A verified IFOL typechecker — final report*

Marcin Benke and Jacek Chrząszcz and Aleksy Schubert and Maciej Zielenkiewicz
{ben,chrzaszcz,alx,maciekz}@mimuw.edu.pl
Institute of Informatics, University of Warsaw, Poland

December 2015

Abstract

We report here the current status of the verification of the prover and its typechecher. The current implementation is able to typecheck formulas and terms for the whole logic, while the prover can derive proofs for the fragment of logic that uses $\forall$ adn $\rightarrow$. We developed functional specifications ACSL for the typechecker and verified them manually against developed code. Part of the specifications was verified mechanically.

1 Introduction

Part of the requirements for the verified software is that the software with which the software is mechanically verified is itself verified. Therefore, it makes sense to build proving tools that are verified themeselves. One possible option to develop a verified proving backend is to use a language that is executed directly by a processor and for which there is a proving framework available. In this way the trusted code base is reduced to: compiler, verification condition generator and the prover with which the proofs are done.

One solution that is available here is to use C programming language with its growing verification toolset. In particular C has a working verified compiler (that covers a large subset of C) and Frama-C toolkit with which it is possible to generate verification conditions that can subsequently be proved by its proving backend Why3 combined with a selection of SMT-solvers and provers (Alt-Ergo, CVC3, CVC4, and Coq in our case). This toolset is used in our project to develop a trustworthy prover.

In order to further reduce the complexity of the difficult verification task, we develop a prover in which the proving engine itself is not verified. However, the result of the proving backend is additionally checked for correctness by a small typechecker. In this way we open the way for relatively quick development of the proving engine, but without compromise in the trustworthiness of the result. However, this comes at the

*This work has been supported by the Polish NCN grant NCN 2012/07/B/ST6/01532.

price of the speed of the final verification. However, the cost is proportional to the size of the final proof, which is way below the cost of the proof search.

2 Trust Architecture

To get a verified typechecker for intuitionistic first-order logic we used the following path.

1. We decided which particular version of the intuitionistic first-order logic the typechecker will handle. This step involved the choice of the connectives to be used and the general form of the logic. We developed a soundness proof for the logic using Coq to make sure that there is no hidden inconsistency in the system. (See Section 2.1).

2. Next we developed a prototype implementation in a functional programming language Haskell. This step gave the general structure of the whole software artifact and after a number of tests it gave us general confidence in this description of the code. (See Section 2.2).

3. Based upon the Haskell implementation we developed code in C programming language. (See Section 2.3).

4. We developed specifications in ACSL specification language [2] that ensured correctness of the operations in all the basic structures used in the C code. These specifications were mechanically verified against the C source code. (See Section 2.4).

5. Based upon the Haskell implementation we developed the specification in ACSL. This specifications were manually verifed against the code in C. (See Section 2.6).

Below we give more details about all these verification steps.

2.1 The Logic

To gain high expressivity of the system and ensure its high flexibility in the future we decided to use slightly stronger system. Our logic is essentially a variant of λP from [6] extended with constructs for existential quantifiers, alternative, conjunction and falsity. There are no separate constructs for implication, equivalence or negation, as these can be easily encoded. The syntax of the logic is given with the following grammar.

$$
\begin{array}{lcl}
\Gamma & ::= & \{\} \mid \Gamma, (x : \phi) \mid \Gamma, (\alpha : \kappa) \\
\kappa & ::= & * \mid (\Pi x : \phi)\kappa \\
\phi & ::= & \alpha \mid (\forall x : \phi)\phi \mid \phi M \mid (\exists x : \phi)\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bot \\
M & ::= & x \mid (\lambda x : \phi.M) \mid (M_1 M_2) \mid [M_1, M_2]_{\exists x:\phi.\phi} \mid \\
 & & \texttt{abstract}\ \langle x : \phi_1, y : \phi_2 \rangle = M_1\ \texttt{in}\ M_2 \mid \langle M_1, M_2 \rangle_{\phi_1 \wedge \phi_2} \mid \\
 & & \pi_1 M \mid \pi_2 M \mid \texttt{in}_{1,\phi_1 \vee \phi_2} M \mid \texttt{in}_{2,\phi_1 \vee \phi_2} M \mid \\
 & & \texttt{case}\ M_1\ \texttt{in}\ (\texttt{left}\ x : \phi_1.M_2)(\texttt{right}\ y : \phi_2.M_3) \\
 & & \varepsilon_\phi(M)
\end{array}
$$

The inference rules for the logic are presented in Figure 1.

Since our logic is a modification of the established system, it is important to make sure that no conceptual error slipped in the course of modification. To this end we encoded our logic in the Coq proof assistant [7] and proved that it can be embedded in the encoding of the Calculus Constructions from the Coq contribution CoqInCoq [1]. Thanks to that we established logical consistency of our logic.

2.2 Implementation in Haskell

The Haskell implementation of the proofchecker is meant as a reference implementation (an executable specification, as it were). Therefore it is kept as simple as possible, with focus on verifiability and portability rather than efficiency. For example, we made a concious decision to avoid HOAS representation of lambda terms, which while known for its efficiency would be difficult to verify and even more difficult to port to C. For the same reason we refrain from using Haskell-specific idioms. About the only exception to these guidelines is usage of Haskell classes to avoid code repetition.

The implementation conists of two modules:

LambdaP.Core defines the abstract syntax of our language in terms of the following types:

```
type Name = String
data Kind = Kstar
          | Kpi Name Type Kind

data Type = Tvar Name
          | Tall Name Type Type
          | Tapp Type Term
          | Texi Name Type Type
          | Tand Type Type
          | Tor Type Type
          | Tbot

data Term = Mvar Name
          | Mapp Term Term
```

Kind formation rules:

$$\vdash * : \Box \qquad \frac{\Gamma, x : \phi \vdash \kappa : \Box}{\Gamma \vdash (\Pi x : \phi)\kappa : \Box}$$

