A verified IFOL typechecker — interim report*

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

June 2014

1 The Logic

Our logic is essentially a variant of λP from [5] extended with constructs for existential quantifiers, alternative, conjunction and falsity. There are no separate constructs for implication or negation, as these can be easily encoded.

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

2 Consistency proof

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 [6] 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.

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

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

2.1 CoqInCoq

The following elements of the calculus of constructions and its metatheory are encoded in Coq:

- syntax in a de Bruijn style
- reduction
- confluence and Church-Rosser
- typing rules
- subject reduction
- strong normalisation
- logical consistency

2.1.1 Syntax in a de Bruijn style

The calculus of construction encoded in Coq has three sorts: `kind`, often in the literature denoted by $\Box$, and two sorts `set` and `prop`, corresponding to the sort usually denoted by $\star$. The duplication

```
Inductive sort : Set :=
  | kind : sort
  | prop : sort
  | set : sort.
```

The terms are: a sort, a reference (or variable) with a de Bruijn index, abstraction with a type of argument and a body, an application or a product with a type of argument and a body type.

```
Inductive term : Set :=
  | Srt : sort → term
  | Ref : nat → term
  | Abs : term → term → term
  | App : term → term → term
  | Prod : term → term → term.
```

These definitions are located in the file `CoqInCoq/Termes.v`.

2.1.2 Reduction and conversion

Beta reduction is introduced as the following relation on terms

```
Inductive red1 : term → term → Prop :=
  | beta : forall M N T, red1 (App (Abs T M) N) (subst N M)
[...]
```

completed with 6 cases defining closure by context (two rules for each syntactic construct with subterms, i.e. `Abs`, `App`, `Prod`). The relation `red1` defines a single step of beta reduction, or $\rightarrow_\beta$. The following relation `red`, defines the transitive reflexive closure of beta, or $\rightarrow^*_\beta$.

```
Inductive red M : term → Prop :=
  | refl_red : red M M
  | trans_red :
      forall (P : term) N, red M P → red1 P N → red M N.
```

Finally, the relation `conv` defines the conversion relation, i.e. the transitive, reflexive and symmetric closure of beta, or $\leftrightarrow^*_\beta$.

```
Inductive conv M : term → Prop :=
  | refl_conv : conv M M
  | trans_conv_red :
      forall (P : term) N, conv M P → red1 P N → conv M N
  | trans_conv_exp :
      forall (P : term) N, conv M P → red1 N P → conv M N.
```

These definitions are located in the file `CoqInCoq/Termes.v`.

2.1.3 Confluence and Church-Rosser

Confluence of beta reduction is proved using the usual trick of introducing a *parallel beta* reduction, proving its strong confluence, hence confluence, and then, because transitive closures of beta and parallel beta are the same, this means that beta is also confluent.

The parallel beta reduction is defined in the file `CoqInCoq/Termes.v`:

```
Inductive par_red1 : term → term → Prop :=
  | par_beta :
     forall M M',
     par_red1 M M' →
     forall N N',
     par_red1 N N' →
     forall T, par_red1 (App (Abs T M) N) (subst N' M')
[...]

Definition par_red := clos_trans term par_red1.
```

The skipped rules from the definition of `par_red1` are the rules for closure by context.

In the same file it is proved that `red` and `par_red` coincide:

```
Lemma red_par_red : forall M N, red M N → par_red M N.
Lemma par_red_red : forall M N, par_red M N → red M N.
```

The next step, in the file `CoqInCoq/Conv.v`, consists in proving that the relation `par_red1` is strongly confluent, e.g. commutes with its transposition:

```
Definition str_confluent (R : term → term → Prop) :=
  commut _ R (transp _ R).

Lemma str_confluence_par_red1 : str_confluent par_red1.
```

which can easily be transformed into proofs for the relations in which we are really interested, e.g. `par_red` and finally `red`:

```
Lemma confluence_par_red : str_confluent par_red.
Lemma confluence_red : str_confluent red.
```

In the end, we get the Church-Rosser property for conversion. i.e.:

```
Theorem church_rosser :
  forall u v, conv u v → exists2 t : term, red u t & red v t.
```

