \mid c^- \cdot S$$
$$R ::= x \cdot S \mid c^+ \cdot S$$

and the M^+ used above has the production

$$M^+ ::= (N : A) \mid (R :)$$

The new syntax $(R :)$ here seems peculiar: it would seem more natural to put simply R. For an atomic term is adequate when a synthesizing term is required, and so is a normal term with a type ascription. However, when we define substitution, it is necessary to know syntactically when we come across an atomic argument in a spine, whether it is in a position that actually *requires* a synthesizing term or not. The $(R :)$ signals that if substitution produces a normal term, then a type ascription must be introduced.

Now we turn to the language of types in LF_*. They are given by the grammar

$$\text{basic types } A, B ::= a \cdot S \mid \Pi^- x{:}A.B$$
$$\text{general types } Z ::= a \cdot S \mid \Pi^\rho x{:}A.Z$$
$$\text{omission modes } \mu ::= s \mid i$$
$$\text{polarities } \sigma ::= + \mid -$$
$$\Pi\text{-annotations } \rho ::= \sigma \mid [\mu]$$

Expanding out the grammar, there are four dependent function types, each of which determines how its argument functions with regard to omission and reconstruction. The Π^- functions are just the ordinary dependent functions from LF. They receive a $-$ superscript to make them stand in contrast with Π^+, which require their argument to be synthesizing. When there are Π^+ arguments, earlier arguments may be omitted via making their functional dependency $\Pi^{[s]}$, which marks a function whose argument is omitted *by synthesis*. Finally, $\Pi^{[i]}$ indicates a function whose argument is omitted *by inheriting* it from the result type the function application is checked against. In this language, the types of the proof rules in the example are (writing $A \to B$ for $\Pi^- x{:}A.B$ when x doesn't appear in B)

$$ande1 : \Pi^{[s]} a{:}o.\Pi^{[s]} b{:}o.\Pi^+ pf\ (\wedge\ a\ b).pf\ a$$

$$impi : \Pi^{[i]} a{:}o.\Pi^{[i]} b{:}o.(pf\ a \to pf\ b) \to pf\ (\supset\ a\ b)$$

Note that there is a distinction between the A, B are 'basic' types, which variables in a context may have, and Z which are the more general types that c^- constants can have. It would be more felicitously uniform if we could have simply one notion of type which constants and variables shared, but so far we have not been able to overcome the technical difficulties that arise when function variables are allowed to omit some of their arguments.

We elide for space reasons the grammar for kinds, and often refrain from mentioning the cases for kinds in the results below. Extending the definitions and results to that level is easy and uninteresting. Sometimes it is useful to write W, V as a 'wildcard' standing for a term or type or kind, for a briefer treatment of judgments and statements that are relevant for all three levels.

2.1 Substitutions

We elect a style of presentation which follows that of the concurrent logical framework CLF [WCPW03], in that we keep all terms in canonical form, that is, β-normal η-long form. This saves us from the complexity of dealing directly with $\beta\eta$-convertibility and the ensuing complex logical relations proofs of decidability of equality (for an example, see [HP01]) This complexity doesn't wholly disappear, though it reappears in a more tractable form: it is delegated to the definition of substitution. Substitution of a normal term in for a variable may create a redex, and the definition of substitution must carry out the

reduction to ensure that the result is still canonical. To show that this process terminates we must pay attention to the decrease in the size of types of redices, logically parallel to the induction in structural cut elimination [Pfe00]. For this reason, CLF indexes the substitution operators with the type at which they operate. In fact, to show just termination of the substitution algorithm, only the skeleton of the type is required, but for our purposes, we need the full type for an independent reason.

Namely, it is possible that a variable-headed term, say, $x \cdot ()$ appears in a spine in a position which needs to be synthesizing. As the matter stands, this is perfectly acceptable, for variables applied to spines are synthesizing. However, we may substitute a term for x, say $c^- \cdot ()$, that produces a result which no longer synthesizes. Therefore, before we set out on the substitution, we must specify what type the substituted object has, so that we can create a type ascription to ensure that the result synthesizes.

We define, therefore, partial operations $[M/x]^A M'$, $[M/x]^A A'$, $[M/x]^A S$, substitution of M for x in M', A', S, respectively, at the type A. Since substituting for a variable in a synthesizing term may require wrapping it in a type, we have have a σ-indexed partial operations $[M/x]^A_\sigma R$. When σ is plus it outputs an M^+, and when it's $-$, an M. The operation $[M \cdot S]^A_\sigma$ resolves the redex $M \cdot S$, for M at type A, and similarly produces an M^+ or M according to σ.

To see that this definition is well-founded, one can analyze the *simple type* of the type in the superscript, that is, the result of erasing all dependencies and changing every Π to a mere $\to$.

The term $subj(M^+)$ is defined by $subj(R :) = R$ and $subj(N : A) = N$. We write $[M^+/x]^A$ to mean $[subj(M^+)/x]^A$.

$$
\begin{gathered}
[M \cdot ()]^{a \cdot S}_- = M \\
[R \cdot ()]^{a \cdot S}_+ = (R :) \\
[N \cdot ()]^{a \cdot S}_+ = (N : a \cdot S) \\
[\lambda x.M \cdot (M'; S)]^{\Pi^- x:A.B}_\sigma = [[M'/x]^A M \cdot S]^{[M/x]^A B}_\sigma
\end{gathered}
$$

$$
\begin{gathered}
[M/x]^A_\sigma x \cdot S = [M \cdot [M/x]^A S]^A_\sigma \\
[M/x]^A_\sigma y \cdot S = y \cdot [M/x]^A S \\
[M/x]^A_\sigma c^+ \cdot S = c^+ \cdot [M/x]^A S
\end{gathered}
$$

$$
\begin{gathered}
[M/x]^A c^- \cdot S = c^- \cdot [M/x]^A S \\
[M/x]^A \lambda y.M = \lambda y.[M/x]^A M \\
[M/x]^A R = [M/x]^A_- R
\end{gathered}
$$

$$
[M/x]^A \mathsf{type} = \mathsf{type}
$$

$$[M/x]^A a \cdot S = a \cdot [M/x]^A S$$
$$[M/x]^A(\Pi^\rho y{:}B.Z) = \Pi^\rho y{:}[M/x]^A B.[M/x]^A Z$$

$$[M/x]^A() = ()$$
$$[M/x]^A(E; S) = ([M/x]^A E; [M/x]^A S)$$

$$[M/x]^A(R :) = [M/x]^A_+ R$$
$$[M/x]^A(N : B) = ([M/x]^A N : [M/x]^A B)$$

$$[M/x]^A * = *$$

2.2 Strictness

We have still so far neglected to pin down formally what it means for, say, one argument to have a sufficiently good occurrence in another argument to allow the former to be omitted. We can see that clearly a has an occurrence in $pf\ (\wedge\ a\ b)$ in such a way that we can 'read it off,' but the general higher-order case can be more complicated. The variable simply appearing in the syntax tree of the type is not enough, for the process of substituting in other arguments may cause β-reductions which make that appearance vanish. We therefore need to define *strict occurrences*, so that an argument which *strictly* occurs in the type of a synthesizing argument, or in the result type of a c^- constant, may be safely omitted.

The definition of strict occurrences that follows closely follows the definition of Pfenning and Schürmann [PS98] used to describe the theory of notational definitions. The notion of *pattern spine* at the heart of it is originally due to Miller [Mil91]. The guiding idea is that a strict position cannot be eliminated by other substitutions, and that, as a result, the operation of substituting $[M/x]N$ is injective in the argument M when x is strict in N. This injectivity means that we can uniquely recover M from $[M/x]N$. That is, the important consequence is that the corresponding matching problem is decidable and has a unique closed solution.

A key limitation of the way strictness is defined here, from the standpoint that more strict occurrences means more opportunities to omit redundant information, is that x cannot generally have a strict occurrence in $(*, S)$, even if it does have a strict occurrence in S. This is because we actually need more than just the term being uniquely determined when it is substituted for a strictly occurring variable: for technical reasons in the unification algorithm, we need its type to be uniquely determined as well.

The strictness judgments are $\Gamma \Vdash^s x \in Z$, (x has a strict occurrence in some argument of the type Z) $\Gamma \Vdash^i x \in Z$, (x has a strict occurrence in the output of the type Z) $\Gamma; \Delta \Vdash x \in W$, ($x$ has a strict occurrence in W in the presence of local bound variables Δ) and $\Delta \vdash S\ \mathsf{pat}$. ($S$ is a pattern spine, that is, a sequence of distinct bound variables)

2.2.1 Top-level

$$\frac{\Gamma;\cdot \Vdash x \in S}{\Gamma \Vdash^{i} x \in a \cdot S} \qquad \frac{\Gamma;\cdot \Vdash x \in A}{\Gamma \Vdash^{s} x \in \Pi^{+} y{:}A.B}$$

$$\frac{\Gamma, y : A \Vdash^{\mu} x \in B}{\Gamma \Vdash^{\mu} x \in \Pi^{\rho} y{:}A.B}$$

2.2.2 Types

$$\frac{\Gamma;\Delta \Vdash x \in S}{\Gamma;\Delta \Vdash x \in a \cdot S}$$

$$\frac{\Gamma;\Delta, y \Vdash x \in B}{\Gamma;\Delta \Vdash x \in \Pi^{\rho} y{:}A.B} \qquad \frac{\Gamma;\Delta \Vdash x \in A}{\Gamma;\Delta \Vdash x \in \Pi^{\rho} y{:}A.B}$$

2.2.3 Spines

$$\frac{\Gamma;\Delta \Vdash x \in M}{\Gamma;\Delta \Vdash x \in (M;S)} \qquad \frac{\Gamma;\Delta \Vdash x \in S}{\Gamma;\Delta \Vdash x \in (M;S)}$$

$$\frac{\Gamma;\Delta \Vdash x \in S}{\Gamma;\Delta \Vdash x \in (M^{+};S)} \qquad \frac{\Gamma;\Delta \Vdash x \in R}{\Gamma;\Delta \Vdash x \in ((R:);S)}$$

$$\frac{\Gamma;\Delta \Vdash x \in N}{\Gamma;\Delta \Vdash x \in ((N:A);S)} \qquad \frac{\Gamma;\Delta \Vdash x \in A}{\Gamma;\Delta \Vdash x \in ((N:A);S)}$$

2.2.4 Pattern Spines

Since all terms are in η-long form, define $x \to^{*}_{\bar{\eta}} H$ ("H is an η-expansion of the variable x") by

$$\frac{y_1 \to^{*}_{\bar{\eta}} H_1 \qquad \cdots \qquad y_n \to^{*}_{\bar{\eta}} H_n}{x \to^{*}_{\bar{\eta}} \lambda y_1 \ldots \lambda y_n.x \cdot (H_1;\cdots;H_n)}$$

Then the definition of pattern spine is

$$\frac{}{\Delta \vdash () \,\mathsf{pat}} \qquad \frac{x \to^{*}_{\bar{\eta}} H \qquad \Delta_1, \Delta_2 \vdash S \,\mathsf{pat}}{\Delta_1, x, \Delta_2 \vdash (H;S) \,\mathsf{pat}}$$

2.2.5 Terms

$$\frac{\Gamma;\Delta, y \Vdash x \in M}{\Gamma;\Delta \Vdash x \in \lambda y.M} \qquad \frac{\Delta \vdash S \,\mathsf{pat}}{\Gamma;\Delta \Vdash x \in x \cdot S}$$

$$\frac{y \in \Delta \qquad \Gamma;\Delta \Vdash x \in S}{\Gamma;\Delta \Vdash x \in y \cdot S} \qquad \frac{\Gamma;\Delta \Vdash x \in S}{\Gamma;\Delta \Vdash x \in c^{\sigma} \cdot S}$$

2.3 Type Checking

We define over the language of LF_* two typing judgments $\Gamma \vdash_{def} M : A$ and $\Gamma \vdash_{alg} M : A$, with analogous judgments at the type and kind levels. The former is definitionally simpler, and consequently far easier to reason about, but nonalgorithmic. The latter, however, is transparently decidable, and can be implemented directly.

Establishing correctness of the system as a whole now has two parts. The first part is to show that the algorithm embodied by $\Gamma \vdash_{alg} M : A$ is sound and complete relative to $\Gamma \vdash_{def} M : A$. After that we must still connect $\Gamma \vdash_{def} M : A$ over LF_* to the same typing judgment over the original language of LF, which we construe as a syntactic subset of LF_*.

In a diagram, the task ahead looks like

$$LF/\vdash_{def} \xrightarrow{(\text{—})^*} LF_*/\vdash_{def} = LF_*/\vdash_{alg}$$

Where $(\text{—})^*$ is a bijective translation from LF to LF_*.

We first give the rules that $\Gamma \vdash_{def} M : A$, $\Gamma \vdash_{alg} M : A$ have in common. This consists of all of the objects in the theory except for spines. Think of each rule with $\vdash$ as implicitly quantified by 'for all $\vdash \in \{\vdash_{def}, \vdash_{alg}\}$, ...'.

When we come to assigning types to spines there are two directions which a spine can be checked. The more familiar one is $\Gamma \vdash S : Z > C$, where the type Z and the spine S are given, and the type C is output. This is read as meaning that if a head (i.e. variable or constant) of type Z is applied S, the result will be of type C. However, we have introduced constants that require the output type to be known, so we also require a judgment $\Gamma \vdash S : Z < C$ which is identical in meaning to the other judgment, except that the type C is input rather than output.

2.3.1 Kinding

$$\frac{a : K \in \Sigma \qquad \Gamma \vdash S : K > \mathsf{type}}{\Gamma \vdash a \cdot S : \mathsf{type}}$$

$$\frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash \Pi^{\sigma} x{:}A.B : \mathsf{type}}$$

Notice here that Π^{μ} types are well-kinded only in the event that the variable they bind actually has a strict occurrence. This is a key property when proving soundness of the system.

$$\frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma, x : A \vdash B : \mathsf{type} \quad \Gamma, x : A \Vdash^{\mu} x \in B}{\Gamma \vdash \Pi^{[\mu]} x{:}A.B : \mathsf{type}}$$

2.4 Typing

$$\frac{\Gamma \vdash A : \mathsf{type}}{\Gamma \vdash (N : A) : A} \qquad \frac{\Gamma \vdash R : A}{\Gamma \vdash (R :) : A}$$

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash S : A > C}{\Gamma \vdash x \cdot S : C}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Pi^{-} x{:}A.B}$$

$$\frac{c^{+} : Z \in \Sigma \qquad \Gamma \vdash S : Z > C}{\Gamma \vdash c^{+} \cdot S : C}$$

$$\frac{c^{-} : Z \in \Sigma \qquad \Gamma \vdash S : Z < C}{\Gamma \vdash c^{-} \cdot S : C}$$

2.5 Spines: Definitional Typing

The definitional typing system $\vdash_{def}$ uses the following rules to typecheck spines. So that we can write down rules only once that work the same way for both $>$ and $<$, say $\gtrless^{s}$ means $>$ and $\gtrless^{i}$ means $<$. Recall that μ,μ' are variables standing for either s or i.

$$\frac{}{\Gamma \vdash_{def} () : \mathsf{type} \gtrless^{\mu'} \mathsf{type}}$$

$$\frac{}{\Gamma \vdash_{def} () : a \cdot S \gtrless^{\mu'} a \cdot S}$$

$$\frac{\Gamma \vdash_{def} M : A \qquad \Gamma \vdash_{def} S : [M/x]^{A} V \gtrless^{\mu'} W}{\Gamma \vdash_{def} (M; S) : \Pi^{-} x{:}A.V \gtrless^{\mu'} W}$$

$$\frac{A = A' \qquad \Gamma \vdash_{def} M^{+} : A' \qquad \Gamma \vdash_{def} S : [M^{+}/x]^{A} V \gtrless^{\mu'} W}{\Gamma \vdash_{def} (M^{+}; S) : \Pi^{+} x{:}A.V \gtrless^{\mu'} W}$$

$$\frac{\Gamma \vdash_{def} M : A \qquad \Gamma \vdash_{def} S : [M/x]^{A} V \gtrless^{\mu'} W}{\Gamma \vdash_{def} (*, S) : \Pi^{[\mu]} x{:}A.V \gtrless^{\mu'} W}$$

These rules as a system are impractical for an implementation because of the final rule. If read bottom-up, it requires the omitted argument M of a spine to be nondeterministically guessed.

2.6 Algorithmic Typing

The algorithmic type checking judgment does higher-order matching (that is, unification where all of the right-hand sides of equations have no free variables) to recover missing arguments.

2.6.1 Matching

We use P to denote sets of equations:

$$P ::= \top \mid (E_1 \doteq E_2) \wedge P \mid (S_1 \doteq S_2) \wedge P \mid (A_1 \doteq A_2) \wedge P$$

Q for sets of typing constraints:

$$Q ::= \top \mid (M : A) \wedge Q$$

and U for unification problems that track two sets of equality constraints, and one set of typing constraints:

$$U ::= \exists \Psi.(P, P', Q)$$

where Ψ denotes a list of variables

$$\Psi ::= \cdot \mid \Psi, x : A$$

It will also be necessary to talk about lists θ of substitutions:

$$\theta ::= \cdot \mid [M/x]^A \theta$$

There are several technical details about such substitutions θ that must be treated (not least of which, typing them) but for space reasons we do not cover them here.

The idea at a high level is that to solve a unification problem

$$\exists x_1{:}A_1, \ldots, x_n{:}A_n.(P, P', Q)$$

is to find a set of instantiations for $x_1, \ldots, x_n$ that make P, P', Q all true. Given that every x_i has a strict occurrence in P, which is maintained as an invariant of the algorithm, we can decompose equations in P while preserving any solutions that might exist, either instantiating variables, or postponing equations by transferring them to P', until P is empty, and all that remains is P' and Q. Since P is empty, our invariant says that no variables remain, so both P' and Q are closed, and can be checked directly. The only potential difficulty is the fact that we recursively call the typechecker on Q. But by inspection, the algorithm only puts strictly smaller type-checking problems into Q.

We define a transition relation $\Longrightarrow_\theta$ 'takes one step, resulting in substitution θ' via the following rules. The basic rules for working on a set of equations are quite straightforward, and all result in the empty substitution.

$$(a \cdot S_1 \doteq a \cdot S_2) \wedge P \Longrightarrow (S_1 \doteq S_2) \wedge P$$

$$(\Pi^\rho x{:}A_1.B_1 \doteq \Pi^\rho x{:}A_2.B_2) \wedge P \Longrightarrow$$

$$(A_1 \doteq A_2) \wedge (B_1 \doteq B_2) \wedge P$$

$$(\lambda x.M_1 \doteq \lambda x.M_2) \wedge P \Longrightarrow (M_1 \doteq M_2) \wedge P$$

$$(x \cdot S_1 \doteq x \cdot S_2) \wedge P \Longrightarrow (S_1 \doteq S_2) \wedge P$$

$$(c^\sigma \cdot S_1 \doteq c^\sigma \cdot S_2) \wedge P \Longrightarrow (S_1 \doteq S_2) \wedge P$$

$$(() \doteq ()) \wedge P \Longrightarrow P$$

$$((E_2; S_1) \doteq (E_2; S_2)) \wedge P \Longrightarrow (E_1 \doteq E_2) \wedge (S_1 \doteq S_2) \wedge P$$

$$(* \doteq *) \wedge P \Longrightarrow P$$

$$(R_1 :) \doteq (R_2 :) \wedge P \Longrightarrow (R_1 \doteq R_2) \wedge P$$

$$(N_1 : A_1) \doteq (N_2 : A_2) \wedge P \Longrightarrow (N_1 \doteq N_2) \wedge (A_1 \doteq A_2) \wedge P$$

These are used via

$$\frac{P \Longrightarrow P_0}{\exists \Psi'(P, P', Q) \Longrightarrow \exists \Psi'(P_0, P', Q)}$$

These less trivial rules handle the occurrence of variable on the left. Recall that we are doing matching, not full unification, so $\exists$-quantified variables do not occur on the right.

$$\exists \Psi, x : A, \Psi'.((x \cdot (H_1; \cdots; H_n) \doteq R) \wedge P, P'Q) \Longrightarrow_{[M/x]^A}$$

$$\exists \Psi, ([M/x]^A \Psi').[M/x]^A(P, P', Q)$$

(if $x_i \rightarrow_{\bar{\eta}}^* H_i$ where $x_1, \ldots, x_n$ are distinct variables not among those in Ψ, x, Ψ', Γ where $M = \lambda x_1 \ldots x_n.R$, if M has no free variables except those in Γ)

$$\exists \Psi, x : A, \Psi'.((x \cdot (H_1; \cdots; H_n) \doteq R) \wedge P, P'Q) \Longrightarrow$$

$$\exists \Psi, x : A, \Psi'.(P, (x \cdot (H_1; \cdots; H_n) \doteq R) \wedge P', Q)$$

(if the above rule doesn't apply)

Iterated $\Longrightarrow_\theta$ is the relation $\Longrightarrow_\theta^*$, defined by

$$\frac{}{U \Longrightarrow_{\cdot}^* U} \qquad \frac{U \Longrightarrow_\theta U' \qquad U' \Longrightarrow_{\theta'}^* U''}{U \Longrightarrow_{\theta'\theta}^* U''}$$

$\models$ is defined, like $\vdash$, uniformly over $\models_{def}$ and $\models_{alg}$ as follows:

$$\frac{}{\Gamma \models \top} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \models Q}{\Gamma \models (M : A) \wedge Q}$$

$$\frac{\Gamma \models P}{\Gamma \models (W \doteq W) \wedge P}$$

$$\frac{\Gamma \models P \qquad \Gamma \models P' \qquad \Gamma \models Q}{\Gamma \models (P, P', Q)}$$

Now we are able to give a definition of the core of the algorithm, the constraint generation judgment, which takes the form

$$\Gamma; \Psi; \Psi' \vdash S : Z \twoheadrightarrow^{\mu'} C/(P, Q)$$

This claims that if we are trying to apply a head of type Z to S, and the resulting type is C, then we must find instantiations for the variables in Ψ' to satisfy the equations P and type constraints Q. Γ, Ψ, S, Z are input to this judgment, and Ψ', P, Q are output. C is input if $\mu' = i$, and output if $\mu' = s$. The judgment is defined by the following rules.

$$\frac{}{\Gamma; \Psi; \cdot \vdash () : a \cdot S < a \cdot S'/(a \cdot S \doteq a \cdot S' \wedge \top, \top)}$$

$$\frac{\dfrac{}{\Gamma; \Psi; \cdot \vdash () : a \cdot S > a \cdot S/(\top, \top)} \qquad \Gamma; \Psi, x : A; \Psi' \vdash S : Z \twoheadrightarrow^{\mu'} C/(P, Q)}{\Gamma; \Psi; x : A, \Psi' \vdash (*; S) : \Pi^{[\mu]} x{:}A.Z \twoheadrightarrow^{\mu'} C/(P, Q)}$$

$$\frac{\Gamma \vdash_{alg} M^+ : A' \qquad \Gamma; \Psi; \Psi' \vdash S : [M^+/x]^A Z \twoheadrightarrow^{\mu'} C/(P, Q)}{\Gamma; \Psi; \Psi' \vdash (M^+; S) : \Pi^+ x{:}A.Z \twoheadrightarrow^{\mu'} C/((A \doteq A') \wedge P, Q)}$$

$$\frac{\Gamma; \Psi; \Psi' \vdash S : [M/x]^A Z \twoheadrightarrow^{\mu'} C/(P, Q)}{\Gamma; \Psi; \Psi' \vdash (M; S) : \Pi^- x{:}A.Z \twoheadrightarrow^{\mu'} C/(P, (M : A) \wedge Q)}$$

Finally, the toplevel rules which tell how to algorithmically typecheck a spine are

$$\frac{\Gamma; \cdot; \Psi' \vdash S : Z < C/(P, Q) \qquad \exists \Psi'.(P, \top, Q) \Longrightarrow^*_{\theta'} (\top, P', Q') \qquad \Gamma \models_{alg} (P', Q')}{\Gamma \vdash_{alg} S : Z < C}$$

$$\frac{\Gamma; \cdot; \Psi' \vdash S : Z > C/(P, Q) \qquad \exists \Psi'.(P, \top, Q) \Longrightarrow^*_{\theta'} (\top, P', Q') \qquad \Gamma \models_{alg} (P', Q')}{\Gamma \vdash_{alg} S : Z > \theta' C}$$

When we have the type as input ($\Gamma \vdash_{alg} S : Z < C$) we invoke constraint generation to produce Ψ', P, Q, and call unification to check that the constraints are satisfied. If unification succeeds, then type-checking does. If we

are to output a type ($\Gamma \vdash_{alg} S : Z < C$) then we furthermore use the substitution returned by unification, and apply it to the type C which constraint generation produced, and return this as the result type of S.

2.7 Correctness

The statements of soundness and completeness of unification are somewhat technical:

Lemma 2.1 (Soundness of Unification) *Suppose that*

$$\exists \Psi'.(P, P', Q) \Longrightarrow^*_{\theta_0} (\top, P'', Q')$$

and $\Gamma \models (\top, P'', Q)$. *Then there is a* θ' *such that* $\theta' = \theta_0$ *and* $\Gamma \vdash \theta' : \Psi'$ *and* $\Gamma \models \theta'(P, P', Q)$.

Lemma 2.2 (Completeness of Unification) *Suppose there exists* θ' *such that* $\Gamma \vdash \theta' : \Psi'$ *and* $\Gamma \models \theta'(P, P', Q)$. *Suppose further that for every* $x \in \Psi'$ *that there is an equation* $W \doteq W'$ *in* P *and a set* Δ_x *of variables disjoint from those declared in* Γ, Ψ' *such that* $\Gamma; \Delta_x \Vdash x \in W$. *Then there exist* P'', Q, θ_0 *such that* $\exists \Psi'.(P, P', Q) \Longrightarrow^*_{\theta_0} (\top, P'', Q')$ *and* $\theta' = \theta_0$ *and* $\Gamma \models (\top, P'', Q')$.

The main thrust of them, however, as is standard with such transition systems, is that (a) all of the individual transitions preserve solutions, and in our case, preserve strict occurrences as well, and (b) each transition decreases the size of the problem, so that solvability of a problem is decidable. The correctness of unification then leads to the correctness of the typing algorithm $\vdash_{alg}$ with respect to the definition $\vdash_{def}$.

Lemma 2.3 (Soundness and Completeness of $\vdash_{alg}$)

(i) *If* $\Gamma \vdash_{alg} M : A$, *then* $\Gamma \vdash_{def} M : A$.

(ii) *If* $\Gamma \vdash_{def} M : A$, *then* $\Gamma \vdash_{alg} M : A$.

3 Equivalence

Having defined LF_* and establishing that the definitional typing judgment is decidable, we turn now to the issue of showing that it is equivalent to LF. As mentioned previously, we construe the language of LF as a strict subset of the language of LF_*. Henceforth we syntactically distinguish every LF object with a $\circ$ in the subscript and every LF_* object with a $*$ subscript. The grammar of LF is

$$M_\circ ::= N_\circ \mid R_\circ$$
$$N_\circ ::= \lambda x.M_\circ$$
$$R_\circ ::= x \cdot S_\circ \mid c^+ \cdot S_\circ$$
$$E_\circ ::= M_\circ$$
$$S_\circ ::= () \mid (E_\circ; S_\circ)$$

$$A_\circ, B_\circ ::= a \cdot S_\circ \mid \Pi^- x{:}A_\circ.B_\circ$$
$$K_\circ ::= \mathsf{type} \mid \Pi^- x{:}A_\circ.K_\circ$$

This is simply the LF_* grammar with $c^-, \Pi^+, \Pi^{[\mu]}, (*;S), (M^+;S)$ removed. The typing judgments and rules that apply to this subset of LF_* are exactly the ordinary typing rules for LF. The only difference is cosmetic: here we say c^+, Π^- where one would of course find merely c, Π in a normal treatment of LF.

It remains to show that LF_* is isomorphic to LF, in the sense that every proof term in LF_* corresponds to one and only one proof term in LF. Fix for the sake of discussion signatures $\Sigma_\circ$ and Σ_*, in LF and LF_* respectively, and assume that they assign types and kinds to exactly the same constant and type family symbols, except that whenever Σ has c^+, we find exactly one of c^+ or c^- in Σ_*. Under suitable further assumptions (described below) that $\Sigma_\circ$ and Σ_* are in fact equivalent signatures, we aim to show that there is a translation from well-formed objects in $\Sigma_\circ$ to well-formed objects in Σ_* that is bijective, homomorphic with respect to typing, and so on.

One difficulty in establishing this result via such a translation comes from the fact that neither LF nor LF_* *prima facie* bears strictly more information than the other: LF_* signatures have more information in the form of Π-annotations, and its terms contain type ascriptions foreign to LF, while an LF term generally contains subterms that are omitted in its LF_* counterpart. Because of this, we cannot simply define an erasure function $W \mapsto W^*$ from LF to LF_* that erases some subterms to $*$. We need another erasure $W \mapsto W^\circ$ which erases Π-annotations, and we need $W \mapsto W^*$ to fill in necessary type ascriptions.

This notation is chosen to suggest that $(\text{—})^*$ takes objects into LF_*, and that $(\text{—})^\circ$ takes objects back to LF, though this latter statement is not strictly true. The general idea is that both mappings erase some information, and that objects $W_\circ$ and W_* ought to be considered equivalent when the mappings bring them together, when '$(W_\circ)^* = (W_*)^\circ$'.

The mapping $(\text{—})^\circ$ for Π-types is defined by $(\Pi^\rho x{:}A.W)^\circ = \Pi^- x{:}A.(W^\circ)$. Otherwise, $W^\circ = W$. However, the definition of $(\text{—})^*$ is less simple. Since it needs to insert type ascriptions, it cannot be merely a function from terms to terms, types to types, and so on. To know which type to insert, we must carry along the type, and in order to know the type of variables, we must carry along a context as well. We write this translation, then, using the same syntax as the typing judgment itself, as $(\Gamma_\circ \vdash M_\circ : A_\circ)^*$ for terms, and $(\Gamma_\circ \vdash A_\circ : \mathsf{type})^*$ for types.

For spines it is still not enough to write something of the form $(\Gamma_\circ \vdash S_\circ : Z_\circ > C_\circ)^*$. We need an additional argument Z_*, because its Π binders carry the required extra annotations required to translate the spine, (dictating, importantly, which arguments to erase) whereas $Z_\circ$ does not. Therefore the translation function for spines takes the form $(\Gamma_\circ \vdash S_\circ : Z_\circ > C_\circ)^*_{Z_*}$.

The translation is defined as follows:

Terms

$$(\Gamma_\circ \vdash \lambda x.M_\circ : \Pi^- x{:}A_\circ.B_\circ)^* =$$
$$\lambda x.(\Gamma_\circ, x : A_\circ \vdash M_\circ : B_\circ)^*$$
$$(\Gamma_\circ \vdash x \cdot S_\circ : C_\circ)^* = x \cdot (\Gamma_\circ \vdash S_\circ : A_\circ > C_\circ)^*_{(\Gamma_\circ \vdash A_\circ : \mathsf{type})^*}$$

(if $x : A_\circ \in \Gamma_\circ$)

$$(\Gamma_\circ \vdash c^+ \cdot S_\circ : C_\circ)^* = c^\sigma \cdot (\Gamma_\circ \vdash S_\circ : A_\circ > C_\circ)^*_{A_*}$$

(if $c^\sigma : A_* \in \Sigma_*$ and $c^+ : A_\circ \in \Sigma_\circ$)

Spines We mention only the case for typed (not kinded) spines. The other case is analogous. We split cases on the subscript Z_*. Make the abbreviations $A_* = (\Gamma_\circ \vdash A_\circ : \mathsf{type})^*$, and $S_* = (\Gamma_\circ \vdash S_\circ : [M_\circ/x]^{A_\circ} Z_\circ > C_\circ)^*_{[M_*/x]^{A_*} Z_*}$, and $M_* = (\Gamma_\circ \vdash M_\circ : A_\circ)^*$. Then for Π^σ we do

$$(\Gamma_\circ \vdash (M_\circ; S_\circ) : \Pi^- x{:}A_\circ.Z_\circ > C_\circ)^*_{\Pi^\sigma x:A_*.Z_*}$$

$$= \begin{cases} ((M_* : A_*); S_*) & \text{if } \sigma = +,\ M_* \text{ normal;} \\ ((M_*:); S_*) & \text{if } \sigma = +,\ M_* \text{ atomic;} \\ (M_*; S_*) & \text{otherwise.} \end{cases}$$

Observe that we only add the type annotation A_* when it is necessary. For $\Pi^{[\mu]}$ we simply erase the argument, and make the same recursive call on $S_\circ$ as before:

$$(\Gamma_\circ \vdash (M_\circ; S_\circ) : \Pi^- x{:}A_\circ.Z_\circ > C_\circ)^*_{\Pi^{[\mu]} x:A_*.Z_*} = (*; S_*)$$
$$(\Gamma_\circ \vdash () : C_\circ > C_\circ)^*_{C_*} = ()$$

Types

$$(\Gamma_\circ \vdash \Pi^- x{:}A_\circ.B_\circ : \mathsf{type})^* =$$
$$\Pi^- x{:}(\Gamma_\circ \vdash A_\circ : \mathsf{type})^*.(\Gamma_\circ, x : A_\circ \vdash B_\circ : \mathsf{type})^*$$
$$(\Gamma_\circ \vdash a \cdot S_\circ : \mathsf{type})^* = a \cdot (\Gamma_\circ \vdash S_\circ : K_\circ > \mathsf{type})^*_{K_*}$$

(if $a : K_* \in \Sigma_*$ and $a : K_\circ \in \Sigma_\circ$)

We may also translate contexts in the evident way, namely by translating each of the types in them. With these maps we can state the correspondence condition for the two signatures:

Definition 3.1 $\Sigma_\circ$ and Σ_* are *equivalent* if

- For every c, we have that $c^\sigma : A_* \in \Sigma_*$ and $c^+ : A_\circ \in \Sigma_\circ$ implies $(A_*)^\circ = (\cdot \vdash A_\circ : \mathsf{type})^*$.
- For every a, we have that $a : K_* \in \Sigma_*$ and $a : K_\circ \in \Sigma_\circ$ implies $(K_*)^\circ = (\cdot \vdash K_\circ : \mathsf{type})^*$.

When two signatures are equivalent, the theories they generate should be equivalent. This essentially amounts to two properties, that the image under the translation of the terms of a type actually belong to the translation of the type itself, and that the translation restricted to any one type is a bijection.

Theorem 3.2 (Type Preservation) *Suppose that* $\Sigma_\circ$ *and* Σ_* *are equivalent. Then*

- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} M_\circ : A_\circ$*, then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash M_\circ : A_\circ)^* : (\Gamma \vdash A_\circ : \mathsf{type})^*$
- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} S_\circ : A_\circ > C_\circ$ *and* $(\Gamma_\circ \vdash A_\circ : \mathsf{type})^* = (A_*)^\circ$*, then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash S_\circ : A_\circ > C_\circ)^*_{A_*} : A_* > (\Gamma \vdash C : \mathsf{type})^*$
- *If* $\Gamma_\circ$ *is a* $\Sigma_\circ$*-context, then* $(\Gamma_\circ)^*$ *is a* Σ_**-context.*
- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} A_\circ : \mathsf{type}$ *then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash A_\circ : \mathsf{type})^* : \mathsf{type}$.
- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} K_\circ : \mathsf{kind}$ *then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash K_\circ : \mathsf{kind})^* : \mathsf{kind}$.

Stating and proving the bijectivity of the translation, though important, is considerably more difficult, and so we do not develop it here.

4 Conclusion

We have described a type system which internalizes facts about which parts of terms can be safely omitted, while preserving representational adequacy. An implementation can achieve significant savings by not representing these omitted parts at all, and still 'prove the same theorems' as before.

The empirical advantage of this species of change of representation has been confirmed by earlier work. Ours retains several of its key properties. By working in a system derived from LF, we have at our disposal all of its representational techniques, such as higher-order abstract syntax. Like LF_i, full unification is not used, and instead only a subset — in our case, higher-order matching — is necessary. This is important for a maximally simple and trustable implementation.

The divergence from LF_i is that LF_* seeks to make type theoretic sense out of the possibility that subterms can be redundant. We do not have LF_i's ability to assign both an 'inference recipe' and a 'checking recipe' to a single constant, since we impose the restriction that a constant has a single type, which gives its reconstruction recipe once and for all. However, preliminary investigation suggests that in many cases — most especially when the object language is a type theory admitting a bidirectional typing algorithm itself — a constant is consistently always or almost always used in one way or the other. Thus, only one recipe is really necessary most of the time.

There is also a possible answer to this difficulty from using notational definitions. It is still an open problem whether notational definitions could feasibly be combined with this system, but if they could, then we could regain the ability to freely use different recipes by introducing a constant as being definitionally equal to an old one: one which, by virtue of being exposed at a new type, specifies a different reconstruction strategy for its arguments.

On the other side of the balance, there are forms of omission which LF_* can handle, which LF_i cannot. Since LF_* places a priority on pushing the mechanics of omission into the language itself at as fundamental a level as

possible, the design of it is such that all terms, types, and kinds can contain placeholders for omitted information as a matter of course: the indices to a type family are general terms, and terms may contain placeholders. LF_i, on the other hand, has restrictions on when placeholders can appear in types. We anticipate, therefore, that encoding techniques that use more high-order and high-level constructions may benefit from the uniform treatment of omission afforded by LF_*. A more precise evaluation of the effectiveness of the proposed system still awaits implementation and experimentation, which we hope to complete soon.

Acknowledgements

Many thanks are due Frank Pfenning for his encouragement and help with both the conceptual and technical portions of this work, and to Kevin Watkins for the original formulation of the type system.

References

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HP01] Robert Harper and Frank Pfenning. On the equivalence and canonical forms in the LF type theory. Technical report, Carnegie Mellon University, 2001.

[HT94] Masami Hagiya and Yozo Toda. On implicit arguments. In *Logic, Language and Computation*, pages 10–30, 1994.

[Lut01] Marko Luther. More on implicit syntax. In *Automated Reasoning. First International Joint Conference (IJCAR'01), Siena, Italy, June 18–23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 386–400, Berlin, 2001. Springer-Verlag.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Miq01] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In S. Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01, Krakow, Poland, 2–5 May 2001*, volume 2044, pages 344–359. Springer-Verlag, Berlin, 2001.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[NL98] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, 1998. IEEE Computer Society Press.

[PE98] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1998.

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, mar 2000.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[WCPW03] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical report, Carnegie Mellon University, 2003.

A logical framework with explicit conversions

Herman Geuvers and Freek Wiedijk [1,2]

Department of Computer Science, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract

The type theory λP corresponds to the logical framework LF. In this paper we present λH, a variant of λP where convertibility is not implemented by means of the customary *conversion rule*, but instead type conversions are made explicit in the terms. This means that the time to type check a λH term is proportional to the size of the term itself.

We define an *erasure* map from λH to λP, and show that through this map the type theory λH corresponds exactly to λP: any λH judgment will be erased to a λP judgment, and conversely each λP judgment can be *lifted* to a λH judgment.

We also show a version of subject reduction: if two λH terms are provably convertible then their types are also provably convertible.

1 Introduction

1.1 Problem

This paper addresses the question whether a formal proof should be allowed to contain the formal equivalent of the sentence '*this is left as an exercise to the reader.*' To explain what we mean here, consider the following 'proof':

> **Theorem.** The non-trivial zeroes of Riemann's $\zeta(s)$ function all lie on the complex line $\Re s = \frac{1}{2}$.
>
> *Proof.* There exists a derivation [3] of this statement with a length less than $10^{10^{100}}$ symbols (finding it is left as an exercise to the reader). Therefore the statement is true. □

[1] Thanks to Thorsten Altenkirch for the suggestion to use John-Major equality in our system.

[2] Email: {herman,freek}@cs.kun.nl

[3] The formal system in which this derivation is constructed does not really matter. Take any system in which one can do practical formal proofs. Some version of ZFC, like Mizar. Or HOL. Or Coq. It does not matter.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Now suppose that the statement in the proof about the existence of the derivation is true. [4] Then would this be an acceptable proof of the Riemann hypothesis? Can we really accept the number $10^{10^{100}}$ (which is the only interesting thing that this 'proof' contains) to be a *proof* here? Somehow it does not seem to contain enough relevant information.

At the TYPES meeting in Kloster Irsee in 1998, there was an interesting discussion after the talk by Henk Barendregt, where he had explained and advocated how to use the $\beta\delta\iota$-reduction of type theory to make Coq [12] automatically do calculations during its type check phase. This uses the technique of *reflection* (see e.g. [3,10] for examples and a discussion), where part of the object language is reflected in itself to make computations and reasoning on the meta-level possible within the system.

Most people clearly considered this way of using Coq's convertibility check to be a *feature*. The only dissenting voice came from Per Martin-Löf, who did not like it at all and seemed to consider this to be a *bug*. We do not have an overview of todays opinions, but the community seems to be quite unanimous that this kind of 'automatic calculations by type checking' is a good thing.

In the Automath system [8] the main performance bottleneck was the convertibility check (if the calculated type of a term M is N, but it is used in a context where the type should be N', then the system needs to verify that $N =_{\beta\delta} N'$.) In fact, the inefficiency of the convertibility check meant that correctness of Automath was in practice only semi-decidable. Although in theory it is decidable whether an Automath text is type correct, in practice when it is *not* correct the system often just would be endlessly reducing and would not terminate in an acceptable time anymore. [5] For this reason the Automath system from the seventies just gave up after having failed to establish convertibility after some given number of reduction steps. Automath apparently *searched* for a 'convertibility proof'. This proof would have to be rediscovered every time the Automath terms would be type checked, and it would not be stored in a 'convertibility proof term'.

The LF system [11] (currently implemented in the Twelf system [5]), which is the best known *logical framework*, has – like Automath and Coq – a conversion rule. But the HOL system [4,6] does not. In HOL β-reduction is not automatically tried by the system, but is one of the derivation rules of the logic. Similarly δ- and ι-reductions are performed using the rules of the logic. If one considers a HOL 'proof term' that stores the HOL rules that have been used to obtain a certain theorem [2], then this proof term somehow documents the 'reduction information' that is not available in a proof term from the type theoretical/LF world. [6]

[4] Which of course we do not know. But suppose.

[5] This problem is less noticeable in Coq because there many definitions are 'opaque' (they cannot be unfolded).

[6] Of course, as the HOL logic does not have dependent types, this kind of reduction is much less important in the first place.

The goal of this paper is to investigate whether it is possible to have a system close to the systems from type theory, but in which the convertibility of types *is* explicitly stored in the proof terms (like it is done in HOL). In such a system the type checker will not need to do a convertibility check on its own. Instead the term will contain the information needed to establish the convertibility. In such a system the type of a term will be *unique*, instead of only being unique *up to conversion.*[7] Because of all this, type checking a term will be cheap. If we consider the substitution operation and term identity checking to take unit time, the time to type check a term will be *linear* in the size of the term.

In the system that we describe in this paper, checking a proof matches much more the image of 'following the proof with your little finger, and checking locally that everything is correct' than is the case with the standard type theoretical proof systems.

1.2 Approach

We define a system called λH.[8] This system is very close to λP, the proper type system that corresponds to LF. However, there is no conversion rule. Instead conversions are made explicit in the terms. If H is a term that shows that A is convertible to A', which we will write as

$$\vdash H : A = A'$$

and if the term a has type A, then the term a^H (the conversion H applied to a) will have type A'.

Note that in our system we have explicit 'equality judgments' just like in Martin-Löf style type theory [9]. However there is a significant difference. In Martin-Löf style type theory there are no terms that prove equalities. The equality judgments in such theories look like:

$$\vdash A = A' : B$$

and the equality is on the *left* of the colon. In contrast, in our system two terms that are provably equal do not need to have the same computed type, so there will not be a common type to the right of the colon. Instead we will have a proof term, and so our equality will be to the *right* of the colon.

The fact that two terms in our system that occur in an equality judgment do not need to have computed types that are syntactically equal, means that our judgmental equality is a version of *John-Major equality* [7].

---

[7] So in such a system the type of a term will be $(\lambda x{:}A.B)\,a$, or it will be $B[x := a]$, but not *both*.

[8] The 'H' in the name of the system reflects the letter that we use for the convertibility proof terms. So λH is 'the logical framework with Hs', i.e., with convertibility proofs.