Kinding rules:

$$\frac{\Gamma \vdash \kappa : \Box}{\Gamma, \alpha{:}\kappa \vdash \alpha : \kappa}\ (tvar)$$

$$\frac{\Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau)\sigma : *}\ (tall) \qquad \frac{\Gamma \vdash \phi : (\Pi x : \tau)\kappa \quad \Gamma \vdash M : \tau}{\Gamma \vdash \phi M : \kappa[x := M]}\ (tapp)$$

$$\frac{\Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\exists x : \tau)\sigma : *}\ (texi) \qquad \vdash \bot : *$$

$$\frac{\Gamma \vdash \phi_1 : * \quad \Gamma \vdash \phi_2 : *}{\Gamma \vdash \phi_1 \wedge \phi_2 : *} \qquad \frac{\Gamma \vdash \phi_1 : * \quad \Gamma \vdash \phi_2 : *}{\Gamma \vdash \phi_1 \vee \phi_2 : *}$$

Typing rules:

$$\frac{\Gamma \vdash \phi : * \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x{:}\phi \vdash x : \phi}\ (var) \qquad \frac{\Gamma \vdash \phi_1 : * \quad y \neq x \quad \Gamma \vdash x : \phi_2}{\Gamma, y{:}\phi_1 \vdash x : \phi_2}\ (varw)$$

$$\frac{\Gamma \vdash M_1 : \phi_1 \quad \Gamma \vdash M_2 : \phi_2}{\Gamma \vdash \langle M_1, M_2 \rangle_{\phi_1 \wedge \phi_2} : \phi_1 \wedge \phi_2}\ (\wedge I) \qquad \frac{\Gamma \vdash M : \phi_1 \wedge \phi_2}{\Gamma \vdash \pi_i(M) : \phi_i}\ (\wedge E)$$

$$\frac{\Gamma \vdash M : \phi_i}{\Gamma \vdash \mathtt{in}_{i,\phi_1 \vee \phi_2} M : \phi_1 \vee \phi_2}\ (\vee I)$$

$$\frac{\Gamma \vdash M_1 : \phi_1 \vee \phi_2 \qquad \Gamma, x : \phi_1 \vdash M_2 : \phi_3 \quad \Gamma, y : \phi_2 \vdash M_3 : \phi_3}{\Gamma \vdash \mathtt{case}\ M_1\ \mathtt{in}\ (\mathtt{left}\ x : \phi_1.M_2)(\mathtt{right}\ y : \phi_2.M_3) : \phi_3}\ (\vee E)$$

$$\frac{\Gamma \vdash M_2[x := M_1] : \phi_2[x := M_1] \quad \Gamma \vdash M_1 : \phi_1}{\Gamma \vdash [M_1, M_2]_{\exists x:\phi_1.\phi_2} : \exists x : \phi_1.\phi_2}\ (\exists I)$$

$$\frac{\Gamma \vdash M_1 : \exists x : \phi_1.\phi_2 \quad \Gamma, x{:}\phi_1, y{:}\phi_2 \vdash M_2 : \phi}{\Gamma \vdash \mathtt{abstract}\ \langle x : \phi_1, y : \phi_2 \rangle = M_1\ \mathtt{in}\ M_2 : \phi}\ (\exists E)^{**}$$

$$\frac{\Gamma, x : \phi_1 \vdash M : \phi_2}{\Gamma \vdash \lambda x : \phi_1.M : \forall x : \phi_1.\phi_2}\ (\forall I)^{\star} \qquad \frac{\Gamma \vdash M_1 : \forall x : \phi_1.\phi_2 \quad \Gamma \vdash M_2 : \phi_1}{\Gamma \vdash M_1 M_2 : \phi_2[x := M_2]}\ (\forall E)$$

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \varepsilon_\phi(M) : \phi}\ (\bot E)$$

---

* $x \notin \mathrm{FV}(\Gamma, \phi_1)$ (eigenvariable condition)

** $x, y \notin \mathrm{FV}(\Gamma, \phi)$ (eigenvariable condition)

Figure 1: The inference rules for the logic

```
  | Mlam Name Type Term
  | Mwit Type Term Term
    -- [m1,m2]_{exists x : phi1.phi2}
  | Mabs Name Type Name Type Term Term
    -- abstract <x:phi1,y:phi2> = m1 in m2
  | Mtup Type Term Term
  | Mpi1 Term
  | Mpi2 Term
  | Min1 Type Term
  | Min2 Type Term
  | Mcas Term (Name,Type,Term) (Name,Type,Term)
  | Meps Type Term -- ex falso
```

LambdaP.GenChecker contains the checker proper

2.2.1 Names and substitution

As is often the case with lambda calculi, the core issue is handling names, substitution and normalization. In our implementation this is handled with help of the class *HasNames*

```
class HasVars a where
  freeNames :: a → [Name]
  freshName :: a → Name

  -- substitution and renaming for types
  substT :: Name → Type → a → a
  renameT :: Name → Name → a → a

  -- substitution and renaming for terms
  substM :: Name → Term → a → a
  renameM :: Name → Name → a → a

  -- replace given variables with fresh ones
  refreshWith :: (Map Name Name) → [Name] → a → a
  refresh :: [Name] → a → a

  whnf :: a → a
  nf :: a → a

  alphaEq :: a → a → Bool
  betaEq :: a → a → Bool
```

Instances of *HasVars* are then provided for the types *Type*, *Kind* and *Term*. We avoid some code repetition by observing that all binders in these types follow the same pattern

$$\mathbf{bind}\ n : t\ \mathbf{in}\ x$$

where t is a type, and provide a generic instance

```
instance HasVars a ⇒ HasVars (Name,Type,a) where
  freeNames (n,t,a) = freeNames t ∪ (freeNames a \\ [n])
  substT n s t@(n1,t1,a)
      | n == n1 = (n1,substT n s t1,a)
      | n1 ∈ freeNames s = (n', substT n s t1, substT n s t')
      | otherwise = (n1, substT n s t1, substT n s a)
        where t' = renameM n1 n' a
              n' = freshName s
  substM n s t@(n1,t1,a)
      | n == n1 = (n1,substM n s t1,a)
      | n1 ∈ freeNames s = (n', substM n s t1, substM n s t')
      | otherwise = (n1, substM n s t1, substM n s a)
        where t' = renameM n1 n' a
              n' = freshName s

  alphaEq (n, t, k) (n', t', k')
    = alphaEq t t'  && alphaEq k (renameM n' n k')

  refreshWith r ns (n,t,k)
      | n ∈ ns = (n', t', refreshWith r' (n':ns) k)
      | otherwise = (n, t', refreshWith r (n:ns) k)
          where
                n' = freshNameFvs ns
                r' = Map.insert n n' r
                t' = refreshWith r ns t
```

given this instance, instances for *Type*, *Kind* and *Term* are then purely routine.