2.1.4 Typing rules

The typing rules are presented as two mutually recursive inductive relations:

```
Inductive wf : env → Prop :=
  | wf_nil : wf nil
  | wf_var : forall e T s, typ e T (Srt s) → wf (T :: e)
with typ : env → term → term → Prop :=
  | type_prop : forall e, wf e → typ e (Srt prop) (Srt kind)
  | type_set : forall e, wf e → typ e (Srt set) (Srt kind)
  | type_var :
     forall e, wf e →
     forall (v : nat) t, item_lift t e v → typ e (Ref v) t
  | type_abs :
     forall e T s1,
     typ e T (Srt s1) →
     forall M (U : term) s2,
     typ (T :: e) U (Srt s2) →
     typ (T :: e) M U → typ e (Abs T M) (Prod T U)
  | type_app :
     forall e v (V : term),
     typ e v V →
     forall u (Ur : term),
     typ e u (Prod V Ur) → typ e (App u v) (subst v Ur)
  | type_prod :
     forall e T s1,
     typ e T (Srt s1) →
     forall (U : term) s2,
     typ (T :: e) U (Srt s2) → typ e (Prod T U) (Srt s2)
  | type_conv :
     forall e t (U V : term),
     typ e t U → conv U V →
     forall s, typ e V (Srt s) → typ e t V.
```

The first relation `wf e` stands for *well-formed environment* `e` and the second relation `typ e t T` stands for *in the (well-formed) environment `e`, the term `t` has type `T`*.

The relation `wf` has two rules, for empty environment and for adding a variable in the environment. Note that the type of variable must by typed with a sort in order for this operation to be legal.

The relation `typ` has 7 rules, 2 first ones for typing sorts `prop` and `set`, 5 rules to type terms of different syntax and the last rule, `type_conv`, called conversion, says that a term `t` of type `U`, has also type `V`, if the latter is a type (i.e. can be typed by a sort) and is convertible to `V`.

The definition of typing relations are located in the file `CoqInCoq/Types.v`.

2.1.5 Subject reduction

After a number of lemmas, including substitution lemmas, inversion lemmas, weakening lemmas, etc. one arrives at

```
Theorem subject_reduction :
  forall e t u, red t u → forall T, typ e t T → typ e u T.
```

This is proved in the file `CoqInCoq/Types.v`.

2.1.6 Strong normalisation

Strong normalisation is proved using the method known as *Reducibility Candidates*, where one declares certain family of sets of terms, subsets of the set of all strongly normalising terms, then defines an interpretation of each type, to be roughly speaking, the set off all strongly normalising terms of that type and finally shows that every typable term is strongly normalising.

Since this fragment of development is very large (more than 3000 lines of code), and we will not need the details here, we present only the definition of strong normalisation, based on the Coq's accessibility predicate and the conclusion.

The predicate that defines the strong normalisation property on terms is defined in the file `CoqInCoq/Termes.v`:

```
Definition sn : term → Prop := Acc (transp _ red1).
```

The conclusion is in the file `CoqInCoq/Strong_Norm.v`:

```
Theorem str_norm : forall e t T, typ e t T → sn t.
```

2.1.7 Logical consistency

Having proved strong normalisation, the proof of logical consistency is now possible. One has to examine the potential normal form proofs of false, to show that none of them is possible. Doing so in a formal way requires some work, but nevertheless, one arrives at the conclusion:

```
(* The absurd proposition : False := forall P:Prop, P *)
Definition absurd_prop := Prod (Srt prop) (Ref 0).
[...]
Theorem coc_consistency :
  forall t, ~ typ nil t absurd_prop.
```

This part of development is done in the file `CoqInCoq/Consistency.v`.

2.2 Encoding of the logic in Coq

We follow the way CC is encoded. Our encoding contains:

- syntax in a de Bruijn style
- reduction
- typing rules

2.2.1 Syntax in a de Bruijn style

Unlike the original CC from CoqInCoq, we have only two sorts, `box`, corresponding to $\Box$ and `star`, corresponding to $\star$. We define them in the file `LambdaP/PTermes.v`:

```
Inductive Psort : Set :=
  | box : Psort
  | star : Psort.
```