1.3 Related Work

Robin Adams is working on a version of pure type systems that have judgmental equalities in the style of Martin-Löf type theories [1]. However, he does not have terms in his system that represent the derivation of the equality judgments. Also, he does not represent the conversions themselves in the terms. Therefore in his system more terms are syntactically identical than in our system. Another difference is that he develops his system for all functional pure type systems, while we only have a system that corresponds to λP.

1.4 Contribution

We define a system λH in which type conversion is represented in the proof term. We show that this system corresponds exactly to the proper type system λP. We also show that this system has a property closely related to subject reduction.

The λH system is quite a bit more complicated than the λP system. It has 13 instead of 4 term constructors, and 15 instead of 7 derivation rules.

1.5 Outline

In Section 2 we define our system. In Section 3 we show that it corresponds to the λP system. In Section 4 we show that an analog of the subject reduction property of λP holds for our system. In Section 5 we define a weak reduction relation for our equality proof terms that is confluent and strongly normalizing and that satisfies subject reduction. Finally, in Section 7 we present a slight modification of our system, where we do not allow conversions to go through the ill-typed terms. Such a system corresponds more closely to a semantical view upon type theory.

2 The system λH

Definition 2.1 The λH expressions are given by the following grammar (the syntactic category $\mathcal{V}$ are the variable names):

$$\mathcal{T} ::= \Box \mid * \mid \Pi \mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \lambda \mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \mathcal{T}\mathcal{T} \mid \mathcal{T}^{\mathcal{E}}$$
$$\mathcal{E} ::= \bar{\mathcal{T}} \mid \mathcal{E}^{\dagger} \mid \mathcal{E} \cdot \mathcal{E} \mid \beta(\mathcal{T}) \mid \iota(\mathcal{T}) \mid \{\mathcal{E}, [\mathcal{V}]\mathcal{E}\} \mid \langle \mathcal{E}, [\mathcal{V}]\mathcal{E}\rangle \mid \mathcal{E}\mathcal{E}$$
$$\mathcal{C} ::= \mid \mathcal{C}, \mathcal{V} : \mathcal{T}$$
$$\mathcal{J} ::= \mathcal{C} \vdash \mathcal{T} : \mathcal{T} \mid \mathcal{E} : \mathcal{T} = \mathcal{T}$$

The $\mathcal{T}$ are the terms of the system, the $\mathcal{E}$ are convertibility proofs, the $\mathcal{C}$ are the contexts, and the $\mathcal{J}$ are the judgments. The *sorts* are the special cases of $\mathcal{T}$ that are the elements of $\{\Box, *\}$.

Definition 2.2 We define the *erasure* operation recursively by:

$$
\begin{aligned}
|x| &\equiv x \\
|\Box| &\equiv \Box \\
|*| &\equiv * \\
|\Pi x{:}A.B| &\equiv \Pi x{:}|A|.|B| \\
|\lambda x{:}A.b| &\equiv \lambda x{:}|A|.|b| \\
|Fa| &\equiv |F||a| \\
|a^H| &\equiv |a|
\end{aligned}
$$

It maps λH terms to λP terms and is extended straightforwardly to contexts.

There are two kinds of judgments in λH: *equality judgments* and *typing judgments*. The first are of the form $H : a = b$, where H codes a proof of convertibility (through not necessarily well-typed terms) of a and b. The rules for equality judgments are independent of typing judgments. In the rules for the typing judgments, equality judgments appear as a side-condition (in the rule for conversion).

Definition 2.3 The rules that inductively generate the λH judgments are the following (in these rules s only ranges over sorts):

definitional equality

$$\frac{}{\bar{A} : A = A}$$

$$\frac{H : A = A'}{H^\dagger : A' = A}$$

$$\frac{H : A = A' \quad H' : A' = A''}{H \cdot H' : A = A''}$$

β-redex

$$\frac{}{\beta((\lambda x{:}A.b)\, a) : (\lambda x{:}A.b)\, a = b[x := a]}$$

erasing equality proofs

$$\frac{}{\iota(a) : a = |a|}$$

congruence rules

$$\frac{H : A = A' \quad H' : B = B'}{\{H, [x]H'\} : \Pi x{:}A.B = \Pi x{:}A'.B'}$$

$$\frac{H : A = A' \quad H' : B = B'}{\langle H, [x]H'\rangle : \lambda x{:}A.B = \lambda x{:}A'.B'}$$

$$\frac{H : F = F' \quad H' : a = a'}{HH' : Fa = F'a'}$$

start & weakening

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash b : B}{\Gamma, x : A \vdash b : B}$$

box & star

$$\vdash * : \Box$$

typed lambda terms

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x{:}A.B : s}$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash b : B : s}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash F : \Pi x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$$

conversion

$$\frac{\Gamma \vdash a : A \quad H : A = A'}{\Gamma \vdash a^H : A'}$$

We write $\Gamma \vdash A : B : C$ as an abbreviation of $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$. We write $A =_{\lambda H} A'$ if we have that $H : A = A'$ for some H. We write 'A is type correct in context Γ' if we have that $\Gamma \vdash A : B$ for some B. We write 'A is type correct' if it is type correct in some context. We write 'Γ is well-formed' if some derivable judgment has Γ as the context. Finally we will write derivability in λH as $\vdash_{\lambda H}$ to distinguish it from derivability in λP which is written $\vdash_{\lambda P}$. (If we omit the subscript, it will be apparent from the context which system is meant.)

The following lemmas about λH are immediate:

Lemma 2.4 *Any subterm of a type correct term is type correct (in the appropriate context).*

Lemma 2.5 *If $\Gamma \vdash A : B : C$ then C is a sort.*

Lemma 2.6 *If $\Gamma \vdash a : A$ then either $\Gamma \vdash A : s$ for some sort s, or $A \equiv \Box$.*

Lemma 2.7 (uniqueness of types) *If* $\Gamma \vdash a : A$ *and* $\Gamma \vdash a : A'$ *then* $A \equiv A'$.

We now show that typing is in linear time by defining a type checking algorithm.

Definition 2.8 Define the function $\mathrm{type} : \mathcal{C} \times \mathcal{T} \to \mathcal{T} \cup \{\mathsf{false}\}$ simultaneously with the functions $\mathrm{wf} : \mathcal{C} \to \{\mathsf{true}, \mathsf{false}\}$ and $\mathrm{comp} : \mathcal{E} \times \mathcal{T} \to \mathcal{T} \cup \{\mathsf{false}\}$ as follows.

$$
\begin{aligned}
\mathrm{type}_\Gamma(*) &= \mathsf{if}\ \mathrm{wf}(\Gamma)\ \mathsf{then}\ \Box\ \mathsf{else}\ \mathsf{false}\\
\mathrm{type}_\Gamma(\Box) &= \mathsf{false}\\
\mathrm{type}_\Gamma(x) &= \mathsf{if}\ \mathrm{wf}(\Gamma) \wedge (x{:}A) \in \Gamma\ \mathsf{then}\ A\ \mathsf{else}\ \mathsf{false}\\
\mathrm{type}_\Gamma(\Pi x{:}A.B) &= \mathsf{if}\ \mathrm{type}_\Gamma(A) \equiv * \wedge \mathrm{type}_{\Gamma,x:A}(B) \in \{*, \Box\}\\
&\quad \mathsf{then}\ \mathrm{type}_{\Gamma,x:A}(B)\ \mathsf{else}\ \mathsf{false}\\
\mathrm{type}_\Gamma(\lambda x{:}A.M) &= \mathsf{if}\ \mathrm{type}_\Gamma(A) \equiv * \wedge \mathrm{type}_{\Gamma,x:A}(M) \neq \Box\\
&\quad \mathsf{then}\ \Pi x{:}A.\mathrm{type}_{\Gamma,x:A}(M)\ \mathsf{else}\ \mathsf{false}\\
\mathrm{type}_\Gamma(MN) &= \mathsf{if}\ \mathrm{type}_\Gamma(M) \equiv \Pi x{:}\mathrm{type}_\Gamma(N).B\\
&\quad \mathsf{then}\ B[x := N]\ \mathsf{else}\ \mathsf{false}\\
\mathrm{type}_\Gamma(M^H) &= \mathsf{if}\ \mathrm{type}_\Gamma(M) \equiv A\ \mathsf{then}\ \mathrm{comp}(H, A)\ \mathsf{else}\ \mathsf{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{wf}(\langle - \rangle) &= \mathsf{true}\\
\mathrm{wf}(\Gamma, x{:}A) &= \mathrm{type}_\Gamma(A) \in \{*, \Box\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{comp}(\bar{A}, B) &= \mathsf{if}\ A \equiv B\ \mathsf{then}\ B\ \mathsf{else}\ \mathsf{false}\\
\mathrm{comp}(H^\dagger, B) &= \mathrm{comp}^{-1}(H, B)\\
\mathrm{comp}(H \cdot H', B) &= \mathrm{comp}(H', \mathrm{comp}(H, B))\\
\mathrm{comp}(\iota(A), B) &= \mathsf{if}\ A \equiv B\ \mathsf{then}\ |A|\ \mathsf{else}\ \mathsf{false}\\
\mathrm{comp}(\beta((\lambda x{:}A.M)N), B) &= \mathsf{if}\ B \equiv (\lambda x{:}A.M)N\ \mathsf{then}\ M[x := N]\ \mathsf{else}\ \mathsf{false}\\
\mathrm{comp}(\{H, [x]H'\}, B) &= \mathsf{if}\ B \equiv \Pi y{:}A.C\\
&\quad \mathsf{then}\ \Pi x{:}\mathrm{comp}(H, A).\mathrm{comp}(H', C[y := x])\\
&\quad \mathsf{else}\ \mathsf{false}\\
\mathrm{comp}(\langle H, [x]H'\rangle, B) &= \mathsf{if}\ B \equiv \lambda y{:}A.C\\
&\quad \mathsf{then}\ \lambda x{:}\mathrm{comp}(H, A).\mathrm{comp}(H', C[y := x])\\
&\quad \mathsf{else}\ \mathsf{false}\\
\mathrm{comp}(HH', B) &= \mathsf{if}\ B \equiv AC\\
&\quad \mathsf{then}\ \mathrm{comp}(H, A)\mathrm{comp}(H', C)\ \mathsf{else}\ \mathsf{false}
\end{aligned}
$$

The function comp^{-1} is defined totally similar to comp, with two exceptions:

$$
\begin{aligned}
\mathrm{comp}^{-1}(\iota(A), B) &= \mathsf{if}\ |A| \equiv B\ \mathsf{then}\ A\ \mathsf{else}\ \mathsf{false}\\
\mathrm{comp}^{-1}(\beta((\lambda x{:}A.M)N), B) &= \mathsf{if}\ B \equiv M[x := N]\ \mathsf{then}\ (\lambda x{:}A.M)N\ \mathsf{else}\ \mathsf{false}
\end{aligned}
$$

Proposition 2.9 (type checking) $\Gamma \vdash_{\lambda H} a : A$ *if and only if* $\mathit{type}(\Gamma, a) \equiv A$ *and the time for type to compute an answer is linear in the length of the inputs.*

Proof. One first proves the fact that, $H : B = C$ if and only if $\mathrm{comp}(H, B) = C$. Then $\Gamma \vdash_{\lambda H} a : A$ implies $\mathrm{type}(\Gamma, a) \equiv A$ is proved by induction on the derivation, simultaneously with 'Γ is well formed' implies $\mathrm{wf}(\Gamma) = \mathsf{true}$. The other way around, one proves simultaneously that $\mathrm{type}(\Gamma, a) \equiv A$ implies $\Gamma \vdash_{\lambda H} a : A$ and that $\mathrm{wf}(\Gamma) = \mathsf{true}$ implies 'Γ is well formed' (by induction over the length of the input: $\mathrm{lth}(\Gamma, a)$, resp. $\mathrm{lth}(\Gamma)$).
$\mathrm{comp}(H, A)$ is clearly linear in the size of the equational proof term H. To make sure that type computes a type in linear time, one has to collect the 'side conditions' $\mathrm{wf}(\Gamma)$ properly to avoid checking the well-foundedness of the (local) context for every variable separately.

3 Correspondence to λP

Lemma 3.1 *If $A =_{\lambda H} A'$ then $|A| =_\beta |A'|$.*

Proof. By induction on the derivation of $A =_{\lambda H} A'$.

Proposition 3.2 ('from λH to λP') *If $\Gamma \vdash_{\lambda H} a : A$ then $|\Gamma| \vdash_{\lambda P} |a| : |A|$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda H} a : A$, using the previous Lemma in the conversion rule.

Lemma 3.3 *For $A, A' \in \mathcal{T}$,*

(i) *if $|A| \equiv |A'|$, then $A =_{\lambda H} A'$,*

(ii) *if $|A| =_\beta |A'|$, then $A =_{\lambda H} A'$.*

Proof.

(i) If $|A| \equiv |A'|$, then $\iota(A) \cdot \overline{\iota A'} : A = A'$.

(ii) If $|A| =_\beta |A'|$, we first prove that $|A| =_{\lambda H} |A'|$, by induction on the proof (in the equational theory of the λ-calculus) of $|A| =_\beta |A'|$. Then we conclude by using that $\iota(A) : A = |A|$. We do some cases
- $A \equiv \Pi x{:}B.C$ and $A' \equiv \Pi x{:}B'.C'$ and $|\Pi x{:}B.C| =_\beta |\Pi x{:}B'.C'|$ was derived from $|B| =_\beta |B'|$ and $|C| =_\beta |C'|$. By IH, $H_0 : |B| = |B'|$ and $H_1 : |C| = |C'|$ for some H_0, H_1, so $\{H_0, [x]H_1\} : |\Pi x{:}B.C| = |\Pi x{:}B'.C'|$.
- $A \equiv (\lambda x{:}B.M)P$, $A' \equiv M'[P'/x]$ with $|(\lambda x{:}B.M)P| \to_\beta |M'[P'/x]|$. Then $\beta((\lambda x{:}B.M)P) : (\lambda x{:}B.M)P = M[P/x]$ and we are done by two applications of (i) (using $|M[P/x]| \equiv |M'[P'/x]|$).

Proposition 3.4 ('from λP to λH') *Let Γ be a λP-context and a, A be λP-terms such that $\Gamma \vdash_{\lambda P} a : A$. Then the following two properties hold.*

(i) *There is a correct λH-context Γ' such that $|\Gamma'| \equiv \Gamma$.*

(ii) *For all λH-contexts Γ' for which $|\Gamma'| \equiv \Gamma$, there are λH-terms a', A' such that $\Gamma' \vdash_{\lambda H} a' : A'$ and $|a'| \equiv a$, $|A'| \equiv A$.*

Proof. Simultaneously by induction on the λP derivation, distinguishing cases according to the last applied rule. We treat four cases and abbreviate 'induction hypothesis' to IH.

- (application)
$$\frac{\Gamma \vdash_{\lambda P} F : \Pi x{:}A.B \quad \Gamma \vdash_{\lambda P} a : A}{\Gamma \vdash_{\lambda P} Fa : B[x := a]}$$
The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. Furthermore, for Γ' any λH-context such that $|\Gamma'| \equiv \Gamma$, by IH, there are F', a', A', A'' and B' such that $\Gamma' \vdash_{\lambda H} F' : \Pi x{:}A'.B'$, $\Gamma' \vdash_{\lambda H} a' : A''$ and $|F'| \equiv F$, $|a'| \equiv a$, $|A'| \equiv |A''| \equiv A$ and $|B'| \equiv B$. By Lemma 3.3, we have $H : A'' = A'$ for some H, so $\Gamma' \vdash_{\lambda H} a'^H : A'$ and $\Gamma' \vdash Fa'^H : B'[a'^H/x]$. We are done, because $|Fa'^H| \equiv Fa$ and $|B'[a'^H/x]| \equiv B[a/x]$.
- (λ)
$$\frac{\Gamma, x{:}A \vdash_{\lambda P} M : B \quad \Gamma \vdash_{\lambda P} A : \star}{\Gamma \vdash_{\lambda P} \lambda x{:}A.M : \Pi x{:}A.B}$$
The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. Furthermore, for Γ' any λH-context such that $|\Gamma'| \equiv \Gamma$, by IH, there is an A' such that $\Gamma' \vdash_{\lambda H} A' : \star$ and $|A'| \equiv A$. So $\Gamma', x{:}A'$ is a correct λH-context. So, by IH there are M' and B' such that $\Gamma', x{:}A' \vdash_{\lambda H} M' : B'$ and $|M'| \equiv M$ and $|B'| \equiv B$. Hence, $\Gamma' \vdash \lambda x{:}A'.M' : \Pi x{:}A'.B'$ and we are done, because $|\lambda x{:}A'.M'| \equiv \lambda x{:}A.M$ and $|\Pi x{:}a'.B'| \equiv \Pi x{:}A.B$.
- (conversion)
$$\frac{\Gamma \vdash_{\lambda P} M : A \quad \Gamma \vdash_{\lambda P} B : s \quad A =_\beta B}{\Gamma \vdash_{\lambda P} M^H : B}$$
The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. Furthermore, for Γ' any λH-context such that $|\Gamma'| \equiv \Gamma$, by IH, there is are M', A', B' such that $\Gamma' \vdash_{\lambda H} M' : A'$, $\Gamma \vdash_{\lambda H} B' : s$ and $|A'| \equiv A$, $|B'| \equiv B$, $|M'| \equiv M$. So $|A'| \equiv A =_\beta B \equiv |B'|$ and by Lemma 3.3, $H : A' = B'$ for some H. Now, $\Gamma' \vdash_{\lambda H} M' : B'$ by the conversion rule in λH and we are done.
- (weakening)
$$\frac{\Gamma \vdash_{\lambda P} A : \star \quad \Gamma \vdash_{\lambda P} M : B}{\Gamma, x{:}A \vdash_{\lambda P} M : B}$$
The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. By IH, there is an A' such that $\Gamma' \vdash_{\lambda H} A' : \star$ and $|A'| \equiv A$. So $\Gamma', x{:}A'$ is a correct λH-context, proving part (1). Now, for any λH context $\Gamma', x{:}A'$ such that $|\Gamma', x{:}A'| \equiv \Gamma, x{:}A$, we know that $|\Gamma'| \equiv \Gamma$, so, by IH there are M' and B' such that $\Gamma' \vdash_{\lambda H} M' : B'$ and $|M'| \equiv M$ and $|B'| \equiv B$. As $\Gamma', x{:}A'$ is correct, we can weaken this to obtain $\Gamma', x{:}A' \vdash_{\lambda H} M' : B'$ and we are done.

Corollary 3.5 (conservativity of λP over λH) *Given a well-formed λH context Γ and λH type A in Γ,*

$$|\Gamma| \vdash_{\lambda P} M : |A| \Rightarrow \exists M'(\Gamma \vdash_{\lambda H} M : A \wedge |M'| \equiv M)$$

Proof. The second part of the Proposition ensures that there are N and B such that $\Gamma \vdash_{\lambda H} N : B$ and $|N| \equiv M$ and $|B| \equiv |A|$. Then $B =_{\lambda H} A$, due to Lemma 3.3, say $H : B = A$. Then $\Gamma \vdash_{\lambda H} N^H : A$.

4 An analogue of subject reduction

The following proposition is the equivalent for λH of the subject reduction property of λP. The system λH does not have a notion of β-reduction, so the statement $a \to_\beta a'$ in the condition of the statement is replaced by $a =_{\lambda H} a'$. Also, we do not get that the type is conserved, it just is conserved up to convertibility (so if $a = a'$ and $a : A$ then we will not always get that $a' : A$, but just that $a' : A'$ for some A' with $A = A'$.)

Proposition 4.1 ('subject reduction') *If* $\Gamma \vdash_{\lambda H} a : A : s$ *and* $\Gamma \vdash_{\lambda H} a' : A' : s'$ *and* $a =_{\lambda H} a'$ *then* $A =_{\lambda H} A'$ *and* $s \equiv s'$.

Proof. From Proposition 3.2 we get that $|\Gamma| \vdash_{\lambda P} |a| : |A| : s$ and $|\Gamma| \vdash_{\lambda P} |a'| : |A'| : s'$ and $|a| =_\beta |a'|$. By subject reduction of λP and uniqueness of types in λP we get that $|A| =_\beta |A'|$ and $s \equiv s'$. From Lemma 3.3 we finally get that $A =_{\lambda H} A'$.

5 Conversion reduction

Definition 5.1 We define the *conversion reduction* relation $\twoheadrightarrow$ as the rewrite relation of the following rewrite rules:

$$\begin{aligned} A^{\overline{A'}} &\to A \\ A^{H \cdot H'} &\to (A^H)^{H'} \\ \bar{A}^\dagger &\to \bar{A} \\ H^{\dagger\dagger} &\to H \\ (H \cdot H')^\dagger &\to H'^\dagger \cdot H^\dagger \end{aligned}$$

We will now list some simple properties of conversion reduction (with some proofs omitted for space reasons):

Proposition 5.2 *Conversion reduction is confluent.*

Proposition 5.3 *Conversion reduction is strongly normalizing.*

(These two propositions even hold for terms that are not type correct.)

Proposition 5.4 (subject reduction for conversion reduction) *If* $\Gamma \vdash_{\lambda H} a : A$ *and* $a \twoheadrightarrow a'$ *then* $\Gamma \vdash_{\lambda H} a' : A$.

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda H} a : A$ one proves that, if $a \to a'$, then $\Gamma \vdash_{\lambda H} a' : A$, distinguishing cases according to the applied reduction step. (The $a \twoheadrightarrow a'$ case then follows immediately.)

Although this proposition is a subject reduction property, it is *not* related to the subject reduction property of λP, as it does not involve β-reduction.

Proposition 5.5 *If* $\Gamma \vdash_{\lambda H} a : A$ *and* $a \twoheadrightarrow a'$ *then* $a =_{\lambda H} a'$.

Proof. If $a \twoheadrightarrow a'$, then $|a| \equiv |a'|$ and hence $a =_{\lambda H} a'$ by Lemma 3.3.

Proposition 5.6 *A term that is in conversion reduction normal form does not contain the operations* $\bar{A}$ *and* $H \cdot H'$*, and it only contains the operation* $H^\dagger$ *in the combinations* $\beta(\ldots)^\dagger$ *and* $\iota(\ldots)^\dagger$.

This last proposition shows that we can do away with the $\bar{A}$ and $H \cdot H'$ operations in our system.

6 Discussion

Imagine formalizing the 'proof' of the Riemann hypothesis on page 1 in Coq. Using reflection this would be doable (even constructively) and, if the Riemann hypothesis has a proof, it would actually be a correct proof too. However, type checking this proof would be completely infeasible.

Now imagine a version of Coq that is built on top of the logical framework λH. When type checking this Coq 'proof' the system would need to store the reduction information in the explicit conversions in the λH proof term that it would build internally. Therefore that proof term would be impractically big. So in such a system the proof would not be considered to be a *real* proof as the underlying λH proof object would be impossible to construct.

For this reason λH adequately represents both our unease with our 'proof' of the Riemann hypothesis, as well as Per's unease with Henk's talk in Kloster Irsee.

(Note that this formalization of the 'proof' of the Riemann hypothesis needs ι-reduction, so it is not possible in LF itself. Therefore for our argument one needs to imagine a version of Coq's type theory that has explicit conversions: a system that relates to the calculus of inductive constructions CiC, in the same way that the system λH relates to λP.)

7 Future work

An interesting thing to do now is to implement λH as the basis of an actual proof assistant, to see whether it is a practical system for doing actual proof checking. Part of such a system might be a term *lifter* that lifts proof terms from λP to λH, inserting the conversions that were needed to make the terms type check.

Another issue is whether it is possible to build such a system in a way that the bulk of the proof terms will not actually be stored in memory, but checked and discarded while it is being generated. This is the way that HOL checks its proofs. Henk Barendregt calls this *ephemeral proofs*. So the question is

whether it will be possible to have a λH implementation with ephemeral proof terms.

7.1 Avoiding ill-typed terms

In the system λH, we have avoided the conversion rule by introducing proof terms that witness an equality (and that can be checked in linear time). But the conversion goes through $\mathcal{T}$, the set of 'pseudo-terms'. This is in line with most implementations of proof checkers, where equality checking is done by a separate algorithm that does not take typing into account. But what if we restrict equalities to conversions that pass through the well-typed terms only? This is more in line with a semantical intuition, where the ill-typed terms just do not exist. We can adapt the syntax of λH to cover this situation and we put the question whether this system is equivalent to λH. We call this new system λF.[9]

The system λF has the same terms and equality proofs as λH, but the judgments are different:

$$\mathcal{J} ::= \mathcal{C} \vdash \mathcal{T} : \mathcal{T} \;\mid\; \mathcal{C} \vdash \mathcal{E} : \mathcal{T} = \mathcal{T}$$

So an equality in λF is always stated and proved *within* a context, in which the terms are well-typed. The rules that inductively generate the λF judgments are the same as for λH, apart from the rules that involve equalities, which are as follows (in these rules s only ranges over sorts):

definitional equality

$$\frac{\Gamma \vdash A : B}{\Gamma \vdash \bar{A} : A = A}$$

$$\frac{\Gamma \vdash H : A = A'}{\Gamma \vdash H^{\dagger} : A' = A}$$

$$\frac{\Gamma \vdash H : A = A' \quad \Gamma \vdash H' : A' = A''}{\Gamma \vdash H \cdot H' : A = A''}$$

β-redex

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash b : B : s \quad \Gamma \vdash a : A}{\Gamma \vdash \beta((\lambda x{:}A.b)\, a) : (\lambda x{:}A.b)\, a = b[x := a]}$$

conversion

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash H : A = A'}{\Gamma \vdash a^{H} : A'}$$

[9] The 'F' stands for 'fully well-typed'.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash H : A = A'}{\Gamma \vdash \iota(a^H) : a = a^H}$$

congruence rules

$$\frac{\begin{array}{c}\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s \\ \Gamma \vdash A' : * \quad \Gamma, x' : A' \vdash B' : s \\ \Gamma \vdash H : A = A' \quad \Gamma, x : A \vdash H' : B = B'[x' := x^H]\end{array}}{\Gamma \vdash \{H, [x{:}A]H'\} : \Pi x{:}A.B = \Pi x'{:}A'.B'}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \quad \Gamma, x : A \vdash b : B : s \\ \Gamma \vdash A' : * \quad \Gamma, x : A' \vdash b' : B' : s \\ \Gamma \vdash H : A = A' \quad \Gamma, x : A \vdash H' : b = b'[x' := x^H]\end{array}}{\Gamma \vdash \langle H, [x{:}A]H'\rangle : \lambda x{:}A.b = \lambda x{:}A'.b'}$$

$$\frac{\begin{array}{c}\Gamma \vdash F : \Pi x{:}A.B \quad \Gamma \vdash a : A \\ \Gamma \vdash F' : \Pi x'{:}A'.B' \quad \Gamma \vdash a' : A' \\ \Gamma \vdash H : F = F' \quad \Gamma \vdash H' : a = a'\end{array}}{\Gamma \vdash HH' : Fa = F'a'}$$

(Note that the $\iota(\ldots)$ of λF just removes one conversion, in contrast to the $\iota(\ldots)$ of λH which removes all conversions at once. Removing all conversions generally leads to a term that is not well-typed, so that is not an option for λF where all terms have to be well-typed, even in the conversion proofs.)

References

[1] Robin Adams. Pure Type Systems with Judgemental Equality. Unpublished, 2003.

[2] Stefan Berghofer. New features of the Isabelle theorem prover – proof terms and code generation, 2000.
http://www4.in.tum.de/~berghofe/papers/TYPES2000_slides.ps.gz

[3] H. Geuvers, F. Wiedijk, J. Zwanenburg, Equational Reasoning via Partial Reflection, in *Theorem Proving for Higher Order Logics*, TPHOL 2000, Portland OR, USA, eds. M. Aagaard and J. Harrison, LNCS 1869, pp. 162–178.

[4] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL.* Cambridge University Press, Cambridge, 1993.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin, A framework for defining logics, in *Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1987, pp. 194–204.

[6] John Harrison. *The HOL Light manual (1.1)*, 2000.
http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz

[7] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.
http://www.dur.ac.uk/c.t.mcbride/thesis.ps.gz

[8] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, Amsterdam, 1994.

[9] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory, An Introduction*. Oxford University Press, 1990.
http://www.cs.chalmers.se/Cs/Research/Logic/book/book.ps

[10] M. Oostdijk and H. Geuvers, Proof by Computation in the Coq system, *Theoretical Computer Science* 272 (1-2), 2001, pp. 293–314.

[11] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems, in *Proceedings of the 16th International Conference on Automated Deduction* (CADE-16), ed. H. Ganzinger, LNAI 1632, 1999, pp. 202–206.

[12] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2002.
ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual-all.ps.gz

Specifying Properties of Concurrent Computations in CLF ⋆

Kevin Watkins [a,1] Iliano Cervesato [b,2] Frank Pfenning [a,3]
David Walker [c,4]

[a] *Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA*
[b] *ITT Industries, AES Division, Alexandria, VA*
[c] *Department of Computer Science, Princeton University, Princeton, NJ*

Abstract

CLF (the Concurrent Logical Framework) is a language for specifying and reasoning about concurrent systems. Its most significant feature is the first-class representation of *concurrent executions as monadic expressions*. We illustrate the representation techniques available within CLF by applying them to an asynchronous pi-calculus with correspondence assertions, including its dynamic semantics, safety criterion, and a type system with latent effects due to Gordon and Jeffrey.

Key words: Please list keywords from your paper here, separated by commas.

1 Introduction

This paper cannot describe the CLF framework in detail; a complete description is available in other work [19,18,3], and the syntax and typing rules of the framework are summarized in Appendix B. Nevertheless, in this introduction, we briefly discuss the lineage of frameworks on which CLF is based, and the basic design of CLF.

A logical framework is a meta-language for the specification and implementation of deductive systems, which are used pervasively in logic and the theory of programming languages. A logical framework should be as simple

⋆ This research was sponsored in part by the NSF under grants CCR-9988281, CCR-0208601, CCR-0238328, and CCR-0306313, and by NRL under grant N00173-00-C-2086.
[1] Email: kw@cs.cmu.edu
[2] Email: iliano@itd.nrl.navy.mil
[3] Email: fp@cs.cmu.edu
[4] Email: dpw@cs.princeton.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and uniform as possible, yet provide intrinsic means for representing common concepts and operations in its application domain. A logical framework is characterized by an underlying meta-logic or type theory and a representation methodology.

The principal starting point for our work is the LF logical framework [7], which is based on a minimal type theory λ^{Π} with only the dependent function type constructor Π. LF directly supports concise expression of variable renaming and capture-avoiding substitution at the level of syntax, and parametric and hypothetical judgments in deductions, following the *judgments-as-types* principle. Proofs are reified as objects, which allows properties of and relations between proofs to be expressed within the framework.

Representations of systems involving state remained cumbersome until the design of the linear logical framework LLF [2] and its close relative RLF [10]. LLF is a conservative extension of LF with the linear function type $A \multimap B$, the additive product type $A \,\&\, B$, and the additive unit type $\top$. The main additional representation of LLF is that of *state-as-linear-hypotheses*. Imperative computations consequently become linear objects in the framework. They can serve as index objects, which means we can express properties of stateful systems at a high level of abstraction.

While LLF solves many problems associated with stateful computation, the encoding of *concurrent computations* remained unsatisfactory for several reasons. One of the problems is that LLF formulations of concurrent systems inherently sequentialize the computation steps.

In this paper we are concerned with CLF, a conservative extension of LLF with intrinsic support for concurrency. Concurrent computations are encapsulated in a monad [15], which permits a simple definitional equality and guarantees conservativity over LF and LLF. The definitional equality on monadic expressions identifies different interleavings of independent steps, thereby expressing *true concurrency*. Dependent types then allow us to specify properties of concurrent computations, as long as they do not rely on the order of independent steps.

We illustrate the framework's expressive power and representation techniques through a sample encoding of the asynchronous π-calculus with correspondence assertions, following Gordon and Jeffrey [6]. Further examples, such as encodings of Petri-nets, Concurrent ML, and the security protocol specification framework MSR can be found in another technical report [3].

The remainder of the paper is organized as follows: Section 2 introduces the π-calculus with which we are concerned and its syntax; Section 3 describes the original static semantics of Gordon et al. and its CLF representation; Section 4 describes the operational semantics of the language and its CLF representation; Section 5 introduces the syntax of traces and describes the abstraction judgment relating computations and traces, and briefly discusses the safety criterion; Section 6 briefly describes related work; and Section 7 concludes. Appendices summarize the π-calculus encoding and the syntax and judgments

$$
\begin{array}{rl}
\mathsf{pr} & : \mathsf{type}. \\
\mathsf{nm} & : \mathsf{type}. \\
\mathsf{tp} & : \mathsf{type}. \\
\mathsf{label} & : \mathsf{type}.
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{stop} & : \mathsf{pr}. \\
\mathsf{par} & : \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{repeat} & : \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{new} & : \mathsf{tp} \to (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{choose} & : \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{out} & : \mathsf{nm} \to \mathsf{nm} \to \mathsf{pr}. \\
\mathsf{inp} & : \mathsf{nm} \to \mathsf{tp} \to \\
 & \qquad (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{begin} & : \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{end} & : \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}.
\end{array}
\qquad
\begin{array}{rl}
\ulcorner \mathsf{stop} \urcorner & = \mathsf{stop} \\
\ulcorner P \mid Q \urcorner & = \mathsf{par}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{repeat}\ P \urcorner & = \mathsf{repeat}\ \ulcorner P \urcorner \\
\ulcorner \mathsf{new}(x{:}\tau); P \urcorner & = \mathsf{new}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner P \urcorner) \\
\ulcorner \mathsf{choose}\ P\ Q \urcorner & = \mathsf{choose}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{out}\ x\langle y \rangle \urcorner & = \mathsf{out}\ x\ y \\
\ulcorner \mathsf{inp}\ x(y{:}\tau); P \urcorner & = \mathsf{inp}\ x\ \ulcorner \tau \urcorner\ (\lambda y. \ulcorner P \urcorner) \\
 & \\
\ulcorner \mathsf{begin}\ L; P \urcorner & = \mathsf{begin}\ \ulcorner L \urcorner\ \ulcorner P \urcorner \\
\ulcorner \mathsf{end}\ L; P \urcorner & = \mathsf{end}\ \ulcorner L \urcorner\ \ulcorner P \urcorner
\end{array}
$$

Fig. 1. Process syntax represented in CLF

of the framework.

2 The asynchronous π-calculus with correspondence assertions

Our asynchronous π-calculus with correspondence assertions follows Gordon and Jeffrey's presentation [6]. Correspondence assertions, originally developed by Woo and Lam [20], come in two varieties, $\mathsf{begin}\ L$ and $\mathsf{end}\ L$, where L is a label that carries information about the state of a communication protocol. Gordon and Jeffrey have shown that a variety of important correctness properties of cryptographic protocols can be stated in terms of matching pairs of these begin and end assertions.

To illustrate the basic ideas, we will examine a simple handshake protocol taken directly from Gordon and Jeffrey's work. This protocol is intended to ensure that if a sender named a receives an acknowledgment message then the receiver named b has actually received the message. In the asynchronous π-calculus with correspondence assertions, we specify the protocol as follows.

$$
\begin{array}{rl}
\mathrm{Send}(a,b,c) = & \mathsf{new}(msg); \mathsf{new}(ack); \\
 & \quad (\mathsf{out}\ c\langle msg, ack \rangle \\
 & \qquad \mid \mathsf{inp}\ ack(); \mathsf{end}\ (a,b,msg)) \\
\mathrm{Rcv}(a,b,c) = & \mathsf{inp}\ c(x,y); \mathsf{begin}\ (a,b,x); \mathsf{out}\ y\langle\rangle
\end{array}
$$

The standard π-calculus process constructors used here are parallel composition ($P \mid Q$), generation of a new name x to be used in a process P ($\mathsf{new}(x); P$ where x is bound in P), asynchronous output on channel c ($\mathsf{out}\ c\langle msg, ack \rangle$), and input on channel c ($\mathsf{inp}\ c(x_1, \ldots, x_n); P$ where variables x_1 through x_n are bound in P). In the protocol, the sending process generates a new message

and a new acknowledgment channel. The sender uses the asynchronous output command to send the pair of message and acknowledgment channel on c and waits for a response on ack. Once the sender receives the acknowledgment, it executes an end assertion which specifies that the sender (named a) requires that the receiver (named b) has already received the input message (msg). The receiver cooperates with the sender by waiting for pairs of message and acknowledgment on channel c. After receiving on c, the begin assertion declares that the receiver b has received the input message. After this declaration, the receiver sends an acknowledgment to the sender. We hope that in all executions of senders in parallel with receivers, begin assertions match up with end assertions. If they do, sender a can be sure that receiver b received the message msg.

Now, consider combining a single sender in parallel with a single receiver: $\mathsf{new}(c); (\mathrm{Send}(a, b, c) \mid \mathrm{Rcv}(a, b, c))$. This configuration is *safe* since in every possible execution, every end (a, b, msg) assertion is preceded in that execution by a distinct corresponding begin (a, b, msg) assertion. On the other hand, placing multiple different senders in parallel with a single copy of a receiver is *unsafe*:

$$\mathrm{Send}(a, b, c) \mid \mathrm{Send}(a', b, c) \mid \mathrm{Rcv}(a, b, c)$$

This configuration is *unsafe* because there exists an execution in which an end L assertion is executed but there has been no prior matching begin L. More specifically, the second sender a' may create a message and send it to the receiver. The receiver, thinking it is communicating with a, receives the message, executes begin (a, b, msg), and returns the acknowledgment. Finally, the second sender executes end (a', b, msg). In this protocol, since the identity of the sender (either a or a') was not included in the message, there has been confusion over who the receiver was communicating with. This is a very simple example, but Gordon and Jeffrey have demonstrated that these assertions can be used to identify serious flaws in much more complicated and important protocols.

2.1 Syntax

The syntax of the π-calculus processes P with correspondence assertions is presented below. We have simplified Gordon and Jeffrey's calculus in a couple of ways, replacing polyadic input and output processes with monadic versions, dropping any data structures other than channels x, y, z and replacing deterministic if statements with non-deterministic choice (choose P Q). Two process forms that did not show up in the informal example above are the process stop, which does nothing, and the replicated process repeat P, which acts as an unbounded number of copies of itself. The static semantics makes use of types τ, which are discussed in the next section; these do not affect the operational semantics of a program.

$$\begin{array}{ll} \mathsf{name} : \mathsf{tp}. & \ulcorner \mathsf{Name} \urcorner = \mathsf{name} \\ \mathsf{chan} : \mathsf{tp} \to (\mathsf{nm} \to \mathsf{eff}) \to \mathsf{tp}. & \ulcorner \mathsf{Ch}(x{:}\tau)e \urcorner = \mathsf{chan}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner e \urcorner) \end{array}$$

Fig. 2. Type syntax represented in CLF

$$\begin{array}{rl} P, Q ::= & \mathsf{stop} \mid (P \mid Q) \mid \mathsf{repeat}\ P \mid \mathsf{new}(x{:}\tau); P \\ & \mid \mathsf{choose}\ P\ Q \mid \mathsf{out}\ x\langle y \rangle \mid \mathsf{inp}\ x(y{:}\tau); P \\ & \mid \mathsf{begin}\ L; P \mid \mathsf{end}\ L; P \end{array}$$

The representation of process syntax follows standard LF methodology. The signature, shown in Figure 1, represents process syntax via CLF types pr (processes), nm (names), tp (types), and label (assertion labels). The representation function mapping processes to CLF objects is shown at the right.

A few comments: The type nm of names does not contain any closed terms; it classifies bound variables within a process expression. The type tp is discussed in Section 3. Channels are a special case of names. We do not specify any particular syntax for assertion labels, but it is assumed that they might mention names bound by new or inp. As is common in LF representations, we use *higher-order abstract syntax*, which allows us to model π-calculus bound variables using framework variables and to implement π-calculus substitution using the framework's substitution.

The most important property of this representation is *adequacy*: every process in the original language has its own representative as a CLF object of type pr, and every object in pr is such a representation. The canonical forms property for CLF renders proofs of such results almost trivial.

3 The static semantics

Gordon and Jeffrey present a static semantics with types and effects for their language. The goal of the static semantics is to ensure that the *correspondence property* for assertions is not violated: for each end L assertion reached in an execution, a distinct begin L assertion for L must have been reached in the past. The static semantics associates an effect e (a multiset of labels) with each program point, such that it is safe to execute end L for each label L in the multiset. (Of course, not all safe programs will necessarily have a valid typing.)

Since CLF includes LLF as a sublanguage, we will be able to represent the static "state" of the effect system as a multiset of linear hypotheses in LLF style [2]. The basic idea is to record a multiset of begins already reached at the current program point as linear hypotheses of the typing judgment.[5] Then each occurrence of begin L contributes a linear hypothesis of type effect L for

[5] Really these are *affine* hypotheses, since the invariant is that the multiset be merely a lower bound: it is perfectly safe to "forget" that a begin was reached at some point in the past. Careful use of the additives $\top$ and & will allow us to simulate affine hypotheses with linear ones.

$$
\begin{aligned}
\mathsf{has} &: \mathsf{nm} \to \mathsf{tp} \to \mathsf{type}.\\
\mathsf{good} &: \mathsf{pr} \to \mathsf{type}.\\
\mathsf{consume} &: \mathsf{eff} \to \mathsf{type}.\\
\mathsf{assume} &: \mathsf{eff} \to \mathsf{pr} \to \mathsf{type}.\\
\\
\mathsf{gd\_stop} &: \mathsf{good}\ \mathsf{stop} \circ\!\!- \top.\\
\mathsf{gd\_par} &: \mathsf{good}\ (\mathsf{par}\ P\ Q) \circ\!\!-\ \mathsf{good}\ P \circ\!\!-\ \mathsf{good}\ Q.\\
\mathsf{gd\_repeat} &: \mathsf{good}\ (\mathsf{repeat}\ P) \circ\!\!-\ \top \leftarrow \mathsf{good}\ P.\\
\mathsf{gd\_new} &: \mathsf{good}\ (\mathsf{new}\ \tau\ (\lambda x.\, P\ x)) \leftarrow \mathsf{wftp}\ \tau\\
&\qquad \circ\!\!-(\Pi x\!:\!\mathsf{nm}.\, \mathsf{has}\ x\ \tau \to \mathsf{good}\ (P\ x)).\\
\mathsf{gd\_choose} &: \mathsf{good}\ (\mathsf{choose}\ P\ Q) \circ\!\!-\ (\mathsf{good}\ P\ \&\ \mathsf{good}\ Q).\\
\mathsf{gd\_out} &: \mathsf{good}\ (\mathsf{out}\ X\ Y) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \leftarrow \mathsf{has}\ Y\ \tau\\
&\qquad \circ\!\!-\mathsf{consume}\ (E\ Y).\\
\mathsf{gd\_inp} &: \mathsf{good}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y))\\
&\qquad \leftarrow (\Pi y\!:\!\mathsf{nm}.\, \mathsf{has}\ y\ \tau \to \mathsf{assume}\ (E\ y)\ (P\ y)).\\
\mathsf{gd\_begin} &: \mathsf{good}\ (\mathsf{begin}\ L\ P) \circ\!\!-\ (\mathsf{effect}\ L \multimap \mathsf{good}\ P).\\
\mathsf{gd\_end} &: \mathsf{good}\ (\mathsf{end}\ L\ P) \circ\!\!-\ \mathsf{effect}\ L \circ\!\!-\ \mathsf{good}\ P.\\
\\
\mathsf{con\_eps} &: \mathsf{consume}\ \{1\} \circ\!\!-\ \top.\\
\mathsf{con\_join} &: \mathsf{consume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\}\\
&\qquad \circ\!\!-\mathsf{effect}\ L \circ\!\!-\ \mathsf{consume}\ E.\\
\\
\mathsf{ass\_eps} &: \mathsf{assume}\ \{1\}\ P \circ\!\!-\ \mathsf{good}\ P.\\
\mathsf{ass\_join} &: \mathsf{assume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\}\\
&\qquad \circ\!\!-(\mathsf{effect}\ L \multimap \mathsf{assume}\ E\ P).
\end{aligned}
$$

Fig. 3. Static semantics represented in CLF

the checking of its continuation, and each end L consumes such a hypothesis.