2.2.2 Infrastructure

For convenience, our checker operates within a monad for error reporting

```
type CM a = ErrorT String Identity a
runCM :: CM a → Either String a
reportError :: MonadError String  m ⇒ [String] → m a
```

however the typechecking functions can be used in any instance of **MonadError** (and can be easily adapted to other error handling schemes).

Handling the environment is a little tricky — morally there are two environments: one for type variables, other for object variables. Since their domains must be disjoint, we decided to actually use a single enironment with appropriately labelled entries, along with dedicated functions for handling both kinds of variables.

```
type Env = [(Name,Either Type Kind)]
emptyEnv :: Env

lookupTvar :: MonadError String m ⇒ Env → Name → m Kind
lookupMvar :: MonadError String m ⇒ Env → Name → m Type
```

```
insertTVar :: MonadError String m ⇒ Name → Kind → Env
    → m Env
insertMVar :: MonadError String m ⇒ Name → Type → Env
    → m Env
```

2.2.3 Proofchecking

With the bookkeeping out of the way, the proof-checker proper is essentially a matter of carefully encoding the typing rules. We use a bidirectional approach with separate (but interdependent) type checking and type inference functions, remembering that we work on three levels: kinds, types and terms.

On the kind level, we have basically just some sanity checks:

```
-- | Check if a kind is well-formed wrt an environment
checkKind :: MonadError String m ⇒  Env → Kind → m ()
checkKind env Kstar = return ()
checkKind env (Kpi n t k) = do
  env' ← insertMVar n t env
  checkKind env' k
```

On the type level, we need to

- check whether given expression is of a given kind;
- in particular check whether it is a type (i.e. of kind $*$)
- infer the kind of a given expression

This is done with the following functions

```
checkType :: MonadError String m ⇒ Env → Type → Kind →
    m ()
checkType env t k = do
  k' ← inferType env t
  if betaEq k k'
    then return ()
    else reportError ["Actual kind:",show k',
                      "isn't equal to expected:",show k]

checkIsType env t = checkType env t Kstar

-- | Infer kind of a type expression
-- env |- t : ?
inferType :: MonadError String m ⇒ Env → Type → m Kind
inferType env (Tvar n) = lookupTvar env n
inferType env (Tall "_" t1 t2) = inferType env t2
inferType env (Tall n t1 t2) = do
  env' ← insertMVar n t1 env
```

```
  inferType env' t2
inferType env (Tapp t m) = do
  k ← inferType env t
  case k of
     Kpi x t1 k1 → do
       checkTerm env m t1
       return (substM x m k1)
     Kstar → reportError ["checkType Tapp: expected
        product kind"]
inferType env (Texi n t1 t2) = do
  env' ← insertMVar n t1 env
  inferType env' t2
inferType env (Tand t u) = mapM_ (checkIsType env) [t,u]
   ≫ return Kstar
inferType env (Tor t u) = mapM_ (checkIsType env) [t,u]
   ≫ return Kstar
inferType env Tbot = return Kstar
```

Checking terms follows a similar pattern:

```
checkTerm :: MonadError String m ⇒ Env → Term → Type →
    m ()
inferTerm :: MonadError String m ⇒ Env → Term → m Type
```

2.3 Implementation in C

2.3.1 Overview of files

The implementation of the typechecker in C was divided into two parts:

- intermediary code for parsing and organisation of typing judgements,
- the code of the actual typechecker.

The goal of the code in the first part is to provide interface to the outside world while the actual typechecking is done in the second part. The verification activities were done for the code in the second part while the code in the first part helped in preparation of tests. Since the code in the first part is fairly standard we do not present details of its construction. However, we make here an overview of code in the second part. Here is a list of the files together with explanation of their role in the typechecker.

1. alloc.h — the header file with declaration of functions that handle allocation, dealocation, deep and shallow copying as well as creation of basic elements of types involved in proof representation.
2. alloc.c — the implementation of most of the allocation and deallocation functions mentioned above.

3. alloc.h.m4 — the code in the header file is very systematic so it pays to generate it automatically from a template; in this case the template is written in the .m4 format.

4. alloc.c.m4 — as the header code, the executable code is also very schematic so we decided to implemnt it with a template written in the .m4 format.

5. checker.h — the header file with definitions of structures that represent the tree structures of the terms (`Term`), types (`Type`) and kinds (`Kind`) together with a wrapper type to hold elements of any of the three types (`Any`) used in the typechecker.

6. checker.c — creation procedures for minimal elements of the above mentioned types.

7. copy.c.pl — a file to generate deep copies of data structures and for procedures to make their deep deallocation.

8. envmap_base.h — a header file with declaration of basic access procedures for a dictionary that can hold elements of the type `Any`.

9. envmap_base.c — implementation of the functions that perform basic access procedures for a dictionary that can hold elements of the type `Any`.

10. typechecker.h — a header file for the typechecking procedures of the typechecker.

11. typechecker.c — implementation of the typechecker procedures.

12. free.h — a header file with declaration of the helping procedures to perform deep freeing of the data structures used to represent typechecker proof tree.

13. free.c — a file with definitions of the procedures to perform deep freeing of the data structures used to represent typechecker proof tree.

14. gen_spec_checker.h — a file with definitions of the procedures that perform structural operations of the typechecher such that alpha conversion, substitution for a variable, variable renaming, checking of beta equality and computation of normal form.

15. gen_spec_checker.c — implementation of the procedures that perform structural operations of the typechecher such that alpha conversion, substitution for a variable, variable renaming, checking of beta equality and computation of normal form.

16. namemap.h — a header file for a simple dictionary to hold strings as values.

17. namemap.c — a file with definitions for the procedures that realise a simple dictionary to hold strings as values.

18. printMessages.h — a header file with declarations of the procedures that perform pretty-printing of the data structures that represent derivations in the typechecker.

19. printMessages.c — a file with definitios for the procedures that perform pretty-printing of the data structures that represent derivations in the typechecker.

20. errorMessages.h — a file with declaration of the procedure to perform printing of error messages together with the necessary infrastructure to define it.

21. errorMessages.c — a file with definition of the procedures to perform printing of error messages.

22. utils.h — a header file with some simple utility functions.

23. utils.c — a file with definitions for some simple utility functions.