The inductive cases of the type `Pterm` correspond to productions of our logic grammar, with the only exception, that they are done in a mixed style, i.e. there is no syntactic separation between terms, types and kinds.

```
Inductive Pterm : Set :=
  | PSrt : Psort → Pterm
  | PRef : nat → Pterm
  | PAbs : Pterm → Pterm → Pterm
  | PApp : Pterm → Pterm → Pterm
  | PProd : Pterm → Pterm → Pterm

  | Ex (phi1 phi2 : Pterm) : Pterm
    (* Ex phi1 phi2 = exists x:phi2, phi2 *)
  | ExCons (phi1 phi2 M1 M2: Pterm) : Pterm
    (* Exists M1 M2 : Ex phi1 phi2 *)
  | ExAbs (phi1 phi2 psi M1 M2: Pterm) : Pterm
    (* ExAbs M1 M2 = abstract <x:phi1 y:phi2> = M1 in M2 : psi*)

  | Or (phi1 phi2 : Pterm) : Pterm
    (* Or phi1 phi2 = phi1 ∨ phi2 *)
  | OrIn1 (phi1 phi2 M : Pterm) : Pterm
    (* : Or phi1 phi2 *)
```

```
| OrIn2 (phi1 phi2 M : Pterm) : Pterm
  (* : Or phi1 phi2 *)
| OrCase (phi1 phi2 psi M M1 M2 : Pterm) : Pterm
  (* = case M in ( left x:phi1.M1)(right y:phi2.M2) : psi*)

| And (phi1 phi2 : Pterm) : Pterm
  (* And phi1 phi2 = phi1 ∧ phi2 *)
| AndPair (phi1 phi2 M1 M2 : Pterm) : Pterm
  (* <M1,M2> : phi1∧ phi2 *)
| AndPi1 (phi1 phi2 M : Pterm) : Pterm
| AndPi2 (phi1 phi2 M : Pterm) : Pterm

| Fals : Pterm
| FalsElim (M phi : Pterm) : Pterm.
```

2.2.2 Reduction

The reduction relation is introduced in a bit different way than the reduction of CC. Since we do not have a single form of redex, which must be propagated through context rules, and since our syntax is much larger, we divide the definition of one step reduction into three parts: the definition `Beta`, which defines a redex, the relation combinator `Ctx`, which defined closure by context of an arbitrary relation and the relation `Pred1`, which is the proper one-step beta reduction relation.

```
Inductive Beta : Pterm → Pterm → Prop :=
 | betaL : forall M N T,
    Beta (PApp (PAbs T M) N)
      (Psubst N M)
 | betaEx : forall phi1 phi2 phi1' phi2' psi M1 M2 M,
    Beta (ExAbs phi1 phi2 psi (ExCons phi1' phi2' M1 M2) M)
      (Psubst M2 (Psubst_rec M1 M 1))
 | betaOr1 : forall phi1 phi2 phi1' phi2' psi M M1 M2,
    Beta (OrCase phi1 phi2 psi (OrIn1 phi1' phi2' M) M1 M2)
      (Psubst M M1)
 | betaOr2 : forall phi1 phi2 phi1' phi2' psi M M1 M2,
    Beta (OrCase phi1 phi2 psi (OrIn2 phi1' phi2' M) M1 M2)
      (Psubst M M2)
 | betaAnd1 : forall phi1 phi2 phi1' phi2' M1 M2,
    Beta (AndPi1 phi1 phi2 (AndPair phi1' phi2' M1 M2))
      M1
 | betaAnd2 : forall phi1 phi2 phi1' phi2' M1 M2,
    Beta (AndPi2 phi1 phi2 (AndPair phi1' phi2' M1 M2))
      M2.
```

The 6 cases correspond to: the (usual) beta reduction for application and abstraction, the reduction of `abstract`, applied to $[\_,\_]$, the two reductions of `case`, applied to $\mathtt{in}_1$ and $\mathtt{in}_2$ respectively, and the two reductions concerned with conjunction, i.e., π_1 and π_2, respectively, applied to $\langle \_,\_ \rangle$.

The closure by context is defined as follows:

```
Inductive Ctx (R: Pterm → Pterm → Prop) :
  Pterm → Pterm → Prop
:=
  | base :
     forall M M', R M M' → Ctx R M M'
  | abs_Pred_l :
     forall M M', Ctx R M M' →
      forall N, Ctx R (PAbs M N) (PAbs M' N)
  | abs_Pred_r :
     forall M M', Ctx R M M' →
      forall N, Ctx R (PAbs N M) (PAbs N M')
[...]
```

The `base` case shows inclusion of `R` in `Ctx R`, and the remaining 45 rules corresponding to subterms of all syntactic constructs.

Finally, the one-step beta reduction is defined as the closure by context of `Beta`

```
Definition Pred1 := Ctx Beta.
```

The reduction relation `Pred` is defined as the transitive reflexive closure of `Pred1`

```
Inductive Pred M : Pterm → Prop :=
  | refl_Pred : Pred M M
  | trans_Pred : forall (P : Pterm) N,
     Pred M P → Pred1 P N → Pred M N.
```

Finally the conversion relation is defined as the transitive, symmetric and reflexive closure of `Pred1`:

```
Inductive Pconv M : Pterm → Prop :=
  | refl_Pconv : Pconv M M
  | trans_Pconv_Pred : forall (P : Pterm) N,
     Pconv M P → Pred1 P N → Pconv M N
  | trans_Pconv_exp : forall (P : Pterm) N,
     Pconv M P → Pred1 N P → Pconv M N.
```

All of the above definitions are located in the file `LambdaP/PTermes.v`.