This accounts for trivial instances of correct programs in which an end is found directly within the continuation of its matching begin. Of course, in actual use, one is more interested in cases in which the end and its matching begin occur in different processes executing concurrently (as in the example of Section 2).

Gordon et al. introduce *latent effects* to treat many such cases. The idea is that each value transmitted across a channel may carry with it a multiset of latent effects, the effects being *debited* from the process sending the value and *credited* to the process receiving it. Since communication synchronizes the sending and receiving processes, it is certain that the begins introducing the debited effects in the sending process will occur before any ends making use of the credited effects in the receiving process. [6]

These considerations lead to a simple type syntax. Each name in the static semantics has a type τ: either Name (really nonsense; i.e., just a nonce) or

[6] Of course, this implicitly relies on the unicast nature of communication in the language. If multicast or broadcast were allowed, more than one process could be credited, violating the non-duplicable nature of effect hypotheses.

$\mathsf{Ch}(x:\tau)e$, representing a channel transmitting names of type τ and a latent effect e. These types ("π-types," for short) are represented by CLF type tp, the constructors of which are shown in Figure 2. Latent effects e are themselves multisets of labels, and are represented in CLF by a type eff discussed below.

Although a latent effect is again a multiset of labels, the LLF strategy of representing multisets by linear hypotheses does not apply, because latent effects must be first-class values. An LF strategy using explicit list constructors (cons and nil) would represent the latent effects as first-class values, but the LF equality on such lists would be too restrictive: $[L_1, L_2]$ and $[L_2, L_1]$ are equal as multisets, but $\mathsf{cons}\ L_1\ (\mathsf{cons}\ L_2\ \mathsf{nil})$ and $\mathsf{cons}\ L_2\ (\mathsf{cons}\ L_1\ \mathsf{nil})$ are not necessarily equal as LF objects.

In CLF, we have a new alternative: expressions are first-class objects, and CLF's concurrent equality on them can model multiset equality precisely. Each label multiset $[L_1, \ldots, L_n]$ will be represented by an expression $\{\mathsf{let}\ \{1\} = \mathsf{latent}\ L_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ \{1\} = \mathsf{latent}\ L_n\ \mathsf{in}\ 1\}$. The equality on the representation then naturally models equality of multisets. We take eff to be a notational abbreviation for the type $\{1\}$ of such expressions, and add the following declaration to the signature.

$$\mathsf{latent} : \mathsf{label} \to \{1\}.$$

In addition, we must axiomatize the objects of type $\{1\}$ that correspond to such multisets; this is the judgment wfeff presented in Appendix A.

Next we represent the π-calculus typing judgment itself as a CLF type family good, defined in Figure 3. We use $A \circ\!\!- B$ and $A \leftarrow B$, which associate to the left, as alternate forms of $B \multimap A$ and $B \to A$, giving the signature the shape of a logic program. The type A in $\Pi u{:}A.\, B$ has been omitted where it is determined by context. We often omit outermost Π quantifiers; in such cases the corresponding arguments to the constant in question are also omitted (implicit). We have also η-contracted some subterms in order to conserve space; these should be read as abbreviations for their η-long (canonical) forms.

Since not every declared effect must actually occur (that is, there is implicitly a *weakening* principle for effects), we must use the additive unit $\top$ to consume any leftover effects at the leaves of a derivation (instances of the $\mathsf{gd\_stop}$ or $\mathsf{con\_eps}$ rules).

The type family wftp, not shown in the figure (see Appendix A), represents the judgment that a π-type is well formed, reducing more or less to the judgment wfeff for any latent effects mentioned in the π-type. The type family has contains no closed objects, but in the course of a derivation of $\mathsf{good}\ P$, hypotheses $\mathsf{has}\ x\ \tau$ will be introduced for each name bound by new or inp in P. Similarly, the family effect has no closed objects, but in the course of a typing derivation, linear hypotheses $\mathsf{effect}\ L$ can be introduced by begin and consumed by end.

The task of assume and $\mathsf{consume}$ is to introduce and consume linear hy-

potheses for the whole multiset of effects contained in a latent effect. Latent effects are consumed by out, which has no continuation, and produced by inp, which does. Accordingly, assume takes the continuation as an argument, and invokes good to check it once the multiset of effects has been introduced into the linear context.

It can be shown by extensions of the standard techniques developed for the LLF fragment of CLF that this representation is *adequate*: a process P is well-typed in the original system just when there is an object of type good P in CLF.

4 The operational semantics

Gordon and Jeffrey's operational semantics [6] is based on a traced transition system $P \xrightarrow{s} P'$, where s is a trace: a sequence of begin and end actions, internal actions τ, and gen actions binding freshly generated names (corresponding to the execution of new). Although we have not specified the language of labels, it is assumed that they may mention such names. Then $P \xrightarrow{s} P'$ when P can evolve to P' while performing the actions in trace s. The traced transition system itself depends on the usual notion of structural congruence $P \equiv P'$ found in the π-calculus literature.

The CLF representation has a somewhat different structure. Since CLF has a first-class notation for concurrent computations, we can factor the traced transition system into two judgments: first, that a process P has a concurrent execution E (which is represented by a CLF expression); and second, that an execution E has a (serialized) trace s. This section is concerned with the first judgment, while the next section treats traces.

Computations in this semantics are represented by CLF expressions

$$x_1 : \mathsf{nm}, \ldots, x_m : \mathsf{nm},\ r_1 : \mathsf{run}\ P_1, \ldots, r_i : \mathsf{run}\ P_i;\\ r_{i+1} :\!\!\uparrow \mathsf{run}\ P_{i+1}, \ldots, r_n :\!\!\uparrow \mathsf{run}\ P_n \ \vdash\ E \leftarrow \top$$

in a context having unrestricted hypotheses of type nm for each generated name, unrestricted hypotheses $r_1 \ldots r_i$ of type run P for each process P that is executing and available unrestrictedly, and linear hypotheses $r_{i+1} \ldots r_n$ of type run P for each process P that is available linearly, where run : pr $\rightarrow$ type.[7] The final result of the computation is taken as the additive unit $\top$, which means that computation can stop at any time, with any leftover resources (linear hypotheses) consumed by $\langle\rangle$, its introduction form.

Then each of the *structural* process constructors stop, par, repeat, and new can be represented by a corresponding synchronous CLF connective:

[7] Here $\leftarrow$ denotes the lax typing judgment, not reverse implication.

$$
\begin{aligned}
\mathsf{ev\_stop} &: \mathsf{run}\ \mathsf{stop} \multimap \{1\}. \\
\mathsf{ev\_par} &: \mathsf{run}\ (\mathsf{par}\ P\ Q) \multimap \{\mathsf{run}\ P \otimes \mathsf{run}\ Q\}. \\
\mathsf{ev\_repeat} &: \mathsf{run}\ (\mathsf{repeat}\ P) \multimap \{!\mathsf{run}\ P\}. \\
\mathsf{ev\_new} &: \mathsf{run}\ (\mathsf{new}\ \tau\ (\lambda u.\ P\ u)) \multimap \{\exists u{:}\mathsf{nm}.\ \mathsf{run}\ (P\ u)\}.
\end{aligned}
$$

The remaining constructors are interpreted according to their semantics:

$$
\begin{aligned}
\mathsf{ev\_choose}_i &: \mathsf{run}\ (\mathsf{choose}\ P_1\ P_2) \multimap \{\mathsf{run}\ P_i\}. \\
\mathsf{ev\_sync} &: \mathsf{run}\ (\mathsf{out}\ X\ Y) \multimap \mathsf{run}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\ P\ y)) \\
&\qquad \multimap \{\mathsf{run}\ (P\ Y)\}. \\
\mathsf{ev\_begin} &: \Pi L{:}\mathsf{label}.\ \mathsf{run}\ (\mathsf{begin}\ L\ P) \multimap \{\mathsf{run}\ P\}. \\
\mathsf{ev\_end} &: \Pi L{:}\mathsf{label}.\ \mathsf{run}\ (\mathsf{end}\ L\ P) \multimap \{\mathsf{run}\ P\}.
\end{aligned}
$$

We depart from the usual practice of leaving outermost Π quantifiers implicit for reasons that will become clear in Section 5.

One interesting feature of the CLF encoding is that many of the structural equivalences of the original presentation of the π-calculus appear automatically (shallowly) as consequences of the principles of exchange, weakening (since $\top$ is present) and so forth satisfied by CLF hypotheses. In the CLF setting the rest of the structural equivalences are captured within a general notion of simulation, discussed briefly in Section 5.

In this representation, each concurrent computation starting from a process P corresponds to a CLF object of type $\mathsf{run}\ P \multimap \{\top\}$; that is, a term $\hat{\lambda} r.\ \{E\}$ where E is a monadic expression of type $\top$ in a context containing a single linear hypothesis r representing the process P. Because CLF's notion of equality identifies monadic expressions differing only in the order of execution of independent computation steps, each such object (modulo equality) represents the dependence graph of a possible computation. Thus judgments (represented by CLF types) concerning such objects, such as the abstraction judgment to be introduced in the next section, are predicates on depedence graphs, not on serialized computations.

There is no simple adequacy result at this stage, since the judgment $P \xrightarrow{s} P'$ of Gordon et al. refers to the trace s, which is not directly available in the CLF operational semantics. (Moreover, the process P' to which P evolves is only available in CLF implicitly as the set of leftover hypotheses consumed by the $\top$ introduction at the end of the CLF expression representing a computation.) Once traces and the abstraction judgment relating a computation to its traces are introduced, it will be possible to state a precise adequacy result.

5 Traces and abstraction

The syntax of the traces s mentioned in the judgment $P \xrightarrow{s} P'$ of Gordon et al. can be represented straightforwardly by LF techniques. Though we have left the label syntax unspecified, it is assumed that labels might depend on names

$$
\begin{array}{rl}
\mathsf{abst} & : \{\top\} \to \mathsf{tr} \to \mathsf{type}. \\
\\
\mathsf{abst\_nil} & : \mathsf{abst}\ E\ \mathsf{tnil}. \\
\mathsf{abst\_stop} & : \mathsf{abst}\ \{\mathsf{let}\ \{1\} = \mathsf{ev\_stop}\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\ \mathsf{in}\ \langle\rangle\}\ s \leftarrow \mathsf{abst}\ E\ s. \\
\mathsf{abst\_par} & : \mathsf{abst}\ \{\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev\_alt}\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\hat{}r_1\hat{}r_2\ \mathsf{in}\ \langle\rangle\}\ s \\
& \qquad \leftarrow (\Pi r_1.\, \Pi r_2.\, \mathsf{abst}\ (E\hat{}r_1\hat{}r_2)\ s). \\
\mathsf{abst\_repeat} & : \mathsf{abst}\ \{\mathsf{let}\ \{!r\} = \mathsf{ev\_repeat}\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\ r\ \mathsf{in}\ \langle\rangle\}\ s \\
& \qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s). \\
\mathsf{abst\_new} & : \mathsf{abst}\ \{\mathsf{let}\ \{[x, r]\} = \mathsf{ev\_new}\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\ x\hat{}r\ \mathsf{in}\ \langle\rangle\} \\
& \qquad\qquad (\mathsf{tgen}\ (\lambda x.\, s\ x)) \\
& \qquad \leftarrow (\Pi x.\, \Pi r.\, \mathsf{abst}\ (E\ x\hat{}r)\ (s\ x)). \\
\mathsf{abst\_choose}_i & : \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_choose}_i\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s) \\
& \qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s). \\
\mathsf{abst\_sync} & : \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_sync}\hat{}R_1\hat{}R_2\ \mathsf{in\ let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s) \\
& \qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s). \\
\mathsf{abst\_begin} & : \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_begin}\ L\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tbegin}\ L\ s) \\
& \qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s). \\
\mathsf{abst\_end} & : \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_end}\ L\hat{}R\ \mathsf{in\ let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tend}\ L\ s) \\
& \qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s).
\end{array}
$$

Fig. 4. The abstraction judgment as a CLF program

introduced in the course of the computation, so the actions gen representing the generation of fresh names in the execution of a new process must bind a variable in the style of higher-order abstract syntax.

The representation of traces is as follows:

$$
\begin{array}{rl}
\mathsf{tr} & : \mathsf{type}. \\
\\
\mathsf{tnil} & : \mathsf{tr}. \\
\mathsf{tint} & : \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tbegin} & : \mathsf{label} \to \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tend} & : \mathsf{label} \to \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tgen} & : (\mathsf{nm} \to \mathsf{tr}) \to \mathsf{tr}.
\end{array}
\qquad
\begin{array}{rl}
\ulcorner \varepsilon \urcorner & = \mathsf{tnil} \\
\ulcorner \tau, s \urcorner & = \mathsf{tint}\ \ulcorner s \urcorner \\
\ulcorner \mathsf{begin}\ L, s \urcorner & = \mathsf{tbegin}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\ulcorner \mathsf{end}\ L, s \urcorner & = \mathsf{tend}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\ulcorner \mathsf{gen}\ \langle x \rangle, s \urcorner & = \mathsf{tgen}\ (\lambda x. \ulcorner s \urcorner)
\end{array}
$$

Now we are equipped with enough tools to write the abstraction judgment relating a computation to its traces, as a CLF type family $\mathsf{abst} : \{\top\} \to \mathsf{tr} \to \mathsf{type}$, the logic program for which is shown in Figure 4. The first argument of this relation is the CLF object representing the dependence graph of the computation, while the second argument is an associated trace. The mode (in the sense of logic programming) is input for the first argument and output for the second. However, the relation is not a function, because from a single execution (as dependence graph) many possible (serial) abstractions as a trace might be extracted. Nevertheless, each execution has at least one abstraction as a trace.

It is also noteworthy that the context in which the abst judgment executes uses unrestricted hypotheses $r : \mathsf{run}\ P$ for each executing process P, whether

or not the corresponding process was represented by a linear hypothesis in the original execution. This is a common phenomenon when writing higher-level judgments in LLF style.

This judgment, taken together with CLF's equality admitting concurrency equations, defines for each concurrent computation the set of possible serializations of that computation as a trace. The traces need not describe the whole computation; the rule abst_nil allows abstraction to stop after computing the trace of any prefix of the computation. This suffices because we are interested only in *safety* properties, which are violated whenever they are violated on a prefix of the computation.

We would like to show that each traced transition $P \xrightarrow{s} P'$ of Gordon and Jeffrey's system corresponds to an object $\hat{\lambda} r.\, E : \mathsf{run}\ P \multimap \{\top\}$ as in Section 4 together with an abstraction $\mathsf{abst}\ E\ s$ yielding the same trace. As it turns out, this is not quite the case, because the structural equivalences considered in that paper induce certain rearrangements of tgen with respect to other actions that are not possible in the CLF variant. However, defining an appropriate notion of "similarity" on traces admitting rearrangement of tgen steps (which, moreover, can be characterized by another CLF judgment), we find that each traced transition is in correspondence with a CLF expression and abstraction yielding a "similar" trace.

The proof technique is illustrative but is not presented here in detail. In brief, one considers the notion of simulation $P_1 \preceq P_2$ induced by the CLF operational semantics of Section 4, abstraction, and "similarity" of traces: whenever P_1 and some context consisting of other processes and names yields a given trace, P_2 yields a similar trace in the same context. Then all the structural equivalences of the traced transition system are simulations in this sense, and it follows easily that each traced transition has its CLF counterpart. The converse is simple, because each rule of the CLF operational semantics is immediately available as a step of the traced transition system (or a structural equivalence). So we have:

Proposition 1 (Adequacy of operational semantics) *The traced transition system proves $P \xrightarrow{s} P'$ for some P' just when there exist $E : \mathsf{run}\ \ulcorner P \urcorner \multimap \{\top\}$ and $A : (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s')$ (in a context binding the free names of P and P'), and s is similar to s'.*

Finally, we can define the *safety criterion* for processes. In a constructive setting, it is easiest to characterize *unsafety*, because it is witnessed by finitary evidence. A process is unsafe precisely when it has an execution admitting some abstraction as a trace that violates the correspondence property (see Section 2). It turns out to be easy to write a CLF judgment characterizing those traces that violate the correspondence property (see Appendix A). Thus, each step of the criterion is modeled by a CLF judgment, and we can write an overall judgment $\mathsf{unsafe}\ P$, which, as a CLF type, contains all the proofs of unsafety of P. This turns out to be the same, mutatis mutandis, as Gordon

and Jeffrey's definition.

6 Related work

Right from its inception, linear logic [5] has been advocated as a logic with an intrinsic notion of state and concurrency. In the literature, many connections between concurrent calculi and linear logic have been observed. Due to space constraints we cannot survey this relatively large literature here. In a logical framework, we remove ourselves by one degree from the actual semantics; we represent rather than embed calculi. Thereby, CLF provides another point of view on many of the prior investigations.

Most closely related to our work is Miller's logical framework Forum [13], which is based on a sequent calculus for classical linear logic and focusing proofs [1]. As shown by Miller and elaborated by Chirimar [4], Forum can also represent concurrency. Our work extends Forum in several directions. Most importantly, it is a type theory based on natural deduction and therefore offers an internal notion of proof object that is not available in Forum. Among other things, this means we can explicitly represent relations on deductions and therefore on concurrent computations.

There have been several formalizations of versions of the π-calculus in a variety of reasoning systems, such as HOL [11], Coq [8,9], Isabelle/HOL [17] or Linc [14]. A distinguishing feature of our sample encoding in this paper is the simultaneous use of higher-order abstract syntax, linearity, modality, and the intrinsic notion of concurrent computations. Also, we are not aware of a formal treatment of correspondence assertions or dependent effect typing for the π-calculus.

Systems based on rewriting logic, such as Maude [12], natively support concurrent specifications (and have been used to model Petri nets, CCS, the π-calculus, etc). However, lacking operators comparable to CLF's dependent types and proof-terms, Maude users must code concurrent computations independently from the concurrent systems that originate them.

As already mentioned above, CLF is a conservative extension of LLF with the asynchronous connectives $\otimes$, 1, !, and $\exists$, encapsulated in a monad. The idea of monadic encapsulation goes back to Moggi's monadic meta-language [15] and is used heavily in functional programming. Our formulation follows the judgmental presentation of Pfenning and Davies [16] that completely avoids the need for commuting conversions, but treats neither linearity nor the existence of normal forms. This permits us to reintroduce some equations to model true concurrency in a completely orthogonal fashion.

7 Conclusions

The goal of this work has been to extend the elegant and logically motivated representation strategies for syntax, judgments, and state available in LF and

LLF to the concurrent world. We have shown how the availability of a *notation* for concurrent executions, admitting a proper truly concurrent equality, enables powerful strategies for specifying properties of such executions.

Ultimately, it should become as simple and natural to manipulate the objects representing concurrent executions as it is to manipulate LF objects. If higher-order abstract syntax means never having to code up α-conversion or capture-avoiding substitution ever again, we hope that in the same way, the techniques explored here can make it unnecessary to code up multiset equality or concurrent equality ever again, so that intellectual effort can be focused on reasoning about deeper properties of concurrent systems.

References

[1] Andreoli, J.-M., *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), pp. 197–347.

[2] Cervesato, I. and F. Pfenning, *A linear logical framework*, Information & Computation **179** (2002), pp. 19–75.

[3] Cervesato, I., F. Pfenning, D. Walker and K. Watkins, *A concurrent logical framework II: Examples and applications*, Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University (2002), revised May 2003.

[4] Chirimar, J. L., "Proof Theoretic Approach to Specification Languages," Ph.D. thesis, University of Pennsylvania (1995).

[5] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[6] Gordon, A. D. and A. Jeffrey, *Typing correspondence assertions for communication protocols*, Theoretical Computer Science **300** (2003), pp. 379–409.

[7] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery **40** (1993), pp. 143–184.

[8] Hirschkoff, D., *A full formalisation of pi-calculus theory in the Calculus of Constructions*, in: E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'97)* (1997), pp. 153–169.

[9] Honsell, F., M. Miculan and I. Scagnetto, *Pi-calculus in (co)inductive type theories*, Theoretical Computer Science **253** (2001), pp. 239–285.

[10] Ishtiaq, S. and D. Pym, *A relevant analysis of natural deduction*, Journal of Logic and Computation **8** (1998), pp. 809–838.

[11] Melham, T., *A mechanized theory of the pi-calculus in HOL*, Nordic Journal of Computing **1** (1995), pp. 50–76.

[12] Meseguer, J., *Software specification and verification in rewriting logic*, Lecture notes for the Marktoberdorf International Summer School, Germany (2002).

[13] Miller, D., *A multiple-conclusion meta-logic*, in: S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science* (1994), pp. 272–281.

[14] Miller, D. and A. Tiu, *A proof theory for generic judgments*, in: P. Kolaitis, editor, *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)* (2003), pp. 118–127.

[15] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[16] Pfenning, F. and R. Davies, *A judgmental reconstruction of modal logic*, Mathematical Structures in Computer Science **11** (2001), pp. 511–540, notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[17] Röckl, C., D. Hirschkoff and S. Berghofer, *Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts*, in: F. Honsell and M. Miculan, editors, *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)* (2001), pp. 364–378.

[18] Watkins, K., I. Cervesato, F. Pfenning and D. Walker, *A concurrent logical framework I: Judgments and properties*, Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University (2002), revised May 2003.

[19] Watkins, K., I. Cervesato, F. Pfenning and D. Walker, *A concurrent logical framework: The propositional fragment*, in: *Types for Proofs and Programs*, Springer-Verlag LNCS, 2004 Selected papers from the *Third International Workshop* Torino, Italy, April 2003. To appear.

[20] Woo, T. and S. Lam, *A semantic model for authentication protocols*, in: *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy* (1993), pp. 178–194.

A π-calculus encoding summarized

Syntax.

$$
\begin{array}{rcl}
\mathsf{eff} = \{1\} : \mathsf{type}. & & \\
\mathsf{latent} & : & \mathsf{label} \to \mathsf{eff}. \\
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner [L_1, \ldots, L_n] \urcorner & = & \\
\multicolumn{3}{r}{\{\mathsf{let}\ \{1\} = \mathsf{latent}\ \ulcorner L_1 \urcorner\ \mathsf{in}\ \ldots} \\
\multicolumn{3}{r}{\mathsf{let}\ \{1\} = \mathsf{latent}\ \ulcorner L_n \urcorner\ \mathsf{in}\ 1\}}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{name} & : & \mathsf{tp}. \\
\mathsf{chan} & : & \mathsf{tp} \to (\mathsf{nm} \to \mathsf{eff}) \to \mathsf{tp}. \\
\\
\mathsf{stop} & : & \mathsf{pr}. \\
\mathsf{par} & : & \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{repeat} & : & \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{new} & : & \mathsf{tp} \to (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{choose} & : & \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{out} & : & \mathsf{nm} \to \mathsf{nm} \to \mathsf{pr}. \\
\mathsf{inp} & : & \mathsf{nm} \to \mathsf{tp} \to \\
 & & \quad (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{begin} & : & \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{end} & : & \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}. \\
\\
\mathsf{tnil} & : & \mathsf{tr}. \\
\mathsf{tint} & : & \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tbegin} & : & \mathsf{label} \to \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tend} & : & \mathsf{label} \to \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tgen} & : & (\mathsf{nm} \to \mathsf{tr}) \to \mathsf{tr}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner \mathsf{Name} \urcorner & = & \mathsf{name} \\
\ulcorner \mathsf{Ch}(x{:}\tau)e \urcorner & = & \mathsf{chan}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner e \urcorner) \\
\\
\ulcorner \mathsf{stop} \urcorner & = & \mathsf{stop} \\
\ulcorner P \mid Q \urcorner & = & \mathsf{par}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{repeat}\ P \urcorner & = & \mathsf{repeat}\ \ulcorner P \urcorner \\
\ulcorner \mathsf{new}(x{:}\tau); P \urcorner & = & \mathsf{new}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner P \urcorner) \\
\ulcorner \mathsf{choose}\ P\ Q \urcorner & = & \mathsf{choose}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{out}\ x\langle y \rangle \urcorner & = & \mathsf{out}\ x\ y \\
\ulcorner \mathsf{inp}\ x(y{:}\tau); P \urcorner & = & \mathsf{inp}\ x\ \ulcorner \tau \urcorner\ (\lambda y. \ulcorner P \urcorner) \\
\\
\ulcorner \mathsf{begin}\ L; P \urcorner & = & \mathsf{begin}\ \ulcorner L \urcorner\ \ulcorner P \urcorner \\
\ulcorner \mathsf{end}\ L; P \urcorner & = & \mathsf{end}\ \ulcorner L \urcorner\ \ulcorner P \urcorner \\
\\
\ulcorner \varepsilon \urcorner & = & \mathsf{tnil} \\
\ulcorner \tau, s \urcorner & = & \mathsf{tint}\ \ulcorner s \urcorner \\
\ulcorner \mathsf{begin}\ L, s \urcorner & = & \mathsf{tbegin}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\ulcorner \mathsf{end}\ L, s \urcorner & = & \mathsf{tend}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\ulcorner \mathsf{gen}\ \langle x \rangle, s \urcorner & = & \mathsf{tgen}\ (\lambda x. \ulcorner s \urcorner)
\end{array}
$$

Dynamic semantics.

$$
\begin{array}{rcl}
\mathsf{ev\_stop} & : & \mathsf{run}\ \mathsf{stop} \multimap \{1\}. \\
\mathsf{ev\_par} & : & \mathsf{run}\ (\mathsf{par}\ P\ Q) \multimap \{\mathsf{run}\ P \otimes \mathsf{run}\ Q\}. \\
\mathsf{ev\_repeat} & : & \mathsf{run}\ (\mathsf{repeat}\ P) \multimap \{!\mathsf{run}\ P\}. \\
\mathsf{ev\_new} & : & \mathsf{run}\ (\mathsf{new}\ \tau\ (\lambda u.\ P\ u)) \multimap \{\exists u{:}\mathsf{nm}.\ \mathsf{run}\ (P\ u)\}. \\
\mathsf{ev\_choose}_i & : & \mathsf{run}\ (\mathsf{choose}\ P_1\ P_2) \multimap \{\mathsf{run}\ P_i\}. \\
\mathsf{ev\_sync} & : & \mathsf{run}\ (\mathsf{out}\ X\ Y) \multimap \mathsf{run}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\ P\ y)) \multimap \{\mathsf{run}\ (P\ Y)\}. \\
\mathsf{ev\_begin} & : & \Pi L{:}\mathsf{label}.\ \mathsf{run}\ (\mathsf{begin}\ L\ P) \multimap \{\mathsf{run}\ P\}. \\
\mathsf{ev\_end} & : & \Pi L{:}\mathsf{label}.\ \mathsf{run}\ (\mathsf{end}\ L\ P) \multimap \{\mathsf{run}\ P\}.
\end{array}
$$

Static semantics.

$$
\begin{aligned}
\mathsf{wflab} &: \mathsf{label} \to \mathsf{type}.\\[1ex]
\mathsf{wfeff} &: \mathsf{eff} \to \mathsf{type}.\\
\mathsf{wff\_eps} &: \mathsf{wfeff}\ \{1\}.\\
\mathsf{wff\_lat} &: \mathsf{wfeff}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\} \leftarrow \mathsf{wflab}\ L \leftarrow \mathsf{wfeff}\ E.\\[1ex]
\mathsf{wftp} &: \mathsf{tp} \to \mathsf{type}.\\
\mathsf{wf\_name} &: \mathsf{wftp\ name}.\\
\mathsf{wf\_chan} &: \mathsf{wftp}\ (\mathsf{chan}\ \tau\ (\lambda x.\, E\ x)) \leftarrow \mathsf{wftp}\ \tau \leftarrow (\Pi x.\, \mathsf{has}\ x\ \tau \to \mathsf{wfeff}\ (E\ x)).\\[1ex]
\mathsf{consume} &: \mathsf{eff} \to \mathsf{type}.\\
\mathsf{assume} &: \mathsf{eff} \to \mathsf{pr} \to \mathsf{type}.\\[1ex]
\mathsf{con\_eps} &: \mathsf{consume}\ \{1\} \circ\!\!- \top.\\
\mathsf{con\_join} &: \mathsf{consume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\}\\
&\qquad \circ\!\!-\mathsf{effect}\ L \circ\!\!-\ \mathsf{consume}\ E.\\[1ex]
\mathsf{ass\_eps} &: \mathsf{assume}\ \{1\}\ P \circ\!\!-\ \mathsf{good}\ P.\\
\mathsf{ass\_join} &: \mathsf{assume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\}\\
&\qquad \circ\!\!-(\mathsf{effect}\ L \multimap \mathsf{assume}\ E\ P).\\[1ex]
\mathsf{has} &: \mathsf{nm} \to \mathsf{tp} \to \mathsf{type}.\\
\mathsf{good} &: \mathsf{pr} \to \mathsf{type}.\\[1ex]
\mathsf{gd\_stop} &: \mathsf{good}\ \mathsf{stop} \circ\!\!-\ \top.\\
\mathsf{gd\_par} &: \mathsf{good}\ (\mathsf{par}\ P\ Q) \circ\!\!-\ \mathsf{good}\ P \circ\!\!-\ \mathsf{good}\ Q.\\
\mathsf{gd\_repeat} &: \mathsf{good}\ (\mathsf{repeat}\ P) \circ\!\!-\ \top \leftarrow \mathsf{good}\ P.\\
\mathsf{gd\_new} &: \mathsf{good}\ (\mathsf{new}\ \tau\ (\lambda x.\, P\ x)) \leftarrow \mathsf{wftp}\ \tau\\
&\qquad \circ\!\!-(\Pi x : \mathsf{nm}.\, \mathsf{has}\ x\ \tau \to \mathsf{good}\ (P\ x)).\\
\mathsf{gd\_choose} &: \mathsf{good}\ (\mathsf{choose}\ P\ Q) \circ\!\!-\ (\mathsf{good}\ P\ \&\ \mathsf{good}\ Q).\\
\mathsf{gd\_out} &: \mathsf{good}\ (\mathsf{out}\ X\ Y) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \leftarrow \mathsf{has}\ Y\ \tau\\
&\qquad \circ\!\!-\mathsf{consume}\ (E\ Y).\\
\mathsf{gd\_inp} &: \mathsf{good}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y))\\
&\qquad \leftarrow (\Pi y : \mathsf{nm}.\, \mathsf{has}\ y\ \tau \to \mathsf{assume}\ (E\ y)\ (P\ y)).\\
\mathsf{gd\_begin} &: \mathsf{good}\ (\mathsf{begin}\ L\ P) \circ\!\!-\ (\mathsf{effect}\ L \multimap \mathsf{good}\ P).\\
\mathsf{gd\_end} &: \mathsf{good}\ (\mathsf{end}\ L\ P) \circ\!\!-\ \mathsf{effect}\ L \circ\!\!-\ \mathsf{good}\ P.
\end{aligned}
$$

Abstraction.

$$\begin{aligned}
\mathsf{abst} &: \{\top\} \to \mathsf{tr} \to \mathsf{type}.\\
\\
\mathsf{abst_nil} &: \mathsf{abst}\ E\ \mathsf{tnil}.\\
\mathsf{abst_stop} &: \mathsf{abst}\ \{\mathsf{let}\ \{1\} = \mathsf{ev_stop}^\wedge R\ \mathsf{in\ let}\ \{_\} = E\ \mathsf{in}\ \langle\rangle\}\ s \leftarrow \mathsf{abst}\ E\ s.\\
\mathsf{abst_par} &: \mathsf{abst}\ \{\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev_alt}^\wedge R\ \mathsf{in\ let}\ \{_\} = E^\wedge r_1{}^\wedge r_2\ \mathsf{in}\ \langle\rangle\}\ s\\
&\qquad \leftarrow (\Pi r_1.\, \Pi r_2.\, \mathsf{abst}\ (E^\wedge r_1{}^\wedge r_2)\ s).\\
\mathsf{abst_repeat} &: \mathsf{abst}\ \{\mathsf{let}\ \{!r\} = \mathsf{ev_repeat}^\wedge R\ \mathsf{in\ let}\ \{_\} = E\ r\ \mathsf{in}\ \langle\rangle\}\ s\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s).\\
\mathsf{abst_new} &: \mathsf{abst}\ \{\mathsf{let}\ \{[x, r]\} = \mathsf{ev_new}^\wedge R\ \mathsf{in\ let}\ \{_\} = E\ x^\wedge r\ \mathsf{in}\ \langle\rangle\}\\
&\qquad\qquad (\mathsf{tgen}\ (\lambda x.\, s\ x))\\
&\qquad \leftarrow (\Pi x.\, \Pi r.\, \mathsf{abst}\ (E\ x^\wedge r)\ (s\ x)).\\
\mathsf{abst_choose}_i &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev_choose}_i{}^\wedge R\ \mathsf{in\ let}\ \{_\} = E^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s).\\
\mathsf{abst_sync} &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev_sync}^\wedge R_1{}^\wedge R_2\ \mathsf{in\ let}\ \{_\} = E^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E^\wedge r)\ s).\\
\mathsf{abst_begin} &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev_begin}\ L^\wedge R\ \mathsf{in\ let}\ \{_\} = E^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tbegin}\ L\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E^\wedge r)\ s).\\
\mathsf{abst_end} &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev_end}\ L^\wedge R\ \mathsf{in\ let}\ \{_\} = E^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tend}\ L\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E^\wedge r)\ s).
\end{aligned}$$

Safety.

$$\begin{aligned}
\mathsf{invalid} &: \mathsf{tr} \to \mathsf{type}.\\
\mathsf{remove} &: \mathsf{label} \to \mathsf{tr} \to \mathsf{tr} \to \mathsf{type}.\\
\neq &: \mathsf{label} \to \mathsf{label} \to \mathsf{type}.\\
\\
\mathsf{inval_end} &: \mathsf{invalid}\ (\mathsf{tend}\ _\ _).\\
\mathsf{inval_int} &: \mathsf{invalid}\ (\mathsf{tint}\ s) \leftarrow \mathsf{invalid}\ s.\\
\mathsf{inval_gen} &: \mathsf{invalid}\ (\mathsf{tgen}\ (\lambda x.\, s\ x)) \leftarrow (\Pi x.\, \mathsf{invalid}\ (s\ x)).\\
\mathsf{inval_begin} &: \mathsf{invalid}\ (\mathsf{tbegin}\ L\ s) \leftarrow \mathsf{remove}\ L\ s\ s' \leftarrow \mathsf{invalid}\ s'.\\
\\
\mathsf{rem_match} &: \mathsf{remove}\ L\ (\mathsf{tend}\ L\ s)\ s.\\
\mathsf{rem_nil} &: \mathsf{remove}\ L\ \mathsf{tnil}\ \mathsf{tnil}.\\
\mathsf{rem_int} &: \mathsf{remove}\ L\ (\mathsf{tint}\ s)\ (\mathsf{tint}\ s') \leftarrow \mathsf{remove}\ L\ s\ s'.\\
\mathsf{rem_gen} &: \mathsf{remove}\ L\ (\mathsf{tgen}\ (\lambda x.\, s\ x))\ (\mathsf{tgen}\ (\lambda x.\, s\ x))\\
&\qquad \leftarrow (\Pi x.\, \mathsf{remove}\ L\ (s\ x)\ (s'\ x)).\\
\mathsf{rem_begin} &: \mathsf{remove}\ L\ (\mathsf{tbegin}\ L'\ s)\ (\mathsf{tbegin}\ L'\ s') \leftarrow \mathsf{remove}\ L\ s\ s'.\\
\mathsf{rem_end} &: \mathsf{remove}\ L\ (\mathsf{tend}\ L'\ s)\ (\mathsf{tend}\ L'\ s') \leftarrow L \neq L' \leftarrow \mathsf{remove}\ s\ s'.\\
\\
\mathsf{invalid} &: \mathsf{tr} \to \mathsf{type}.\\
\mathsf{unsafe} &: \mathsf{pr} \to \mathsf{type}.\\
\\
\mathsf{show_unsafe} &: \Pi E : (\mathsf{run}\ P \multimap \{\top\}).\, \mathsf{unsafe}\ P \leftarrow (\Pi r.\, \mathsf{abst}\ (E^\wedge r)\ s) \leftarrow \mathsf{invalid}\ s.
\end{aligned}$$

B CLF type theory summarized

See the technical report [18] for further details.

Syntax.

$$
\begin{aligned}
K, L &::= \mathsf{type} \mid \Pi u{:}A.\, K \\
A, B, C &::= A \multimap B \mid \Pi u{:}A.\, B \mid A \,\&\, B \\
&\quad \mid \top \mid \{S\} \mid P \\
P &::= a \mid P\ N \\
S &::= \exists u{:}A.\, S \mid S_1 \otimes S_2 \mid 1 \mid\ !A \mid A \\
\Gamma &::= \cdot \mid \Gamma, u{:}A \\
\Delta &::= \cdot \mid \Delta, x \hat{:} A \\
\Sigma &::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\
N &::= \hat{\lambda} x.\, N \mid \lambda u.\, N \mid \langle N_1, N_2 \rangle \\
&\quad \mid \langle\rangle \mid \{E\} \mid R \\
R &::= c \mid u \mid x \mid R \hat{} N \mid R\ N \mid \pi_1 R \mid \pi_2 R \\
E &::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\
M &::= [N, M] \mid M_1 \otimes M_2 \mid 1 \mid\ !N \mid N \\
p &::= [u, p] \mid p_1 \otimes p_2 \mid 1 \mid\ !u \mid x \\
\Psi &::= p \hat{:} S, \Psi \mid \cdot
\end{aligned}
$$

Typing.

Judgments.

$$
\begin{array}{lcr}
\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind} & \Gamma; \Delta \vdash_\Sigma N \Leftarrow A & \vdash \Sigma\ \mathsf{ok} \\
\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} & \Gamma; \Delta \vdash_\Sigma R \Rightarrow A & \vdash_\Sigma \Gamma\ \mathsf{ok} \\
\Gamma \vdash_\Sigma P \Rightarrow K & \Gamma; \Delta \vdash_\Sigma E \leftarrow S & \Gamma \vdash_\Sigma \Delta\ \mathsf{ok} \\
\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type} & \Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S & \Gamma \vdash_\Sigma \Psi\ \mathsf{ok} \\
 & \Gamma; \Delta \vdash_\Sigma M \Leftarrow S &
\end{array}
$$

$$
\begin{aligned}
\mathsf{inst\_k}_A(u.\, K, N) &= K' \\
\mathsf{inst\_a}_A(u.\, B, N) &= B' \\
\mathsf{inst\_s}_A(u.\, S, N) &= S'
\end{aligned}
$$

Rules.

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \qquad \frac{\vdash \Sigma\ \mathsf{ok} \quad \cdot \vdash_\Sigma K \Leftarrow \mathsf{kind}}{\vdash \Sigma, a{:}K\ \mathsf{ok}} \qquad \frac{\vdash \Sigma\ \mathsf{ok} \quad \cdot \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash \Sigma, c{:}A\ \mathsf{ok}}$$

$$\frac{}{\vdash_\Sigma \cdot\ \mathsf{ok}} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash_\Sigma \Gamma, u{:}A\ \mathsf{ok}}$$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ \mathsf{ok}} \qquad \frac{\Gamma \vdash_\Sigma \Delta\ \mathsf{ok} \quad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\Gamma \vdash_\Sigma \Delta, x\hat{:}A\ \mathsf{ok}}$$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ \mathsf{ok}} \qquad \frac{\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type} \quad \Gamma \vdash_\Sigma \Psi\ \mathsf{ok}}{\Gamma \vdash_\Sigma p\hat{:}S, \Psi\ \mathsf{ok}}$$

Henceforth, it will be assumed that all judgments are considered relative to a particular fixed signature Σ, and the signature indexing each of the other typing judgments will be suppressed.

$$\frac{}{\Gamma \vdash \mathsf{type} \Leftarrow \mathsf{kind}}\ \mathsf{type}\mathbf{KF} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u{:}A \vdash K \Leftarrow \mathsf{kind}}{\Gamma \vdash \Pi u{:}A.\, K \Leftarrow \mathsf{kind}}\ \Pi\mathbf{KF}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \multimap B \Leftarrow \mathsf{type}}\ \multimap\mathbf{F} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u{:}A \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash \Pi u{:}A.\, B \Leftarrow \mathsf{type}}\ \Pi\mathbf{F}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \,\&\, B \Leftarrow \mathsf{type}}\ \&\mathbf{F} \qquad \frac{}{\Gamma \vdash \top \Leftarrow \mathsf{type}}\ \top\mathbf{F}$$

$$\frac{\Gamma \vdash S \Leftarrow \mathsf{type}}{\Gamma \vdash \{S\} \Leftarrow \mathsf{type}}\ \{\}\mathbf{F} \qquad \frac{\Gamma \vdash P \Rightarrow \mathsf{type}}{\Gamma \vdash P \Leftarrow \mathsf{type}}\ \Rightarrow\mathsf{type}\Leftarrow$$

$$\frac{}{\Gamma \vdash a \Rightarrow \Sigma(a)}\ a \qquad \frac{\Gamma \vdash P \Rightarrow \Pi u{:}A.\, K \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma \vdash P\ N \Rightarrow \mathsf{inst\_k}_A(u.\, K, N)}\ \Pi\mathbf{KE}$$

$$\frac{\Gamma \vdash S_1 \Leftarrow \mathsf{type} \quad \Gamma \vdash S_2 \Leftarrow \mathsf{type}}{\Gamma \vdash S_1 \otimes S_2 \Leftarrow \mathsf{type}}\ \otimes\mathbf{F} \qquad \frac{}{\Gamma \vdash 1 \Leftarrow \mathsf{type}}\ 1\mathbf{F}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u{:}A \vdash S \Leftarrow \mathsf{type}}{\Gamma \vdash \exists u{:}A.\, S \Leftarrow \mathsf{type}}\ \exists\mathbf{F} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type}}{\Gamma \vdash\, !A \Leftarrow \mathsf{type}}\ !\mathbf{F}$$

$$\frac{\Gamma; \Delta, x\hat{:}A \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \hat{\lambda}x.\, N \Leftarrow A \multimap B}\ \multimap\mathbf{I} \qquad \frac{\Gamma, u{:}A; \Delta \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \lambda u.\, N \Leftarrow \Pi u{:}A.\, B}\ \Pi\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash N_1 \Leftarrow A \quad \Gamma; \Delta \vdash N_2 \Leftarrow B}{\Gamma; \Delta \vdash \langle N_1, N_2 \rangle \Leftarrow A \,\&\, B}\ \&\mathbf{I} \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle \Leftarrow \top}\ \top\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta \vdash \{E\} \Leftarrow \{S\}}\ \{\}\mathbf{I} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow P' \quad P' \equiv P}{\Gamma; \Delta \vdash R \Leftarrow P}\ \Rightarrow\Leftarrow$$

$$\frac{}{\Gamma;\cdot \vdash c \Rightarrow \Sigma(c)}\ c \qquad \frac{}{\Gamma;\cdot \vdash u \Rightarrow \Gamma(u)}\ u \qquad \frac{}{\Gamma; x\hat{:}A \vdash x \Rightarrow A}\ x$$

$$\frac{\Gamma;\Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma;\Delta_2 \vdash N \Leftarrow A}{\Gamma;\Delta_1,\Delta_2 \vdash R\hat{}N \Rightarrow B}\ \multimap\mathbf{E} \qquad \frac{\Gamma;\Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma;\Delta \vdash \pi_1 R \Rightarrow A}\ \&\mathbf{E}_1$$

$$\frac{\Gamma;\Delta \vdash R \Rightarrow \Pi u\!:\!A.\, B \quad \Gamma;\cdot \vdash N \Leftarrow A}{\Gamma;\Delta \vdash R\ N \Rightarrow \mathsf{inst\_a}_A(u.\, B, N)}\ \Pi\mathbf{E} \qquad \frac{\Gamma;\Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma;\Delta \vdash \pi_2 R \Rightarrow B}\ \&\mathbf{E}_2$$

$$\frac{\Gamma;\Delta_1 \vdash R \Rightarrow \{S_0\} \quad \Gamma;\Delta_2; p\hat{:}S_0 \vdash E \leftarrow S}{\Gamma;\Delta_1,\Delta_2 \vdash (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) \leftarrow S}\ \{\}\mathbf{E} \qquad \frac{\Gamma;\Delta \vdash M \Leftarrow S}{\Gamma;\Delta \vdash M \leftarrow S}\ \Leftarrow\!\leftarrow$$

$$\frac{\Gamma;\Delta; p_1\hat{:}S_1, p_2\hat{:}S_2, \Psi \vdash E \leftarrow S}{\Gamma;\Delta; p_1 \otimes p_2\hat{:}S_1 \otimes S_2, \Psi \vdash E \leftarrow S}\ \otimes\mathbf{L} \qquad \frac{\Gamma;\Delta;\Psi \vdash E \leftarrow S}{\Gamma;\Delta; 1\hat{:}1, \Psi \vdash E \leftarrow S}\ 1\mathbf{L}$$

$$\frac{\Gamma, u\!:\!A;\Delta; p\hat{:}S_0, \Psi \vdash E \leftarrow S}{\Gamma;\Delta; [u,p]\hat{:}\exists u\!:\!A.\, S_0, \Psi \vdash E \leftarrow S}\ \exists\mathbf{L} \qquad \frac{\Gamma, u\!:\!A;\Delta;\Psi \vdash E \leftarrow S}{\Gamma;\Delta; !u\hat{:}!A, \Psi \vdash E \leftarrow S}\ !\mathbf{L}$$

$$\frac{\Gamma;\Delta \vdash E \leftarrow S}{\Gamma;\Delta;\cdot \vdash E \leftarrow S}\ \leftarrow\!\leftarrow \qquad \frac{\Gamma;\Delta, x\hat{:}A;\Psi \vdash E \leftarrow S}{\Gamma;\Delta; x\hat{:}A, \Psi \vdash E \leftarrow S}\ \mathrm{A}\mathbf{L}$$

$$\frac{\Gamma;\Delta_1 \vdash M_1 \Leftarrow S_1 \quad \Gamma;\Delta_2 \vdash M_2 \Leftarrow S_2}{\Gamma;\Delta_1,\Delta_2 \vdash M_1 \otimes M_2 \Leftarrow S_1 \otimes S_2}\ \otimes\mathbf{I} \qquad \frac{}{\Gamma;\cdot \vdash 1 \Leftarrow 1}\ 1\mathbf{I}$$

$$\frac{\Gamma;\cdot \vdash N \Leftarrow A \quad \Gamma;\Delta \vdash M \Leftarrow \mathsf{inst\_s}_A(u.\, S, N)}{\Gamma;\Delta \vdash [N, M] \Leftarrow \exists u\!:\!A.\, S}\ \exists\mathbf{I} \qquad \frac{\Gamma;\cdot \vdash N \Leftarrow A}{\Gamma;\cdot \vdash !N \Leftarrow !A}\ !\mathbf{I}$$

A Coq Library for Verification of Concurrent Programs

Reynald Affeldt [a,1] Naoki Kobayashi [b,2]

[a] *Department of Computer Science, University of Tokyo, Tokyo, Japan*

[b] *Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan*

Abstract

Thanks to recent advances, modern proof assistants now enable verification of realistic sequential programs. However, regarding the concurrency paradigm, previous work essentially focused on formalization of abstract systems, such as pure concurrent calculi, which are too minimal to be realistic. In this paper, we propose a library that enables verification of realistic concurrent programs in the Coq proof assistant. Our approach is based on an extension of the π-calculus whose encoding enables such programs to be modeled conveniently. This encoding is coupled with a specification language akin to spatial logics, including in particular a notion of fairness, which is important to write satisfactory specifications for realistic concurrent programs. In order to facilitate formal proof, we propose a collection of lemmas that can be reused in the context of different verifications. Among these lemmas, the most effective for simplifying the proof task take advantage of confluence properties. In order to evaluate feasibility of verification of concurrent programs using this library, we perform verification for a non-trivial application.