24. vars_set.h — a header file with declarations for a simple dictionary with the possibility to remove elements.

25. vars_set.c — a file with defintions for functions that realise a simple dictionary with the possibility to remove elements.

Out of these files the files alloc.c, envmap_base.c, namemap.c, printMessages.c, errorMessages.c and vars_set.c were mechanically verified. The verification for the rest of the files was manual.

In addition to these files that were subject to verification, the implementation contains files that itermediate between the logic of typechecking and the external environment. These files provide handling of command line options and parsing of textual form of both formulas and proofs. The files are

- main.c — it contains the procedures that handle command line options and direct control to appropriate functions of the actual typechecker.

- main.h — it contains the headers of the procedures that handle command line options. In principle this file could be omitted as there is no intent to provide this functionality to any other modules of the programme. However, we decided to create the file to provide documentation in a consistent and comprehensive way.

- syntaxAnalyzer.y — the TPTP yacc parser. It contains the structure from tptp-parser project on code.google.com as well as our own code that constructs internal representation of formulas and proofs.

- lexicalAnalyzer.l — the code of lexical analyser.

- parserHelpers.c — the code of functions that help in parsing of the formulas and proofs.

- parserHelpers.h — the header files with the functions that help in parsing of the formulas and proofs.

2.3.2 Overview of code

The implementation of the typechecker in C follows the Haskell implementation. We realize the datatypes as structures. For instance the Haskell definition of the `Kind` datatype is realised in C as the following set of declarations

```
/* data Kind = Kstar
             | Kpi Name Type Kind */
...

struct Kind {
    eKind  which;
    Kstar kStar;
    Kpi   kPi;
};
```

As we can see the elements of the `Kind` type are represented in a C structure of the same name. The first field of the structe contains information on which particular case of the type is represented. This field can assume values from the enumeration type `eKind` that has two values `eKstar` and `eKpi`.

```
typedef enum { eKstar, eKpi } eKind;
```

The meaning of the values is such that when `eKstar` occurs in the `which` field then the field `kStar` of the structure contains meaningful content and can be used to represent the case `Kstar` of the `Kind` type. Similarlny, when `eKpi` occurs in the `which` field then the field `kPi` of the structure contains meaningful content and can be used to represent the case `KPi` of the `Kind` type.

We can see that the fields `kStar` and `kPi` must contain vaues of specific corresponding types `Kstar` and `Kpi`, respectively. The types represent the particular cases of the original datatype. The `Kstar` type is a singleton type. It would be best to represent it as the structure with no fields, but such structures are not covered by Why3 and Frama-C. Therefore we decided to realise it as a structure with a

```
typedef struct Kstar { int nothing; } Kstar;
// something must be inside a structure for WHY3 to digest
```

At last the `Kpi` structure is designed to provide representation for the three argument of the original Haskell `Kpi` constructor.

```
typedef struct Kpi {
    Name name;
    Type* type;
    Kind* kind;
} Kpi;
```

The field `name` represents the `Name` argument, the field **`type`** represents the `Type` argument and at last `kind` field represents the `Kind` argument. Notably all these fields are actually pointer types.

Datatypes for `Type` and `Term` are realised in a similar fashion. We present here only their main structures.

```
/*
data Term = Mvar Name
          | Mapp Term Term
          | Mlam Name Type Term
          | Mwit Type Term Term
          | Mabs Name Type Name Type Term Term
          | Mtup Type Term Term
          | Mpi1 Term
          | Mpi2 Term
          | Min1 Type Term
          | Min2 Type Term
          | Mcas Term (Name,Type,Term) (Name,Type,Term)
*/
struct Term {
        eTerm   which;
        Mvar    mVar;
        Mapp    mApp;
        Mlam    mLam;
        Mwit    mWit;
        Mabs    mAbs;
        Mtup    mTup;
        Mpi1    mPi1;
        Mpi2    mPi2;
        Min1    mIn1;
        Min2    mIn2;
        Mcas    mCas;
        Meps    mEps;
};

...

/*
data Type = Tvar Name
          | Tall Name Type Type
          | Tapp Type Term
          | Texi Name Type Type
          | Tand Type Type
          | Tor Type Type
          | Tbot
*/
struct Type {
    eType  which;
    Tvar   tVar;
    Tall   tAll;
    Tapp   tApp;
    Texi   tExi;
    Tand   tAnd;
    Tor    tOr;
    Tbot   tBot;
};
```

As can be seen from these examples we maintain in the C source code direct visibility of the original Haskell code so that the expressions can be immediately confronted. The same holds for the source code of funcitons. We present here the code for the function `checkKind`

```
//! 383 388
//! -- | Check if a kind is well-formed wrt an environment
//! checkKind :: MonadError String m ⇒  Env → Kind → m ()
//! checkKind env Kstar = return ()
//! checkKind env (Kpi n t k) = do
//!   env' ← insertMVar n t env
//!   checkKind env' k
bool checkKind(const envmap env, Kind k) {
  if (k.which == eKstar)
    return true;

  bool res = true;
  envmap* env_ = envmap_copy(env);
  if (env_ == NULL) return false;
  if (envmap_insertMVar(k.kPi.name, k.kPi.type, env_) == NULL)
    res = false;
  else
    res = checkKind(*env_, *k.kPi.kind);

  envmap_free(*env_);
  free(env_);
  return res;
}
```

As this listing contains the Haskell code we can directly refer to its content. We can see that the C code indeed checks first which case of the `Kind` datatype should be handled. In case the case to handle is `Kstar` then immediately `true` value can be returned. Otherwise we know that the case is `Kpi`. In this case we make a local copy of the environment `env_` not to pollute the external environment with local definitions. Subsequently, we insert with `envmap_insertMVar` an association of the type to the name introduced by the current kind (and there, implicitly, the type in the field `k.kPi .type` is checked for correctness). As we can see `envmap_insertMVar` corresponds directly to `insertMVar` from the Haskell code. In case the operation does not succeed we prepare to return `false`. Otherwise, we recursively check the Kind that remains in `*k.kPi.kind`. Finally, the resources allocated locally are freed and the accumulated result is returned.

2.4 Specifications in ACSL and their verification in Frama-C

2.4.1 Memory Model and Type Casts

The most difficult part of the whole proof development was handling of the memory management. The basic difficulty we encountered is that the most advanced memory model offered by Frama-C was the model *Typed+cast* [3].