Key words: Coq, concurrent programs, π-calculus

1 Introduction

Concurrent programs are ubiquitous: multi-threaded programs in network servers, distributed programs for database applications, etc. In order to guarantee their correctness and security properties, it is important to verify them formally. The main difficulty in formally verifying concurrent programs is the size of their state space. The latter can be very large (because of non-determinism) and even infinite (for non-terminating applications, such as reactive systems).

[1] Email: affeldt@yl.is.s.u-tokyo.ac.jp

[2] Email: kobayasi@kb.cs.titech.ac.jp

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Proof assistants and model checkers can be regarded as complementary tools for formal verification. Model checkers are fully automated but can only handle finite state space systems (without appropriate abstraction techniques). Proof assistants are interactive but they can handle infinite state space systems directly, using inductive reasoning. In this paper, we are concerned with formal verification based on proof assistants.

Proof assistants have been applied successfully to the formal verification of sequential programs. There exist tools to enable practical verification of imperative programs (e.g., [11]). Proof assistants have been used to verify realistic programs (e.g., [3,24,2]).

Regarding concurrency, previous work using proof assistants has focused on abstract concurrent systems rather than on realistic concurrent programs. There are many formalizations of pure concurrent calculi (e.g., [14,10,15,25,26]) and experiments with the combined use of proof assistants and model checkers for minimal concurrent languages (e.g., [29,19]). This work demonstrates the usefulness of proof assistant-based formal verification for concurrent programs. However, the minimality of formalized calculi and languages makes it cumbersome to verify realistic concurrent programs. Moreover, in view of the large proof developments in previous work, it is even questionable whether such verifications can be done in practice. For these reasons, we think that formal verification of realistic concurrent programs has not yet been addressed satisfactorily.

In this paper, we introduce a library that enables verification of realistic concurrent programs using a general-purpose proof assistant, namely Coq [27]. This library consists of:

- A modeling language with attractive features for verification of realistic concurrent programs. This modeling language is based on the π-calculus [20] (a foundational language for the study of concurrent systems) but is different from encodings developed in previous work in that it allows Coq datatypes and control structures to be used. Consequently, it makes it easy to model realistic concurrent programs and to run these models using existing virtual machines and compilers.
- A specification language for realistic concurrent programs. In particular, it provides a notion of fairness that is necessary to write satisfactory specifications for realistic concurrent programs.
- A collection of lemmas in order to facilitate formal proof. The most effective lemmas are based on confluence properties. They allow for smaller formal proofs by reducing the state space that needs to be explored for the purpose of verification.

To evaluate the feasibility of verification of concurrent programs using our library, we have performed formal verification of an existing mail server.

Paper Outline

We explain the three parts of the library in turn (the modeling language, the specification language and the collection of lemmas) and then report on the case study. We use the syntax of Coq (version 7).

2 Modeling Language

In this section, we introduce (a Coq encoding of) a simple concurrent language that can be used to model a wide range of realistic concurrent programs. Simplicity and generality are inherited from the π-calculus, on which this modeling language is based. Because of its minimality, the (pure) π-calculus is not well-suited to modeling of realistic concurrent programs. The main reason is that datatypes and control structures (conditionals and functions) need to be encoded by means of the concurrent primitives. Our modeling language addresses this shortcoming by extending the π-calculus with datatypes and functions, similarly to the Pict programming language [23]. We call our modeling language applπ, which stands for "applied π-calculus" [3].

In Sect. 2.1 and Sect. 2.2 we discuss the encoding of the syntax and the operational semantics of applπ, respectively.

2.1 Syntax Encoding

The syntax of applπ consists of *channels* and *processes*. Intuitively, processes perform computations and exchange values with other processes through channels.

Channels are encoded by means of the functional type `chan`. Any type in `Set` can be used as a datatype for communicated values, and channels themselves can be communicated:

```
Axiom chan : Set -> Set.
```

Processes are encoded by means of the inductive type `proc`. Each constructor of `proc` corresponds to a concurrent primitive of the π-calculus [4]:

```
Inductive proc : Type :=
  zeroP: proc
| inP: (A:Set)(chan A) -> (A -> proc) -> proc
| rinP: (A:Set)(chan A) -> (A -> proc) -> proc
| outP: (A:Set)(chan A) -> A -> proc -> proc
| parP: proc -> proc -> proc
| nuP: (A:Set)((chan A) -> proc) -> proc.
```

[3] We introduce this abbreviation to avoid confusion with Abadi and Fournet's applied π-calculus [1] which is an extension of the π-calculus to study security protocols.

[4] The concurrent primitives of applπ are more precisely a subset of those of the π-calculus: replication is restricted to input processes and there is no external choice. These restrictions have little impact on expressiveness, as discussed in [23].

Intuitively, `zeroP` represents the inert process. `(inP c [x:A]P)` represents an input process: it waits for some value `v` of type `A` along the channel `c` and then behaves as process `([x:A]P v)`. `(outP c v P)` represents an output process: it sends the value `v` along the channel `c` and then behaves as process `P`. `(parP P Q)` represents the parallel composition of the processes `P` and `Q`. `(rinP c [x:A]P)` represents replicated input: it waits for some value `v` of type `A` along the channel `c` and then behaves as process `(parP (rinP c [x:A]P) ([x:A]P v))`. The process `(nuP [x:(chan A)]P)` represents channel creation: it creates a new channel `c'` and then behaves as the process `([x:(chan A)]P c')`. Processes are in the `Type` universe so that they cannot be sent as data.

This encoding allows the Coq language to be used as the functional core of applπ. This effect is achieved by *higher-order abstract syntax* (HOAS), an encoding technique used to ease the management of binders. Concretely, process continuations for input and channel creation primitives are taken to be Coq functions. Thus, one can use the Coq language to write applπ processes. Our encoding can be said to be a *deep embedding* because we define the syntax as an inductive type that we use in the next section to define the operational semantics. However, the ability to integrate Coq functions gives it also the flavor of a *shallow embedding*. (See for instance [21] or [25] for definitions.)

The use of dependent types guarantees that channels are used consistently according to their type. For instance, `(inP c [x:A]P)` is rejected by Coq if `c` has not type `chan A`. Without dependent types, we would have to introduce a sum type for values and insert explicit tagging/untagging to perform data emission and reception, what would make modeling in applπ cumbersome. The combined use of HOAS and dependent types makes our encoding different from previous work on encoding of the π-calculus in Coq.

The following definitions are used in the rest of the paper. They represent output and input processes without continuations:

```
Definition OutAtom [A:Set; x:(chan A); v:A] := (outP x v zeroP).
Definition InAtom [A:Set; x:(chan A)] := (inP x [x:A]zeroP).
```

Before discussing the formal operational semantics, we illustrate the practical advantages of applπ as a modeling language.

2.1.1 Modeling Realistic Concurrent Programs

By way of example, we show below how to model a simple client/server program.

The process below represents a simple server. It waits on the channel `i` for a request, more precisely a pair of a natural number and a channel. It computes the successor of the received natural number and sends it back using the received channel:

```
Definition server [i:(chan nat*(chan nat))]:proc :=
(rinP i [ar:?]
  let a = (Fst ar) in let r = (Snd ar) in
  (OutAtom r (plus a (1)))).
```

We observe that it is easy to write realistic programs because our encoding provides us with the Coq language and the Coq standard library (here: `let` construct; `plus`, `Fst`, `Snd` functions).

The process below represents a client for the above server. It sends a request and waits for the answer of the server along a channel it has created. Eventually, it displays the server response along the channel `o`:

```
Definition client [i:(chan nat*(chan nat));o:(chan nat)]:proc :=
(nuP [r:?]
  (parP (OutAtom i ((0),r)) (inP r [x:?](OutAtom o x)))).
```

The parallel composition `(parP (server i) (client i o))` models a simple client/server program. We discuss further modeling issues in our case study.

2.1.2 Executing applπ Models

It is possible to run applπ models with little modification by using the extraction facility of Coq. For instance, the server above can be turned into OCaml code:

```
Coq < Recursive Extraction server.
...
(* various OCaml data structures and functions, including
a datatype for concurrent primitives and the plus function *)
...
let server i =
  RinP (False, i, (fun ar -> OutP (True, (snd ar),
    (plus (fst ar) (S O)), ZeroP)))
```

To run that program using existing virtual machines or compilers, it is sufficient to replace the type constructors for concurrent primitives by OCaml functions with the appropriate semantics. For a sample OCaml module with such functions, see http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi.

This facility can be used to run applπ models as programs on their own. More radically, one can use applπ not as a modeling language but as a programming language, the Coq interface providing static type checking (a la polyadic π-calculus, thanks to our use of dependent types) and OCaml providing an efficient execution environment for formally verified programs.

The operational semantics of applπ is a relation between processes, which defines what it means for a process to execute actions such as data emission/reception and channel creation. Similarly to the syntax, the operational semantics is borrowed from the π-calculus. More precisely, it is a non-standard labeled transition semantics. Before explaining the encoding, we justify the need for a non-standard semantics.

Our use of HOAS makes it difficult to encode the standard semantics of the π-calculus. The difficulty comes from the fact that ν-bound channels and conditionals are handled at the meta-level in our syntax encoding (respectively by Coq variables and Coq case analysis). For illustration, let us consider the following applπ process:

```
(nuP [x:?](parP (inP x [_:?](OutAtom x v)) (OutAtom x v)))
```

Using a standard semantics, we would expect it to reduce by communication along channel `x` to the process:

```
(nuP [x:?](OutAtom x v))
```

It is difficult to write in Coq a rule to perform such reductions because the processes that are reduced are inside a meta-level λ-abstraction. Honsell et al. [15] solve this problem in their HOAS encoding of the (pure) π-calculus by introducing several artifacts. For instance, their encoding of the standard rule for channel creation requires an additional predicate to check occurrence of a channel in a process (predicate `notin` in the rule `fRES` in [15]). However, such a predicate cannot be defined for applπ because conditionals are represented by Coq case analysis (whereas they are represented by type constructors in [15]).

Our solution is to distinguish between channels already created and channels to be created. For this purpose, instead of considering sole processes, we consider *states*, i.e. pairs of a process with the list of the channels created so far. In a state, channels already created appear in the list and the channels to be created appear as ν-bound channel in the process. (In comparison, both kinds of channels are represented by ν-bound channels in standard semantics.) We denote by `L#P` the state composed of the list `L` and the process `P`, by `nilC` the empty list, and by `&` the addition of an element to a list. The applπ process above is rewritten into the state:

```
nilC#(nuP [x:?](parP (inP x [_:?](OutAtom x v)) (OutAtom x v)))
```

Using our non-standard semantics, it first creates a new channel `x'` to replace `x`:

```
x'&nilC#(parP (inP x' [_:?](OutAtom x' v)) (OutAtom x' v))
```

and then reduces by communication along channel `x'`:

```
x'&nilC#(OutAtom x' v)
```

Concretely, the operational semantics is encoded by means of two inductive

predicates `Trans` and `Redwith`.

`(Trans P l Q)` means that process `P` reduces to process `Q` by performing the elementary action `l` (of type `TrLabel`, representing either data emission, data reception, channel creation or communication). The formal definition of the `Trans` predicate is similar to standard labeled transition semantics except for the rule for channel creation:

```
Inductive Trans: proc-> TrLabel -> proc -> Prop :=
...
| tr_new: (A:Set)(C:(chan A)->proc)(x:(chan A))
        (Trans (nuP C) (NewL x) (C x))
...
```

`(Redwith S l S')` means that state `S` reduces to state `S'` by performing a communication or a channel creation (action of type `RedLabel`). In particular, it captures what it means for a channel to be new (or fresh): simply that it does not appear in the list of channels created so far.

```
Inductive Redwith: state -> RedLabel -> state -> Prop :=
...
| red_new: (L:ChanList)(P,Q:proc)(A:Set)(x:(chan A))
        (Trans P (NewL x) Q) -> (fresh x L) ->
                (Redwith L#P (New x) (x&L)#Q).
```

In the following, when `(Redwith S l S')` is true for some `l`, we write `(Red S S')`. We also write `Reds` for the reflexive, transitive closure of `Red`.

3 Specification Language

Specification of concurrent programs deals with questions such as reachability of desirable states. There are several specification languages (or logics) designed for that purpose, such as spatial logics [4] or Dam's π-μ-calculus [9]. The specification language provided in our library is based on Cardelli and Gordon's spatial logic [4], because we found it expressive enough for our purpose.

Concerning temporal formulas, an important issue developed in our specification language is formalization of strong fairness. Intuitively, strong fairness is a system property enjoyed by execution environments in which communications that can execute infinitely often are eventually scheduled for execution. It is an important assumption without which we cannot write satisfactory specifications for realistic concurrent programs. For instance, let us consider the following program:

```
(parP (parP (OutAtom d v) (inP d [_:?](OutAtom e v)))
      (parP (OutAtom c v) (rinP c [_:?](OutAtom c v))))
```

A property that one might want to check is that the process `(OutAtom e v)` is eventually revealed. However, without the fairness assumption, this property

does not even hold.

We show in Sect. 3.1 how we encode the fairness assumption and in Sect. 3.2 we give the semantics of the formulas of our specification language.

3.1 Encoding of the Fairness Assumption

Fairness is expressed by means of quantifications over runs of concurrent programs. We first explain how we encode runs.

3.1.1 Encoding of Runs

A run intuitively consists of a maximal sequence of successive reductions.

A *state sequence* is an indexed set of optional states. A *stable* state is a state that cannot evolve anymore.

```
Definition stateSeq : Type := nat -> (optionT state).
Definition Stable [S:state] : Prop := ~(EXT T:state | (Red S T)).
```

A *reduction sequence* is a state sequence such that each state is obtained by a reduction of its predecessor:

```
Definition isRedSeq [PS:stateSeq] : Prop :=
(n:nat)((S:state)(PS n)==(SomeT ? S) ->
        (EXT S':state | (PS (S n))==(SomeT ? S')/\(Red S S')) \/
        (PS (S n))==(NoneT ?)) /\
     ((PS n)==(NoneT ?) -> (PS (S n))==(NoneT ?)).
```

A *maximal reduction sequence* (or a *run*) is a reduction sequence whose last state is stable, or an infinite reduction sequence:

```
Definition isMaxRedSeq [PS:stateSeq] : Prop := (isRedSeq PS) /\
 ((n:nat)(P:state)
   ((PS n)==(SomeT ? P) -> (PS (S n))==(NoneT ?) -> (Stable P))).
```

One may observe that empty sequences are valid runs. In the encoding of formulas, we enforce the condition that a run starts with some state.

3.1.2 Encoding of Fairness

We formalize the notion of *strong fairness*. Informally, strong fairness says that any process that is infinitely often enabled is eventually reduced [5].

We need a few intermediate definitions. We say that `P` is a subprocess of `Q` when `Q` consists of the parallel composition of `P` with some other process(es). The predicate `(reduced P Q R)` intuitively means `Q` reduces to `R` by reducing its subprocess `P`. The formal definition is omitted for lack of space.

We define what it means for a subprocess to be *enabled* and *eventually reduced*:

[5] *Weak fairness* says that a continuously and infinitely enabled process is eventually reduced. Strong fairness subsumes weak fairness.

```
Definition enabled [P:proc; Q:state] : Prop :=
 (EXT R:state | (reduced P Q R)).
Definition ev_reduced [P:proc; PS:stateSeq] : Prop :=
 (EX n:nat | (EXT S:state | (EXT S':state |
    (PS n)==(SomeT ? S) /\ (PS (S n))==(SomeT ? S') /\
       (reduced P S S')))).
```

We define what it means for a property to hold *infinitely often*:

```
Definition is_postfix [PS',PS:stateSeq] : Prop :=
 (EX n:nat | (m:nat)(PS' m)==(PS (plus m n))).
Definition infinitely_often [p:state->Prop; PS:stateSeq] : Prop :=
 (PS':stateSeq)(is_postfix PS' PS) ->
      (EX n:nat | (EXT S:state | (PS' n)==(SomeT ? S) /\ (p S))).
```

A *fair reduction sequence* is a state sequence such that there is no process that is infinitely often enabled but never reduced:

```
Definition isFairRedSeq [PS:stateSeq] : Prop :=
   (PS':procSeq)(is_postfix PS' PS)->
        (P:proc)(infinitely_often [Q:state](enabled P Q) PS') ->
              (ev_reduced P PS').
```

3.2 Available Formulas

Our specification language consists of a set of logical and spatial formulas (of type `form`) and a set of temporal formulas (of type `tform`). The semantics of formulas is implemented by means of two satisfaction relations (`sat` of type `form->state->Prop` and `tsat` of type `tform->state->Prop`). The explicit distinction between logical and spatial formulas, and temporal formulas is required for the confluence properties introduced in the next section to hold. The informal semantics of basic formulas can be found in Table 1. Observe that we make use of a predicate `Cong` that encodes the standard notion of *structural congruence* (which intuitively relates processes that only differ by spatial rearrangements).

By way of example, we show the formal semantics of the `FMUSTEV` temporal formula. It is defined by quantification over all possible fair runs, as defined in the previous section:

```
Axiom FMUSTEV_satisfaction : (P:state)(f:form)
(tsat (FMUSTEV f) P) <->
 ((PS:stateSeq)(PS 0)==(SomeT ? P) ->
  (isMaxRedSeq PS) -> (isFairRedSeq PS) ->
  (EXT S:state | (EX n:nat | (PS n)==(SomeT ? S) /\ (sat f S)))).
```

In the implementation, the satisfaction relations are axiomatized. This is because the formula for negation does not respect the positivity constraints imposed by Coq. This problem has already been observed in [26]. This is not

	Logical Formulas	
`(sat ISANY S)`	iff	`True`
`(sat NEG f S)`	iff	`~(sat f S)`
`(sat OR f g S)`	iff	`(sat f S)\/(sat g S)`
	Spatial Formulas	
`(sat INPUTS c f L#P)`	iff	`(Cong P (parP (inP c Q) R))` and `(sat f L#(Q v))` for any `v`
`(sat OUTPUTS c v f L#P)`	iff	`(Cong P (parP (outP c v Q) R))` and `(sat f L#Q)`
`(sat CONSISTS f g L#P)`	iff	`(Cong P (parP Q R))` with `(sat f L#Q)` and `(sat g L#R)`
	Temporal Formulas	
`(tsat (MAYEV f) S)`	iff	for some run, there exists `S'` such that `(Reds S S')` and `(sat f S')`
`(tsat (FMUSTEV f) S)`	iff	for any fair run, there exists `S'` such that `(Reds S S')` and `(sat f S')`

Table 1
Basic Formulas

problematic as long as we do not study formally the properties of the formulas.

4 Collection of Lemmas

At this point, we are able to write a concurrent program `P`, a (temporal) property `f`, and we can try to prove `(tsat f P)` using Coq tactics. This direct approach is tedious because the Coq native tactics are too low-level and not adapted to the problem at hand. Our solution is to propose a collection of lemmas (and accompanying tactics) to facilitate formal proof.

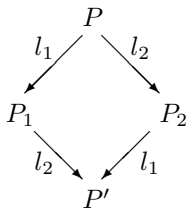
The main difficulty in proving properties of concurrent programs is non-determinism. In order to prove a property for some program, one often needs to check all possible runs. This is at best costly and often impossible because there may be infinitely many runs or because some process is unknown. To deal with these situations, we propose several lemmas based on confluence properties.

In Sect. 4.1, we explain lemmas based on confluence properties and in Sect. 4.2, we give an overview of the whole library.

4.1 Confluence Properties

4.1.1 Basic Idea

We say that two reductions are *confluent* when they can be executed in either order to reach the same result. More precisely, if P is a process such that $P \xrightarrow{l_1} P_1$ and $P \xrightarrow{l_2} P_2$ are confluent, then for any P' such that $P_2 \xrightarrow{l_1} P'$, we have $P_1 \xrightarrow{l_2} P'$. Graphically, P has the following "diamond property":



Since we know that, no matter the run, P necessarily reduces to P', it is not always necessary to explore both runs to verify a `FMUSTEV` property. This is the basic idea behind lemmas based on confluence properties.

4.1.2 Partial Confluence and Linearized Channels

In order to identify "diamond properties", we appeal to the notion of partial confluence, which is more general than confluence and often occurs in pratice.

We say that a reduction is *partially confluent* [18] when it is confluent with any other reduction. More precisely, if P is a process such that $P \xrightarrow{l_1} P_1$ is a partially confluent reduction, then for any reduction $P \xrightarrow{l_2} P_2$ and for any P' such that $P_2 \xrightarrow{l_1} P'$, we have $P_1 \xrightarrow{l_2} P'$.

The property of partial confluence is enjoyed by *linearized channels* [18]. A linearized channel is a generalization of a linear channel. It can be used more than once, but only in a sequential manner: an output process `(outP c v P)` can reuse `c` again for output in `P`, an input process `(inP c P)` can reuse `c` again for input in `(P v)` for any `v` of the appropriate type.

We introduce linearized channels in applπ by adding some boolean information to the type of channels `chan`:

```
Axiom chan : Set -> bool -> Set.
```

and by adding a new constructor to the type of processes `proc`:

```
Inductive proc : Type :=
...
| nuP: (A:Set)((chan A false)->proc) -> proc (* non-linearized *)
| nuPl: (A:Set)((chan A true)->proc) -> proc. (* linearized *)
```

The operational semantics is modified accordingly.

For the time being, we assume that linearized channels are correctly annotated. Verification that a process is well-annotated can be done by the type system proposed in [17], Sect. 6.

4.1.3 Sample Confluence Property

The following example is taken from our library:

```
Axiom conf_red_com :
 (L:ChanList; P,P':proc; A:Set; c:(chan A true))
 (well_annotated L#P) ->
 (Redwith L#P (epsilon c) L#P')->
 (f:form)(K:ChanList)(free_chans K f) ->
 ~(in_ChanList c K) ->          (* f does not depend on the     *)
 (M:ChanList)(guard M L#P') -> (* channels that are consumed   *)
 (inter K M nilC) ->            (* or revealed by communication *)
 (tsat (FMUSTEV f) L#P')->
 (tsat (FMUSTEV f) L#P).
```

Intuitively, it says that if `L#P` reduces to `L#P'` by a linearized communication, then in order to prove `(tsat (FMUSTEV f) L#P)`, it is sufficient to prove `(tsat (FMUSTEV f) L#P')` (modulo some conditions that we do not explain here in detail for lack of space).

Currently, these lemmas are axiomatized. They are similar to partial order reduction techniques used in model checking and can be informally justified as such (see for instance [6], Chap. 10).

4.2 Library Overview

The library consists of the applπ language as defined in Sect. 2 (extended with linearized channels), the specification language as defined in Sect. 3, and a collection of lemmas. Although the library is very large (at the time of this writing, 35 proof scripts, 11178 commands for 14951 lines), only a few lemmas are axiomatized (most axioms have actually been discussed in this paper). Not all lemmas are equally important.

During formal proof, the most important lemmas are those that simplify the goal. For instance, confluence properties such as the one seen above are such lemmas: they basically act by simplifying the process that appears in the goal. Similarly, the properties of the formulas of the specification language (distributivity laws, etc.) act by simplifying the formula that appears in the goal.

There are a large number of lemmas that are not intended to be used directly during formal proof but that are very important because they are ubiquitously used to prove other lemmas. Such technical lemmas prove properties about the applπ language (injection, inversion) whose proofs are not immediate because of our use of dependent types, and properties about structural congruence (structural congruence is a bisimulation, etc.).

See http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi for details.

5 Case Study

We evaluate feasibility of verification of concurrent programs using our library. Our case study is the SMTP receiver part of an existing mail server. In short, this program receives and processes SMTP commands, sends back SMTP replies and queues received electronic mail.

We chose this application for the purpose of comparison. Indeed, we have already performed verification of this application in Coq using a different approach that consists of building a faithful functional model [2]. In short, the original Java implementation was turned into a Coq function using monadic style programming, third-party programs (client and file system) were modeled using Coq predicates and non-software aspects were modeled using functional constructs (for instance, non-deterministic system failures were modeled using infinite lists to serve as test oracles). Arguably, this approach has little overhead because it takes advantage of the Coq built-in support for functional programs. Therefore, comparison should highlight the overhead of using our library for verification.

In the following, we first explain how we model the mail server using our library, and then we comment on the formal proof that it correctly implements the SMTP protocol.

See http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/applpi for details.

Modeling of the Main Program

The mail server is modeled as a process (`work`) that is itself the parallel composition of several subprocesses that handle incoming SMTP requests (`get_helo_def`, etc.). The state of the application is reified as a data structure that is communicated from one subprocess to the other. The flow of communication reproduces the flow of control of the Java program. Subprocesses correspond to the Java methods that (are supposed to) implement the SMTP protocol. The reified state corresponds to fields of the server object:

```
Definition work
   [c1:InputStream;c2:OutputStream;tofs:ToFileSystem]:proc :=
let st = initial_state in
   (nuPl [heloc:(chan STATE ?)]
   (nuPl [mailc:(chan STATE ?)]
   (nuPl [rcptc:(chan STATE ?)]
        (parP (rinP heloc (get_helo_def heloc mailc))
        (parP (rinP mailc (get_mail_def mailc rcptc))
        (parP (rinP rcptc (get_rcpt_def mailc rcptc))
        (OutAtom heloc st)))))).
```

Modeling benefits from the fact that applπ is based on the π-calculus. The connections between the mail server and third-party programs (client and file-system) are modeled using channels (instead of sockets in the original Java

implementation) that are aggregated into the reified state and "move" around with the state during computation. Acknowledgments are modeled by a typical π-calculus idiom: a fresh channel is sent to receive the acknowledgment on it.

Modeling of Third-party Programs

Third-party programs are modeled by Coq predicates. For instance, the client is modeled by predicates (`speaks_valid_protocol` and `acknowledges_replies`) that implement the SMTP protocol as defined in RFC 821.

Modeling of System Errors

System errors are modeled by channels. A system failure (resp. a network error) is modeled by outputting some value along the channel `system_failure_chan` (resp. `IOexn_chan`). Since non-determinism is inherent to applπ, we can model non-deterministic system failures by a process:

```
Definition may_fail :=
(nuP [x:?](parP (InAtom x)
          (parP (OutAtom x tt)
                (inP x [_:?](OutAtom system_failure_chan tt))))).
```

This is more elegant than infinite lists that serve as test oracles in the functional model discussed above. Indeed, the process `may_fail` is clearly separated from the model of the main program, so that we can extract an ML program for the server without pollution from the modeling of system errors.

Formal Proof

We have formally proved that the parallel composition of the mail server, a valid client, a valid file-system, and a non-deterministic system failures generator ends up with a successful termination (modeled by channel `result_chan`, similarly to system errors), a system failure or a network error (formula `reports_succ_or_error`):

```
Definition reports_succ_or_error : form :=
 (OR (OUTPUTS result_chan tt ISANY)
   (OR (OUTPUTS IOexn_chan tt ISANY)
     (OUTPUTS system_failure_chan tt ISANY))).

Goal
  (Client:InputStream->OutputStream->proc)
    (s:InputStream)(y:OutputStream)(valid_client (Client s y))->
  (file_system:ToFileSystem->proc)
    (tofs:ToFileSystem)(valid_fs tofs (file_system tofs)) ->
  (* channels for termination detection are distinct *)
  (is_set result_chan&(IOexn_chan&(system_failure_chan&nilC)))->
(tsat (FMUSTEV reports_succ_or_error)
```

```
(result_chan&(IOexn_chan&(system_failure_chan&nilC)))#
(nuPl [s1:InputStream]
(nuPl [s2:OutputStream]
(nuPl [tofs:ToFileSystem]
    (parP (Client s1 s2) (parP (file_system tofs)
    (parP (work s1 s2 tofs) may_fail)))))).
```

Verification using our library requires 3927 commands. This is large compared to the 1059 commands required by the verification using the functional model. However, there are several ways to reduce the size of the proof. In particular, we used for verification a confluence property that is weaker (but easier to use in practice) than the one presented in Sect. 4.1. Also, it should be observed that the applπ model is more satisfactory than the functional model in many respects: the `work` process takes multi-threading into account and it can easily be run as an ML program, which was not the case of the functional model.

6 Conclusion

In this paper, we proposed a Coq library to verify realistic concurrent programs. We have formalized a modeling language based on the π-calculus that is convenient to write and run (models of) realistic concurrent programs. We have introduced a specification language based on spatial logics extended with the notion of strong fairness. In order to facilitate formal proof, we have built a collection of lemmas among which confluence properties of the modeling language significantly simplify proofs. We have evaluated the feasibility of our approach by verifying a non-trivial application using our library.

Related Work

There exist several formalizations of pure concurrent calculi in proof assistants. In Coq, Hirschkoff proposes a first-order abstract syntax encoding of the π-calculus and formalizes proof techniques [14]; Despeyroux formalizes a proof of subject reduction for the π-calculus [10]; Honsell et al. formalize the foundational paper on the π-calculus [15]; Scagnetto and Miculan formalize the ambient calculus (a derivative of the π-calculus) and its spatial logic [26]. In Isabelle, Röckl et al. propose a HOAS encoding of the π-caculus and formalize the Theory of Contexts [25]. This work focuses on the formalization of the theory of pure concurrent calculi. In contrast, we are concerned with verification of realistic concurrent programs and we aim at building a practical library for that purpose.

The verification of concurrent programs using proof assistants is also addressed using the UNITY formalism. In particular, there exist several formalizations of compositional reasoning (e.g., [13,22]) that is useful to tackle realistic examples. Our work is complementary: we use the π-calulus as an

underlying formalism and therefore we can benefit from known analyses to formalize additional proof techniques (e.g., lemmas based on confluence properties).

Watkins et al. propose a logical framework [28] with built-in facilities for reasoning about concurrency. Using this concurrent logical framework (CLF), Cervesato et al. encode several concurrent systems [5], including the π-calculus. Although the authors have not addressed directly the issue of verification of realistic concurrent programs, it seems that an implementation of CLF may ease the development of a library similar to ours.

Coupet-Grimal [7] proposes an encoding of linear temporal logic in Coq. Temporal formulas are defined for an abstract transition system and their properties are collected into a library that has been used to prove correctness of a garbage-collection algorithm [8]. It would be useful to integrate similar reasoning on temporal formulas for our language.

Future Work

As stated in Sect. 4.1, we assume that channels are annotated so as to reflect partial confluence. In order to verify that channels are correctly annotated, we plan to formalize an adequate type system inside Coq, similarly to the work by Gay [12] who formalizes the type system of Kobayashi et al. [18] in Isabelle. We also plan to provide mechanical proofs for the lemmas based on confluence properties that are axiomatized for the time being.

We have been formalizing new formulas (such as fixed points) to enhance expressiveness but they are not yet integrated in the library.

In order to reduce the size of formal proofs, we are improving automation and investigating additional proof techniques based on other type systems, such as Kobayashi's type system for lock-freedom [16].

References

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 104–115, Jan. 2001.

[2] Reynald Affeldt and Naoki Kobayashi. Formalization and verification of a mail server in Coq. In Mitsuhiro Okada, Benjamin Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *International Symposium on Software Security, Tokyo, Japan, November 8–10, 2002*, volume 2609 of *Lecture Notes in Computer Science*, pages 217–233. Springer, Feb. 2003.

[3] Paul E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Department of Computer Science, Brigham Young University, Feb. 1998.

[4] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 365–377, 2000.

[5] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Mar. 2002. Revised May 2003.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[7] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[8] Solange Coupet-Grimal and Catherine Nouvet. Formal verification of an incremental garbage collector. *Journal of Logic and Computation*, 13(6):815–833, 2003.

[9] Mads Dam. *Logic for Concurrency and Synchronisation*, chapter Proof Systems for the pi-calculus Logics. Kluwer Academic Publishers, 2003.

[10] Joëlle Despeyroux. A higher-order specification of the π-calculus. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 425–439. Springer, Aug. 2002.

[11] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003.

[12] Simon J. Gay. A framework for the formalisation of pi calculus type systems in Isabelle/HOL. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, Edinburgh, Scotland, UK*, volume 2152 of *Lecture Notes in Computer Science*, pages 217–232. Springer, Sep. 2001.

[13] Barbara Heyd and Pierre Crégut. A modular coding of UNITY in Coq. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 251–266. Springer, Aug. 1996.

[14] Daniel Hirschkoff. *Mise en œuvre de preuves de bisimulation*. PhD thesis, École Nationale des Ponts et Chaussées, 1999.

[15] Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, Feb. 2001.

[16] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, Sep. 2002.

[17] Naoki Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 10th Anniversary Colloquium, March 2002, Lisbon, Portugal*. Springer-Verlag, 2002. Tutorial.

[18] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'96)*, pages 358–371. ACM Press, 1996.

[19] Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, Austin, TX, Aug. 2001.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, Sep. 1992.

[21] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[22] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 23(5):626–656, 2001.

[23] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[24] Benjamin C. Pierce and Jérôme Vouillon. Specifying a file synchronizer (full version). Draft, Mar. 2002.

[25] Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In *FOSSACS'01*, number 2030 in Lecture Notes in Computer Science. Springer, 2001.

[26] Ivan Scagnetto and Marino Miculan. Ambient calculus and its logic in the calculus of inductive constructions. In Frank Pfenning, editor, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.

[27] The Coq Development Team, LogiCal Project. *The Coq Proof Assistant, Reference Manual*. INRIA, 2002.

[28] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Mar. 2002. Revised May 2003.

[29] Shen-Wei Yu. *Formal Verification of Concurrent Programs Based on Type Theory*. PhD thesis, Department of Computer Science, University of Durham, Oct. 1998.

Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic

Richard Bubel [1] Andreas Roth [2] Philipp Rümmer [3]

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, Germany

Abstract

The interactive theorem prover developed in the KeY project, which implements a sequent calculus for JavaCard Dynamic Logic (JavaCardDL) is based on taclets. Taclets are lightweight tactics with easy to master syntax and semantics. Adding new taclets to the calculus is quite simple, but poses correctness problems. We present an approach how derived (non-axiomatic) taclets for JavaCardDL can be proven sound in JavaCardDL itself. Together with proof management facilities, our concept allows the safe introduction of new derived taclets while preserving the soundness of the calculus.

Key words: taclets, lightweight tactics, dynamic logic, theorem proving

1 Introduction

Background

Taclets are a new approach for constructing powerful interactive theorem provers [**?**]. First introduced as *Schematic Theory Specific Rules* [**?**], they are an efficient and convenient framework for lightweight tactics. Their most important advantages are the restricted and, thus, easy to master syntax and semantics compared to an approach based on meta languages like ML, and their seamless integration with graphical user interfaces of theorem provers which they can be efficiently compiled [**?**] into.

Taclets contain three kinds of information, the logical content of the rule to be applied, information about side-conditions on their applicability, and pragmatic information for interactive and automatic use. Due to their easy

[1] Email:bubel@ira.uka.de
[2] Email:aroth@ira.uka.de
[3] Email:ruemmer1@ira.uka.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

syntax and intuitive operational semantics, a person with some familiarity in formal methods should be able to write own taclets after a short time of study.

The interactive theorem prover developed in the KeYproject [?] is based on taclets implementing a sequent calculus for JavaCard Dynamic Logic (*JavaCardDL*) [?]. *JavaCard* is a subset of *Java* lacking multi-threading, garbage collection and graphical user interfaces, but with additional features like transactions.

JavaCardDL has around three hundred axiomatic rules, this means taclets that capture the *JavaCard* semantics. Correctness of rules is crucial since new taclets can be introduced quite easily. The work presented here ensures the correctness of derived taclets for *JavaCardDL* by providing means to prove them correct relatively to the core set of *JavaCardDL* axioms (possibly enriched with further axioms for certain theories). The soundness of taclets is proven in the calculus itself and without escaping to higher order logics. By our work, we extend an approach already described in [?] for classical first-order logic to *JavaCardDL*.

Related Work

Related to our approach are other projects for program verification like Bali [?,?], where consistence and correctness of rules that cover the Java semantics are ensured using Isabelle, or the LOOP project [?] where PVS is used as foundation, and the calculus rules are thus obtained as higher order logic theorems. Complementary to the presented approach—ensuring correctness for *derived* taclets—further work has been carried out in the KeY project in order to cross-validate a few selected axiomatic rules against the *Java* axiomatisation of Bali [?].

Structure of this Paper

In Sect. 1.1 we repeat the most important concepts of classical dynamic logics and *JavaCardDL*. A formal description of taclets and a definition of the basic vocabulary used throughout the paper is given in Sect. 2. The different steps to be performed in order to prove the correctness of derived taclets are described in Sect. 3–5. In Sect. 6 we give a justification of the complete procedure as main theorem. Finally, in Sect. 7 we discuss the current and future work to be done.

1.1 Dynamic Logic

Classical Dynamic Logics

The family of dynamic logics (DL) [?] belongs to the class of multi modal logics. As programs are first-class citizens of DL formulas, they are well-suited for program analysis and reasoning purposes. For the sake of simplicity and as a consequence of using a non-concurrent and real world programming language, we will only consider deterministic programs.

Let p be an arbitrary program and ϕ a first-order or dynamic logic formula, then

- $\langle \mathrm{p} \rangle \phi$ ("diamond p ϕ"): p terminates and after the execution of p formula ϕ holds
- $[\mathrm{p}]\phi$ ("box p ϕ"): if p terminates then after the execution of p formula ϕ holds

are typical representatives of DL formulas. Deterministic propositional dynamic logic (DPDL) is defined over a signature $\Sigma = (At_0, Prg_0, Op)$, where At_0, Prg_0 are enumerable sets of propositional variables and atomic programs respectively. Besides the classical propositional operators $\neg, \rightarrow$ the operator set Op contains box $[\mathrm{p}]$ and diamond $\langle \mathrm{p} \rangle$ modalities for each program p. The set of formulas is the smallest set defined inductively over At_0 and Prg_0:

- all classical propositional formulas are formulas in DPDL
- if ϕ, ψ are DPDL formulas then $\phi \rightarrow \psi$ and $\neg\phi$ are DPDL formulas
- if $\mathrm{p} \in Prg$ is a program and ϕ a formula in PDL then $\langle \mathrm{p} \rangle \phi$ and $[\mathrm{p}]\phi$ are DPDL formulas
- the set Prg of programs is the smallest set satisfying

(i) $Prg_0 \subseteq Prg$

(ii) if $\mathrm{p}, \mathrm{q} \in Prg$ and $\psi \in$ DPDL then
'p;q' (concatenation), '**if** (ψ) {p} **else** {q}' and '**while** (ψ) {p}' are programs.

The semantics can be defined in terms of Kripke frames $(S, (\rho_\mathrm{p})_{\mathrm{p} \in Prg})$ with a set S of states, and transition relations $\rho_\mathrm{p} : S \rightarrow S$ which define the semantics of each program $\mathrm{p} \in Prg$. The relations ρ_p have to adhere certain conditions w.r.t. the program constructors (;, **if**−**else**, etc.) from the definition above, for example, program composition $\rho_{\mathrm{p};\mathrm{q}} = \rho_\mathrm{q} \circ \rho_\mathrm{p}$.

An excerpt from an axiom system for DPDL in terms of sequent calculus rules is given in Table 1.

DPDL is useful to reason about program properties induced by the program constructors. However, as a consequence of constructing programs from atomic (anonymous) programs without any fixed semantics, they lack possibilities to talk about individual programs and, thus, about functional properties.

Like the step from propositional to first-order logic, one extends DPDL to deterministic quantified dynamic logics (DQDL). DQDL extends the propositional part to full first-order logic (with equality and universe D), and on the program side it replaces the anonymous atomic programs with assignments of the form v=t, where v is a variable and t an arbitrary term. In general, each program state $s \in S$ is assigned a first order structure (D, I) and a variable valuation $\sigma : Var \rightarrow D$ respecting $\rho_{x=t}(s) = s'$ with $\sigma' = \sigma_x^{t^{(D,I),\sigma}}$.

Again a relatively [4] complete calculus can be given, the corresponding

[4] Usually DQDL is interpreted in an arithmetic structure.

$$\frac{\Gamma \vdash \langle \textbf{if } (\psi) \ \{\mathrm{p};\ \textbf{while } (\psi) \ \{\mathrm{p}\}\ \} \textbf{ else } \{\}\rangle\phi, \Delta}{\Gamma \vdash \langle \textbf{while } (\psi) \ \{\mathrm{p}\}\rangle\phi, \Delta} \quad (1)$$

$$\frac{\Gamma, \psi \vdash \langle \mathrm{p}\rangle\phi,\ \Delta \quad \Gamma, \neg\psi \vdash \langle \mathrm{q}\rangle\phi,\ \Delta}{\Gamma \vdash \langle \textbf{if } (\psi) \ \{\mathrm{p}\} \textbf{ else } \{\mathrm{q}\}\rangle\phi,\ \Delta} \quad (2)$$

$$\frac{\Gamma \vdash \langle \mathrm{p}\rangle\langle \mathrm{q}\rangle\phi,\ \Delta}{\Gamma \vdash \langle \mathrm{p};\mathrm{q}\rangle\phi,\ \Delta} \quad (3) \qquad \frac{\Gamma^{\{\mathrm{x}\leftarrow\mathrm{z}\}}, \mathrm{x} \doteq \mathrm{t}^{\{\mathrm{x}\leftarrow\mathrm{z}\}} \vdash \phi, \Delta^{\{\mathrm{x}\leftarrow\mathrm{z}\}}}{\Gamma \vdash \langle \mathrm{x}{=}\mathrm{t}\rangle\phi, \Delta} \quad (4)$$

Table 1
DPDL/DQDL Axiomatisation (excerpt). z is a new variable.

assignment rule is shown in Table 1.

Example 1.1 *For the universe $D = \mathbb{N}$ of natural numbers, the DQDL formula* $\langle \mathrm{x}{=}3;;\mathrm{y}{=}\mathrm{x};\rangle \mathrm{y} \doteq \mathrm{x}$ *can be proven valid with the rules of Table 1 as the proof on the right shows.*

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{\mathrm{x} \doteq 3, \mathrm{y} \doteq x \vdash \mathrm{y} \doteq \mathrm{x}}\,(close)}{\mathrm{x} \doteq 3 \vdash \langle \mathrm{y}{=}\mathrm{x};\rangle \mathrm{y} \doteq \mathrm{x}}\,(4)}{\vdash \langle \mathrm{x}{=}3;\rangle\langle \mathrm{y}{=}\mathrm{x};\rangle \mathrm{y} \doteq \mathrm{x}}\,(4)}{\vdash \langle \mathrm{x}{=}3;\mathrm{y}{=}\mathrm{x};\rangle \mathrm{y} \doteq \mathrm{x}}\,(3)$$

JavaCardDL

The step from academic languages as described in the previous paragraphs to real world programming languages like *JavaCard* [**?**,**?**] leads to several complications. In the next few paragraphs, we introduce some features of *JavaCardDL* [**?**]. First some preliminaries:

- Formulas must not occur in *JavaCardDL* programs, instead *Java* expressions of type **boolean** are used as guards.
- The set of variables $Var = PVar \uplus LVar$ is the disjoint union of program variables *PVar* and logical variables *LVar*. In contrast to logical variables, program variables can occur in programs as well as in formulas, but cannot be bound by quantifiers. For instance, let $x \in LVar$ and $\mathrm{o}, \mathrm{u} \in PVar$, then $\forall x.\langle \mathrm{o}{=}\mathrm{u};\rangle x \doteq \mathrm{u}$ is a well-formed *JavaCardDL* formula, whereas $\forall x.\langle \mathrm{o}{=}x;\rangle x \doteq \mathrm{u}$ is not.
- All states have the same universe D, and predicates and logical variables are assumed to be rigid.

A sequent calculus covering *JavaCard* has to cope with aliasing, side-effects, abrupt termination as result of thrown exceptions, **break**s, **continue**s or **return**s and more. The KeY approach is led by the symbolic execution paradigm, thus a majority of the calculus rules realises a *JavaCard* interpreter reducing expressions and statements stepwise to side-effect free assignments.

Example 1.2 *An* easy-to-use *decomposition rule similar to (3) is not available in JavaCardDL due to abrupt termination. For example*

$$\vdash \langle \mathrm{l}:\{\ \textbf{if}\ (\mathrm{v} == 0)\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \textbf{else}\ \{\ \mathrm{v} = 0;\ \}\ \mathrm{v} = 3;\}\ \rangle \mathrm{v} \doteq 3$$

cannot be decomposed to

$$\vdash \langle \mathrm{l}:\{\ \textbf{if}\ (\mathrm{v} == 0)\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \textbf{else}\ \{\ \mathrm{v} = 0;\ \}\ \}\rangle\langle \mathrm{v} = 3\rangle \mathrm{v} \doteq 3,$$

as this is obviously not equivalent for $\mathrm{v} = 0$.

Decomposition was essential for DPDL and DQDL in order to reduce the complexity of programs stepwise to atomic programs or assignments, which can be handled by rules of the calculus without having a dedicated rule for each program.

JavaCardDL therefore introduces the notion of a *first active statement* to which a rule applies, and a program context '.. $\circ_1$ ...' whose inactive prefix '..' matches on all preceding labels, opening braces or **try** blocks. Consider the following rule:

$$\frac{\#\mathsf{b} \doteq \textbf{true} \vdash \langle\ ..\ \{\ \#\mathsf{sta1}\}\ ...\rangle\phi \qquad \#\mathsf{b} \doteq \textbf{false} \vdash \langle\ ..\ \{\ \#\mathsf{sta2}\}\ ...\rangle\phi}{\vdash \langle\ ..\ \ \textbf{if}\ (\#\mathsf{b})\ \{\#\mathsf{sta1}\}\ \textbf{else}\ \{\#\mathsf{sta2}\}\ ...\rangle\phi} \tag{5}$$

where #b is a side-effect free boolean expression and #sta1, #sta2 are arbitrary *JavaCard* statements.

Example 1.3 (Example 1.2 continued) *Applying rule (5) to*

$$\vdash \langle \mathrm{l}:\{\ \textbf{if}\ (\mathrm{v} == 0)\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \textbf{else}\ \{\ \mathrm{v} = 0;\ \}\ \mathrm{v} = 3;\}\rangle \mathrm{v} \doteq 3$$

where $\circ_1$ *corresponds to the program between* 'l:{ ' *(inactive program prefix) and* 'v = 3;} ' *(suffix of the program context) now yields the two sequents*

(i) $(\mathrm{v}==0) \doteq \textbf{true} \vdash \langle \mathrm{l}:\{\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \mathrm{v} = 3;\ \}\rangle \mathrm{v} \doteq 3$ *and*

(ii) $(\mathrm{v}==0) \doteq \textbf{false} \vdash \langle \mathrm{l}:\{\ \{\ \mathrm{v} = 0;\ \}\ \mathrm{v} = 3;\ \}\rangle \mathrm{v} \doteq 3$

2 Taclets

Taclets are lightweight, stand-alone tactics with simple syntax and semantics. Their introduction was motivated by the observation that only few basic actions in proof construction are sufficient to implement most rules for first-order modal logic. These are:

- to recognise sequents as an axiom, and to close the according proof branch,
- to modify at most one formula per rule application,
- to add a finite (and fixed) number of formulas to a sequent,
- to let a proof goal split in a fixed number of branches,

- to restrict the applicability according to context information.

These are the only actions which taclet constructs are provided for. This restriction turns out to reduce the complexity for users of a proof system significantly [**?**].

Taclets by Example

Taclets describe rule schemas in a concise and easily readable way. A very simple example rewrites terms $1 + 1$ with 2. In taclet notation such a rule schema is written:

$$\texttt{find}(1 + 1)\ \texttt{replacewith}(2)$$

In a taclet—in addition to the logical content of the described rule—an operational meaning is encoded: If a user of a taclet-based prover selects the term of the `find`-part (i.e. $1 + 1$) of a taclet and chooses the taclet for application, the `find`-part is replaced with (an instantiation of) the `replacewith`-term (i.e. 2).

In this simple form, the rule schemas described by taclets are not expressive enough for practical use; *schema variables* and more constructs besides `find` and `replacewith` make them powerful enough to fulfil the requirements posed above.

Schema Variables and Instantiations

Expressions[5] in taclets may contain elements from a set SV of *schema variables*. An *instantiation* $\iota(\mathsf{v})$ of a schema variable $\mathsf{v} \in SV$ is a concrete expression that must fulfil certain conditions depending on the kind of the schema variable (see below). We may, e.g., define a schema variable i such that $\iota(\mathsf{i})$ must be a term of an integer sort.

Expressions e in taclets containing schema variables from SV are called *schematic expressions* over SV. The instantiation map ι can be canonically continued on schematic expressions:

$$\iota(op(e_1, \ldots, e_n)) = \begin{cases} \iota(op) & \text{if } n = 0 \text{ and } op \in SV \\ op(\iota(e_1), \ldots, \iota(e_n)) & \text{otherwise} \end{cases} \tag{6}$$

Thus e describes a set $\{\iota(e) \mid \iota \text{ is an instantiation map for every } \mathsf{v} \in SV\}$ of concrete expressions. For instance, a taclet $\texttt{find}(\mathsf{i} + \mathsf{i})\ \texttt{replacewith}(2 * \mathsf{i})$ contains schematic terms over $\{\mathsf{i}\}$. Applied on a sequent containing the term $3 + 3$, i is instantiated with $\iota(\mathsf{i}) = 3$ and the taclet replaces $\iota(\mathsf{i} + \mathsf{i}) = 3 + 3$ with $\iota(2 * \mathsf{i}) = 2 * 3$ in the new goal.

[5] By *expression* we denote syntactical elements like terms or formulas, but in the context of *JavaCardDL* also *Java* programs.

Taclet Syntax

`replacewith`(2) is an example of a *goal template*, this means the description of how a goal changes by applying the taclet. More than one goal template may be part of a taclet, separated by semicolons, which describes that a goal is split by the taclet. If there is no goal template in a taclet, applications close the proof branch. Additionally, goal templates may contain the following clauses:

- While in the example taclets above the `find`- and `replacewith`-parts consisted of terms, they can also be sequents. *All* `find`- and `replacewith`-parts of a taclet must *either* be terms or sequents. These sequents indicate that the described expression must be a top-level formula in either the antecedent or succedent, e.g. a taclet `find`$(\vdash \phi \rightarrow \psi)$ `replacewith`$(\phi \vdash \psi)$ (over the schema variables $\{\phi, \psi\}$) is applicable only to top-level formulas in the succedent. A sequent in the `find`-part must have either an empty antecedent or succedent.
- Taclet applications can *add* formulas to the antecedent or succedent. This is denoted by the keyword `add` followed by a schematic sequent (similarly to `replacewith`).
- Taclets support the dynamic enlargement of the taclet rule base by adding new taclets described behind the key word `addrules`. Though a theoretical treatment in accordance with the concepts of this paper is possible [**?**] we omit this feature in the sequel.

Often more information on the sequent a taclet is applicable to is needed. Such side conditions are described by the following optional taclet constituents:

- A taclet that contains an `if` followed by a schematic sequent *context* is only directly applicable if *context* is a "sub-sequent" of the sequent the taclet is applied to. If this is not the case, the taclet is however still applicable but, by an automatic cut, it is required to show the `if`-condition.
- Predefined clauses in a `varcond`-part describe conditions on the instantiations of schema variables. The most important ones are:
  - `v not free in s`, which disallows logical variables $\iota($`v`$)$ to occur unbound in $\iota($`s`$)$.
  - `v new depending on s`, which introduces a new skolem symbol $\iota($`v`$)$ (possibly depending on free "meta variables" occurring in $\iota($`s`$)$).

The complete syntax of taclets is reiterated here as an overview:

$$\begin{array}{l} [\texttt{if (}\textit{context}\texttt{)}]\ [\texttt{find (}f\texttt{)}]\ [\texttt{varcond (}c_1,\ldots,c_k\texttt{)}] \\ \quad [\texttt{replacewith (}rw_1\texttt{)}]\ [\texttt{add (}add_1\texttt{)}]; \\ \qquad\qquad \vdots \qquad\qquad\quad \vdots \\ \quad [\texttt{replacewith (}rw_n\texttt{)}]\ [\texttt{add (}add_n\texttt{)}] \end{array} \tag{7}$$

For $i = 1 \ldots n$, *context* and add_i stand for a schematic sequent, f and rw_i for a schematic term, formula, or sequent but all of the same kind. $c_1, \ldots, c_k$ are variable conditions.

Additionally—though out of scope of this paper—taclets can be assigned to one or several rule sets, which makes them available to be automatically executed by strategies. For a homogenous treatment in this paper f and rw_i are declared to be never empty: we assume that skipping `replacewith` is a shorthand for $rw_i = f$, a skipped `find` means $f = \vdash \mathit{false}$, and *false* always occurs in succedents of sequents. A skipped `if-` or `add`-part means $\mathit{context} = \vdash$ or $add_i = \vdash$ (resp.).

Schema Variable Types

While the above definitions have been general enough to be applied to every first-order modal logic, we are now focusing on special schema variables for *JavaCardDL*. Let SV_{tac} denote the schema variables contained in a taclet *tac*. Schema variables $\mathsf{v} \in SV_{tac}$ are assigned to one out of a predefined list of types, each having special properties concerning admissible instantiations $\iota(\mathsf{v})$. An instrument to define these properties is to introduce *prefix sets* (denoted by $\Pi_l(\mathsf{v})$, $\Pi_{pv}(\mathsf{v})$, and $\Pi_{jmp}(\mathsf{v})$) for schema variables v. A selection of the most relevant schema variable types is given below. If v is of type

- Variable, then v is assigned a *sort*, $\iota(\mathsf{v})$ must be of that sort. Moreover, $\iota(\mathsf{v})$ must be a logical variable. For $\mathsf{v} \neq \mathsf{v}' \in SV$: if v' is a Variable schema variable then $\iota(\mathsf{v}) \neq \iota(\mathsf{v}')$. $\iota(\mathsf{v})$ must not occur bound in $\iota(\mathsf{v}'')$ for all $\mathsf{v}'' \in SV$.
- Term, then v is assigned a *sort*, $\iota(\mathsf{v})$ must be of that sort.

  v is assigned a set $\Pi_l(\mathsf{v}) \subseteq SV$ of schema variables. $\Pi_l(\mathsf{v})$ is defined to be the smallest set with, for all constituents of *tac*, if v occurs in the scope of a Variable schema variable $\mathsf{v}' \in SV$ then $\mathsf{v}' \in \Pi_l(\mathsf{v})$ except there is a variable condition v' `not free in` v declared in t. v is assigned a set $\Pi_{pv}(\mathsf{v})$ which is the smallest set of program variables that occur but are not declared in *tac* or are declared above[6] every occurrence of v.

  We require from instantiations ι: If, for some $\mathsf{v}' \in SV_{tac}$, $\iota(\mathsf{v}')$ is a logical variable that occurs unbound in $\iota(\mathsf{v})$ then $\mathsf{v}' \in \Pi_l(\mathsf{v})$; if $\iota(\mathsf{v}')$ is a program variable that occurs undeclared in $\iota(\mathsf{v})$ then $\mathsf{v}' \in \Pi_{pv}(\mathsf{v})$.
- Formula, then as for Term, v is assigned $\Pi_l(\mathsf{v})$ and $\Pi_{pv}(\mathsf{v})$. v must fulfil the same conditions concerning these sets.
- Statement, then $\iota(\mathsf{v})$ is a *JavaCard* statement. Again, v is assigned $\Pi_{pv}(\mathsf{v})$ and it must satisfy the same conditions as above concerning this set.

  v is assigned a set $\Pi_{jmp}(\mathsf{v})$ consisting of *JavaCard* statements **break**, **continue**, **break** l, **continue** l for all labels l, if v is enclosed with a suitable jump target. If *jst* is a **break** or **continue** statement of $\iota(\mathsf{v})$ with a target

[6] If we consider *tac* as abstract syntax tree.

not in $\iota(\mathsf{v})$ then $jst \in \Pi_{jmp}(\mathsf{v})$.[7] Usually Statement schema variables have names starting with # to distinguish them from regular *Java* elements.

- ProgramVariable, then $\iota(\mathsf{v})$ is a local program variable or class attribute of *Java*. v is assigned a *Java* type and $\iota(\mathsf{v})$ must be of that type. Again, names of this kind of variable start with #.
- ProgramContext, then $\iota(\mathsf{v})$ is a program transformation[8] pt that takes a *Java* program element α and delivers a new program element $pt(\alpha)$, such that $pt(\alpha)$ is a sequence of statements of which the first one contains α and has only opening braces, opening **try** blocks, etc., in front. For this case, the continuation of the instantiation map (6) is then modified to

$$\iota(op(e_1, \ldots, e_n)) := pt(\iota(e_1))$$

if $n = 1$ and op is a ProgramContext schema variable.

Usually, v is denoted by .. e_1 ... containing the schematic *Java* program e_1, as introduced in Sect. 1.1.

Example 2.1 *The following taclet performs a cut with the condition that the focused term (*t*) equals* 0 *and replaces it in the respective goal by* 0*. We declare* t *as* Term *schema variable of an integer sort.*

$$\begin{array}{lll} \texttt{find(t)} & \texttt{replacewith(0)} & \texttt{add(t} \doteq 0 \vdash \texttt{);} \\ & \texttt{replacewith(t)} & \texttt{add(} \vdash \texttt{t} \doteq 0\texttt{)} \end{array} \tag{8}$$

*As an example that represents a rule of JavaCardDL, we take a taclet that replaces the postfix increment operator applied to a program variable (*x*)* behind *a statement (#*sta*) with an equivalent statement using assignment and the + operator, and leaves the formula (*ϕ*) behind the diamond unchanged. #*sta *is a* Statement *schema variable and* ϕ *a* Formula *schema variable.*

$$\texttt{find}(\langle \#\texttt{sta x++;}\rangle\phi) \quad \texttt{replacewith}(\langle \#\texttt{sta x=x+1;}\rangle\phi) \tag{9}$$

Another taclet splits a proof for an **if** *statement with the condition* x==0 *(where* x *is a concrete local variable) and produces goals, reducing the formula to the statements of the appropriate branch and the if condition put to the correct side of the sequent. #*sta1 *and #*sta2 *are* Statement *schema variables and* ϕ *is a* Formula *schema variable.*

$$\begin{array}{l} \texttt{find}(\langle \texttt{l:}\ \textbf{if}\ (\texttt{x==0})\ \#\texttt{sta1}\ \textbf{else}\ \#\texttt{sta2}\rangle\phi) \\ \quad \begin{array}{lll} \texttt{replacewith}(\langle \texttt{l:}\ \#\texttt{sta1}\ \rangle\phi) & \texttt{add}(\texttt{x} \doteq 0 \vdash \texttt{);} \\ \texttt{replacewith}(\langle \texttt{l:}\ \#\texttt{sta2}\ \rangle\phi) & \texttt{add(} \vdash \texttt{x} \doteq 0\texttt{)} \end{array} \end{array} \tag{10}$$

[7] For a complete treatment of *JavaCardDL* it is furthermore necessary to consider **return**-statements, which are left out in this paper

[8] Thus being an exception from the statement above that $\iota(\mathsf{v})$ must be an expression.

Semantics

Taclets have a precise operational semantics, which is described in detail in [**?**], and which we have sketched informally above. For the purposes of this paper it is sufficient to fix the logical meaning of a taclet in the traditional rule schema notation.

We denote the union of two sequents and the subset relation between two sequents as follows:

$$\big(\Gamma_1 \vdash \Delta_1\big) \cup \big(\Gamma_2 \vdash \Delta_2\big) := \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$$
$$\big(\Gamma_1 \vdash \Delta_1\big) \subseteq \big(\Gamma_2 \vdash \Delta_2\big) \text{ iff } \Gamma_1 \subseteq \Gamma_2 \text{ and } \Delta_1 \subseteq \Delta_2$$

First, we assume that f is a schematic sequent, i.e. taclet *tac* can only be applied to top-level formulas. By the operational semantics of taclets [**?**], *tac* represents the rule schema:

$$\frac{rw_1 \cup add_1 \cup (\Gamma \vdash \Delta) \qquad \ldots \qquad rw_k \cup add_k \cup (\Gamma \vdash \Delta)}{f \cup (\Gamma \vdash \Delta)} \tag{11}$$

where $\Gamma \vdash \Delta$ is an arbitrary sequent with *context* $\subseteq f \cup (\Gamma \vdash \Delta)$.

Similarly, if f is a schematic term or formula ($seq[e]$ denotes a sequent with an arbitrary but for a rule fixed occurrence of an expression e):

$$\frac{seq[rw_1] \cup add_1 \cup (\Gamma \vdash \Delta) \qquad \ldots \qquad seq[rw_k] \cup add_k \cup (\Gamma \vdash \Delta)}{seq[f] \cup (\Gamma \vdash \Delta)}$$

where $\Gamma \vdash \Delta$ is an arbitrary sequent with *context* $\subseteq seq[f] \cup (\Gamma \vdash \Delta)$.

In Sect. 4, the notion of *meaning formulas* is derived that makes the meaning of these rule schemas induced by taclets more precise.

Because of their simplicity and operational meaning, taclets can be schematically compiled into the GUI of taclet-based interactive theorem provers: In the KeY system a mouse click over an expression displays only those taclets whose `find`-part can be matched with the expression in focus. This reduces drastically the cognitive burden on the user. For an extensive account on user interaction see [**?**].

3 Outline of Bootstrapping Taclets

After having introduced basic notions and notations, we can focus on the task of how to ensure correctness of derived taclets. We aim to prove their soundness within the *JavaCardDL* calculus itself. Our approach is based on [**?**] which has already provided this kind of bootstrapping for classical first-order logic.

Given a taclet *tac*, we first derive a *meaning formula* $M(tac)$ (see Sect. 4), which is supposed to be valid if and only if all possible applications of *tac* are

correct proof steps. For example, consider the following taclet tac_0:

$$\texttt{find}(true \wedge \ \phi \ \vdash \ ) \quad \texttt{replacewith}(\phi \vdash)$$

with a Formula schema variable ϕ. The corresponding meaning formula is

$$M(tac_0) = \neg\phi \rightarrow \neg(true \wedge \phi) \text{ or equivalently } (true \wedge \phi) \rightarrow \phi$$

Intuitively, the meaning formula states that if a formula in an antecedent is replaced, the new formula must be at most as strong as the old one. If this can be proven for all instantiations of ϕ, i.e. for all formulas, then obviously tac_0 is sound.

Unfortunately, meaning formulas contain schema variables (here: ϕ) and are thus no *JavaCardDL* formulas. Moreover, we have to quantify somehow over all formulas. Skolemisation of schema variables (see Sect. 5) helps us, however, not having to leave our original logic and not having to employ higher order logics on the object level. Skolemisation of meaning formula $M(tac_0)$ leads to

$$M_{\mathrm{Sk}}(tac_0) = (true \wedge \phi_{\mathrm{Sk}}) \rightarrow \phi_{\mathrm{Sk}},$$

where ϕ_{Sk} is a new nullary predicate. We call these formulas $M_{\mathrm{Sk}}(tac)$ *taclet proof obligations*. $M_{\mathrm{Sk}}(tac)$ is a *JavaCardDL* formula (with a slightly extended vocabulary) and can be loaded into our interactive theorem prover. If the proof obligation can be proven successfully then correctness of the taclet is ensured for all possible applications according to the definition of the meaning formula. The proof of the corresponding theorem is given in [**?**] and sketched in Sect. 6.

On a semantic level, this theorem can be justified by arguing that if an application of the taclet *tac* leads to an incorrect proof, a suitable interpretation D can be constructed such that the meaning formula $M(tac)$ is not satisfied under D (which is a direct consequence of the definition of meaning formulas) and thus $M(tac)$ could not have been proven. This semantic argumentation works fine for first-order logics [**?**], but when *JavaCardDL* comes into play, the complete complex *JavaCard* semantics would have to be incorporated in the reasoning.

Instead, we take a syntactic approach getting the *JavaCard* semantics via the *JavaCardDL* calculus for free. The basic idea is to show that an application of a taclet *tac* can always be replaced by a transformed proof of $M_{\mathrm{Sk}}(tac)$.

4 Meaning Formulas of Taclets

The basis for our reasoning about the correctness of taclets is a *meaning formula* [**?**] derived in this section. It is declared to be the meaning of a taclet independently from concrete taclet application mechanisms, thus providing a very flexible way to address soundness issues. In fact we define a taclet application mechanism to be correct if (and only if) taclets with valid meaning

formulas are translated into sound rules.[9] To show that a taclet is correct it is thus sufficient to prove the validity of its meaning formula.

For the whole section we define $(\Gamma \vdash \Delta)^* := \bigwedge \Gamma \to \bigvee \Delta$, in particular $(\vdash \phi)^* = \phi$ and $(\phi \vdash)^* = \neg\phi$. Furthermore, in this section by the validity of a sequent we mean the validity of $(\Gamma \vdash \Delta)^*$. We define a (sequent) calculus C to be *sound* if only valid sequents are derivable in C. We conceive rules

$$\frac{P_1 \quad \ldots \quad P_n}{Q}$$

as relations between tuples of sequents (the premisses) and single sequents (the conclusion) and define that a rule $R \in C$ is sound if for all tuples $(\langle P_1, \ldots, P_k\rangle, Q) \in R$:

$$\text{if } P_1, \ldots, P_k \text{ are valid, then } Q \text{ is valid.} \tag{12}$$

For the calculus C we can state:

Lemma 4.1 *C is sound if all rules $R \in C$ are sound.*

The rules R_{tac} we are interested in are defined through taclets *tac* over a set SV of schema variables in the form as defined in (7). Assuming first that the `find`-part is a sequent, taclets induce the rule schema (11). To apply Lem. 4.1, for each instantiation ι of SV, (12) must be shown for $k = n$, $P_i = \iota(rw_i \cup add_i \cup \Gamma \vdash \Delta)$ $(i = 1 \ldots n)$, and $Q = \iota(f \cup \Gamma \vdash \Delta)$. Since the formulas of $\Gamma \vdash \Delta$ which are not in *context* are arbitrary and not influenced by the rule application we can simply omit them and show the lemma for $P_i = \iota(rw_i \cup add_i \cup context)$ $(i = 1 \ldots n)$ and $Q = \iota(f \cup context)$. We assume that *tac* does not introduce skolem functions, i.e. does not contain such a variable condition. Then by the deduction theorem, the global condition (12) can be strengthened to the local implication, namely that $P_1^* \wedge \ldots \wedge P_n^* \to Q^*$ must be valid.

Since ι, as defined by (6), treats propositional junctors as a homomorphism and the operator $(\cdot)^*$ is a homomorphism regarding the union of sequents up to propositional transformations, this formula can now be simplified as follows:

$$P_1^* \wedge \ldots \wedge P_n^* \to Q^* = \bigwedge_{i=1}^{n} \iota(rw_i \cup add_i \cup context)^* \to \iota(f \cup context)^* \tag{13}$$

$$= \iota\Big(\bigwedge_{i=1}^{n} (rw_i^* \vee add_i^*) \to (f^* \vee context^*)\Big). \tag{14}$$

If (14) is proven for all instantiations ι, then the rule R_{tac} represented by *tac* is sound.

In the next definition our previously made assumptions are revoked: the variable condition sv_i `new depending on`... introduces new skolem functions.

[9] As a schematic formula, the meaning formula is by definition valid iff all instances of the formula are valid.

If $P_1, \ldots, P_n$ contain skolem symbols that do not occur in Q, the interpretation of the symbols can be regarded as universally quantified in (12) by the usual definition of 'valid'. Because of their negation in (13), they are existentially bound in the meaning formula. Moreover, taclets that have terms or formulas instead of sequents as `find`-part and `replacewith`-parts are reduced to a rule that adds an equivalence $f \leftrightarrow rw_i$ or equation $f = rw_i$ to the antecedent.

Definition 4.2 (Meaning Formula) *Each taclet tac, as declared in (7), is assigned an* unquantified meaning formula *tac^*, which is defined by:*

$$tac^* := \begin{cases} \bigwedge_{i=1}^{n}(rw_i^* \vee add_i^*) \rightarrow (f^* \vee context^*) & \text{if } f \text{ is a sequent} \\ \bigwedge_{i=1}^{n}\left(f \doteq rw_i \rightarrow add_i^*\right) \rightarrow context^* & \text{if } f \text{ is a term} \\ \bigwedge_{i=1}^{n}\left((f \leftrightarrow rw_i) \rightarrow add_i^*\right) \rightarrow context^* & \text{if } f \text{ is a formula} \end{cases}$$

Suppose $\mathsf{sv}_1, \ldots, \mathsf{sv}_k \in SV_{tac}$ are all schema variables, which tac contains a variable condition sv_i `new depending on`*... for. $M(tac) := \exists \mathsf{x}_1 \ldots \exists \mathsf{x}_k.\phi$ is defined to be the* meaning formula *of tac where ϕ is obtained from tac^* by replacing each sv_i with a new* Variable *schema variable x_i with the same sort as sv_i.*

Example 4.3 (Example 2.1 continued) *The taclets tac_1, tac_2, and tac_3 defined through (8), (9), and (10), resp., have (after applying some propositional equivalence transformations) the following meaning formulas:*

$$M(tac_1) = \left(\mathsf{t} \doteq 0 \wedge \mathsf{t} \doteq 0\right) \vee \left(\mathsf{t} \doteq \mathsf{t} \wedge \neg(\mathsf{t} \doteq 0)\right) \tag{15}$$

$$M(tac_2) = \langle \#\mathsf{sta}\ \mathrm{x{+}{+};}\rangle\phi \leftrightarrow \langle \#\mathsf{sta}\ \mathrm{x{=}x{+}1;}\rangle\phi \tag{16}$$

$$\begin{aligned} M(tac_3) = \ & \big((\langle \mathrm{l}:\ \textbf{if}\ (\mathrm{x{=}{=}0})\ \#\mathsf{sta1}\ \textbf{else}\ \#\mathsf{sta2}\rangle\phi \leftrightarrow \langle \mathrm{l}:\ \#\mathsf{sta1}\ \rangle\phi) \\ & \wedge\ \mathrm{x} \doteq 0\big) \\ & \vee\big((\langle \mathrm{l}:\ \textbf{if}\ (\mathrm{x{=}{=}0})\ \#\mathsf{sta1}\ \textbf{else}\ \#\mathsf{sta2}\rangle\phi \leftrightarrow \langle \mathrm{l}:\ \#\mathsf{sta2}\ \rangle\phi) \\ & \wedge\ \neg(\mathrm{x} \doteq 0)\big) \end{aligned} \tag{17}$$

5 Construction of Proof Obligations

Except for trivial taclets, the meaning formula $M(tac)$ of a taclet *tac* contains schema variables, which is at least inconvenient for proving $M(tac)$. Variables of these types do however not occur bound within the formula (resp. when considering validity, they can be regarded as implicitly universally quantified), and hence it is possible to replace them in a suitable way without altering the validity of the meaning formula:

- Schema variables for logical variables or program variables can simply be replaced with new concrete variables. It has to be taken in account, however, that when instantiating a schematic expression it is possible that two

different schema variables of type ProgramVariable are instantiated with the same concrete variable (which is not possible for Variable schema variables by the definitions of Sect. 2). By the presence or absence of such collisions, the set of instances of a schematic expression is divided into (finitely many) classes, which all have to be considered to capture the meaning of the schematic expression.

- Schema variables for terms, formulas or *Java* statements can be replaced with suitable "skolem" symbols, which are similar to the atomic programs of DPDL for Statement schema variables. To model the notion of abrupt termination, which does not exist in DPDL, tuples of *Java* jump statements are attached to occurrences of symbols for statements.
- Schema variables for program contexts can be replaced with a surrogate *Java* block containing atomic programs.

From now on, we only consider the replacement of schema variables for logical variables, terms, formulas and statements, and we also assume that the concerned taclets only contain schema variables of these kinds. Other kinds of schema variables are treated in a similar way in [**?**].

5.1 Skolem symbols

We define two syntactic domains that consist of symbols for the skolemisation of schema variables:

- Symbols that are placeholders for terms and formulas, and which are similar to ordinary function and predicate symbols
- Symbols that are placeholders for *Java* statements, similar to the atomic programs of DPDL.

As usual, the elements of both domains are assigned signatures that determine syntactically well-formed expressions. Their shape is described in more detail as follows.

Skolem Symbols for Terms and Formulas

The sets of symbols for terms and formulas are denoted with $\mathit{Func}_{\mathrm{Sk}}$ and $\mathit{Pred}_{\mathrm{Sk}}$ (resp.). The signature

$$\alpha(s_{\mathrm{Sk}}) = \begin{cases} (S, S_1, \ldots, S_n, T_1, \ldots, T_k) & \text{for } s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \\ (S_1, \ldots, S_n, T_1, \ldots, T_k) & \text{for } s_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}} \end{cases}$$

of a symbol $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$ consists of

- a result sort S, if $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}}$,
- a finite sequence $S_1, \ldots, S_n$ of sorts that determines the number and kinds of term arguments; this sequence corresponds to the signature of ordinary predicate symbols,

- a finite sequence $T_1, \ldots, T_k$ of *Java* types, which are the component types of a tuple of program variables with which occurrences of s_{Sk} are equipped.

Accordingly, the inductive definition of well-formed terms and formulas is extended by:

If $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$ is a symbol with the signature $\alpha(s_{\mathrm{Sk}})$ as above, $t_1, \ldots, t_n$ are terms of the sorts $S_1, \ldots, S_n$ and $\mathrm{pv}_1, \ldots, \mathrm{pv}_k \in \mathit{PVar}$ are program variables of the types $T_1, \ldots, T_k$, then

$$s_{\mathrm{Sk}}(t_1, \ldots, t_n; \mathrm{pv}_1, \ldots, \mathrm{pv}_k)$$

is a term of sort S or a formula (resp.).

Skolem Symbols for Statements

The set of skolem symbols used for statements is denoted with $\mathit{Statement}_{\mathrm{Sk}}$. The signature $\alpha(st_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m)$ of a symbol $st_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ consists of

- a finite sequence $T_1, \ldots, T_k$ of *Java* types (analogously to the symbols for terms or formulas),
- a natural number m that gives the size of the *jump table*; this is a tuple of *Java* statements that are arguments of occurrences of st_{Sk} within programs.

The symbols $\mathit{Statement}_{\mathrm{Sk}}$ extend the definition of well-formed *Java* programs, i.e. the following (informal) rule is added to the *Java* grammar [**?**]:

If $st_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ is a symbol with $\alpha(st_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m)$, $\mathrm{pv}_1, \ldots, \mathrm{pv}_k$ are program variables of the types $T_1, \ldots, T_k$ and $\mathit{jst}_1, \ldots, \mathit{jst}_m$ are *Java* statements of the following kinds [10]

- **return**-statements, with or without an argument (a plain program variable)
- **break**- and **continue**-statements, with or without a label
- **throw**-statements whose argument is a program variable

then

$$st_{\mathrm{Sk}}(\mathrm{pv}_1, \ldots, \mathrm{pv}_k; \mathit{jst}_1; \ldots; \mathit{jst}_m)$$

is a statement.

5.2 From Meaning Formula to Proof Obligation

From now on we suppose that a taclet *tac* with meaning formula $M(\mathit{tac})$ is fixed. Let SV_{tac} be the set of schema variables $M(\mathit{tac})$ contains. We define an instantiation ι_{Sk} over SV_{tac} that replaces each schema variable either with a *JavaCardDL* variable or with an appropriate skolem expression. The definition refers to the properties of schema variables as introduced in Sect. 2:

[10] Which are exactly the reasons that can lead to an abrupt termination of a statement, see [**?**].

- If $\mathsf{x} \in SV_{tac}$ is of type Variable, then $\iota_{\mathrm{Sk}}(\mathsf{x}) \in LVar$ is a new logical variable that has the same sort as x.
- If $\mathsf{sv} \in SV_{tac}$ is of type Term, Formula or Statement, then let $\{\mathrm{pv}_1, \ldots, \mathrm{pv}_k\} = \Pi_{pv}(\mathsf{sv})$ be the program variables that can occur undeclared in instantiations of sv. Let $T_1, \ldots, T_k$ be the *Java* types of $\mathrm{pv}_1, \ldots, \mathrm{pv}_k$.
- If $\mathsf{sv} \in SV_{tac}$ is of type Term, then

$$\iota_{\mathrm{Sk}}(\mathsf{sv}) = f_{\mathrm{Sk}}(v_1, \ldots, v_l; \mathrm{pv}_1, \ldots, \mathrm{pv}_k)$$

  is a term, where
  - $v_1, \ldots, v_l$ with $v_i = \iota_{\mathrm{Sk}}(\mathsf{x}_i)$ are the instantiations of $\mathsf{x}_1, \ldots, \mathsf{x}_l \in SV_{tac}$, which are distinct Variable schema variables determined by the prefix $\Pi_l(\mathsf{sv}) = \{\mathsf{x}_1, \ldots, \mathsf{x}_l\}$ of sv in *tac*
  - and $f_{\mathrm{Sk}} \in Func_{\mathrm{Sk}}$ denotes a new skolem symbol with signature

$$\alpha(f_{\mathrm{Sk}}) = (S, S_1, \ldots, S_l, T_1, \ldots, T_k)$$

    where S is the sort of sv and $S_1, \ldots, S_l$ are the sorts of $v_1, \ldots, v_l$.
- Analogously, if $\mathsf{sv} \in SV_{tac}$ is a schema variable of type Formula, then

$$\iota_{\mathrm{Sk}}(\mathsf{sv}) = p_{\mathrm{Sk}}(v_1, \ldots, v_l; \mathrm{pv}_1, \ldots, \mathrm{pv}_k)$$

  is a formula containing a new skolem symbol $p_{\mathrm{Sk}} \in Pred_{\mathrm{Sk}}$ for formulas.
- If $\mathsf{sv} \in SV_{tac}$ is a schema variable of type Statement, then two additional (and new) program variables are needed: t_{sv} of *Java* type **Throwable**, and d_{sv} of *Java* type **int** (the latter variable is used in Sect. 5.3). Let $\{jst_1, \ldots, jst_m\} = \Pi_{jmp}(\mathsf{sv})$ be jump statements that can occur uncaught in instantiations of sv. The instantiation $\iota_{\mathrm{Sk}}(\mathsf{sv})$ of sv is the statement [11]

$$\iota_{\mathrm{Sk}}(\mathsf{sv}) = st_{\mathrm{Sk}}(\mathrm{pv}_1, \ldots, \mathrm{pv}_k, \mathrm{t}_{\mathsf{sv}}, \mathrm{d}_{\mathsf{sv}}; jst_1; \ldots; jst_m; \textbf{throw } \mathrm{t}_{\mathsf{sv}})$$

  where st_{Sk} denotes a new skolem symbol for statements with signature $\alpha(st_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m+1)$.

Finally, the *taclet proof obligation* of *tac* is defined to be the formula

$$M_{\mathrm{Sk}}(tac) := \iota_{\mathrm{Sk}}(M(tac))$$

Example 5.1 (Example 4.3 continued) *From the meaning formulas of the taclets tac_1, tac_2 and tac_3 the following proof obligations are derived:*

[11] We always add a **throw**-statement, as instantiations of sv may always terminate abruptly through an exception regardless of $\Pi_{jmp}(\mathsf{sv})$.

$$M_{\mathrm{Sk}}(tac_1) = \big(t_{\mathrm{Sk}} \doteq 0 \wedge t_{\mathrm{Sk}} \doteq 0\big) \vee \big(t_{\mathrm{Sk}} \doteq t_{\mathrm{Sk}} \wedge \neg(t_{\mathrm{Sk}} \doteq 0)\big) \tag{18}$$

$$M_{\mathrm{Sk}}(tac_2) = \tag{19}$$

$$\begin{aligned} &\langle sta_{\mathrm{Sk}}(\mathrm{v},\, \mathrm{t}_{\#\mathsf{sta}},\, \mathrm{d}_{\#\mathsf{sta}};\ \textbf{throw}\ \mathrm{t}_{\#\mathsf{sta}});\ \mathrm{v{+}{+};}\rangle p_{\mathrm{Sk}}(\mathrm{v}) \leftrightarrow \\ &\langle sta_{\mathrm{Sk}}(\mathrm{v},\, \mathrm{t}_{\#\mathsf{sta}},\, \mathrm{d}_{\#\mathsf{sta}};\ \textbf{throw}\ \mathrm{t}_{\#\mathsf{sta}});\ \mathrm{v{=}v{+}1;}\rangle p_{\mathrm{Sk}}(\mathrm{v}) \end{aligned} \tag{20}$$

$$M_{\mathrm{Sk}}(tac_3) = \tag{21}$$

$$\begin{aligned} &\big((\langle \mathrm{l}:\ \textbf{if}\ (\mathrm{x{=}{=}0})\ \beta_1\ \textbf{else}\ \beta_2\rangle p_{\mathrm{Sk}}(\mathrm{x}) \leftrightarrow \langle \mathrm{l}:\ \beta_1\rangle p_{\mathrm{Sk}}(\mathrm{x})) \wedge \mathrm{x} \doteq 0\big) \vee \\ &\big((\langle \mathrm{l}:\ \textbf{if}\ (\mathrm{x{=}{=}0})\ \beta_1\ \textbf{else}\ \beta_2\rangle p_{\mathrm{Sk}}(\mathrm{x}) \leftrightarrow \langle \mathrm{l}:\ \beta_2\rangle p_{\mathrm{Sk}}(\mathrm{x})) \wedge \neg(\mathrm{x} \doteq 0)\big) \end{aligned} \tag{22}$$

where we use the abbreviations

$$\begin{aligned} \beta_1 &= sta1_{\mathrm{Sk}}(\mathrm{x},\, \mathrm{t}_{\#\mathsf{sta1}},\, \mathrm{d}_{\#\mathsf{sta1}};\ \textbf{break}\ \mathrm{l};\ \textbf{throw}\ \mathrm{t}_{\#\mathsf{sta1}});\\ \beta_2 &= sta2_{\mathrm{Sk}}(\mathrm{x},\, \mathrm{t}_{\#\mathsf{sta2}},\, \mathrm{d}_{\#\mathsf{sta2}};\ \textbf{break}\ \mathrm{l};\ \textbf{throw}\ \mathrm{t}_{\#\mathsf{sta2}}); \end{aligned}$$

5.3 Decomposition Rules

Calculus rules for *JavaCardDL* programs always modify the leading statements within a program block (see Sect. 1). Unfortunately, the addition of skolem symbols for statements would destroy the (relative) completeness of a set of rules: If a skolem symbol turns up as the first active statement of a program block, no *JavaCardDL* rule will be applicable to that block.

As we have stated in Sect. 1.1 that a "naive" decomposition rule for *JavaCardDL* cannot be posed due to abrupt termination, we define a family of decomposition rules specifically for statement skolem symbols. These rules cope with abrupt termination by applying a transformation to the statement $\alpha = st_{\mathrm{Sk}}(\ldots)$. This transformation splits α in two parts $\alpha_1 = st'_{\mathrm{Sk}}(\ldots)$ and α_2, such that the concatenation $\alpha_1;\alpha_2$ is equivalent to the original statement α. Furthermore, the first program fragment α_1 is constructed in a way that prevents abrupt termination, and thus, the equivalence

$$\langle ..\ st_{\mathrm{Sk}}(\ldots);\beta\ ...\rangle\phi \leftrightarrow \langle st'_{\mathrm{Sk}}(\ldots)\rangle\langle ..\ \alpha_2;\beta\ ...\rangle\phi \tag{23}$$

holds. The remaining statement α_2 does no longer contain any skolem symbols, i.e. it is a pure *JavaCard* program, and hence it is possible to handle α_2 by the application of regular *JavaCardDL* rules.

We assume that for each statement skolem symbol $st_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ that occurs within ι_{Sk} a second new skolem symbol $\mathit{Dec}(st_{\mathrm{Sk}})$ is introduced, which has the same signature as st_{Sk} except for the jump table:

$$\alpha(st_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m) \quad \Longrightarrow \quad \alpha(\mathit{Dec}(st_{\mathrm{Sk}})) = (T_1, \ldots, T_k, 0).$$

Following equivalence (23), two decomposition taclets $D^{\Diamond}_{st_{\mathrm{Sk}}}$ and $D^{\Box}_{st_{\mathrm{Sk}}}$ for diamond and box modalities (resp.) are introduced for each statement skolem

symbol st_{Sk} that occurs in ι_{Sk}. We only give the definition of $D^{\diamond}_{st_{\mathrm{Sk}}}$, as the taclet for boxes is obtained analogously:

$$D^{\diamond}_{st_{\mathrm{Sk}}}: \quad \{ \quad \texttt{find (} \langle .. \ st_{\mathrm{Sk}}(\mathsf{p}_1, \ldots, \mathsf{p}_k; \#\mathsf{jst}_1; \ldots; \#\mathsf{jst}_m); \ ...\rangle\phi \ \texttt{)}$$
$$\texttt{replacewith (} \langle Dec(st_{\mathrm{Sk}})(\mathsf{p}_1, \ldots, \mathsf{p}_k);\rangle\langle .. \ ic \ ...\rangle\phi \ \texttt{)} \ \}$$

where $\mathsf{p}_1, \ldots, \mathsf{p}_k$ are schema variables for program variables, $\#\mathsf{jst}_1, \ldots, \#\mathsf{jst}_m$ are variables for statements corresponding to the signature $\alpha(st_{\mathrm{Sk}})$ and ϕ is a schema variable for formulas. Furthermore the taclet contains an if-cascade ic, which is denoted by α_2 in equivalence (23):

$$\begin{array}{lll} \{ & \textbf{if} \ (\ \mathsf{p}_k == 1 \ ) & \#\mathsf{jst}_1 \\ \textbf{else} & \textbf{if} \ (\ \mathsf{p}_k == 2 \ ) & \ldots \\ \textbf{else} & \textbf{if} \ (\ \mathsf{p}_k == m \ ) & \#\mathsf{jst}_m \ \} \end{array}$$

In this statement at most one of the jump statements represented by the schema variables $\#\mathsf{jst}_1, \ldots, \#\mathsf{jst}_m$ is selected and executed, depending on the value of the last program variable argument p_k (note that the type of p_k is **int** by the definitions of the last section).