This model has a specific representation of the heap. Instead of a single variable on the model side, the heap is modelled by three variables that represent arrays of integers, floats and addresses. Actually the memory of heap is split into three regions of values of specific type. This makes the proving process easier, especially one does not need to prove separation conditions for variables from different classes of these three. However, that convenience comes at the price that not all casts are possible. The *+cast* suffix in the memory model makes it possible to perform some casts and these are necessary to use the system primitives for memory allocation. Since the only kind of memory cast that is done in our typechecker is after the allocation, the generated verification conditions are sound.

Arithmetic model The intended approach to use integers in the typechecker is to use them as mathematical integer numbers (not in the two's complement arithmetic). Therefore, runtime conditions are generated that guard arithmetic operations not to exceed the ranges of machine integers.

2.4.2 Axiomatisation of Allocation

The original specifications of the standard library allocation turned out to be too difficult for provers we use [4]. Therefore, we decided to provide our own specification of memory allocation primitives.

We decided to axiomatise a simplified allocator in which there is an allocation table, modelled as a ghost array (`__allocated`) together with two integer variables that hold the size of the allocated heap (`__size_allocated`) and the number of entries occupied in the allocation table (`__num_allocated`). They are declared in ACSL as follows:

```
//@ ghost char* __allocated[1024];
//@ ghost size_t  __size_allocated = 0;
//@ ghost size_t  __num_allocated = 0;
```

In addition to these ghost variables we defined limits within the variables can operate. In particular, we introduced a limit on the number of entries in the allocation table (`MAX_ALLOCS`) and the limit on the size of the heap (`HEAP_SIZE`). (These are accompanied by definitions of ranges for unsigned and signed integers.).

```
/*@ axiomatic dynamic_allocation_sizes {
  @  logic size_t MAX_ALLOCS;
  @  logic size_t HEAP_SIZE;
  @  logic size_t UINT32_MAX;
  @  logic size_t SINT32_MAX;
  @  axiom uint32_max_def:
  @       UINT32_MAX == 4294967295;
  @  axiom sint32_max_def:
  @       SINT32_MAX == 2147483647;
  @  axiom max_allocs_range:
  @       0 ≤ MAX_ALLOCS ≤ 2147483647;
  @  axiom max_allocs_size:
  @       MAX_ALLOCS == 1024;
```

```
@  axiom heap_size_range:
@       0 ≤ HEAP_SIZE ≤ 2147483647;
@ }
@*/
```

Already proofs with this simplified model turned out to be very difficult so this simplification was productive at this stage of work. Still, this approach to memory allocation gives good approximation of the real allocator that indeed operates with an allocation table and has limited memory at its disposal. Moreover, already this model makes it possible to control if everything that was allocated indeed is allocated and there are no unintended memory leaks.

Predicates to Describe Allocation

We need a number of predicates to describe the allocation process in a concise way. They are all gathered in the `axiomatic` block called `dynamic_allocation` and placed in the `stdlib.h` header file.

Predicate `is_allocable` The first predicate tells that in the particular memory (`L`) the particular (`size`) of memory and number of allocation blocks (`no`) are available.

```
@ predicate is_allocable{L}(size_t size, size_t no) =
@   size > 0 && no > 0 &&
@   \at(__num_allocated, L) + no < MAX_ALLOCS &&
@   \at(__size_allocated, L) + size <= HEAP_SIZE;
```

Predicate `allocation_footprint` This is a predicate that describes which is the difference in allocation state between two memories `L` and `M`. More precisely it tells that the size of the allocated area in `M` is the size of the allocated area in `L` increased by `s`. Additionally the number of allocated slots in `M` is the number of allocated slots in `L` increased by `no`.

```
@ predicate allocation_footprint{L,M}(size_t s, size_t no) =
@   \at(__num_allocated, L) + no == \at(__num_allocated,M) &&
@   \at(__size_allocated, L) + s == \at(__size_allocated,M);
```

Predicate `correctt_allocation` This is a predicate that holds true when the allocation structures have correct layout, i.e. the number of allocated pointers and the size of allocated area do not exceed available sizes. For technical reasons the predicate has an argument, which is ignored.

```
@ predicate correct_allocation{L}(size_t i) =
@     \at(__num_allocated, L) < MAX_ALLOCS &&
@     \at(__size_allocated, L) <= HEAP_SIZE;
```

Predicate `is_allocated_size` This is a predicate that holds true when the given pointer `p` is stored in one of the entries of the allocation table, which means that it was allocated at some point of program execution, and the size of the allocated area starting at the pointer is `i`.

```
@ predicate is_allocated_size{L}(char* p, size_t i) =
@    \at(__num_allocated, L) < MAX_ALLOCS ==>
@      \exists size_t j; (0 <= j < \at(__num_allocated, L) &&
@           \at(__allocated[j], L)==p &&
@           \valid{L}(__allocated[j]+(0..i-1)));
```

Predicate `is_allocated` This is a variant of the previous predicate in which we abstract from the size of the allocated block. It holds true when the given pointer `p` is stored in one of the entries of the allocation table.

```
@ predicate is_allocated{L}(char* p) =
@    \at(__num_allocated, L) < MAX_ALLOCS ==>
@    \exists size_t i; is_allocated_size{L}(p, i);
```

Predicate `old_allocated` This is a predicate describes the operation of a pointer freeing. Two memories `L` and `M` are related here so that their allocation status differs by one freeing operation. This means that all the pointers that are located in the allocation table in `L` on indices before the given address `p` remain at their positions wherease the ones on indices after the address are moved by one downwards.

```
@ predicate old_allocated{L,M}(char* p) =
@   \forall size_t i, j;
@     ((0 <= i < \at(__num_allocated,L) &&
@      (0 <= j < i &&
@      \at(__allocated[i],L)==p) ==>
@         \at(__allocated[j],L) == \at(__allocated[j],M)))
@     &&
@
@     ((0 <= i < \at(__num_allocated,L) &&
@      (i < j < \at(__num_allocated,L) &&
@      \at(__allocated[i],L)==p) ==>
@         \at(__allocated[j],L) == \at(__allocated[j-1],M)));
```

Axioms that Describe Allocation

To facilitate the proving process we need to express certain invariants of the allocator. These are written in the form of axioms that are also located in the `axiomatic` block called `dynamic_allocation` and placed in the `stdlib.h` header file.