Example 5.2 *An application of the decomposition rule for diamond modalities could look as follows:*

$$\frac{\vdash \ \langle st'_{\mathrm{Sk}}(\mathrm{t}, \mathrm{d})\rangle\langle \textbf{try} \ \{ \ \textbf{if} \ (\mathrm{d} == 1) \ \textbf{throw} \ \mathrm{t}; \ \} \ \textbf{catch} \ (\textbf{Exception} \ \mathrm{e}) \ \{...\}\rangle\phi}{\vdash \ \langle \textbf{try} \ \{ \ st_{\mathrm{Sk}}(\mathrm{t}, \ \mathrm{d}; \ \textbf{throw} \ \mathrm{t}); \ \} \ \textbf{catch} \ (\textbf{Exception} \ \mathrm{e}) \ \{...\}\rangle\phi}$$

6 Main Result

To show that *tac* is derivable, which is by Sect. 4 equivalent to the derivability of all instances of $M(tac)$, we assume that there is a closed proof H of $M_{\mathrm{Sk}}(tac)$ using the sequent calculus for *JavaCardDL* (extended by the skolem symbols and the decomposition taclets of Sect. 5). It is possible to transform H into a proof H_ϕ for each instance ϕ of $M(tac)$:

Theorem 6.1 (Main Result) *Suppose that a proof H of $M_{\mathrm{Sk}}(tac)$ exists. Then for each instance $\phi = \kappa(M(tac))$ of the meaning formula $M(tac)$ there is a proof H_ϕ.*

In the following we will sketch a proof of Theorem 6.1. Due to lack of space we skip most of the details of the proof; a more detailed account can be found in [**?**].

The proof obligation $M_{\mathrm{Sk}}(tac) = \iota_{\mathrm{Sk}}(M(tac))$ differs from other instances $\phi = \kappa(M(tac))$ of the meaning formula in the instantiation of schema variables for terms, formulas and statements: In $M_{\mathrm{Sk}}(tac)$ such variables are replaced with skolem symbols as introduced in Sect. 5.1.[12] Hence it is possible

[12] Schema variables for logical variables are in both cases simply instantiated with logical variables.

to obtain a "proof" H' of ϕ by replacing each occurrence of a skolem symbol $s_{\mathrm{Sk}}(\ldots) = \iota_{\mathrm{Sk}}(sv)$ in H with the instantiation $\kappa(sv)$ from ϕ. In general, the tree H' cannot be expected to be a proof, as it is possible that the replacement of skolem symbols leads to invalid rule applications. But by a slightly more complex transformation, as sketched below, it is possible to obtain a legal proof:

For the replacement of skolem symbols we define an appropriate kind of substitutions: We assume that a mapping σ of the skolem symbols

$$\mathit{Sym}_{\mathrm{Sk}} := \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}} \cup \mathit{Statement}_{\mathrm{Sk}}$$

to terms, formulas and *Java* statements "with holes" is given. Namely, we allow that for a symbol s_{Sk} with signature $\alpha(s_{\mathrm{Sk}}) = ([S,]S_1, \ldots, S_n, T_1, \ldots, T_k)$ (or $\alpha(s_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m)$ for statement symbols), the value $\sigma(s_{\mathrm{Sk}})$ contains a number of holes $\circ_i$ labelled with natural numbers $i \in \{1, \ldots, n+k\}$ (or $i \in \{1, \ldots, k+m\}$, resp.).

Example 6.2 *For a predicate skolem symbol $p_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}}$, an example of a substitution is given by the following mapping:*

$$\sigma(p_{\mathrm{Sk}}) = r(\circ_2, a) \wedge q(\circ_1) \wedge \langle \circ_2{=}1;\rangle\phi \qquad \text{for } p_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}},\ \ \alpha(p_{\mathrm{Sk}}) = (S, \mathbf{int}).$$

The mapping σ is continued to terms, formulas, *Java* programs, sequents, proof trees and taclets as a morphism, and by the replacement of skolem symbols. Holes are replaced with the arguments of occurrences of skolem symbols: [13]

$$\sigma(s_{\mathrm{Sk}}(r_1, \ldots, r_l)) := \{\circ_1/r_1, \ldots, \circ_l/r_l\}\big(\sigma(s_{\mathrm{Sk}})\big)$$

Example 6.3 (Example 6.2 continued) *The mapping σ is applied in the following way to a formula containing the symbol p_{Sk}:*

$$\sigma\big(\forall x.p_{\mathrm{Sk}}(x;\mathrm{i})\big) = \forall x.\big(r(\mathrm{i}, a) \wedge q(x) \wedge \langle \mathrm{i}{=}1;\rangle\sigma(\phi)\big)$$

6.1 Treatment of Taclets

The most important observation to prove Theorem 6.1 is the following lemma:

Lemma 6.4 (Lifting of Taclet Applications) *Suppose that $R_{tac'}$ is a rule schema that is described by a taclet tac', and that tac' does not contain skolem symbols (as introduced in Sect. 5.1). If an instance of $R_{tac'}$ is given by*

$$\frac{P_1 \quad \cdots \quad P_n}{Q}$$

[13] Extensive considerations about possible collisions are omitted in this document; see [**?**] for details.

and σ is a substitution of skolem symbols, then there is a proof tree with root sequent $\sigma(Q)$, whose open goals are exactly the sequents $\sigma(P_1), \ldots, \sigma(P_n)$.

Proof. First suppose that the considered rule application is not the application of a rewrite taclet within an argument of a skolem symbol occurrence. Then it can be shown that

$$\frac{\sigma(P_1) \quad \cdots \quad \sigma(P_n)}{\sigma(Q)}$$

is an instance of $R_{tac'}$.

Otherwise, if a rewrite taclet is applied to a term t within an argument of a skolem symbol occurrence, it is possible that a single occurrence of t in Q produces more than one occurrence of $\sigma(t)$ in $\sigma(Q)$ (like in example 6.3, where a single occurrence of the program variable i in the original formula yields multiple occurrences after the application of σ). Provided that the cut-rule and rules treating equations are available, it is then possible to perform a cut with the equation $\sigma(t) \doteq \sigma(t)$ and apply tac' to one side of the equation. Afterwards the equations $\sigma(t) \doteq \sigma(t_i)$ can be used to replace all occurrences of $\sigma(t)$ successively. This is illustrated by the following proof tree fragment, in which we use the notation $(\Gamma \vdash \Delta) = \sigma(Q)$:

$$\dfrac{\dfrac{\begin{array}{c}\sigma(P_1)\\ \vdots \\ \Gamma_1, \sigma(t) \doteq \sigma(t_1) \vdash \Delta_1\end{array} \quad \cdots \quad \begin{array}{c}\sigma(P_n)\\ \vdots \\ \Gamma_n, \sigma(t) \doteq \sigma(t_n) \vdash \Delta_n\end{array}}{\Gamma, \sigma(t) \doteq \sigma(t) \vdash \Delta}\, tac' \qquad \dfrac{*}{\Gamma \vdash \Delta, \sigma(t) \doteq \sigma(t)}}{\Gamma \vdash \Delta}$$

□

Corollary 6.5 *Suppose that the proof H of $M_{\mathrm{Sk}}(tac) = \iota_{\mathrm{Sk}}(M(tac))$ only consists of applications of taclets tac', and that the concerned taclets tac' do not contain skolem symbols. Then for each instance $\phi = \kappa(M(tac))$ of the meaning formula $M(tac)$ there is a proof H_ϕ.*

Proof. W.l.o.g. we may assume that ι_{Sk} and κ are equal w.r.t. the instantiations of schema variables of type Variable. Each taclet application within H can then be replaced with the proof tree fragment that is obtained from Lem. 6.4, for a σ that substitutes skolem expressions $s_{\mathrm{Sk}}(\ldots) = \iota_{\mathrm{Sk}}(sv)$ with the concrete instantiation $\kappa(sv)$, i.e. in a way such that $\sigma(M_{\mathrm{Sk}}(tac)) = \phi$. □

6.2 Treatment of Decomposition Rules

Lem. 6.4 of the last section is not directly applicable to applications of the taclets $D^{\diamond}_{s_{\mathrm{Sk}}}$, $D^{\Box}_{s_{\mathrm{Sk}}}$ (Sect. 5.3), as these taclets contain statement skolem symbols s_{Sk} and $Dec(s_{\mathrm{Sk}})$. If these symbols are replaced with arbitrary *Java* statements by the application of a substitution σ (as introduced in the previous section), then the obtained taclet will furthermore be unsound in general.

We circumvent these problems by constructing particular substitutions σ of the symbols s_{Sk} and $Dec(s_{\mathrm{Sk}})$ with the property that $\sigma(D^{\diamond}_{s_{\mathrm{Sk}}})$, $\sigma(D^{\Box}_{s_{\mathrm{Sk}}})$ are sound taclets, so that subsequently Lem. 6.4 can be applied for obtaining a proof tree.

Lemma 6.6 *Suppose that σ is a substitution that replaces all skolem symbols of a formula ψ, and s_{Sk} is a skolem symbol for statements. Then there is a substitution σ' that differs from σ only in the symbols s_{Sk}, $Dec(s_{\mathrm{Sk}})$, such that*

(i) *$\sigma'(D^{\diamond}_{s_{\mathrm{Sk}}})$, $\sigma'(D^{\Box}_{s_{\mathrm{Sk}}})$ are sound taclets*

(ii) *There is a proof tree (fragment) whose root is $\vdash \sigma(\psi)$, such that the only goal left is $\vdash \sigma'(\psi)$.*

Referring to this lemma it is possible to formulate an analogue of Lem. 6.4 for decomposition taclets:

Lemma 6.7 (Lifting of Decompositions) *Suppose that R_D is a rule that is described by a decomposition taclet D ($D = D^{\diamond}_{s_{\mathrm{Sk}}}$ or $D = D^{\Box}_{s_{\mathrm{Sk}}}$). If an instance of R_D is given by*

$$\frac{P}{Q}$$

and σ' is a substitution of skolem symbols as in Lem. 6.6 w.r.t. D, then there is a proof tree of $\sigma'(Q)$, whose only goal left is the sequent $\sigma'(P)$.

Proof. First the application of D is replaced with an application of the taclet $\sigma'(D)$, which is sound by Lem. 6.6, (i) (this substitutes certain occurrences of s_{Sk}, $Dec(s_{\mathrm{Sk}})$ within P and Q). Subsequently Lem. 6.4 can be applied to the resulting rule application w.r.t. σ'. □

Corollary 6.8 *Suppose that the proof H of $M_{\mathrm{Sk}}(tac) = \iota_{\mathrm{Sk}}(M(tac))$ only consists of applications of taclets tac' that do not contain skolem symbols, and of applications of decomposition taclets. Then for each instance $\phi = \kappa(M(tac))$ of the meaning formula $M(tac)$ there is a proof H_ϕ.*

Proof. σ is chosen as in the proof of Cor. 6.5. By repeated application of Lem. 6.6, (ii) it is possible to construct a proof tree with root sequent $\vdash \phi$ and a single goal $\vdash \sigma'(M_{\mathrm{Sk}}(tac))$, with a substitution σ' that is chosen according to Lem. 6.6 for each skolem symbol s_{Sk} for statements.

It is then possible to construct a closed proof tree of $\vdash \sigma'(M_{\mathrm{Sk}}(tac))$ by transforming H: Each taclet application within H is replaced with the proof tree fragment that is obtained from Lem. 6.4 or Lem. 6.7 (according to the kind of the taclet). □

7 Conclusions

In this paper, we have outlined how to ensure correctness of derived taclets. Because of limited space, we have only sketched the basic idea and covered only some few kinds of schema variables. The presented concept is completely

integrated in the taclet-based KeY prover. Even a greater class of possible *JavaCardDL* taclets is supported.

As future work, it remains

- to generalise the concept of skolemisation of meaning formulas,
- to study quantified first-order logics with skolemised statements as 'atomic' programs, and
- to explore further areas of application, as for example, proofs of program transformation properties.

Taclets are a simple but powerful concept. By their syntactic and semantic simplicity, users are enabled to write new rules and add them to the system easily. We have shown that, despite this fact, the correctness of the rule base can be efficiently ensured—even for a special purpose logic like *JavaCardDL*.

Acknowledgements

We would like to thank Martin Giese and Steffen Schlager for useful comments on earlier versions of this paper, as well as Bernhard Beckert and P.H. Schmitt for fruitful discussions. Also we want to thank the anonymous referees and workshop organisers.

Meta-programming With Built-in Type Equality

Tim Sheard and Emir Pasalic

Computer Science & Engineering Department
OGI School of Science & Engineering
Oregon Health & Sciences University

{sheard,pasalic}@cse.ogi.edu

Abstract

We report our experience with exploring a new point in the design space for formal reasoning systems: the development of the programming language Ωmega. Ωmega is intended as both a practical programming language and a logic. The main goal of Ωmega is to allow programmers to describe and reason about semantic properties of programs from within the programming language itself, mainly by using a powerful type system.

We illustrate the main features of Ωmega by developing an interesting meta-programming example. First, we show how to encode a set of well-typed simply typed λ-calculus terms as an Ωmega data-type. Then, we show how to implement a substitution operation on these terms that is guaranteed by the Ωmega type system to preserve their well-typedness.

Key words: Meta-programming, Meta-language, Equality types

1 Introduction

There is a large semantic gap between what a programmer knows about his program and the way he has to express this knowledge to a formal system for reasoning about that program. While many reasoning tools are built on the Curry-Howard isomorphism, it is often hard for the programmers to conceptualize how they can put this abstraction to work. We propose the design of a language that makes this important isomorphism concrete – proofs are real object that programmers can build and manipulate without leaving their own programming language.

We have explored a new point in the design space of formal reasoning systems and developed the programming language Ωmega. Ωmega is *both* a

practical programming language *and* a logic. These sometimes irreconcilable goals are made possible by embedding the Ωmega logic in a type system based on *equality qualified* types[6]. This design supports the construction, maintenance, and propagation of semantic properties of programs using powerful old ideas about types in novel ways.

For what kind of programming would a language like Ωmega be useful? The rest of this paper describes one possibility.

Meta-programming in Ωmega

Meta-programs manipulate object-programs represented as data. Traditionally, object-language programs are represented with algebraic data-types as *syntactic objects*. This representation preserves syntactic properties of object-language programs (i.e., it is impossible to represent syntactically incorrect object-language programs). In this paper, we explore the benefits of representing object-language programs as data in a manner that preserve important *semantic properties*, in particular *scoping* and *typing*. Representing typed object-languages in a way which preserves semantic properties can lead to real benefits. By preserving typing and scoping properties, we gain assurance in the correctness of a particular language processor (e.g. compiler, interpreter, or program analysis). Such semantics preserving representations statically catch errors introduced by incorrect meta-language programs.

Contributions

The first contribution is an approach to manipulating strongly typed object languages in a manner which is semantics preserving. This approach encodes *well-typed* and *statically scoped* object-language programs as data-types which embed the type of the object-language program in the type of its representation. While this can be done using only the standard extensions to the Haskell 98 type system (using equality types), we use Ωmega, an extension to Haskell inspired by Cheney and Hinze's work on phantom types [6].

The second contribution is an implementation of Cheney and Hinze's ideas that makes programming with well-typed object-language programs considerably less tedious than using equality types in Haskell alone. Our implementation of Ωmega also supports several other features, such as extensible kinds and staging, which we shall not discuss in this paper. This integration creates a powerful meta-programming tool.

The third contribution is a demonstration that semantic properties of meta-programs (i.e., preserving object-language types) can be encoded in the type of the meta-program itself – the programmer need not resort to using another meta-logic to (formally) assure himself that his substitution algorithm preserves typing. We demonstrate this by implementing a type-preserving substitution operation on the object-language of simply typed λ-calculus.

The last contribution is the demonstration that these techniques support the embedding of logical frameworks style judgments into a programming language such as Haskell. This is important because it moves logical style reasoning about programs from the meta-logical level into the programming language.

2 Ωmega: A Meta-language with Type Equality

Type Equality in Haskell. A key technique that inspired the work described in this paper is the encoding of *equality between types* as a Haskell type constructor (`Equal a b`). Thus a non-bottom value (`p::Equal a b`), can be regarded as a proof of the proposition that `a` equals `b`.

The technique of encoding the equality between types `a` and `b` as a polymorphic function of type $\forall\varphi.\ (\varphi\ \texttt{a}) \rightarrow (\varphi\ \texttt{b})$ was proposed by both Baars & Swierstra [2], and Cheney & Hinze [6] at about the same time, and is described somewhat earlier in a different setting by Weirich [20]. We illustrate this by the data-type `Equal :: * → * → *`

```
data Equal a b = Equal (∀φ. (φ a) → (φ b))
cast :: Equal a b → (φ a) → (φ b)
cast (Equal f) = f
```

The logical intuition behind this definition (also known as Leibniz equality [12]) is that two types are equal if, and only if, they are interchangeable in any context. This context is represented by the arbitrary Haskell type constructor φ. Proofs are useful, since from a proof `p :: Equal a b`, we can extract functions that *cast* values of type $(C[\texttt{a}])$ to type $(C[\texttt{b}])$ for type contexts $C[\ ]$. For example, we can construct functions `a2b::Equal a b → a → b` and `b2a::Equal a b → b → a` which allow us to cast between the two types `a` and `b` in the identity context. Furthermore, it is possible to construct combinators that manipulate equality proofs based on the standard properties of equality (transitivity, reflexivity, congruence, and so on).

Equality types are described elsewhere [2], and we shall not belabor their explanation any further. The essential characteristic of programming with type equality in Haskell is the requirement that programmers manipulate proofs of equalities between types using equality combinators. This has two practical drawbacks. First, manipulation of proofs using combinators is tedious. Second, while present throughout a program, the equality proof manipulations have no real computational content – they are used solely to leverage the power of the Haskell type system to accept certain programs that are not typable when written without the proofs. With all the clutter induced by proof manipulation, it is sometimes difficult to discern the difference between the truly important algorithmic part of the program and mere equality proof manipulation. This, in turn, makes programs brittle and rather difficult to

change.

2.1 Type Equality in Ωmega

What if we could extend the type system of Haskell, in a relatively minor way, to allow the type-checker itself to manipulate and propagate equality proofs? Such a type system was proposed by Cheney and Hinze [6], and is one of the ideas behind Ωmega [17]. In the remainder of this paper, we shall use Ωmega, rather than pure Haskell to write our examples. We conjecture that, in principle, whatever it is possible to do in Ωmega, it is also possible to do in Haskell (plus the usual set of extensions), only in Ωmega it is expressed more cleanly and succinctly.

The syntax and type-system of Ωmega has been designed to closely resemble Haskell (with GHC extensions). For practical purposes, we could consider (and use) it as a conservative extension to Haskell. In this section, we will briefly outline the useful differences between Ωmega and Haskell.

In Ωmega, the equality between types is not encoded explicitly (as the type constructor `Equal`). Instead, it is built into the type system, and is used implicitly by the type-checker. Consider the following (fragmentary) data-type definitions. (We adopt the GHC syntax for writing the existential types with a universal quantifier that appears to the left of a data-constructor. We also replace the keyword `forall` with the symbol ∀. We shall write explicitly universally or existentially quantified variables with Greek letters. Arrow types (`->`) will be written as $\rightarrow$, and so on.)

```
data Exp e t
  = Lit Int where  t=Int
  | V (Var e t)
data Var e t
 = ∀γ.  Z where e = (γ,t)
 | ∀γα. S (Var γ t) where e = (γ,α)
```

Each data-constructor in Ωmega may contain a `where` clause which contains a list of equations between types in the scope of the constructor definition. These equations play the same role as the Haskell type `Equal` in Section 2, with one important difference. The user is not required to provide any actual evidence of type equality – the Ωmega type checker keeps track of equalities between types and proves and propagates them automatically.

The mechanism Ωmega uses to keep track of equalities between types is very similar to the constraints that the Haskell type checker uses to resolve class-based overloading. A special qualified type [8] is used to assert equality between types, and a constraint solving system is used to simplify and discharge these assertions. When assigning a type to a type constructor, the equations specified in the where clause just become predicates in a qual-

ified type. Thus, the constructor `Lit` is given the type `∀e t.(t=Int) => Int → Exp e t`. The equation `t=Int` is just another form of predicate, similar to the class membership predicate in the Haskell type (for example, `Ord a => a → a → Bool`).

Tracking equality constraints. When type-checking an expression, the Ωmega type checker keeps two sets of equality constraints: *obligations* and *assumptions*.

Obligations. The first set of constraints is a set of *obligations*. Obligations are generated by the type-checker either when (a) the program constructs data-values with constructors that contain equality constraints; or (b) an explicit type signature in a definition is encountered.

For example, consider type-checking the expression `(Lit 5)`. The constructor `Lit` is assigned the type `∀e t.(t=Int) => Int → Exp e t`. Since `Lit` is polymorphic in `e` and `t`, the type variable `t` can be instantiated to `Int`. Instantiating `t` to `Int` also makes the equality constraint obligation `Int=Int`, which can be trivially discharged by the type checker.

`Lit 5 :: Int → Exp e Int` *with obligation* `Int = Int`

One practical thing to note is that in this context, the data-constructors of `Exp` and `Var` are given the following types:

```
Lit :: ∀e t.t=Int => Int → Exp e t
Z   :: ∀e e' t.e=(e',t) => Var e t
S   :: ∀e t e' t'. e=(e',t') => (Var e' t) → (Var e t)
```

It is important to note that the above qualified types can be *instantiated* to the following types:

```
Lit :: Int → Exp e Int
Z   :: Var (e,t) t
S   :: (Var e' t) → (Var (e',t') t)
```

We have already seen this for `Lit`. Consider the case for `Z`. First, the type variable `e` can be instantiated to `(e',t)`. After this instantiation, the obligation introduced by the constructor becomes `(e',t)=(e',t)`, which can be immediately discharged by the built-in equality solver. This leaves the instantiated type `(Var (e',t) t)`.

Assumptions. The second set of constraints is a set of *assumptions* or *facts*. Whenever, a constructor with a `where` clause is pattern-matched, the type equalities in the where-clause are added to the current set of assumptions in the scope of the pattern. These assumptions can be used to discharge obligations. For example, consider the following partial definition:

```
evalList ::  Exp e t → e → [t]
evalList exp env =
 case exp of Lit n → [n]
```

When the expression exp of type `(Exp e t)` is matched against the pattern `(Lit n)`, the equality `t=Int` from the definition of `Lit` is introduced as an assumption. The type signature of `evalList` induces the obligation that the body of the definition has the type `[t]`. The right-hand side of the `case` expression, `[n]`, has the type `[Int]`. The type checker now must discharge (prove) the obligation `[t]=[Int]`, while using the fact, introduced by the pattern `(Lit n)` that `t=Int`. The Ωmega type-checker uses an algorithm based on congruence-closure [11], to discharge equality obligations. It automatically applies the laws of equality to solve such equations. In this case, the equation is discharged easily using congruence.

3 Ωmega Example: Substitution

Now, we shall develop our main example, showcasing the meta-programming facilities of Ωmega. First, we shall define a sample object-language of simply typed λ-calculus judgments, and then implement a type-preserving substitution function on those terms. While this object-language is quite simple, useful perhaps only for didactic purposes, we have applied our techniques on a wider range of meta-programs and object-languages (e.g., tagless staged interpreters for typed imperative languages, object-languages with modal type systems, and so on [13,14]).

This example demonstrates type-preserving *syntax-to-syntax* transformations between object-language programs. Substitution, which we shall develop in the remainder of this paper, is one such transformation. Furthermore, a correct implementation of substitution can be used to build more syntax-to-syntax transformations: we shall provide an implementation of big-step semantics that uses substitution.

The substitution operation we present preserves object-language typing. As a meta-program, it not only analyzes object-language typing judgments, but also builds new judgments based on the result of that analysis.

3.1 The Simply Typed λ-calculus with Typed Substitutions

Figures 1 and 2 define two sets of typed expressions. The first figure of expressions (Figure 1) is just the simply typed λ-calculus. The second figure (Figure 2) defines a set of typed substitutions. The substitution expressions are taken from the $\lambda\upsilon$-calculus [4]. There are several of other ways to represent substitutions explicitly as terms (see Kristoffer Rose's excellent paper [16] for a comprehensive survey), but we have chosen the notation of $\lambda\upsilon$ for its simplicity.

A substitution expression σ is intended to represent a mapping from de-Bruijn indices to expressions (i.e., a substitution), the same way that λ-expressions are intended to represent functions. As in $\lambda\upsilon$, we define three

Expressions and types

$$\tau \in \mathbb{T} ::= \mathsf{b} \mid \tau_1 \to \tau_2$$
$$\Gamma \in \mathbb{G} ::= \langle\rangle \mid \Gamma, \tau$$
$$e \in \mathbb{E} ::= \mathsf{Var}\ n \mid \lambda_\tau e \mid e_1\ e_2$$

$$\frac{}{\Gamma, \tau \vdash 0 : \tau}\,(\text{Base}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau' \vdash (n+1) : \tau}\,(\text{Weak}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma \vdash \mathsf{Var}\, n : \tau}\,(\text{Var})$$

$$\frac{\Gamma, \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda_{\tau_1}.e : \tau_1 \to \tau_2}\,(\text{Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\,(\text{App})$$

Fig. 1. The simply typed λ-calculus fragment.

Substitutions à la $\lambda\upsilon$ [4]

$$\sigma \in \mathbb{S} ::= e/ \mid \Uparrow(\sigma) \mid \uparrow$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e/ : \Gamma, \tau}\,(\text{Slash}) \qquad \frac{}{\Gamma, \tau \vdash \uparrow : \Gamma}\,(\text{Shift}) \qquad \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma, \tau \vdash \Uparrow(\sigma) : \Gamma', \tau}\,(\text{Lift})$$

Fig. 2. Explicit substitutions fragment.

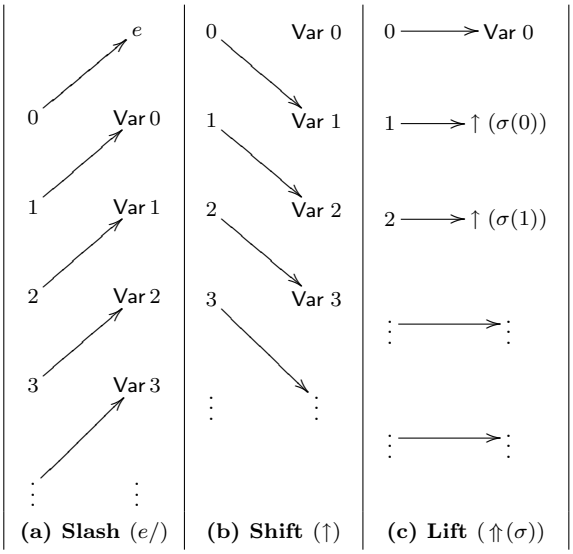


Fig. 3. Substitutions

kinds of substitutions in Figure 2 (see Figure 3 for a graphical illustration):

(i) *Slash* $(e/)$. Intuitively, the slash substitution maps the variable with the index 0 to e, and any variable with the index $n+1$ to $\mathsf{Var}\ n$.

(ii) *Shift* ($\uparrow$). The shift substitution adjusts all the variable indices in a term by incrementing them by one. It maps each variable n to the term $\mathsf{Var}\ (n+1)$.

(iii) *Lift* ($\Uparrow(\sigma)$). The lift substitution ($\Uparrow(\sigma)$) is used to mark the fact that the substitution σ is being applied to a term in a context in which index 0 is bound and should not be changed. Thus, it maps the variable with the index 0 to $\mathsf{Var}\ 0$. For any other variable index $n+1$, it maps it to the term that σ maps to n, with the provision that the resulting term must be adjusted with a shift: $((n+1) \mapsto\uparrow (\sigma(n)))$.

Typing substitutions

The substitution expressions are typed. The typing judgments of substitutions, written $\Gamma_1 \vdash \sigma : \Gamma_2$, indicate that the type of a substitution, in a given type assignment, is another type assignment. The intuition behind the substitution typing judgment is the following: given a term whose variables are assigned types by Γ_2, applying a the substitution σ yields an expression whose variables are assigned types by Γ_1.

Example. We describe a couple of example substitutions.

(i) Consider the substitution $(\mathsf{True}/)$. This substitution maps the variable with the index 0 to the Boolean constant `True`. The type of this substitution is $\Gamma \vdash \mathsf{True}/ : \Gamma, \mathsf{Bool}$. In other words, given any type assignment, the substitution $(\mathsf{True}/)$ can be applied in any context where the variable 0 is assigned type Bool.

(ii) Consider the substitution $\sigma = (\Uparrow(\mathsf{True}/))$. σ is the substitution that replaces the variable with the index 1 with the constant True.

Recall that the type of any substitution θ under a type assignment Γ, is a type assignment Δ (written $\Gamma \vdash \theta : \Delta$), such that for any expression e' to which the substitution θ is applied, the following must hold $\Delta \vdash e' : \tau$ and $\Gamma \vdash \theta(e') : \tau$.

So, what type should we assign to σ? When applied to an expression, a lift substitution ($\sigma = \Uparrow(\mathsf{True}/)$) does not change the variable with the index 0. Thus, when typing σ as $\Gamma \vdash \sigma : \Delta$, we know something about the shape of Γ and Δ. Namely, for some Δ', we know that $\Delta = (\Delta', \tau)$, and for some Γ', we know that $\Gamma = (\Gamma', \tau)$. The type assignments Δ' and Γ' are determined by the sub-substitution $\mathsf{True}/$, yielding the following

typing derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{True} : \mathsf{Bool}}\,\text{Const}}{\Gamma \vdash \mathsf{Bool}/ : \Gamma, \mathsf{Bool}}\,\text{Slash}}{\Gamma, \tau \vdash \Uparrow(\mathsf{Bool}/) : \Gamma, \mathsf{Bool}, \tau}\,\text{Lift}$$

There are three typing rules for the substitutions (Figure 2):

(i) *Slash* ($e/$). A slash substitution $e/$ replaces the 0-index variable in an expression by e. Thus, in any context Γ, where e can be given type τ, the typing rule requires the substitution to work only on expressions in the type assignment Γ, τ, where the 0-index variable is assigned the type τ. Since the slash substitution also decrements the indexes of the remaining variables, they are all shifted to the right by one place, so that the remaining free variables can be assigned their old types in Γ after the substitution is applied.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e/ : \Gamma, \tau}\,(\text{Slash})$$

(ii) *Shift* ($\uparrow$). The shift substitution maps all variables n to $\mathsf{Var}\ (n+1)$. Thus, given a term whose variables are assigned type a by Γ, after performing the shift substitution, the types in the type assignment must for each variable must "move" to the left by one position. This is done by appending an arbitrary type τ for the variable with the index 0, which cannot occur free in the term after the substitution is performed.

$$\frac{}{\Gamma, \tau \vdash \uparrow : \Gamma}\,(\text{Shift})$$

(iii) *Lift* ($\Uparrow(\sigma)$). For any variable index $(n+1)$ in a term, the substitution $\Uparrow(\sigma)$ applies σ to n and then shifts the resulting term. Thus, the 0-index term in the type assignment remains untouched, and the rest of the type assignment is as specified by σ:

$$\frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma, \tau \vdash \Uparrow(\sigma) : \Gamma', \tau}\,(\text{Lift})$$

Applying substitutions

In the remainder of this Section, we show how to implement a function (we call it subst) that takes a substitution expression σ, a λ-expression e, and returns an expression such that all the indices in e have been replaced according

Substitution on expressions

$$(\cdot,\cdot) \Rightarrow \cdot \subset \mathbb{S} \times \mathbb{E} \times \mathbb{E}$$

$$\frac{2}{\begin{array}{l}(\sigma, e_1) \Rightarrow e_1' \\ (\sigma, e_2) \Rightarrow e_2'\end{array}} \quad (\sigma, (e_1\ e_2)) \Rightarrow e_1'\ e_2' \quad \frac{(\Uparrow(\sigma), e) \Rightarrow e'}{(\sigma, \lambda.e) \Rightarrow \lambda e'} \quad \frac{(\sigma, n) \Rightarrow e}{(\sigma, \mathsf{Var}\ n) \Rightarrow e}$$

Substitution on variables

$$(\cdot,\cdot) \Rightarrow \cdot \subset \mathbb{S} \times \mathbb{N} \times \mathbb{E}$$

$$\frac{}{(e/, 0) \Rightarrow e} \quad \frac{}{(e/, n+1) \Rightarrow \mathsf{Var}\ n} \quad \frac{}{(\Uparrow(\sigma), 0) \Rightarrow \mathsf{Var}\ 0}$$

$$\frac{(\sigma, n) \Rightarrow e \quad (\uparrow, e) \Rightarrow e'}{(\Uparrow(\sigma), n+1) \Rightarrow e'} \quad \frac{}{(\uparrow, n) \Rightarrow \mathsf{Var}\ (n+1)}$$

Fig. 4. Applying substitutions to terms

the substitution. In the simply typed λ-calculus, substitution preserves typing, so we expect the following property to be true of the substitution function subst: if $\Gamma \vdash \sigma : \Delta$ and $\Delta \vdash e : \tau$, then $\Gamma \vdash \mathsf{subst}\ \sigma\ e : \tau$.

How should subst work? Figure 4 presents two judgments, $(\sigma, e_1) \Rightarrow e_2$ and $(\sigma, n) \Rightarrow e$, which describe the action of substitutions on expressions and variables, respectively. These judgments are derived from the reduction relations of the $\lambda\upsilon$-calculus [4]. It is not difficult to show that this reduction strategy indeed does implement capture avoiding substitution sufficient to perform β reductions (see Benaissa, Lescanne & al. [4] for proofs).

4 Implementing Substitution in Ωmega

Next, we show how to implement this substitution operation in Ωmega, using expression and substitution judgments instead of expressions and substitution expressions.

4.1 Judgments

The expression and substitution judgments can be easily encoded in Ωmega. The data-types `Var` and `Exp` encode expression and variable judgments presented in Figure 1.

```
data Var e t = ∀d.    Z where e = (d,t)
             | ∀d t2. S (Var d t) where e = (d,t2)

data Exp e t =         V   (Var e t)
```

```
  | ∀t1 t2. Abs (Exp (e,t1) t2)
            where t = t1 → t2
  | ∀t1.    App (Exp e (t1 → t))
                (Exp e t1)
```

The judgment `Var` implements the lookup and weakening rules for variables. Just as in the judgment of Figure 1, there are two cases:

(i) First, there is the constructor `Z`. This constructor translates the definition of Figure 1 directly: the `where`-clause requires the type system of Ωmega to prove that there exists some environment γ such that the environment `t` is equal to γ extended by `t`.

(ii) The second constructor, `S` takes a judgment of type (`Var` γ `t`), and a requirement that the environment `e` is equal to the pair (γ, α), where both γ and α are existentially quantified.

The names `S` and `Z` are chosen to show how the judgments for variable are structurally the same as the natural number indices. Finally, the subjudgments for the variable case are "plugged" into the definition of `Exp e t` using the constructor `V`.

The type of expression judgments (`Exp e t`) is constructed in a similar fashion. We shall only explain the abstraction case in some detail. The constructor `Abs` takes as its argument a judgment of type (`Exp (e,t1) t2`): an expression judgment of type `t2` in the type assignment `e`, extended so that it assigns the variable 0 the type `t1`. If this argument can be supplied, then the result type of the `Abs` judgment is the function type (`t1` → `t2`), as indicated by the `where`-clause.

Next, we define a data-constructor `Subst gamma delta` that represents the typing judgments for substitutions . The type constructor `Subst gamma delta` represents the typing judgment $\Gamma \vdash \sigma : \Delta$ presented in Figure 2.

```
data Subst gamma delta =
  ∀t1.    Shift
          where gamma = (delta,t1)
| ∀t1.    Slash (Exp gamma t1)
          where delta = (gamma,t1)
| ∀del1 gam1 t1. Lift (Subst gam1 del1)
          where delta = (del1,t1),
                gamma = (gam1,t1)
```

4.2 Substitution

Finally, we define the substitution function `subst`. It has the following type:

```
subst :: Subst gamma delta →
```

```
1 subst :: Subst gamma delta →
2           Exp delta t → Exp gamma t
3 subst s (App e1 e2)  = App (subst s e1) (subst s e2)
4 subst s (Abs e) = Abs (subst (Lift s) e)
5 subst (Slash e) (V Z) =  e
6 subst (Slash e) (V (S n)) = V n
7 subst (Lift s) (V Z) = V Z
8 subst (Lift s) (V (S n)) = subst Shift (subst s (V n))
9 subst (Shift) (V n) = V (S n)
```

Fig. 5. Substitution in simply typed λ-calculus.

```
Exp delta  t → Exp gamma t
```

It takes a substitution whose type is `delta` in some type assignment `gamma`, an expression of type `t` that is typed in the type assignment `delta`, and produces an expression of type `t` typable in the type assignment `gamma`.

We will discuss the implementation of the function `subst` (Figure 5) in more detail. In several relevant cases, we shall describe the process by which the Ωmega type-checker makes sure that the definitions are given correct types. Recall that every pattern-match over one of the `Exp` or `Subst` judgments may introduce zero or more equations between types, which are then available to the type-checker in the body of a case (or function definition). The type checker may use these equations to prove that two types are equal. In the text below, we sometimes use the type variables `gamma` and `delta` for notational convenience, but also Skolem constants like `_1`. These are an artifact of the Ωmega type-checker (they appear when pattern-matching against values that may contain existentially quantified variables) and should be regarded as type constants.

(i) The application case (line 3) simply applies the substitution to the two sub-expression judgments and then rebuilds the application judgment from the results.

(ii) The abstraction case (line 4) pushes the substitution under the λ-abstraction. It may be interesting to examine the types of the various subexpressions in this definition.

`Abs e`	:	`Exp delta t`, where `t=t1 → t2`
`e`	:	`Exp (delta,t1) t2`
`s`	:	`Subst gamma delta`
`Lift s`	:	`Subst (gamma,t1) (delta,t1)`
`subst (Lift s) e`	:	`Exp (gamma,t1) t2`

The body of the abstraction, `e` has the type `(delta,t1)`, where `t1`

is the type of the domain of the λ-abstraction. In order to apply the substitution `s` to the body of the abstraction (`e`), we need a substitution of type `(Subst (gamma,t1) (delta,t1))`. This substitution can be obtained by applying `Lift` to `s`. Then, recursively applying `subst` with the lifted substitution to the body `e`, we obtain an expression of type `(Exp (gamma,t1) t2)`, from which we can construct a λ-abstraction of the `(Exp gamma (t1 → t2))`.

(iii) The variable-slash case (line 5-6). There are two cases when applying the slash substitution to a variable expression:

(a) Variable 0. The substitution `(Slash e)` has the type `(Subst (gamma) (gamma,t))`, and contains the expression `e :: Exp gamma t`. The expression `(V Z)` has the type `(Exp (delta,t) t)`. Pattern matching introduces the equation `gamma=delta`, and we can use `e` to replace `(V Z)`.

```
Slash e  ::  (Subst (gamma) (gamma,t))

e        ::  Exp gamma t
```

(b) Variable $n+1$. Pattern matching on the substitution argument introduces the equation `delta=(gamma,t1)`. Pattern matching against the expression `(V (S n))` introduces the equation `delta=(gamma',t)`, for some `gamma'`. The expression result expression `(V n)` has the type `(Exp gamma' t)`. The type checker then uses the two equalities to prove that it has the type `(Exp gamma t)`. It does this by first using congruence to prove that `gamma=gamma'`, and then by applying this equality to obtain `Exp gamma' t = Exp gamma t`.

```
Slash e     ::  Subst gamma (gamma,t)

(V (S n))   ::  Exp delta t
```

(iv) The variable-lift case (lines 7-8). There are two cases when applying the lift substitution to a variable expression.

(a) Variable 0. This case is easy because the lift substitution places makes no changes to the variable with the index 0. We are able simply to return `(V Z)` as a result.

(b) Variable $n+1$. The first pattern (`Lift s :: Subst gamma delta`), on the substitution, introduces the following equations:

```
delta = (d',_1),
gamma = (g',_1)
```

The pattern on the variable (`V(S n):: Var delta t`) introduces the equation

```
delta = (d2,_2)
```

The first step is to apply the substitution `s` of type `(Subst g' d')` to a decremented variable index `(V n)` which has the type `n ::`

`Var d2 t`. To do this, the type checker has to show that `g'=d2`, which easily follows from the equations introduced by the pattern, yielding a result of type `(Exp g' t)`. Applying the `Shift` substitution to this result yields an expression of type `(Exp (g',a) t)` (where `a` is can be any type). Now, equations above can be used to prove that this expression has the type `(Exp gamma t)` using the equation `gamma=(g',_1)`.

(v) Variable-shift case (line 9). Pattern matching on the `Shift` substitution introduces the equation `gamma = (delta,_1)`. The expression has the type `(Exp delta t)`. Applying the successor to the variable results in an expression `(V (S n))` of type `(Exp (delta,a) t)`. Immediately, the type checker can use the equation introduced by the pattern to prove that this type is equal to `(Exp gamma t)`.

We have defined type-preserving substitution simply typed λ-calculus judgments. Recall, that since equality proofs can be encoded in Haskell, it should be possible (with certain caveats) to implement the function `subst` in Haskell (with a couple of GHC extensions). It is worth noting that Ωmega has proven very helpful in writing such complicated functions: explicitly manipulating equality proofs for such a function in Haskell, would result in code that is both verbose and difficult to understand.

5 A Big-step Evaluator

Finally, we implement a simple evaluator based on the big-step semantics for the λ-calculus. The evaluation relation is given by the following judgment:

$$\frac{}{\lambda e \Rightarrow \lambda e} \qquad \frac{}{x \Rightarrow x} \qquad \frac{e_1 \Rightarrow \lambda e' \quad (e_2/, e') \Rightarrow e_3 \quad e_3 \Rightarrow e''}{e_1\ e_2 \Rightarrow e''}$$

Note that in the application case, we first use the substitution $(e_2/, e') \Rightarrow e_3$ to substitute the argument e_2 for the variable with index 0 into the body of the λ-abstraction.

The big-step evaluator is implemented as the function `eval` which takes a well-typed expression judgment of type `(Exp delta t)`, and returns judgments of the same type. The evaluator reduces β-redices using a call-by-name strategy, relying upon the substitution implemented above.

```
eval :: Exp delta t -> Exp delta t
eval (App e1 e2) =
  case eval e1 of
    Abs body -> eval (subst (Slash e2) body)
eval x = x
```

Note that the type of the function `eval` statically ensures that it preserves the typing of the object language expressions it evaluates, with the usual caveats that the `Exp`s faithfully encode well-typed λ-expressions.

Finally, let us apply the big-step evaluator to a simple example. Consider the expression, `example`.

```
example :: Exp gamma (a → a)
example = (Abs (V Z)) `App` ((Abs (Abs (V Z)))
          `App` (Abs (V Z)))
-- example = (λ x.x) ((λ y. (λ z.z))) (λ x.x)
```

The expression `example` evaluates to the identity function. Applying `eval` to it yields precisely that result:

```
evExample = eval example
-- evExample = (Abs (V Z)) : Exp gamma (a → a)
```

6 Related Work

Implementations of simple interpreters that use equality proof objects implemented as Haskell datatypes, have been given by Weirich [20] and Baars and Swierstra [2]. Baars and Swierstra use an untyped syntax, but use equality proofs to encode dynamically typed values. Hinze and Cheney [5,6] have recently resurrected the notion of "phantom type," first introduced by Leijen and Meijer [10]. Hinze and Cheney's phantom types are designed to address some of the problems that arise when using equality proofs to represent type-indexed data. Their main motivation is to provide a language in which polytypic programs, such as generic traversal operations, can be more easily written. Cheney and Hinze's system bears a strong similarity to Xi et al.'s *guarded recursive datatypes* [21], although it seems to be a little more general.

We adapt Cheney and Hinze's ideas to meta-programming and language implementation. We incorporate their ideas into a Haskell-like programming language. The value added in our work is additional type system features (extensible kinds and rank-N polymorphism, not used in this paper) applying these techniques to a wide variety of applications, including the use of typed syntax, the specification of semantics for patterns, and its combination with staging to obtain tagless interpreters, and the encoding of logical framework style judgments as first class values within a programming language.

Simonet and Pottier [18] proposed a system of *guarded algebraic data types,* which seem equivalent in expressiveness to *phantom types*, *guarded recursive datatype constructors*, and Ωmega's equality qualified (data)types. They present a type system for guarded algebraic data types as an extension to the HM(X) [19] type system, and describe a type inference algorithm. They prove a number of important properties about the type system and the inference

algorithm (e.g., type soundness, correctness, and so on).

The technique of manipulating well-typedness judgments has been used extensively in various logical frameworks [7,15]. We see the advantage of our work here in translating this methodology into a more main-stream functional programming idiom. Although our examples are given in Ωmega, most of our techniques can be adapted to Haskell with some fairly common extensions.

In previous work, we have used the techniques and programming language extensions described above to address the problem of tagless interpreters in meta-programming [14]. Tagless interpreters can easily be constructed in dependently typed languages such as Coq [3] and Cayenne [1]. These languages, however, do not support staging, nor have they gained a wide audience in the functional programming community. Programming with well-typed object-language syntax, applied to the problem of constructing tagless staged interpreters, has been shown possible in a meta-language (provisionally called MetaD) with staging and dependent types [14]. The drawback of this approach is that there is no "industrial strength" implementation for such a language. In fact, the judgment encoding technique presented in this paper is basically the same, except that instead of using a dependently typed language, we encode the necessary machinery in a language which is arguably more recognizable to Haskell programmers. By using explicit equality types, everything can be encoded using the standard GHC extensions to Haskell 98. Ωmega adds further ease of use to these techniques, relieving the programmer of the responsibility of explicitly manipulating equality proofs.

A technique using *indexed type systems* [22], a restricted and disciplined form of dependent typing, has been used to write interpreters and source-to-source transformations on typed terms [21]. The meta-language with guarded recursive datatype constructors, used by Xi & al., seems to be roughly equivalent in expressive power to Ωmega. Ωmega, however, is equipped with additional features, such as staging, which may give it a wider range of useful applications.

7 Discussion and Future Work

Meta-language Implementation. The meta-language used in this paper can be seen as a (conservative) extension of Haskell, with built-in support for equality types. It was largely inspired by the work of Cheney and Hinze. The meta-language we have used in our examples in this papers is the functional language Ωmega, a language designed to be as similar to Haskell. We have implemented our own Ωmega interpreter, similar in spirit and capabilities to the Hugs interpreter for Haskell [9].

Theoretical work demonstrating the consistency of type equality support in a functional language has been carried out by Cheney and Hinze. We

have implemented these type system features into a type inference engine, combining it with an equality decision procedure to manipulate type equalities. The resulting implementation has seen a good deal of use in practice, but more rigorous formal work on this type inference engine is required.

Polymorphism and Binding Constructs in Types. The object-language of the example presented in this paper (Figure 1), is simply typed: there are no binding constructs or structures in any index arguments to `Exp`. If, however, we want to represent object languages with universal or existential types, we will have to find a way of dealing with *type constructors* or *type functions* as index arguments to judgments, which is difficult to do in Haskell or Ωmega. We are currently working on extending the Ωmega type system to do just that. This would allow us to apply our techniques to object languages with more complex type systems (e.g., polymorphism, dependent types, and so on).

Logical Framework in Ωmega. The examples presented in this paper succeed because we manage to encode the usual logical-framework-style inductive predicates into the type system of Ωmega. We have acquired considerable experience in doing this for typing judgments, lists with length, logical propositions, and so on. What is needed now is to come up with a formal and general scheme of translating such predicates into Ωmega type constructors, as well as to explore the range of expressiveness and the limitations of such an approach. We intend to work on this in the future.

References

[1] Augustsson, L. and M. Carlsson, *An exercise in dependent types: A well-typed interpreter*, in: *Workshop on Dependent Types in Programming*, Gothenburg, 1999, available online from `www.cs.chalmers.se/~augustss/cayenne/interp.ps`.

[2] Baars, A. I. and S. D. Swierstra, *Typing dynamic typing*, in: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, SIGPLAN Notices 37(9) (2002).

[3] Barras, B., S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi and B. Werner, *The Coq Proof Assistant Reference Manual – Version V6.1*, Technical Report 0203, INRIA (1997).

[4] Benaissa, Z.-E.-A., D. Briaud, P. Lescanne and J. Rouyer-Degli, *$\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation*, Journal of Functional Programming **6** (1996), pp. 699–722.

[5] Cheney, J. and R. Hinze, *A lightweight implementation of generics and dynamics.*, in: *Proc. of the workshop on Haskell* (2002), pp. 90–104.

[6] Cheney, J. and R. Hinze, *Phantom types* (2003), available from http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf.

[7] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, in: *Proceedings Symposium on Logic in Computer Science* (1987), pp. 194–204, the conference was held at Cornell University, Ithaca, New York.

[8] Jones, M. P., "Qualified types :–theory and practice," Ph.D. thesis, Keble College, Oxford University (1992).

[9] Jones, M. P., *The hugs 98 user manual* (200).

[10] Leijen, D. and E. Meijer, *Domain-specific embedded compilers*, in: *Proceedings of the 2nd Conference on Domain-Specific Languages* (1999), pp. 109–122.

[11] Nelson, G. and D. C. Oppen, *Fast decision procedures based on congruence closure*, Journal of the ACM **27** (1980), pp. 356–364.

[12] Nordström, B., K. Peterson and J. M. Smith, "Programming in Martin-Lof's Type Theory," International Series of Monographs on Computer Science **7**, Oxford University Press, New York, NY, 1990, currently available online from first authors homepage.

[13] Pašalić, E., "Heterogeneous Meta-programming," Ph.D. thesis, Oregon Health and Sciences University, OGI School of Science & Engineering (2004), forthcoming.

[14] Pašalić, E., W. Taha and T. Sheard, *Tagless staged interpreters for typed languages*, in: *The International Conference on Functional Programming (ICFP '02)*, ACM, Pittsburgh, USA, 2002.

[15] Pfenning, F. and C. Schürmann, *System description: Twelf — A meta-logical framework for deductive systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, LNAI **1632** (1999), pp. 202–206.

[16] Rose, K. H., *Explicit substitution – tutorial & survey*, Technical Report LS-96-3, BRICS, Universitty of Århus (1996), bRICS Lecture Series.

[17] Sheard, T., E. Pasalic and R. N. Linger, *The ωmega implementation.*, Available on request from the author. (2003).

[18] Simonet, V. and F. Pottier, *Constraint-based type inference for guarded algebraic data types* (2003), submitted for publication.

[19] Sulzmann, M., M. Odersky and M. Wehr, *Type inference with constrained types*, in: *FOOL4: 4th. Int. Workshop on Foundations of Object-oriented programming Languages*, 1997.

[20] Weirich, S., *Type-safe cast: functional pearl.*, in: *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, ACM Sigplan Notices **35.9** (2000), pp. 58–67.

[21] Xi, H., C. Chen and G. Chen, *Guarded recursive datatype constructors*, in: C. Norris and J. J. B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, ACM SIGPLAN Notices **38, 1** (2003), pp. 224–235.

[22] Xi, H. and F. Pfenning, *Dependent types in practical programming*, in: *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, ACM, New York, NY, 1999, pp. 214–227.

8 Acknowledgment

The work described in this paper is supported by the National Science Foundation under the grant CCR-0098126.

Imperative LF Meta-Programming

Aaron Stump

Dept. of Computer Science and Engineering Washington University in St. Louis St. Louis, Missouri, USA Web: http://cl.cse.wustl.edu/

Abstract

Logical frameworks have enjoyed wide adoption as meta-languages for describing deductive systems. While the techniques for representing object languages in logical frameworks are relatively well understood, languages and techniques for meta-programming with them are much less so. This paper presents work in progress on a programming language called Rogue-Sigma-Pi (RSP), in which general programs can be written for soundly manipulating objects represented in the Edinburgh Logical Framework (LF). The manipulation is sound in the sense that, in the absence of runtime errors, any putative LF object produced by a well-typed RSP program is guaranteed to type check in LF. An important contribution is an approach for soundly combining imperative features with higher-order abstract syntax. The focus of the paper is on demonstrating RSP through representative LF meta-programs.

Key words: Meta-Programming, Logical Frameworks, Rewriting Calculus

1 Introduction

Applications using a logical framework such as the Edinburgh Logical Framework (LF) [7] very frequently need meta-programs which produce or manipulate LF encodings of derivations. For example, proof-producing decision procedures like the CVC ("Cooperating Validity Checker") system or the Touchstone theorem prover from Necula's PCC system emit proof objects for formulas they report valid [18,10]. The Princeton and Yale Foundational Proof-Carrying Code (FPCC) projects both rely on tools that automatically generate LF derivations [21,6].

The present work contributes to the study of sound meta-programming for LF. A meta-programming language for LF is described called Rogue-Sigma-Pi (RSP). RSP extends LF in a type-safe way with convenient meta-programming constructs: pattern-matching, general recursion, a limited form of dependent records, and expression attributes (for mutable state). The combination of mutable state with *higher-order abstract syntax* (HOAS) [12] is quite delicate.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

An important contribution of the present work in progress is a (currently just conjectured) sound approach supporting this combination.

The meta-theoretic properties of RSP necessary for its intended use are type safety and conservativity with respect to LF. Due to the inclusion of dependent records (as also happens with the inclusion of pairs: see, e.g., [16]), unicity of types fails in LF and hence in RSP. The approach adopted here is to rely on bottom-up type computation, but add support for explicit ascriptions to guide the type computation to a desired type. Conservativity with respect to LF has one qualification. In RSP, run-time errors like failure of pattern matching can occur, which result in an RSP term's evaluating to `Null`. The statement of conservativity is this: any RSP value of LF type is guaranteed to be an LF object, as long as it contains no `Null`s. This form of conservativity together with type safety guarantees the property mentioned above: in the absence of run-time errors, any putative LF object produced by evaluation of an RSP term is truly an LF object.

The paper informally introduces RSP (Section 3), and then shows how the language is used in practice on several example meta-programs (Section 4). While it is not hard to formalize the basic idea of RSP's type system, the exact formalization needed to achieve the meta-theoretic results is work in progress. The paper presupposes knowledge of LF (see, e.g., [11]).

2 Related Work

LF meta-programming plays a crucial role in several application domains. In the CVC project, a cooperating validity checker was instrumented to produce proofs in a variant of LF [18]. It was originally hoped that producing proofs would help catch bugs in CVC. This was actually quite rare, since most bugs were failures of completeness, where validity proofs are of no obvious relevance. Nevertheless, there were constantly bugs in CVC's proof-producing code. Given a straightforward implementation in C++, it is easy to write code which erroneously generates malformed proofs. Such an error is caught only when a malformed proof is produced for some input formula and then run through a proof checker. Tracking down the causes of such failed proofs is extremely time consuming.

Several approaches have been proposed for writing type-safe LF meta-programs. Appel and Felty use Twelf to implement partially correct tactics and decision procedures [1]. In Twelf, sets of LF types receive a computational interpretation as logic programs [13]. LF's typing establishes that any proof produced by a successful computation is guaranteed to check. The program may still fail due to run-time errors such as non-termination or match failure. The Delphin language of Schürmann is a pure functional language for meta-programming with LF encodings [17]. Delphin carefully places pattern-matching and recursion over LF for type safe manipulation for LF encodings. Other less closely related systems include Cayenne and Alf [2,9].

3 Overview of RSP

RSP is a proper extension of LF. We adopt the following notation for LF. We write `x:A => B` for dependent product type $\Pi^x A : B.$, and `x:A -> M` for $\lambda x : A.\, M$. We call the latter *representational* abstractions, since they will be used in RSP solely for representation using HOAS. Another kind of abstraction will be used for computation. In RSP's operational semantics, computation occurs in the bodies of representational abstractions, although not in the bodies of computational abstractions. Insofar as a representational abstraction is simply a parameterization of its body, evaluating that body without any argument given is not unreasonable. Notationally, if `x` does not occur in `B`, we write `A => B` instead of `x:A => B`. Application is written explicitly with infix `@`, and `x @ y` is sometimes written `x(y)` when `x` is a variable or constant.

The features RSP adds to LF are briefly these. RSP has dependently typed **expression attributes**, which can be read (`X.a`) and written (`X.a := Y`). The type of an attribute is just like a dependent function type, except with `=a>` instead of `=>`. Types of attribute reads are computed as for applications. The type system restricts attributes in attribute reads and attribute writes to be just constant symbols. Using such attributes, we support recursion by writing recursive equations: a functional expression containing attribute read $a.b$, say, is written into the b attribute of a.

The computational abstractions mentioned above are dependently typed **pattern abstractions**, typed with yet another arrow, `=c>`. These are of the form `x\P\Δ -> M`. Here, `x` is a name for the whole input to the abstraction, `P` is the pattern the input should match, Δ is a context for pattern variables in `P`, and `M` is the body of the abstraction. If Δ is empty we write `null` for it. Pattern abstractions are applied to target expressions by matching the pattern against the target. The notion of matching used in RSP is just syntactic first-order matching. **Deterministic choice** allows pattern abstractions of the same type to be combined: we write `M|N` for deterministic choice between `M` and `N`. An application of a |-expression is evaluated by using the first abstraction (from left to right) whose pattern matches the target expression. RSP uses `Null(A)` for match failure and also for reads of uninitialized attributes, for every type `A`.

RSP's pattern abstractions are inspired by those of Pure Pattern Type Systems (P^{2}TS) [3], with an important difference. In P^{2}TS, the range type of a pattern abstraction is allowed to depend on the pattern variables. Hence, the pattern and its context must become part of the domain type of the abstraction, and pattern abstractions receive types $\Pi P : \Delta\,.\, B$, where P is the pattern, Δ is the context for pattern variables, and B is the type of the body. Abstractions can still only be connected by choice if they have exactly the same type. Since patterns are part of types, this leads to the following serious drawback of the P^{2}TS approach: abstractions can only be connected if they are attempting to match the same pattern. This is a severe restriction, since

$$\textit{(c-arrow-I)}\quad \frac{\begin{array}{l}\text{pattern } \Gamma\ \Delta\ P \\ \Gamma, \Delta \vdash P :: A \qquad \Gamma, \Delta, x = P \vdash M :: B \qquad \Gamma \vdash x : A \texttt{ =c> } B :: *\end{array}}{\Gamma \vdash x \backslash P \backslash \Delta \rightarrow M \ :: \ x : A \texttt{ =c> } B}$$

$$\textit{(rec-I)}\quad \frac{\Gamma \vdash M : A \qquad \Gamma, y = M \vdash N :: B \qquad \Gamma \vdash x : A, B :: *}{\Gamma \vdash (x = M, y.N) :: x : A, B}$$

Fig. 1. Selected RSP typing rules

it means programs cannot use case analysis to take different action based on the form of expressions. The present approach solves this problem by not including patterns as part of the domain types of abstractions. But this requires the range type not to depend on pattern variables. The range type can still depend, however, on the name `x` for the entire target expression to which the pattern abstraction is being applied.

This approach to dependent pattern abstractions is enforced by the typing rule (c-arrow-I) in Figure 1. As mentioned above, the exact formalization of RSP is not yet complete, so this is just the essential idea: note that things like the definition of pattern used in the first premise must be formulated with great care. The third premise adds an equation to the context, which is used when checking convertibility. The fourth premise ensures that the pattern variables from Δ are not used in B.

Finally, it turns out to be a practical necessity to have some kind of pairing mechanism. This is mainly to allow pattern abstractions to perform simultaneous pattern matching on a dependently typed *bundle* of objects. Initially, RSP adopted dependent sum types, following [16]. It has become clear, however, that a limited form of **dependent record types** would lead to more readable meta-programs. This is because in some applications, it becomes necessary to manipulate rather large bundles, and it is easier to read code which refers to elements of a large bundle by name instead of by a sequence of projections. Fortunately, the uses of bundles in RSP does not seem to require some of the features which complicate record types. We do not need subtyping on record types, nor, apparently, manifest fields in record types. We adopt right-associating records as in [14]. We write $\{x : A, B\}$ for the right-associating record with leftmost field x and remaining fields B. Then `(x = M, y.N)` denotes the dependent record with leftmost field x storing element `M`, and remaining fields `N`, where `N` is allowed to use `y` as another name for `M`. This follows the approach originally proposed in [8], where each field has a label and an associated bound variable (to avoid variable capture during substitution). We often elide the binding and write just `(x = M, N)`. By using `x`

```
O : type.
IMP : (O => O => O).
FALSE : O.

Pf : (O => type).
Dn : (P : O => Pf(IMP @ (IMP @ (IMP @ P @ FALSE) @ FALSE) @ P)).
K : (P : O => Q : O => Pf(IMP @ P @ (IMP @ Q @ P))).
S : (P : O => Q : O => R : O =>
     Pf(IMP @ (IMP @ P @ (IMP @ Q @ R))
            @ (IMP @ (IMP @ P @ Q) @ (IMP @ P @ R)))).
MP : (P : O => Q : O => Pf(IMP @ P @ Q) => Pf(P) => Pf(Q)).

I : type.
EQUALS : (I => I => O).
Eqrefl : (x : I => Pf(EQUALS @ x @ x)).
Eqsymm : (x : I => y : I => Pf(EQUALS @ x @ y) =>
          Pf(EQUALS @ y @ x)).
Eqtrans : (x : I => y : I => z : I => Pf(EQUALS @ x @ y) =>
           Pf(EQUALS @ y @ z) => Pf(EQUALS @ x @ z)).
```

Fig. 2. LF signature: classical logic with equality (no quantifiers)

as an alias for `M` in `N` (which is discussed but not supported in [14]), we can often guide bottom-up type computation to some desired type for `M`. In some cases, however, we still need explicit ascriptions `M:A`. The typing rule should be essentially the (`rec-I`) rule of Figure 1.

RSP terms are parsed with the following precedences from tightest to loosest binding: attribute read and projections, application, record formation, arrows, and deterministic choice. So the first term below is fully parenthesized as the second (and evaluates to whatever value is stored for attribute `b` of expression `a`)

```
(x\a\null -> x.b | x\y\y:I -> x.c) @ a
((x\a\null -> (x.b)) | (x\y\y:I -> (x.c))) @ a.
```

4 Meta-Programming Examples

We consider two example meta-programs that manipulate LF encodings of proofs in classical first-order logic with equality. All the code has been type checked and run on sample inputs using a prototype implementation of RSP [1]. This prototype is written in Rogue, a version of the untyped Rewriting Calcu-

[1] This implementation does not support all the features of records yet, in particular field accesses. Versions of the examples using projections instead of field accesses have been checked and run on sample inputs.

```
1.  rank : (I =a> Int).
2.  findp : (x : I =a> {y:I, Pf(Equals @ x @ y)}).
3.  find : (base =a> x : I =c> {y:I, d:Pf(Equals @ x @ y)}).
4.
5.  uf.find := x \ y \ y : I ->
6.    Let(fx, x.findp,
7.    Ite(fx,
8.    Let(ffx, uf.find @ fx.1,
9.       x.findp := (y = ffx.1,
10.                  d = Eqtrans @ x @ fx.1 @ y @ fx.2 @ ffx.2)),
11.   Drop1(x.rank := 0, (y = x,
12.                       d = Eqrefl(x) : Pf(Equals @ x @ y))))).
```

Fig. 3. The RSP code for find

lus [19], which is essentially an untyped version of RSP. We encode our logic in a standard way as the LF signature of Figure 2. Our examples do not deal with quantifiers, so they are omitted for space reasons. Also, constructs to form first-order logic terms are omitted. The examples also make use of the following non-logical LF declarations, whose role is further explained below:

```
base : type.
uf : base.
dt : base.
```

4.1 Proof-Producing Union-Find

The first example is a proof-producing version of the well-known union-find algorithm (see, e.g., [4, Chapter 22]). Recall that this algorithm maintains disjoint sets of elements in balanced lazily path-compressed trees. The `union` operation takes two elements and merges their trees by making the root of the shallower one (as bounded by its *rank*) point to the root of the deeper one. The `find` operation takes an element `x` and returns the root of its tree. It modifies the pointers (called *find pointers*) encountered along the path from `x` to the root so that they all point directly to the root. Proof-producing `union` additionally takes in (the LF encoding of) a proof that the two given elements are equal. Proof-producing `find` additionally returns a proof that `x = r` where `x` is the input element and `r` is the root element which `find` returns. The underlying data structure is augmented so that every find pointer from a node `x` to a node `x.findp` has associated with it a proof that `x = x.findp`.

Figure 3 shows typing declarations and the code just for `find`. Lines are numbered for reference. The typing declarations declare attributes `rank` and `findp`, as well as `find` (lines 1-3). The latter is just so we can write a recursive definition, which occupies the rest of the Figure (lines 5-12). We implement find pointers using the `findp` attribute. The idea is that `x.findp` will store a dependent record of type `y:I, Pf(Equals @ x @ y)`. That is,

```
Let(x,M,N)     ≡   (x \ q \ q :type(M) -> N) @ M
Ite(M,N,N')    ≡   (Null(type(M)) -> N' |
                     q \ q2 \ q2 :type(M) -> N) @ M
Drop1(M,N)     ≡   Let(ignore, M, N)
```

Fig. 4. RSP abbreviations used in the examples

a record consisting of an individual `y` together with a proof that `x` equals `y`. Such a record is also what `uf.find` returns. We set `uf.find` to be a pattern abstraction taking in an individual `x` matching a pattern which is a single variable `y` (line 5). Since a variable matches anything, this pattern does not constrain the input to the abstraction at all (and syntactic sugar can be introduced to omit it). We first put `x`'s find pointer in temporary variable `fx` using a Let statement (line 6). We then use an Ite statement (if-then-else) to check (line 7) whether or not `x`'s find pointer is `Null` (at the appropriate type). This relies on the fact, mentioned above, that attributes without a stored value default to `Null`. Let and Ite forms (as well as Drop1, used on line 11) are abbreviations, given in Figure 4, where we write `type(M)` for the type in the current context of `M`.

Consider then the first case of the if-then-else statement (lines 8-10). We make a recursive call to `uf.find` on the first component of `fx`, which is the individual pointed to by `x`'s find pointer, and put the result in temporary `ffx` (line 8). Then we set `x`'s find pointer to be a new record, consisting of `ffx.1` (line 9), which by induction is the top element of the chain of find pointers from `fx.1`; and the appropriate transitivity proof (line 10). Note the careful use of y as the third argument to `Eqtrans`. This ensures that the type computed bottom-up for the record (i.e., the one being stored in `x.findp`) is `y:I, d:Pf(Equals @ x @ y)`, as required by the stated return type for `uf.find`. The "else" branch of the Ite expression (lines 11-12) sets `x`'s rank to 0 (for the benefit of `uf.union`), and then returns a record consisting of `x` and a reflexivity proof. Bottom-up type computation for `Eqrefl(x)` will compute the type `Pf(Equals @ x @ x)`. In order for the two branches of the Ite expression to have the same type, an ascription must be used (line 12) so that the reflexivity proof will be viewed as having type `Pf(Equals @ x @ y)`. Since `y` is an alias for `x` at this point, this ascription is legal.

4.2 Imperative Deduction Theorem

The union-find example constructs proofs but never applies a pattern abstraction to a proof to analyze it. In this second example, we consider a meta-program that does analyze proofs using pattern abstractions. For the Hilbert-style formulation of classical logic we have adopted (Figure 2), it is standard to prove the so-called Deduction Theorem by induction on the structure of proofs (cf. [20, Chapter 2]) with case analysis:

```
dedthm : (base =a> A : O =c> B : O =c>
          (Pf(A) => Pf(B)) =c> Pf(IMP @ A @ B)).

dedthm_h : (base =a> bridge : (u:O => Pf(u)) =>
             bundle : {A : O, B : O, d : Pf(B)} =c>
             Pf(IMP @ bundle.A @ bundle.B)).

dedthm_cached : (bundle : {A : O, B : O, d : Pf(B)} =a>
                  Pf(IMP @ bundle.A @ bundle.B)).
```

Fig. 5. Declarations for the Deduction Theorem

```
1.  dt.dedthm := A:O \ null -> B:O \ null ->
2.   D : (Pf(A) => Pf(B)) \ null ->
3.     (bridge : (u:O => Pf(u)) ->
4.       dt.dedthm_h @ bridge @
5.          (q = A, p = B, d = D @ bridge(A) : Pf(p))
6.          : Pf(IMP @ A @ B))
7.     @ Null(u:O => Pf(u))
```

Fig. 6. Deduction Theorem, outer routine

Theorem 1 (Deduction Theorem) *If formula B is derivable possibly using assumption u of formula A, then the formula "A implies B" is derivable without assumption u.*

The inductive proof corresponds exactly to a certain recursive program, where case analysis is implemented by pattern matching. Such a program is a standard example for meta-programming in Twelf [11]. The Twelf program implementing the Deduction Theorem uses HOAS to represent the hypothetical judgment that `B` is derivable from assumption `A` as a representational (i.e., λ-) abstraction. Careful use of higher-order pattern unification enables computation to proceed beneath such abstractions.

We develop here an implementation of the Deduction Theorem in RSP. Since RSP supports imperative programming using attributes, we will actually be able to write an imperative version of this function, which caches intermediate results. Caching intermediate results is a simple but highly effective optimization in automated reasoning systems. In the case of the Deduction Theorem, we will cache intermediate proofs using an attribute `dedthm_cached`. Since the type of the cached proof, `IMP @ A @ B`, depends on both formulas `A` and `B`, we have to store cached results in the `dedthm_cached` attribute of dependent records of type `A:O, B:O, d:Pf(B)`. This explains the declared type for `dedthm_cached` in Figure 5.

Just as in Twelf, it will be necessary to compute under representational abstractions. RSP is able to achieve this using just first-order matching. We borrow a technique of Fegaras and Sheard, developed for implementing cata-

```
1. dt.dedthm_h := bridge : (u:O => Pf(u)) ->
2.  bundle : {A:O, B : O, d:Pf(B)} \ null ->
3.  Ite(bundle.dedthm_cached, bundle.dedthm_cached,
4.  bundle.dedthm_cached := ((A : O ->
5.  (B \ A \ null -> F \ bridge @ B \ null -> IMP_REFL |
6.
7.  B:O \ null ->
8.  (F \ MP @ P @ B @ d1 @ d2
9.     \ (P : O, d1 : Pf(IMP @ P @ B), d2 : Pf(P)) ->
10.    MP @ (IMP @ A @ P) @ (IMP @ A @ B)
11.       @ (MP @ (IMP @ A @ (IMP(P) @ B))
12.             @ (IMP @ (IMP @ A @ P) @ (IMP @ A @ B))
13.             @ (S @ A @ P @ B)
14.             @ (dt.dedthm_h @ bridge @
15.                  (x \ A, y \ (IMP @ P @ B), d1 : Pf(y))))
16.       @ (dt.dedthm_h @ bridge @ (x \ A, y \ P, d2 : Pf(y))) |
17.     D : Pf(B) \ null -> MP @ B @ (IMP @ A @ B) @
18.                           (K @ B @ A) @ D))) @
19.   bundle.A @ bundle.B @ bundle.d)).
```

Fig. 7. Deduction Theorem, main routine

morphisms over datatypes with embedded functions, to program with HOAS in RSP [5]. The function `dt.dedthm` of Figure 6 takes in the function from `Pf(A)` to `Pf(B)` representing the hypothetical judgment. It calls this function on a placeholder term `bridge(A)`, thus replacing (representations of) uses of the assumption `A` in the proof with the placeholder. The helper routine `dt.dedthm_h` of Figure 7 then operates on objects of type `Pf(B)` which may contain occurrences of the placeholder. Enountering the placeholder signals that the assumption is being used, and the appropriate action may be taken. One nice twist here is that unlike in [5], the placeholder does not need to be built in (either to our LF signature or to RSP). We simply introduce it using a representational abstraction (Figure 6, line 3). Since we compute in the bodies of representational abstractions, we will then call the helper (line 4) with the placeholder deployed in the term (line 5). Finally, the placeholder is eliminated after the helper is done computing by applying the representational abstraction to `Null` at the appropriate type (line 7). Bugs in our implementation might lead to occurrences of `Null` appearing in the resulting proof, but this is consistent with our statement of conservativity with respect to LF.

The main routine of the Deduction Theorem (Figure 7) has few surprises. The code begins by checking to see if there is a cached result, and uses it if so (line 3). Otherwise (lines 4-19), it sets the cached result to the appropriate proof, computed by applying cases to the parts of the input bundle. Recursive calls are needed (line 14 and line 16) just when the input proof is an application of `MP`. For typographic reasons the proof `IMP_REFL` for one of the cases (in line

```
MP @ (IMP @ A @ (IMP @ B @ B)) @ (IMP @ A @ B)
   @ (MP @ (IMP @ A @ (IMP(IMP @ B @ B) @ B))
         @ (IMP @ (IMP @ A @ (IMP @ B @ B)) @ (IMP @ A @ B))
         @ (S @ A @ (IMP @ B @ B) @ B)
         @ (K @ A @ (IMP @ B @ B))) @ (K @ A @ B)
```

Fig. 8. IMP_REFL (reflexivity of implication, where A = B)

5) appears in Figure 8. Note that this proof is in terms of A and B, but it is supposed to prove `IMP @ A @ A`. This is indeed what it proves, because at the point in Figure 7 where `IMP_REFL` is used (line 5), it is known that A and B are identical. This is because in line 5, the pattern abstraction matches iff the argument given for `B` matches pattern `A`. Bottom-up type computation for the proof of Figure 8 will, however, compute type `IMP @ A @ B`, which is just what we need to match the return type of `dt.dedthm_h`.

5 Mutable State and HOAS

The above examples combine mutable state and HOAS. Without some restrictions, this would quickly lead to failure of type preservation. For example, consider the term

```
x:I -> (a.b := x)
```

Since this is a representational abstraction, evaluation will occur in the body, causing variable `x` to be stored in attribute `b` of expression `a`. Reading this attribute subsequently returns an open term, which is hardly typable!

Our solution to this problem is based on the following very simple observation. Suppose that instead of the above term we had something like

```
x:I -> (x.b := x)
```

Then there would be no unsoundness, because outside the scope of this binding for `x`, we cannot reference `x`. Hence, we cannot attempt to read its `b` attribute.

We generalize this observation as follows. In an attribute write expression `x.a := y`, if the set of free variables FV(`x`) is a superset of FV(`y`), then we know that it cannot happen that a variable of `y` goes out of scope while `x.a` could still be evaluated. The above examples all are safe in this regard. For instance, take the attribute write in `uf.find` (Figure 3, lines 9-10). Assume by induction that terms `t` cannot evaluate to terms `t'` such that FV(`t'`) $\supsetneq$ FV(`t`). Then no term in the body of `uf.find` can evaluate to a term containing more than the free variables of `x`. Hence, the attribute write is safe. A similar observation applies to the attribute write in `dt.dedthm_h` (Figure 7, line 4). A suitable analysis can enforce this approach; in some cases, it appears some annotations may need to be supplied relating the free variables sets of different arguments to a function. This is the case with the `union` function of union-find, for example, whose code we omit for space reasons.

6 Conclusion and Future Work

This paper has presented work in progress on imperative LF meta-programming in Rogue-Sigma-Pi (RSP). RSP overlays LF with standard programming constructs, including syntactic pattern matching and unrestricted recursion. Imperative programming is supported through dependently typed attributes, which are very convenient for numerous examples, including those of proof-producing union-find and the imperative Deduction Theorem which were considered here. It is well-known that great care is required to combine programming constructs with LF. Imperative features pose special problems, particularly due to the interaction with HOAS. A conservative solution was proposed: we can store values with free variables in attributes as long as we know that the attribute read expressions become inaccessible at least as soon as the values do. The main future work is proving the meta-theoretic properties of type safety and conservativity with respect to LF for a formalization of the system.

References

[1] A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 2002. to appear.

[2] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM Press, 1998.

[3] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages*. ACM, 2003.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[5] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294. ACM Press, 1996.

[6] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A Syntactic Approach to Foundational Proof Carrying-Code. In *IEEE Symposium on Logic in Computer Science*, 2002.

[7] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[8] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[9] Lena Magnusson. *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, 1995.

[10] G. Necula and P. Lee. Proof Generation in the Touchstone Theorem Prover. In David McAllester, editor, *17th International Conference on Automated Deduction*, 2000.

[11] F. Pfenning. *Logical Frameworks*, chapter 21. Volume 2 of Robinson and Voronkov [15], 2001.

[12] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[13] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[14] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[15] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.

[16] J. Sarnat. LF-Sigma: The Metatheory of LF with Sigma types. Technical Report 1268, Yale CS department, 2004.

[17] C. Schürmann. Recursion for higher-order encodings. In *Proceedings of Computer Science Logic*, number 2142 in LNCS, pages 585–599, 2001.

[18] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[19] A. Stump, R. Besand, J. Brodman, J. Hseu, and B. Kinnersley. From Rogue to MicroRogue. In *International Workshop on Rewriting Logic and Applications*, 2004.

[20] A. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2nd edition, 2000.

[21] D. Wu, A. Appel, and A. Stump. Foundational Proof Checkers with Small Witnesses. In D. Miller, editor, *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2003.

A Meta Linear Logical Framework

Andrew McCreight and Carsten Schürmann [1,2]

Yale University
New Haven, CT, USA

Abstract

Logical frameworks serve as meta-languages to represent deductive systems, sometimes requiring special purpose meta logics to reason about the representations. In this work, we describe $\mathcal{L}_\omega^+$, meta logic for the linear logical framework LLF [CP96] and illustrate its use via a proof of the admissibility of cut in the sequent calculus for the tensor fragment of linear logic. $\mathcal{L}_\omega^+$ is first-order, intuitionistic, and not linear. The soundness of $\mathcal{L}_\omega^+$ is shown.

1 Introduction

Logical frameworks are meta languages designed for representing various formal systems prevalent in programming language semantics, logics, and protocol design. By design, a logical framework is foundationally uncommitted, meaning that it is primarily concerned with the way formal systems are represented and not with reasoning about their properties. Logical frameworks have, in this spirit, undergone significant extensions, leaving the design of meta logics far behind. Modern logical frameworks incorporate linear types to model resource awareness (useful when designing programming languages with effects), ordered types (to model formal systems that access resources in a particular order), and even monadic types that capture concurrency.

By separating meta languages from meta logics, we get a quite substantial design space for special purpose meta logics. Each meta logic is tailored toward a particular logical framework, responding to its requirements, expressiveness and idiosyncrasies, with the sole purpose of formalizing meta theoretic arguments about encodings in the logical framework. A logical framework together with a meta logic defines a meta logical framework. One example of a meta logic is $\mathcal{M}_\omega^+$ [Sch00],

[1] This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519 and NFS grant CCR-0325808. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of this agency.

[2] Email:{aem,carsten}@cs.yale.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

designed specifically for the logical framework LF [HHP93]. Conversely, McDowell and Miller [MM97], have chosen a fixed meta logic and to study how to encode and reason about various meta languages in their system. However, their design is also not immune to change. Non-standard extensions of their first-order meta logic with definitions and natural number induction have become necessary to facilitate reasoning about terms with open parameters [MT03].

The absence of well-understood meta logics has often been interpreted as a severe impediment to the deployment and acceptance of the technology among researchers and scientists as well as developers and industry. Consequently, the prevalent use of logical framework technology is as a representation language for one particular logic that is then used to describe and reason about the object systems in question. Higher-order logic is a popular candidate used in Isabelle/HOL [Pau94] and Twelf/HOL [App01] which have been instrumental in the formal study of programming languages, such as Java [NvO98], hardware verification [Har97], and protocol verification [Pau97], among other things. Higher-order logic is well-understood, clean, expressive, and when enriched with induction principles a good choice for many applications. However, it limits the ways in which deductive systems can be encoded, and therefore cannot take advantage of the advanced representation technology provided by modern logical frameworks.

In this work, we propose a special purpose meta logic for the linear logical framework LLF [CP96] which plays the role of a linear meta logical framework. LLF's distinguishing feature over LF is a set of linear operators capable of handling depletable resources. LLF has been successfully employed in representing and experimenting with a variety of security and authentication protocols [CDL+99]. Although the theory behind LLF is well-understood, our work is to our knowledge the first research towards a sound meta logic for LLF.

$\mathcal{L}^+_\omega$ extends the meta logic $\mathcal{M}^+_\omega$ for LF developed by the second author [Sch00] into the LLF setting. $\mathcal{L}^+_\omega$ is first-order, intuitionistic, and not linear. Aside from $\top$, it does not define any logical constant symbols. It does however inherit proofs by induction over arbitrary higher-order types without the restrictive positivity condition, including those that take advantage of both linear and intuitionistic assumptions. Furthermore, it supports quantification over LLF contexts.

The paper is organized in the following way: in Section 2 we review the linear logical framework LLF and illustrate its representational expressiveness in terms of a sequent calculus for the tensor fragment of linear logic. In Section 3, we present a formal meta logic $\mathcal{L}^+_\omega$ that serves as the formalization of theorems as well as meta theoretic proofs. We start by describing the interface between the meta logic and the logic, first by giving extensions to LLF, before describing the meta logic proper and its proof theory. Next, in Section 4, we use as an example the proof of the theorem that cuts are admissible in the previously defined sequent calculus encoding. Then $\mathcal{L}^+_\omega$'s soundness is shown in Section 5, before we conclude in Section 6 and assess results.

$$
\begin{array}{lll}
(Kinds) & K & ::= \mathrm{type} \mid \Pi u{:}A.\ K \\
(Types) & A, B & ::= a \mid A\ M \mid \Pi u{:}A.\ B \mid A \multimap B \mid A\ \&\ B \mid \top \\
(Objects) & M, N & ::= c \mid u \mid \lambda u{:}A.\ M \mid M\ N \mid \hat{\lambda} u{:}A.\ M \mid M \hat{} N \\
 & & \quad \mid \langle M, N \rangle \mid \pi_1 M \mid \pi_2 M \mid \langle\rangle \\
(Signatures) & \Sigma & ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
(Contexts) & \Gamma, \Delta & ::= \cdot \mid \Gamma, u : A \\
(Substitution) & \rho & ::= \cdot \mid \rho, M/u
\end{array}
$$

Fig. 1. LLF syntax

2 The Linear Logical Framework LLF

The linear logical framework LLF [CP96] extends the the logical framework LF [HHP93] with linear resources that may be created, used, or modified. Its feature set supersedes that of LF, supporting dependent types. Every term in LLF reduces to a canonical form. LLF has established itself as an elegant tool for adequate encodings of judgments as types, derivations as objects, and hypothetical judgments as (linear) functions including an elegant treatment of depletable resources.

For example, the well-known derivability judgment for linear classical logic of the form $A_1, \ldots, A_n \Longrightarrow B_1, \ldots, B_n$ can be represented in LLF as a function of the form

$$\mathrm{neg}\ A_1 \ldots \multimap \mathrm{neg}\ A_n \multimap \mathrm{pos}\ B_1 \ldots \multimap \mathrm{pos}\ B_m \to \#.$$

neg and pos are families of types, representing assumptions to the left and right of the $\Longrightarrow$ symbol, respectively, while $\#$ is a type that stands for the empty sequent. Encoding lists of assumptions as linear functions instead of making them explicit as lists has several advantages, namely that lookup, consumption, and substitution are directly supported by LLF through variables names, linear application, and β-reduction, which renders encodings of resource oriented formal systems brief, concise, and readable.

LLF borrows its linear operators from linear logic [Gir87] and uses $\beta\eta$ as the underlying notion of definitional equality [Coq91]. Furthermore, it conservatively extends LF. LLF does not provide a dependent linear function space. The syntax for standard LLF [CP96] is given in Figure 1.

Kinds can either be the kind for types or a dependent product. *Types* can either be a type constant, an application, a dependent function type, the linear function type, the additive product type, or the additive unit. *Objects* can either be an object constant, a variable, an intuitionistic function or application, a linear function or application, a linear additive pair or projection, or the constructor for the additive unit. A *signature* binds type and object constants. An LLF *context* is either empty, or a smaller context extended with an object binding.

We write LLF judgments using $\triangleright$ to separate assumptions from the rest of

$$\frac{}{\Psi;\Gamma;\cdot \triangleright c : \Sigma(c)} \quad \frac{}{\Psi;\Gamma;\cdot \triangleright u : \Gamma(u)} \quad \frac{}{\Psi;\Gamma;u : A \triangleright u : A}$$

$$\frac{\Psi;\Gamma,u : A;\Delta \triangleright M : B}{\Psi;\Gamma;\Delta \triangleright \lambda u{:}A.\ M : \Pi u{:}A.\ B} \quad \frac{\Psi;\Gamma;\Delta \triangleright M : \Pi u{:}A.\ B \quad \Psi;\Gamma;\cdot \triangleright N : A}{\Psi;\Gamma;\Delta \triangleright M\ N : [\mathrm{id}_{\Gamma;\Delta}, N/u]B}$$

$$\frac{\Psi;\Gamma;\Delta,u : A \triangleright M : B}{\Psi;\Gamma;\Delta \triangleright \hat{\lambda} u{:}A.\ M : A \multimap B} \quad \frac{\Psi;\Gamma;\Delta_1 \triangleright M : A \multimap B \quad \Psi;\Gamma;\Delta_2 \triangleright N : A}{\Psi;\Gamma;\Delta_1,\Delta_2 \triangleright M \hat{} N : B}$$

$$\frac{\Psi;\Gamma;\Delta \triangleright M : A \quad \Psi;\Gamma;\Delta \triangleright N : B}{\Psi;\Gamma;\Delta \triangleright \langle M, N\rangle : A \,\&\, B}$$

$$\frac{\Psi;\Gamma;\Delta \triangleright M : A \,\&\, B}{\Psi;\Gamma;\Delta \triangleright \pi_1 M : A} \quad \frac{\Psi;\Gamma;\Delta \triangleright M : A \,\&\, B}{\Psi;\Gamma;\Delta \triangleright \pi_2 M : B} \quad \frac{}{\Psi;\Gamma;\Delta \triangleright \langle\rangle : \top}$$

Fig. 2. Typing rules of LLF.

$$\frac{}{A \Longrightarrow A}\ \mathsf{ax} \qquad \frac{\Gamma_1 \Longrightarrow C, \Delta_1 \quad \Gamma_2, C \Longrightarrow \Delta_2}{\Gamma_1,\Gamma_2 \Longrightarrow \Delta_1,\Delta_2}\ \mathsf{cut}$$

$$\frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \otimes B \Longrightarrow \Delta}\ \otimes\mathsf{L} \qquad \frac{\Gamma_1 \Longrightarrow A, \Delta_1 \quad \Gamma_2 \Longrightarrow B, \Delta_2}{\Gamma_1,\Gamma_2 \Longrightarrow A \otimes B, \Delta_1, \Delta_2}\ \otimes\mathsf{R}$$

Fig. 3. Tensor fragment of linear logic

the judgment. The meta context Ψ, yet unused, holds meta-level assumptions, which we will discuss in Section 3. The form for the object typing judgment is $\Psi;\Gamma;\Delta \triangleright M : A$, which states that under the meta assumptions in Ψ, the intuitionistic assumptions in Γ and the linear assumptions in Δ, the object M has type A. Figure 2 defines the static semantics of LLF. Kinds and types must be linearly closed, and thus the judgments that define their validity ($\Psi;\Gamma \triangleright K : \mathrm{kind}$ and $\Psi;\Gamma \triangleright A : K$), given in Appendix **??**, are declared without a linear context.

Throughout the paper we use simultaneous substitutions ρ that are defined simultaneously on the intuitionistic and the linear variables. Out of notational convenience, we write $\mathrm{id}_{\Gamma;\Delta}$ for the identity substitution on $\Gamma;\Delta$.