Axiom `__allocated_small` This axiom ensures that the number of allocated positions in the memory `L` fits in the range between 0 and the maximal number of available positions, i.e. `MAX_ALLOCS` (excluding the situation that the number of positions is equal to `MAX_ALLOCS`).

```
@
@   axiom __allocated_small{L}:
@         0 <= \at(__num_allocated,L) < MAX_ALLOCS;
@
```

Axiom `__allocated_valid` This axiom provides the link between our model of allocation and validity of memory reads and writes. It tells that every position in a block allocated in our model is valid for reading and writing.

```
@   axiom __allocated_valid{L}:
@    \forall char* keys, size_t s;
@         is_allocated_size{L}(keys,s) ==>
@         \valid{L}(keys+(0..s-1));
```

Axiom `valid__allocated` This axiom tells that all blocks available for reading and writing (valid) have to be located in some of the blocks allocated by our allocator.

```
@ axiom valid__allocated{L}:
@  \forall char* keys, size_t s;
@    \valid{L}(keys+(0..s-1)) ==>
@    \exists char* keys0, size_t s0, i;
@       is_allocated_size{L}(keys0, s0) &&
@       keys0+i==keys && i+s<=s0;
```

Axiom `unique_in__allocated` This axiom tells that the allocation table contains only unique vauels, i.e. no pointer is stored there twice.

```
@   axiom unique_in__allocated{L}:
@    \forall integer i, j; 0 <= i <= j < MAX_ALLOCS &&
@                  __allocated[i] == __allocated[j] ==> i==j;
```

Axiom `sep__allocated` This axiom tells that all the entries in the allocation table are separate from the allocation table itself.

```
@   axiom sep__allocated{L}:
@     \forall size_t i, size_t j;
@         0 <= i < __num_allocated &&
@         \valid((__allocated[i])+(0..j)) ==>
@         \separated(__allocated+(0..MAX_ALLOCS-1),(
    __allocated[i])+(0..j));
```

Axiom `sep__allocated_between` This axiom tells that each entry in the allocation table is separated from all the remaining ones presen there.

```
@   axiom sep__allocated_between{L}:
@     \forall int i, j, size_t k,l;
@         0 <= i <= __num_allocated &&
```

```
@          0 <= j <= __num_allocated &&
@          i != j &&
@          \valid((__allocated[i])+(0..k)) &&
@          \valid((__allocated[j])+(0..l)) ==>
@          \separated((__allocated[i])+(0..k),
@                     __allocated[j]+(0..l));
```

Axiom `allocated_block_length` This axiom tells that the length of each entry in the allocation table is actually equal to the built-in allocated block length function.

```
@  axiom allocated_block_length{L}:
@    \forall char* p, size_t l; is_allocated_size{L}(p,l)
@      <==>
@    \block_length{L}(p) == l;
```

Axiom `allocated_in_heap_size` This axiom tells that the allocated block size is less than available heap size.

```
@  axiom allocated_in_heap_size{L}:
@    \forall char* p, size_t l; is_allocated_size{L}(p,l)
@       ==> l <= HEAP_SIZE;
```

Axiom `non_null_for_allocated` This axiom tells that each block located in the allocated area of the allocation table is not null.

```
@   axiom non_null_for_allocated:
@      \forall size_t i; 0 <= i <= __num_allocated ==>
    __allocated[i] != \null;
```

Lemmas that facilitate proving

The ability to prove correctness of programs offered by the axioms and predicates is extended through a number of lemmas.

- `is_allocable_monotone` tells that the predicate `is_allocable` is monotone in both of its arguments.
- `valid_different_separated` tells that one byte blocks that are available for reading and writing (i.e. are λ`valid`) are separated when they are different.
- `is_allocable_det` tells that the predicate `is_allocable` is deterministic, i.e. that the excluded middle law holds for it.
- `is_allocated_det` tells that the predicate `is_allocated_size` is deterministic, i.e. that the excluded middle law holds for it.
- `later_allocated_different` tells that entries in the allocation table that are allocated later than a given pointer `p` are different than the pointer.

- `allocated_in_separated` tells that entries in the allocation table that are allocated and valid are separated from the allocation table.
- `old_allocated_is_allocated` tells that if two different pointers are allocated and one of them gets freed then the other one is still allocated.
- `is_allocated_size_no_size` tells that the predicate `is_allocated_size` implies `is_allocated`.

Input-output specification of Allocation Functions

These axioms concerning allocation table have to be also connected to the interface of the allocation and deallocation in C. Here is a description of the two functions: `calloc` and `free` that operate on the allocator and are used in our code (we use only these two).

Specifications for `calloc` The interface of the `calloc` function is

```
void *calloc(size_t nmemb, size_t size);
```

We describe now step by step the specified properties of this function. The general structure of the specification is as follows

```
/*@ requires 0 <= nmemb*size <= UINT32_MAX;
  @ behavior allocation:
  @   ...
  @ behavior no_allocation:
  @   ...
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
```

We require here unconditionally (i.e. all calls to `malloc` must fulfil this requirement) that the overall number of allocated bytes (`nmemb*size`) is between 0 and the maximal number representable in unsigned integer type. Then we have two possible behaviours: `allocation`, where the intent is to describe the operation of the procedure in case the allocation operation is possible and `no_allocation`, where the intent is to describe the operation of the procedure in case the allocation operation is not possible. We specify in addition that these two behaviours cover all the possible situations in which the procedure can be called (`complete behaviours`) and that the two behaviours are not overlapping (`disjoint behaviours`).

We describe now the specifications associated with the two distinguished behaviours.

As for the behaviour `allocation` we specify first when this behaviour describes the operation of the `calloc` function. It is specified in the `assumes` clause

```
@  assumes is_allocable{Pre}((size_t)(nmemb*size),(size_t)1);
```

It postulates that this behaviour takes place when the `is_allocable` predicate in the precondition state (`Pre`) holds that tells there is at least `nmemb*size` bytes free in the allocation table and that the allocation table has at least one free entry.

The next item in the specification describes what cells of the memory can possibly be assigned in the course of the procedure

```
@   assigns __allocated[__num_allocated], __size_allocated,
@           __num_allocated;
```

Here we describe that the variables that describe the internal state of our allocator change. In particular the first unallocated so far entry (under `__num_allocated`) is changed and the size of allocated area (`__size_allocated`) as well as the number of allocated entries in the allocation table (`__num_allocated`).

Now, a number of properties that are guaranteed to hold after the call to the `calloc` function. The first one tells that the result of the function is not null

```
@ ensures \result!=\null;
```

The next one tells that the difference in allocation state between the state before the call (`Pre`) and after is (`Post`) is exactly that `nmemb*size` more bytes are marked as allocated on the heap and one more entry in the allocation table is occupied. Note that the projections are necessary here as ACSL has strict typing discipline. Moreover, the projection to `size_t` is guaranteed to be identical due to the main requires clause above.

```
@ ensures allocation_footprint{Pre,Post}((size_t)(nmemb*size),
@             (size_t)1);
```

We have to make sure that the allocated block at the result address of the respective size (`nmemb*size`) is understood as allocated after the call returns. Note that we have to project the λ`result` to **`char`**`*` as the function returns values of the type **`void`**`*`, but our allocation table stores values of the type **`char`**`*`.

```
@ ensures is_allocated_size{Post}((char*)\result, (size_t)(
    nmemb*size));
```

The following property tells that the allocator leaves all the remaining intact, i.e. whatever pointer is allocated before the call, it is different than the result of the function.

```
@ ensures \forall char* p;
@   \result!=\null && (is_allocated{Pre}(p) ==>
@      p != (char*)\result);
```

The previous statement is accompanied by one that states that whatever pointer pointed to an allocated block before the call to the function must point to an allocated block of the same size after the call.

```
@ ensures \forall char* p, size_t n;
@   is_allocated_size{Pre}(p,n) ==>
@     is_allocated_size{Post}(p,n);
```

The statement above is strenghtened by the information that all elements of the allocation table remain in their positions after the call to the allocation procedure.

```
@ ensures \forall size_t i;
@   0 <= i <= \at(__num_allocated,Post)-1 ==>
@        \at(__allocated[i],Post) == \at(__allocated[i],Pre);
```

This, in turn, supplements the information conveyed in the previous fact by the statement that the freshly allocated memory block is placed under the `__num_allocated` position (understood in the pre-state of the method).

```
@ ensures __allocated[__num_allocated] == (char*)\result;
```

The following statement tells that the result is valid for reading and writing for the length of `nmemb*size` bytes.

```
@ ensures \valid((char*)\result+(0..(nmemb*size)-1));
```

The following statement tells that the block pointed out by the result is separate from the allocation table.

```
@ ensures \separated(__allocated+(0..MAX_ALLOCS-1),
@                    (char*)\result+(0..(nmemb*size)-1));
```

Except from the separation from the allocation table we prescribe that the freshly allocated block is separate from all the remaining blocks that were allocated before.

```
@ ensures \forall char** a, size_t s;
@   is_allocated_size{Pre}((char*)a, s) ==>
@       \separated((char*)\result+(0..(size_t)(nmemb*size)-1),
@                  a+(0..s-1));
```

As for the behaviour `no_allocation` we specify first when this behaviour describes the operation of the `calloc` function. It is specified in the `assumes` clause

```
@ assumes !is_allocable{Pre}((size_t)(nmemb*size),(size_t)1);
```

It postulates that this behaviour takes place when the `is_allocable` predicate in the precondition state (`Pre`) does not hold that tells there is at least `nmemb*size` bytes free in the allocation table and that the allocation table has at least one free entry.

The next item in the specification describes that no cells of the memory can possibly will be assigned in the course of the procedure.

```
@ assigns \nothing;
```

As for the properties that are guaranteed to hold after the call to the `calloc` function, the first statement tells that the result of the function is null.

```
@ ensures \result==\null;
```

The next one tells that there is no difference in allocation state between the state before the call (`Pre`) and after is (`Post`).

```
@ ensures allocation_footprint{Pre,Post}((size_t)0,(size_t)0);
```

The previous statement is accompanied by one that states that whatever pointer pointed to an allocated block before the call to the function must point to an allocated block of the same size after the call.

```
@ ensures \forall char* p, size_t n;
@   is_allocated_size{Pre}(p,n) ==>
@      is_allocated_size{Post}(p,n);
```

As in the previous behaviour section, the statement above is strenghtened by the information that all elements of the allocation table remain in their positions after the call to the allocation procedure.

```
@ ensures \forall size_t i;
@     0 <= i <= \at(__num_allocated,Post) ==>
@       \at(__allocated[i],Post) == \at(__allocated[i],Pre);
```

At last we specify that before the call one cannot allocate one block of the size `nmemb *size`. This statement should in principle be provable at the entry to the function, but it is given here explicitely facilitate proving.

```
@ ensures !is_allocable{Pre}((size_t)(nmemb*size),(size_t)1);
```

Specifications for `free` The interface of the `free` function is

void free(void *p);

The specification of its input-output relation looks as follows:

```
/*@
  @   requires  is_allocated{Pre}((char*)p);
  @   assigns  __allocated[__num_allocated], __size_allocated,
  @            __num_allocated;
  @   frees p;
  @   ensures !is_allocated{Post}((char*)p);
  @   ensures old_allocated{Pre,Post}((char *)p);
  @   ensures \forall size_t s;
  @     is_allocated_size{Pre}((char*)p,s) ==>
  @      \at(__size_allocated,Post) ==
  @      \at(__size_allocated,Pre) - s;
  @   ensures allocation_footprint{Post,Pre}(
  @             (size_t)(\block_length{Pre}((char*)p)),
  @             (size_t)1);
  @*/
```

2.5 Verification Conditions in the Code

The verfication process required us to write a sizable number of specifications of the input-output behaviour of the functions that are used in our implementation. We provide an example of such specifications to give the impression on how they work. The whole body of specifications is given in the respective C files.

We present here the input-output relation specification for the function

```
envmap* envmap_copy(const envmap in);
```

The function allocates a new structure to hold keys, values pairs and copies there the arrays that store this information in the argument structure **in**.