As an example, consider the representation of the tensor fragment of classical linear logic depicted in Figure 3. The rules cut and $\otimes$R illustrate how resources on either side of the sequent symbol are distributed as resources to either of the two premisses. A derivation can only then be closed by ax if the left and the right context contain a single formula A. Each inference rule is represented as a constant in LLF as shown in Figure 4. As usual, we omit the leading Π-quantifiers for inferable types. LLF's meta theory guarantees the existence of β-normal, η-long canonical forms [VC02] used in order to establish the adequacy of this encoding.

$$
\begin{array}{ll}
\text{ax} & : \text{neg } A \multimap \text{pos } A \multimap \#. \\
\text{cut} & : (\text{pos } C \multimap \#) \multimap (\text{neg } C \multimap \#) \multimap \#. \\
\text{tensorL} & : (\text{neg } A \multimap \text{neg } B \multimap \#) \multimap (\text{neg } (A \otimes B) \multimap \#). \\
\text{tensorR} & : (\text{pos } A \multimap \#) \multimap (\text{pos } B \multimap \#) \multimap (\text{pos } (A \otimes B) \multimap \#).
\end{array}
$$

Fig. 4. Encoding of Figure 3 in LLF

$$
\begin{array}{lll}
(Objects) & M, N & ::= \ldots \mid n[\rho] \mid \pi_{\mathrm{p}} m \\
(Modules) & m & ::= \alpha \mid \pi_{\mathrm{m}} m \\
(Contexts) & \Gamma, \Delta & ::= \ldots \mid \Gamma, \pi_{\mathrm{p}} m : A \mid \Gamma, \gamma \in \Phi \\
(Substitution) & \rho & ::= \ldots \mid \rho, M/\pi_{\mathrm{p}} m
\end{array}
$$

Fig. 5. LLF extensions

3 The Meta Logic $\mathcal{L}_\omega^+$

The meta logic $\mathcal{L}_\omega^+$ provides the syntactic and proof-theoretic means to express properties about encodings in LLF and their respective proofs, should they exist. Following the general philosophy underlying this and other meta logical frameworks [Sch00,BCM00], the elegance and scalability of our approach emerges from the clear distinction between the language of representation and the language for reasoning. The meta logic $\mathcal{L}_\omega^+$'s noteworthy properties include that it is first-order, i.e. only a universal and an existential quantifier are available, minimal, i.e. no other propositional constants but truth can be defined, and non-linear, i.e. $\mathcal{L}_\omega^+$ is an intuitionistic logic designed to reason about linearity.

We first present extensions to LLF that allow our meta logic $\mathcal{L}_\omega^+$ to express properties about LLF objects in Section 3.1. Next, we describe how the meta level deals with LLF contexts, and how the interface there works. The necessary vocabulary having been built, we then discuss the meta logic proper, starting with its syntax and semantics, then moving to the proof theory. The running example will be continued to illustrate the concepts in question.

3.1 Extensions to LLF

In a meta logic, we wish to reason abstractly about the existence and form of hypothetical LLF objects. LLF must be extended to allow the inclusion of these objects bound at the meta-level. In a closed meta level context, any LLF objects will be standard, as described in the previous section. Figure 5 gives an exact account of these extensions, which are discussed in detail in the following paragraphs. All of LLF's fundamental properties, including conservative extension over LF, type soundness, and the existence of canonical forms remain unchanged under these extensions.

Meta variables

LLF objects may refer to other hypothetical LLF objects whose existence is postulated by the meta logic, usually in form of a universally quantified variable. Those *meta variables*, denoted by n, are bound on the meta level and visible from within LLF terms.

Since meta variables are bound outside of any LLF context, they are given an explicit fixed context (of linear and intuitionistic variables). Consequently, each occurrence of a meta variable n requires an explicit mediating substitution ρ that casts an occurrence of n into the appropriate ambient context. This combination of meta variable and explicit substitution is written as $n[\rho]$. [3]

Context variables

To control the flow of resources inside a meta-theoretic proof, the meta level has to communicate to LLF how many resources are available, how many are to be consumed, and which hypothetical objects are consuming which ones. Context variables γ that are declared as part of LLF contexts in Figure 5 communicate this information and stand for slices of LLF contexts (including the intuitionistic and linear part). Within LLF, context variables are virtually invisible. For example, they can neither be consumed, substituted into, nor can they occur inside LLF objects or types. In fact, the only places where they may occur are in the contexts to other hypothetical objects, characterized by the previously described meta variables. Context variables are declared in the context Ψ that is part of the LLF typing judgment described in Figure 2.

Module variables

Meta variables and context variables form the basic interface between LLF and the meta level. This would be sufficient if we only wanted to reason about closed LLF terms. But the goal of the paper is significantly more ambitious than this, i.e. to reason about all higher-order LLF encodings, including those that may very well be open. The meta theoretic view of openness inevitably impacts the LLF level. For reasons that have not been discussed so far (but will be in the next section), the open parameters are grouped into modules, made visible to LLF in the form of *module projections* (such as $\pi_{\mathrm{p}}(\alpha)$, $\pi_{\mathrm{p}}(\pi_{\mathrm{m}}(\alpha))$, and $\pi_{\mathrm{p}}(\pi_{\mathrm{m}}(\pi_{\mathrm{m}}(\alpha)))$) of module variables α. These projections behave like any other LLF variables, and are thus subject to declaration in an LLF context and to instantiation by a substitution, as described in Figure 5.

3.2 Module contexts and worlds

We have thus far discussed the required extensions to LLF from the point of view of LLF. For the remainder of this section, we switch our point of view to that of the

[3] Our extension of LLF with meta variables is similar to a system developed for a different purpose [PP03], from which we take the syntax for meta variable binders.

$$
\begin{array}{ll}
(Module\ Kinds) & k \ ::= \mathrm{sig} \mid \Pi u\!:\!A.\ k \\
(Module\ Sigs) & s \ ::= \epsilon \mid \exists u\!:\!A.\ s \mid \lambda u\!:\!A.s \\
(Worlds) & \Phi \ ::= s \mid \Phi^* \mid \Phi_1 + \Phi_2 \\
(Module\ Contexts) & \chi \ ::= \cdot \mid \chi_1, \chi_2 \mid \gamma \in \Phi \mid \alpha : s \\
(Meta\ Contexts) & \Psi \ ::= \cdot \mid \Psi, n :: (\chi \triangleright A) \mid \Psi, \gamma \in (\chi \triangleright \Phi) \mid \Psi, \alpha :: (\chi \triangleright s)
\end{array}
$$

Fig. 6. Module context syntax

$$
\begin{array}{ll}
[\![\cdot]\!] & = \cdot \\
[\![\chi, \chi']\!] & = [\![\chi]\!], [\![\chi']\!] \\
[\![\gamma \in \Phi]\!] & = \gamma \in \Phi \\
[\![m : \epsilon]\!] & = \cdot \\
[\![m : \exists u\!:\!A.\ s]\!] & = \pi_{\mathrm{p}} m : A,\ [\![\pi_{\mathrm{m}} m : [\pi_{\mathrm{p}} m/u]s]\!]
\end{array}
$$

Fig. 7. Flattening

$$
\begin{array}{ll}
\lfloor \cdot \rfloor_A^P & = \cdot \\
\lfloor \Gamma, u : B \rfloor_A^P & = \begin{cases} \lfloor \Gamma \rfloor_A^P, u : B \text{ if } P(B, A) \\ \lfloor \Gamma \rfloor_A^P \qquad \text{otherwise} \end{cases} \\
\lfloor \Gamma, \pi_{\mathrm{p}} m : B \rfloor_A^P & = \begin{cases} \lfloor \Gamma \rfloor_A^P, \pi_{\mathrm{p}} m : B \text{ if } P(B, A) \\ \lfloor \Gamma \rfloor_A^P \qquad \text{otherwise} \end{cases} \\
\lfloor \Gamma, \gamma \in \Phi \rfloor_A^P & = \lfloor \Gamma \rfloor_A^P, \gamma \in \lfloor \Phi \rfloor_A^P \\
\lfloor \epsilon \rfloor_A^P & = \epsilon \\
\lfloor \exists u\!:\!B.\ s \rfloor_A^P & = \exists u\!:\!B.\ \lfloor s \rfloor_A^P \text{ if } P(B, A) \\
\lfloor \exists u\!:\!B.\ s \rfloor_A^P & = \lfloor s \rfloor_A^P \text{ if not } P(B, A) \\
\lfloor \lambda u : A.w \rfloor_A^P & = \lambda u : A. \lfloor w \rfloor_A^P \\
\lfloor \Phi^* \rfloor_A^P & = (\lfloor \Phi \rfloor_A^P)^* \\
\lfloor \Phi_1 + \Phi_2 \rfloor_A^P & = \lfloor \Phi_1 \rfloor_A^P + \lfloor \Phi_2 \rfloor_A^P
\end{array}
$$

Fig. 8. Filtering modulo P

meta level. In the full generality of higher-order encodings, inductive arguments often require reasoning under λ-binders, which is tantamount to reasoning about open objects. The argument often calls for more than one hypothesis that seem un-

related at first sight. It is the simultaneous presence of these hypotheses that make a base case go through, or justify the application of a previously proved lemma.

Thus, instead of dealing with individual parameters, the meta level deals with collections of related LF parameters called *modules*. Modules are classified by *module signatures* s. A module is either empty (classified by the signature ϵ) or a pair, where the first element is an LLF parameter and the second element is another module (classified by the signature $\exists u : A.\ s$). The final possible classification, $\lambda u : A.s$, denotes a module of signature s parameterized by an LLF object of type A. Module kinds k are used to keep track of whether a module is fully instantiated (sig) or parameterized ($\Pi u : A.\ k$).

Modules themselves remain abstract, so no concrete module constructors are needed. Instead, a module can consist of a variable (α), or the second element of some other module m ($\pi_{\mathrm{m}} m$), with the m subscript indicating this is the module subcomponent. The typing rules for modules are standard, as they are simply an instance of dot notation [CL90].

The meta logic's view of LLF (intuitionistic and linear) contexts $\Gamma; \Delta$ is called a *module context*, defined in Figure 6 by the syntactic category χ. Informally, the meta level does not distinguish between the intuitionistic and linear contexts, it merely stipulates the existence of particular modules α (of module signature s), or slices γ whose linear part is known to be consumed by a quantified LLF object (expressed as a meta variable).

Module contexts χ must not be thought of as a collection of meta level bindings of γ and α variables, but rather as an abstract description of LLF level bindings. The actual meta level binding takes place in meta contexts Ψ (Figure 6), that we have already used (however not defined) in Figure 2. Meta variables, context variables, and module variables are declared in Ψ, and each declaration is indexed by a module context (denoted by the leading $\chi\ \triangleright$) describing its free variables.

The colorful collection of α's and γ's fully describes a hypothetical pair of valid LLF contexts. The precise relation between the two is discussed in the next subsection. It is important to note, however, that the particular order of declarations in χ is irrelevant and does not reflect the order or declarations within $\Gamma; \Delta$. For example, the module context γ, γ' stands for an arbitrary valid interleaving of two valid contexts $\Gamma; \Delta$ and $\Gamma'; \Delta'$.

The type of a module context is defined by *world* Φ, that, intuitively speaking, describes the shape of a context in the form of a regular expression built from module signatures, repetition and alternation. Worlds have been extensively studied in prior work by the second author [Sch01]. Module contexts may contain only modules valid in Φ. We write $\Psi; \chi' \vdash \chi : \Phi$ for the judgment that decides when (χ', χ) is a valid module context, and χ is in world Φ. For space reasons it is defined in Appendix **??**.

3.3 *Context conversion*

Module contexts χ, while useful at the meta level, cannot directly be used by LLF. For instance, the aggregation of parameters into modules complicates the splitting of contexts required to type linear application ($M \hat{} N$). Additionally, we want to be able to relate the intuitionistic and linear LLF contexts, so we must derive them both from a single χ.

A module context χ is converted to an LLF context Γ in a two step process. First, χ is *flattened* into an LLF context $[\![\chi]\!]$, as defined in Figure 7. This process simply breaks apart each module m in χ into its individual parameters.

Flattening keeps all parameters, which leads to unwanted parameter duplication if used to produce both the Γ and the Δ from a single χ. Furthermore, in the case of the linear context, we must cull extra variables that may occur in χ that simply cannot occur in an LLF object of a certain type.

We solve both of these problems by *filtering*. Filtering, given by $\lfloor\Gamma\rfloor^P_A$, eliminates from an LLF context any variables of type B that do not match the binary predicate $P(B, A)$. It is defined in Figure 8. Similarly, in the case of context variables, we apply filtering to the world annotation $\lfloor\Phi\rfloor^P_A$ and remove all references to module projections that do not match the predicate, creating a narrower view of γ. Our notion of filtering is very general because we permit two seemingly unrelated predicates to transform χ into the intuitionistic and linear context. We require that the resulting $\Gamma; \Delta$ always forms a valid LLF context.

A good choice for each P is one based on the subordination relation [Vir99]. In LLF, all types must be linearly closed. Therefore for the linear context, we use the predicate $A \mathrel{\hat{\prec}} B$, which holds when objects of type A can occur in objects of type B, but not at the type level. For the intuitionistic context, we use the predicate $A \prec\!: B$, which holds if some object of type A can occur in an object of type B, possibly at the type level. This pair of predicates makes as many things as possible linear. If instead the predicate used for the intuitionistic context holds for all pairs of LLF types and the predicate for the linear context holds for none, $\mathcal{L}^+_\omega$ reduces to a meta logic of the logical framework LF [HHP93].

We write the composition of filtering with flattening as $[\![\chi]\!]^P_A$. This composition is used any time we are transitioning from the meta logic level to the logic level.

3.4 *LLF typing rules revisited*

The additional typing rules of our extension to LLF in Figure 9 can now be explained in detail. The two bottom rules for the intuitionistic and linear use of module parameters follow the axiom rule of LLF. The top rule in that figure is the typing rule for meta variable n of type A in context χ declared in Ψ. The judgment $\Psi; \Gamma'; \Delta' \triangleright \rho : \Gamma; \Delta$, defined in Appendix **??**, ensures that the substitution ρ will, when applied to an object well-typed under $\Psi; \Gamma; \Delta$, produce an object well-typed under $\Psi; \Gamma'; \Delta'$. The second premiss of the typing rule for meta variables therefore checks that the substitution associated with the meta variable will correctly map an object substituted for n into the ambient context.

$$\frac{\Psi(n) = (\chi \triangleright A) \quad \Psi; \Gamma; \Delta \triangleright \rho : [\![\chi]\!]^{\prec:}_{A}; [\![\chi]\!]^{\hat{\prec}}_{A}}{\Psi; \Gamma; \Delta \triangleright n[\rho] : [\rho]A}$$

$$\frac{}{\Psi; \Gamma; \cdot \triangleright \pi_{\mathrm{p}} m : \Gamma(\pi_{\mathrm{p}} m)} \qquad \frac{}{\Psi; \Gamma; \pi_{\mathrm{p}} m : A \triangleright \pi_{\mathrm{p}} m : A}$$

Fig. 9. Typing rules of extended LLF.

$$\begin{array}{ll}
(Formulas) & F ::= \forall n :: (\chi \triangleright A).\ F \mid \forall \gamma \in (\chi \triangleright \Phi).\ F \mid \exists n :: (\chi \triangleright A).\ F \mid \mathbb{T} \\
(Programs) & P ::= \Lambda n :: (\chi \triangleright A).\ P \mid \Lambda \gamma \in (\chi \triangleright \Phi).\ P \mid P\ M \mid P\ \chi \\
 & \quad \mid \langle\!\langle \chi \triangleright M; P \rangle\!\rangle \mid \langle\!\langle\rangle\!\rangle \mid x \mid \mathrm{case}\ \Omega \mid \mu\, x \in F.\ P \\
 & \quad \mid \nu\, \alpha :: (\chi \triangleright s).\ P \\
(Cases) & \Omega ::= \cdot \mid \Omega, (\Psi \vdash \sigma \mapsto P) \\
(Meta\ Contexts) & \Psi ::= \ldots \mid \Psi, x \in F \\
(Substitutions) & \sigma ::= \cdot \mid \sigma, M/n \mid \sigma, \chi/\gamma \mid \sigma, P/x \mid \sigma, \alpha/\alpha
\end{array}$$

Fig. 10. $\mathcal{L}^{+}_{\omega}$ syntax

3.5 *Formulas and their semantics*

$\mathcal{L}^{+}_{\omega}$ itself is a first-order meta logic custom designed for LLF. Similar to $\mathcal{M}^{+}_{\omega}$ [Sch00] its syntactic categories consist of formulas, programs, and cases, given in Figure 10.

The universal quantifiers of $\mathcal{L}^{+}_{\omega}$ range over meta-variables n and context variables γ, where χ is the aforementioned module context that describes all free variables of the term in question. There are no quantifiers for module variables α. We do not include existential quantification over module contexts because it does not seem to serve any useful purpose, as opposed to universal quantification, which is required for induction. $\mathbb{T}$ stands for the only propositional constant truth expressible in $\mathcal{L}^{+}_{\omega}$.

The semantic entailment for $\mathcal{L}^{+}_{\omega}$ is written in terms of $\models$, a relation that is defined as follows (in terms of the flattening and filtering operation $[\![\chi]\!]^{P}_{A}$ described in Section 3.3):

$$\begin{array}{lll}
\models \forall \gamma \in (\chi \triangleright \Phi).\ F & \text{iff} & \models [\chi'/\gamma]F \text{ for all } \cdot ; \chi \vdash \chi' : \Phi \\
\models \forall n :: (\chi \triangleright A).\ F & \text{iff} & \models [M/x]F \text{ for all } \cdot ; [\![\chi]\!]^{\prec:}_{A}; [\![\chi]\!]^{\hat{\prec}}_{A} \triangleright M : A \\
\models \exists n :: (\chi \triangleright A).\ F & \text{iff} & \models [M/x]F \text{ for some } \cdot ; [\![\chi]\!]^{\prec:}_{A}; [\![\chi]\!]^{\hat{\prec}}_{A} \triangleright M : A \\
\models \mathbb{T} & &
\end{array}$$

The existential is the dual to the universal quantifier, and true is always valid.

The semantics of $\mathcal{L}_\omega^+$ portrays its intended use as a meta logic to reason about LLF encodings. Any proof within this meta logic should convince a critical observer of the validity of the statement, lemma, or theorem. It is almost certainly possible to give a categorical or model theoretic explanation of proof. We have instead chosen to view proofs as *total programs* via a realizability interpretation. A proof hence acts as a transformation in between LLF encodings. Its input/output behavior is fixed by the formula, its type.

Figure 10 describes the syntactic category for *programs*. $\Lambda n :: (\chi \triangleright A).\ P$ and $\Lambda\gamma \in (\chi \triangleright \Phi).\ P$ are the two binding constructs of $\mathcal{L}_\omega^+$ for LLF objects n and module contexts γ, respectively. Symmetrically, two forms of application $P\ M$ and $P\ \chi$ serve as the respective elimination forms. $\langle\!\langle \chi \triangleright M; P \rangle\!\rangle$ is a proof term for an existential formula, pairing an LLF term with a program. Next, the Figure shows the familiar unit $\langle\!\langle\rangle\!\rangle$ and program variables x and three more constructs that we will explain next: the case construct with cases Ω, the recursion operator μ, and finally the new operator ν.

Case and recursion are necessary to express inductive proofs as programs. The formulation of case ($\mathrm{case}\ \Omega$), the elimination form for LLF objects, looks peculiar, but is in fact quite natural. There is no explicit case subject, because implicitly, case matches against the ambient context in which a "case" may occur. This choice will prove useful in the meta theoretic investigation in Section 5, because dependencies render matching a non-local operation. Each individual case in $\Omega, (\Psi \vdash \sigma \mapsto P)$, consists of a substitution σ that serves as the pattern for that particular case. Each free variable that occurs in a pattern must be declared in Ψ and the body P may not refer to any other variables other than the ones declared in Ψ. The fixed point operator $\mu\ x \in F.\ P$ provides the most general form of the induction hypotheses.

Unbounded recursion and case with an empty Ω illustrate, that without further side condition, $\mathcal{L}_\omega^+$ programs may be partial and hence non-total. The following three side conditions to $\mathrm{case}\ \Omega$ and $\mu\ x \in F.\ P$, respectively, remedy that problem and enforce totality.

Strictness. Each $x \in \Psi$ must have at least one occurrence in the pattern that leads to an unambiguous solution of the higher-order matching algorithm to be used.

Coverage. For all patterns σ within Ω, and or all ambient environments η, there exists a new ambient environment η', such that $[\eta']\sigma = \eta$.

Termination. For all arguments $M_1 \ldots M_n$ to P it holds that for all x that occur in P and arguments $N_1 \ldots N_n$ to x, it holds that $(N_1 \ldots N_n) < (M_1 \ldots M_n)$ with respect to some well-founded order $<$.

Finally, $\nu\ \alpha :: (\chi \triangleright s).\ P$ introduces a new module variable during runtime. Often, for proofs about higher-order encodings the corresponding program has to recurse under an LLF λ binder, be it linear or intuitionistic. Afterwards modules can always be discharged via the mediating substitutions attached to meta variables. There are no other elimination forms for modules.

$$\frac{\Psi;\cdot \vdash \chi : \Phi \quad \Psi;\, [\![\chi]\!]^{\prec:}_{A} \triangleright A : \mathrm{type} \quad \Psi, n :: (\chi \triangleright A) \vdash P \in F}{\Psi \vdash \Lambda n :: (\chi \triangleright A).\ P \in \forall n :: (\chi \triangleright A).\ F}$$

$$\frac{\Psi \vdash P \in \forall n :: (\chi \triangleright A).\ F \quad \Psi;\, [\![\chi]\!]^{\prec:}_{A};\, [\![\chi]\!]^{\hat{\prec}}_{A} \triangleright M : A}{\Psi \vdash P\ M \in [\mathrm{id}_\Psi, M/n]F}$$

$$\frac{\Psi;\cdot \vdash \chi : \Phi \quad \Psi, \gamma \in (\chi \triangleright \Phi) \vdash P \in F}{\Psi \vdash \Lambda \gamma \in (\chi \triangleright \Phi).\ P \in \forall \gamma \in (\chi \triangleright \Phi).\ F} \qquad \frac{\Psi \vdash P \in \forall \gamma \in (\chi \triangleright \Phi).\ F \quad \Psi; \chi \vdash \chi' : \Phi}{\Psi \vdash P\ \chi' \in [\mathrm{id}_\Psi, \chi'/\gamma]F}$$

$$\frac{\Psi;\cdot \vdash \chi : \Phi \quad \Psi \vdash P \in [\mathrm{id}_\Psi, M/n]F \quad \Psi;\, [\![\chi]\!]^{\prec:}_{A};\, [\![\chi]\!]^{\hat{\prec}}_{A} \triangleright M : A}{\Psi \vdash \langle\!\langle \chi \triangleright M; P \rangle\!\rangle \in \exists n :: (\chi \triangleright A).\ F}$$

$$\frac{}{\Psi \vdash \langle\!\langle\rangle\!\rangle \in \top} \qquad \frac{\Psi \vdash \Omega \in F}{\Psi \vdash \mathrm{case}\ \Omega \in F} \qquad \frac{}{\Psi \vdash x \in \Psi(x)} \qquad \frac{\Psi, x \in F \vdash P \in F}{\Psi \vdash \mu\, x \in F.\ P \in F}\ (**)$$

$$\frac{\Psi;\cdot \vdash \chi : \Phi \quad \Psi;\, [\![\chi]\!] \triangleright s : \mathrm{sig} \quad \Psi, \alpha :: (\chi \triangleright s) \vdash P \in F \quad \Psi \vdash F\ \mathrm{ok}}{\Psi \vdash \nu\, \alpha :: (\chi \triangleright s).\ P \in F}$$

$$\frac{}{\Psi \vdash \cdot \in F} \qquad \frac{\Psi \vdash \Omega \in F \quad \Psi' \triangleright \sigma : \Psi \quad \Psi' \vdash P \in [\sigma]F}{\Psi \vdash \Omega, (\Psi' \vdash \sigma \mapsto P) \in F}\ (*)$$

Fig. 11. Derivability in $\mathcal{L}^{+}_{\omega}$

3.7 *Proof theory for* $\mathcal{L}^{+}_{\omega}$

$\mathcal{L}^{+}_{\omega}$'s design is based on the realizability interpretation of total programs as proof. The type system for programs that is described in this section plays the role of a meta logic, whose soundness is shown in Section 5. Our approach to developing the meta logic follows closely [Sch00] and differs significantly from [MT03], who show the soundness of their design by a cut-elimination argument.

$\mathcal{L}^{+}_{\omega}$'s type theory is defined in Figure 11 in terms of two mutually dependent typing judgments: $\Psi \vdash P \in F$ for programs and $\Psi \vdash \Omega \in F$ for cases, using three auxiliary judgments. Two of those judgments $\Psi; \Gamma \rhd s : \mathrm{sig}$ and $\Psi \vdash F : \mathrm{ok}$ ensure the respective validity of module signatures and formulas but do not contribute much to the understanding of the rules. Their definition is given in Appendix **??**. The other judgment $\Psi' \rhd \sigma : \Psi$ ensures the validity of substitutions, that play the role of patterns. Patterns are important for the understanding $\mathcal{L}^{+}_{\omega}$ and given in Figure 12. The first rule exhibits the need for flattening and filtering when instantiating meta variables.

All of the rules in Figure 11 lend themselves to two complementary interpretations. Type-theoretically speaking, the first four rules account for well-typed ab-

$$\frac{\Psi' \triangleright \sigma : \Psi \quad \Psi'; [\![[\sigma]\chi]\!]^{\prec:}_{[\sigma]A}; [\![[\sigma]\chi]\!]^{\hat{\succ}}_{[\sigma]A} \triangleright M : [\sigma]A}{\Psi' \triangleright (\sigma, M/n) : (\Psi, n :: (\chi \triangleright A))}$$

$$\frac{}{\Psi \triangleright \cdot : \cdot} \qquad \frac{\Psi' \triangleright \sigma : \Psi \quad \Psi'; [\sigma]\chi \vdash \chi' : \Phi}{\Psi' \triangleright (\sigma, \chi'/\gamma) : (\Psi, \gamma \in (\chi \triangleright \Phi))}$$

$$\frac{\Psi' \triangleright \sigma : \Psi \quad \Psi' \vdash P \in [\sigma]F}{\Psi' \triangleright (\sigma, P/x) : \Psi, x \in F} \qquad \frac{\Psi' \triangleright \sigma : \Psi \quad \alpha \in \Psi'}{\Psi' \triangleright (\sigma, \alpha/\alpha) : \Psi, \alpha}$$

Fig. 12. Typing rules for patterns

stractions and applications, and logically speaking, they are merely introduction and elimination rules for the universal quantifiers, albeit ones using the flattening and filtering operation $[\![\chi]\!]^P_A$ described in Section 3.3. The fifth rule is the typing rule for pairs, and simultaneously an introduction rule for the existential quantifier. The corresponding elimination rule is subsumed by the case rules defined in below the dotted line [Sch01], and thus need not be considered separately. The typing rule for unit is standard. Ω, the argument to case, is a list of all of the cases (which must all have the same type). The type of a variable x can be inferred from the meta context, and recursion is standard. The rule for ν extends Ψ by a new module constant. The typing rule ensures that α does not escape during evaluation by requiring that the type of the body not contain α.

4 Example

We have considered a few examples from programming language and logic design to exercise and experiment with the meta logic $\mathcal{L}^+_\omega$ for LLF. Our case studies include stateful computations (we managed to represent all meta theoretic proofs about Mini-ML with references in the original LLF paper [CP96]), linear lambda calculi and of linear logic itself. We found that the proof of the admissibility of cut for the tensor fragment of linear logic (see Figure 3), illustrates $\mathcal{L}^+_\omega$'s unique characteristics the best. Of course, there is a certain risk of confusing the reader with two conceptually different yet linear logics.

Theorem 4.1 (Admissibility of cut) *If* $\mathcal{P} :: \Gamma_1 \Longrightarrow C, \Delta_1$ *and* $\mathcal{Q} :: \Gamma_2, C \Longrightarrow \Delta_2$ *then* $\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2$.

Proof. By lexicographic structural induction on the subformula A and simultaneously on $\mathcal{P}$ and $\mathcal{Q}$ [Pfe94]. We show only the essential case between $\otimes$R and $\otimes$L. The remaining cases can be found in Appendix **??**.

$\mathcal{P} :: \Gamma'_1, \Gamma''_1 \Longrightarrow A \otimes B, \Delta'_1, \Delta''_1$ (by assumption)
$\mathcal{P}_1 :: \Gamma'_1 \Longrightarrow A, \Delta'_1$ (by assumption)
$\mathcal{P}_2 :: \Gamma''_1 \Longrightarrow B, \Delta''_1$ (by assumption)

$\mathcal{Q} :: \Gamma_2, A \otimes B \Longrightarrow \Delta_2$ (by assumption)
$\mathcal{Q}_1 :: \Gamma_2, A, B \Longrightarrow \Delta_2$ (by assumption)
$\mathcal{R}_1 :: \Gamma_1', \Gamma_2, B \Longrightarrow \Delta_1', \Delta_2$ (by ind. hyp. on $\mathcal{P}_1, \mathcal{Q}_1$)
$\mathcal{R} :: \Gamma_1', \Gamma_1'', \Gamma_2 \Longrightarrow \Delta_1', \Delta_1'', \Delta_2$ (by ind. hyp. on $\mathcal{P}_2, \mathcal{R}_1$)
□

Theorem 4.1 corresponds to the following formula in $\mathcal{L}_\omega^+$.

(1) $\forall \gamma_1 \in (\cdot \triangleright \Phi).\, \forall \gamma_2 \in (\gamma_1 \triangleright \Phi).\, \forall C : (\cdot \triangleright \mathrm{o}).$
$\forall P : (\gamma_1 \triangleright \mathrm{pos}\ C \to \#).\, \forall Q : (\gamma_2 \triangleright \mathrm{neg}\ C \to \#).$
$\exists R : (\gamma_1, \gamma_2 \triangleright \#).\, \top$

where

(2) $\Phi = ((\lambda A : \mathrm{o}.\, \exists n : \mathrm{neg}\ A.\, \epsilon)$
$+ (\lambda A : \mathrm{o}.\, \exists p : \mathrm{pos}\ A.\, \epsilon))^*.$

The first two quantifiers in (1) range over module contexts γ_1 (valid in the empty context $\cdot$) and γ_2 (valid in γ_1). γ_1 represents the list of hypotheses of both Γ_1 and Δ_1, while γ_2 represents Γ_2 and Δ_2. Φ is the world of these contexts, ensuring that γ_1 and γ_2 only contain assumptions of the form "pos A" and "neg A". For example,

$$p_1 : \mathrm{pos}\ A_1, p_2 : \mathrm{pos}\ A_2, n_3 : \mathrm{neg}\ A_3 \in \Phi.$$

In (1), C ranges over closed formulas, P over sequent derivations in γ_1 with formula C on the left, and Q over sequent derivations in γ_2 with formula C on the right. R stands for the result derivation, necessarily valid in the union of γ_1 and γ_2.

The proof Formula (1), on the other hand is a total program that maps contexts Δ_1, Δ_2, and LLF objects C, P and Q such that $\cdot; \cdot \vdash C : \mathrm{o}$, $\cdot; \Delta_1 \vdash P : \mathrm{pos}\ C \to \#$, and $\cdot; \Delta_2 \vdash Q : \mathrm{neg}\ C \to \#$ into an LLF object R such that $\cdot; \Delta_1, \Delta_2 \vdash R : \#$. In the interest of clarity, the surface language used in Figure 13 that depicts only the essential case of the proof above, making use of a significant amount of syntactic sugar. The remaining cases of "ca" are given in the Appendix **??**.

fun defines the recursive program "ca" by cases. "ca" expects five arguments, including two contexts, all in the form of patterns. $\Gamma_1', \Gamma_1'', \Gamma_2, A, B, P_1, P_2$ and Q_2 occur free in the pattern and in the body of that case. Thus **fun** is a shorthand for a leading μ, followed by several Λ binders and a case expression. For uniformity reasons, we write **new**...**in** ...**end** for $\nu\ \alpha :: (\chi \triangleright s).\ P$. And finally, the **let** ...**in** ...**end** is the standard local binding construct that can be directly expressed using $\mathcal{L}_\omega^+$ programs by combining nested program application with implicit case analysis.

Figure 13 illustrates the novel and distinct features of $\mathcal{L}_\omega^+$ including pattern-matching against linear patterns, hypothetical reasoning, and context splitting. We describe the program in greater detail in the rest of this section, in the context of an analysis of ca's properties regarding strictness, coverage, and termination.

Strictness.

Upon application, matching will always instantiate all free variables in the pattern of "ca". The claim follows directly for Γ_2, A, B, P_1, P_2, and Q_1, which leaves

$$
\begin{array}{l}
\textbf{fun}\ \text{ca}\ (\Gamma_1', \Gamma_1'')\ \Gamma_2\ (A \otimes B) \\
\quad (\hat{\lambda} p\!:\!\text{pos}\ (A \otimes B).\ \text{tensorR}\hat{}\,(\Gamma_1' \triangleright P_1)\hat{}\,(\Gamma_1'' \triangleright P_2)\hat{}\,p) \\
\quad (\hat{\lambda} n\!:\!\text{neg}\ (A \otimes B).\ \text{tensorL}\hat{}\ \\
\qquad (\Gamma_2 \triangleright (\hat{\lambda} n_1\!:\!\text{neg}\ A.\ \hat{\lambda} n_2\!:\!\text{neg}\ B.\ Q_1\hat{}\,n_1\hat{}\,n_2))\hat{}\,n) = \\
\quad \textbf{new}\ \alpha :: (\Gamma_1', \Gamma_2 \triangleright \exists n : \text{neg}\ B.\ \epsilon)\ \textbf{in} \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ \langle R_1, \langle\rangle\rangle = \text{ca}\ \Gamma_1'\ (\Gamma_2, \alpha : \exists n : \text{neg}\ B.\ \epsilon)\ A\ P_1 \\
\qquad\qquad\qquad\qquad (\hat{\lambda} n_1\!:\!\text{neg}\ A.\ Q_1\hat{}\,n_1\hat{}\,\pi_{\text{p}}(\alpha)]) \\
\qquad\quad \textbf{val}\ \langle R, \langle\rangle\rangle = \text{ca}\ \Gamma_1''\ (\Gamma_1', \Gamma_2)\ B\ P_2\ (\hat{\lambda} n\!:\!\text{neg}\ B.\ R_1[n/\pi_{\text{p}}(\alpha)]) \\
\qquad \textbf{in} \\
\qquad\quad \langle R, \langle\rangle\rangle \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Fig. 13. Admissibility of cut, essential case

Γ_1' and Γ_1'' to be explained. In $\mathcal{L}_\omega^+$, every object carries its own context, which means that any instantiation of P_1 and P_2 decides the instantiations for Γ_1' and Γ_1'', respective, rendering matching a deterministic operation.

Coverage.

The first two arguments to "ca" are the context patterns (γ_1', γ_1'') and γ_2. How the context is split into γ_1' and γ_1'' is determined by how the two contexts are used. This is fixed by ascribing context information to the two variables P_1 and P_2 bound in the fourth argument to "ca": $(\hat{\lambda} p\!:\!\text{pos}\ (A \otimes B).\ \text{tensorR}\hat{}\,(\gamma_1' \triangleright P_1)\hat{}\,(\gamma_1'' \triangleright P_2)\hat{}\,p)$. Context and type ascription are features of the syntax we have chosen to present proofs in $\mathcal{L}_\omega^+$ in, with counterparts in the formal development of $\mathcal{L}_\omega^+$ in Section 3.

The challenge is to verify that "ca" covers all cases. Canonical forms are patterns and in the interest of completeness, two additional cases (described in Appendix **??**) related to "tensorR" must be considered, depending on if p is consumed in P_1 or P_2.

Termination.

"ca" must be total in order to be considered a proof. Therefore any evaluation of "ca", independent of what arguments are applied, must terminate. Consider the body of "ca" in Figure 13. The two recursive calls to "ca" correspond to appeals to the induction hypothesis in the proof of Theorem 4.1, yielding result objects R_1 and R, respectively.

The first instruction is the **new** instruction that introduces a new hypotheses of type neg B. Recall from the proof of Theorem 4.1 that $\mathcal{R}_1$ is the result of the induction hypothesis applied to $\mathcal{P}_1$ and $\mathcal{Q}_1$, which is parametric in B. Since hypothetical arguments are encoded via higher-order functions, "ca" can only execute a recursive call after traversing the binder $(\hat{\lambda} n_2 : \text{neg } B)$. In general one can only do this by applying it to a new parameter $n_2 : \text{neg } B$, in form of the module declaration

$$\alpha :: (\gamma_1', \gamma_2 \triangleright \exists n : \text{neg } B.\, \epsilon). \tag{3}$$

α is a new variable, that ranges over groups of new parameters, and is similar to $\underline{x}$ in [Sch01]. Intuitively, one can think of a module as a temporary list of new constant symbols that act as placeholders within the body of **new**. The γ_1', γ_2 resolve all ambiguities related to the naming of α. We write π_{p} to project the head of the list, and π_{m} for the tail. $\pi_{\mathrm{p}}(\alpha)$, for example, is a new name for the newly introduced parameter, and should be used instead of n_2.

The first recursive call cuts P_1 and Q_1 with cut-formula A. Eventually, the computation will finish and the resulting derivation R_1 will use all resources of the set $\gamma_1', \gamma_2, \alpha : \exists n : \text{neg } B.\, \epsilon$, which corresponds directly to the informal proof. Recall that γ_1' represents assumption lists Γ_1' and Δ_1', γ_2 the assumption lists Γ_2 and Δ_2, and α to the additional hypothesis B that occurs to the left of the sequence arrow.

The other recursive call for cutting P_2 and R_1 is similar to the first except that this time the cut formula is B. R_1 is parametric in $\pi_{\mathrm{p}}(\alpha)$, which is subsequently replaced by a linear variable n *before* the second recursive call is invoked. Replacements of this kind are supported in $\mathcal{L}_\omega^+$, expressed by substituting n for $\pi_{\mathrm{p}}(\alpha)$. The resulting R is valid in $\gamma_1', \gamma_1'', \gamma_2$, and does therefore not depend on α. Hence, it can safely escape the scope of **new**.

"ca" terminates because the arguments that correspond to derivations $\mathcal{P}$ and $\mathcal{Q}$ are smaller with respect to a well-founded lexicographical order on the cut formula and simultaneously on $\mathcal{P}$ and $\mathcal{Q}$. In this work, we consider only lexicographic and simultaneous extensions of the subterm ordering. In particular the first recursive call terminates because A and B are subterms of $A \otimes B$.

5 Meta Theory of $\mathcal{L}_\omega^+$

The totality of every program in $\mathcal{L}_\omega^+$ is a sufficient and necessary condition for the soundness of $\mathcal{L}_\omega^+$. The argument relies on a small-step operational semantics given in Appendix **??**. We define a evaluation meta context E to be a meta context Ψ binding only module variables α. For the purposes of the operational semantics, we extend the set of programs with a closure $\{\sigma; P\}$, in which σ is a substitution that maps P from whatever meta context it is well-typed under into the outer meta context. The evaluation judgment $E \vdash P \to P'$ relates a program P to the outcome of a single evaluation step P'. For a sequence of zero or more evaluation steps, we write $E \vdash P \to^* P'$. The set of values is V.

$$V ::= \Lambda n :: (\chi \triangleright A).\, P \mid \Lambda \gamma \in (\chi \triangleright \Phi).\, P \mid \langle\!\langle \chi \triangleright M; V \rangle\!\rangle \mid \langle\!\langle\rangle\!\rangle$$

For functions, applications, existentials and fixed points, evaluation proceeds in the standard fashion. The evaluation of a closure $\{\sigma; P\}$ is essentially carrying out a single step of lazily applying the substitution σ to P. This is done because eager substitution is not sound in the presence of case. Evaluation of $(\mathrm{case}\ \Omega, (\Psi \vdash \sigma' \mapsto P))$ in a closure proceeds by attempting to generate a substitution σ'' that, when composed with σ', is equivalent to the σ of the closure. If one is found, then evaluation of P continues in a closure under σ''. The evaluation of $\nu\, \alpha :: (\chi \triangleright s).\ P$ proceeds by evaluating P until it becomes a value. When it finally becomes a value, the ν binding is pushed into any non-values (as occur in a function) that may exist in the value. In addition to the usual weakening and exchange lemmas, the following properties hold:

Lemma 5.1 (LLF module variable strengthening) *If* $\Psi; \cdot \vdash \chi : \Phi$ *and* $\Psi, \alpha :: (\chi' \triangleright s); [\![\chi]\!]^{\prec:}_A; [\![\chi]\!]^{\succeq}_A \triangleright M : A$ *then* $\Psi; [\![\chi]\!]^{\prec:}_A; [\![\chi]\!]^{\succeq}_A \triangleright M : A$.

Proof. If χ does not contain α, then the LLF context produced by flattening and filtering χ cannot contain any projections from α, and thus neither can M. □ □

Theorem 5.2 (Type preservation) *If* $E \vdash P \in F$ *and* $E \vdash P \longrightarrow P'$ *then* $E \vdash P' \in F$.

Proof. By induction on the structure of the evaluation relation. The cases for ν rely on the fact that the type of the body of the ν must not use the bound module variable, and on Lemma 5.1. This allows the ν to be pushed inward while preserving the type. The substitution cases rely on the soundness of the substitution of σ into χ, A, M and x. □ □

Theorem 5.3 (Progress) *If* $E \vdash P \in F$ *then either* P *is a value or* $E \vdash P \longrightarrow P'$.

Proof. By induction on the structure of the typing derivation. The progress proof uses the fact that E binds only module variables, and on the usual canonical forms lemma. It also relies on the coverage condition holding, which ensures that the program $(\mathrm{case}\ \cdot)$ is never evaluated. □ □

Theorem 5.4 (Termination) *If* $E \vdash P \in F$ *then* $E \vdash P \longrightarrow^* V$.

Proof. By induction on the typing derivation, keeping track of the instantiations of the values bound by reductions of μ, using the termination condition. □ □

Theorem 5.5 (Soundness) *If* $\cdot \vdash P \in F$ *then* $\models F$.

Proof. By induction on F, using Theorems 5.2, 5.3 and 5.4. □ □

6 Conclusion

We have described the meta logic $\mathcal{L}^+_\omega$ for the linear logical framework LLF. LLF is useful for the representation of formal systems that rely on a notion of deletable resource. Surprisingly, many such systems can be represented in LLF, among them programming languages with effects, state transition systems, such as the infamous

blocks world often used in AI, and of course resource oriented logics such as linear logic itself.

The meta logic $\mathcal{L}_\omega^+$ is custom-made for LLF, which means that it incorporates knowledge about linear assumptions, how they are consumed, split in the multiplicative, and duplicated in the additive fragment. It enables the formalization of meta theoretic properties, the mechanization of reasoning about LLF encodings, and leads to relatively short proof terms. The soundness of $\mathcal{L}_\omega^+$ follows from a realizability argument that shows that every function in $\mathcal{L}_\omega^+$ is total, i.e. it terminates and covers all cases.

In future work, we plan to implement a proof checker and an automated theorem prover for $\mathcal{L}_\omega^+$, and consider extensions to the ordered logical framework and the concurrent logical framework.

References

[App01] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Boston, USA, June 2001.

[BCM00] David Basin, Manuel Clavel, and Jos Meseguer. Rewriting logic as a metalogical framework. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 55–80. Springer-Verlag LNCS 1974, 2000.

[CDL+99] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. In *12th Computer Security Foundations Workshop — CSFW-12*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Proc. Programming Concepts and Methods*, pages 479–504. North Holland, 1990.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Har97] John Harrison. Floating point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, June 1997.

[MT03] Dale Miller and Alwen Tiu. A proof theory for generic judgments. In *Proceedings of LICS 2003*, pages 118–127, Ottawa, July 2003.

[NvO98] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, January 1998. ACM Press.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[Pau97] Lawrence C. Paulson. Proving properties of security protocols by induction. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, June 1997.

[Pfe94] Frank Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, November 1994.

[PP03] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *CADE-19*, pages 473–487, Miami Beach, Florida, July 2003.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

[Sch01] Carsten Schürmann. Recursion for higher-order encodings. In Laurent Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.

[VC02] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear lf. *Electronic Notes in Theoretical Computer Science*, 70(2), 2002.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. Forthcoming.