The input-output relation specification is as follows.

```
/*@
@ requires default_size_inv(DEFAULT_SIZE) &&
@          valid_envmap(in) &&
@          correct_allocation((size_t)0);
```

```
@ requires 0 <= in.bufsize*sizeof(char*)
@             + in.bufsize*sizeof(Any*)
@             + sizeof(envmap) <= UINT32_MAX;
@ requires \forall size_t i; 0 <= i < in.size ==>
@    \exists integer j; j>0 &&
@                       depth_Any((in.vals)[i])==j;
@
@ behavior alloc_new:
@  assumes is_allocable{Pre}((size_t)(in.bufsize*sizeof(char*)
@                                    +in.bufsize*sizeof(Any*)
@                                    +sizeof(envmap)),
@                              (size_t)3);
@  assigns __allocated[__num_allocated..__num_allocated+1],
@          __size_allocated, __num_allocated;
@  ensures valid_envmap_ptr(\result);
@  ensures allocation_footprint{Pre,Post}(
@             (size_t)(in.bufsize*sizeof(char*)
@                     +in.bufsize*sizeof(Any*)
@                     +sizeof(envmap)),
@             (size_t)3);
@ behavior alloc_no:
@  assumes !is_allocable{Pre}((size_t)(in.bufsize*sizeof(char*)
@                                     +in.bufsize*sizeof(Any*)
@                                     +sizeof(envmap)),
@                               (size_t)3);
@  assumes __num_allocated <= UINT32_MAX - 1;
@  assigns __allocated[__num_allocated];
@  ensures \result == \null;
@  ensures allocation_footprint{Pre,Post}((size_t)0,
@                                         (size_t)0);
@ complete behaviors;
@ disjoint behaviors;
@*/
```

2.6 Functional Specifications

Functional specifications are based upon the code written in Haskell. However, the specifications had to be rewritten by hand. After this process, we checked the specifications against the existing code in C.

3 Prover

The prover is based on Eden automata as presented in [5] and implemented in Haskell. Description of the automaton state consists of automaton definition $\langle A, \leq, Q, q_0, I\rangle$ which depends on the formula and does not change during computation and instantenous description $\langle q, T, k, w\rangle$. The instantenous description is updated by either adding an element to T or removing the last added element from T (i.e. the elements of T form

a stack) and changing of current position (q, w). The automaton description can be implemented efficiently in the following way:

- $\langle A, \leq \rangle$ is represented as a simple tree
- elements of Q are represented as numbers, as well as formulae
- set of instructions I is a map from left-hand state to instruction type and right-hand state.

The instantenous descriptions needs more attention. Instead of keeping separate tree and mapping k from nodes of T to nodes of A we instead label each node of T with pointer to node of A. Moreover in order to have fast access to children of given kind (=node of A), which is used for jumps, instead of keeping typical list of children we keep in each node a map from kind to first child of that kind, and each node has links to left and right siblings of the same kind. Description of a node consists of

- pointer to parent node
- pointer to node of A (kind)
- pointer to left and right siblings of the same kind
- map from kind to first child of that kind.

This allows for efficient traversals and updating of links while adding nodes and removing them if we know which node to delete now. In order to keep track of nodes which are added and other state changes we keep a stack of *frames*. A frame indicate operation on the state which can be reversed while backtracking, and is one of:

- jump — stores previous apple (current node) and map of visited instrucitons in that node
- new — stores information about newly created node.

4 Problems to Be Solved

We discovered the following problems with the existing state-of-the-art of program verification tools and theories.

- The basic problem that occurs in such formalisations is the management of the assumptions. It turns out that most of the statements do not need many assumptions to be proved. However, in order to prove some of them we need to add more statements in the form of asserts. These asserts help in proving some of the facts, but extend the context for all other ones and the proving backends have to deal with them, instantiate them etc. This increases the search space for proof and sometimes statements that were provable cannot be discharged by existing proving backends as the solution is beyond the current search space.

- The current proving backends do not deal well with quantifier instantiation. In many cases instantiations are very obvious to humans and are not very complicated, but the proving backends cannot establish proper ones. This especially concerns cases when some natural strenghtening of a fact to prove must be done, e.g. when the context guarantees that a bigger buffer is valid for reading and writing than what is actually needed.

  This situation could be mitigated by some directives at the source code level that could direct the instantiations to take certain shape. Actually proving backends can handle this kind of hint, but specification languages have rarely support for this kind of hinting.

- The problems above are especially visible when the strenghtening to be done must go through the existential quantifier. The value abstracted by the quantifier is reluctantly used by automatic theorem provers, which makes many proofs difficult to carry out automatically. Similar difficulties can be observed when one of the assumptions has the form of alternative. In case it is crucial to split the proof into two cases depending on the branch of the alternative, automatic theorem provers try to avoid this kind of step.

- In those cases when the proofs had to be done manually in Coq or any other theorem prover the proving process is very tedious due to enormous number of assumptions to deal with. This could be mitigated by appropriate Coq tactics that would pick only those assumptions that are relevant for the current proof.

- Coq proof assistant or any other tool that gives the user the opportunity to do the proof step by step is most often used not as a tool to actually carry out the proof, but as a tool to investigate the specifications and the code. It is often the case that mistakes or missing descritptions in specifications are discovered in an attept to do a proof step by step. However, this process is usually very tedious since the proof assistants have rather weak support for browsing the assumptions or the available lemmas.

- One of the most demanding processes in the course of program verification is the task of discharging separation specifications. This could be made easier by development of dedicated provers that discharge specifically separation conditions.

- Usually a particular property of interest can be expressed in many ways. For instance the fact that an array A is ordered can be written in at least these two ways

  - $\forall xy.x \leq y \implies A[x] \leq A[y]$,
  - $\forall x.A[x] \leq A[x+1]$.

  It turns out that the former statement is much easier to deal with when automated theorem provers are concerned. It would be of great use to have a kind of guide on which formulations are better for automated theorem prooving and which give rise to problems.

References

[1] B. Barras. Coq in coq, 1997. `http://coq.inria.fr/pylons/pylons/contribs/view/CoqInCoq/v8.4`.

[2] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, and Y. M. an d Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009.

[3] P. Baudina, F. Bobot, L. Correnson, and Z. Dargaye. *WP Plug-in Manual*. CEA LIST, Software Safety Laboratory.

[4] M. Benke, J. Chrząszcz, and A. Schubert. A verified ifol typechecker — interim report. Technical report, Institute of Informatics, University of Warsaw,, 2014.

[5] A. Schubert, P. Urzyczyn, and D. Walukiewicz-Chrzaszcz. How hard is positive quantification? To appear.

[6] M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

[7] T. C. D. Team. *The Coq Proof Assistant. Reference Manual*. INRIA, March 2014.