2.2.3 Typing rules

The typing rules for our logic are defined in a similar fashion to the definition of typing for CC, i.e. as a mutual inductive definition of well-formed contexts and correct typing judgements.

```
Inductive Pwf : Penv → Prop :=
[...]
with Ptyp : Penv → Pterm → Pterm → Prop :=
| type_star : forall e, Pwf e →
   Ptyp e (PSrt star) (PSrt box)
```

```
[...]
| type_ex :
   forall e phi1,
   Ptyp e phi1 (PSrt star) →
   forall phi2,
   Ptyp (phi1 :: e) phi2 (PSrt star) →
   Ptyp e (Ex phi1 phi2) (PSrt star)
| type_exCons :
   forall e M1 phi1,
   Ptyp e M1 phi1 →
   Ptyp e phi1 (PSrt star) →
   forall M2 phi2,
   Ptyp (phi1::e) phi2 (PSrt star) →
   Ptyp e M2 (Psubst M1 phi2) →
   Ptyp e (ExCons phi1 phi2 M1 M2) (Ex phi1 phi2)
| type_exAbs :
   forall e M1 phi1 phi2,
   Ptyp e phi1 (PSrt star) →
   Ptyp (phi1 :: e) phi2 (PSrt star) →
   Ptyp e M1 (Ex phi1 phi2) →
   forall M2 psi,
   Ptyp e psi (PSrt star) →
   Ptyp (phi2 :: phi1 :: e) M2 (Plift 2 psi) →
   Ptyp e (ExAbs phi1 phi2 psi M1 M2) psi

| type_or :
   forall e phi1,
   Ptyp e phi1 (PSrt star) →
   forall phi2,
   Ptyp e phi2 (PSrt star) →
   Ptyp e (Or phi1 phi2) (PSrt star)
| type_in1 :
   forall e M phi1,
   Ptyp e M phi1 →
   Ptyp e phi1 (PSrt star) →
   forall phi2,
   Ptyp e phi2 (PSrt star) →
   Ptyp e (OrIn1 phi1 phi2 M) (Or phi1 phi2)
| type_in2 :
   forall e M phi1,
   Ptyp e phi1 (PSrt star) →
   forall phi2,
   Ptyp e M phi2 →
   Ptyp e phi2 (PSrt star) →
   Ptyp e (OrIn2 phi1 phi2 M) (Or phi1 phi2)
| type_orCase :
```

```
   forall e M phi1 phi2,
   Ptyp e phi1 (PSrt star) →
   Ptyp e phi2 (PSrt star) →
   Ptyp e M (Or phi1 phi2) →
   forall M1 M2 psi,
   Ptyp e psi (PSrt star) →
   Ptyp (phi1 :: e) M1 (Plift 1 psi) →
   Ptyp (phi2 :: e) M2 (Plift 1 psi) →
   Ptyp e (OrCase phi1 phi2 psi M M1 M2) psi

| type_and :
   forall e phi1,
   Ptyp e phi1 (PSrt star) →
   forall phi2,
   Ptyp e phi2 (PSrt star) →
   Ptyp e (And phi1 phi2) (PSrt star)
| type_andPair :
   forall e M1 phi1,
   Ptyp e M1 phi1 →
   Ptyp e phi1 (PSrt star) →
   forall M2 phi2,
   Ptyp e M2 phi2 →
   Ptyp e phi2 (PSrt star) →
   Ptyp e (AndPair phi1 phi2 M1 M2) (And phi1 phi2)
| type_andPi1 :
   forall e M phi1 phi2,
   Ptyp e phi1 (PSrt star) →
   Ptyp e phi2 (PSrt star) →
   Ptyp e M (And phi1 phi2) →
   Ptyp e (AndPi1 phi1 phi2 M) phi1
| type_andPi2 :
   forall e M phi1 phi2,
   Ptyp e phi1 (PSrt star) →
   Ptyp e phi2 (PSrt star) →
   Ptyp e M (And phi1 phi2) →
   Ptyp e (AndPi2 phi1 phi2 M) phi2

| type_fals :
   forall e,
   Pwf e → Ptyp e Fals (PSrt star) (* star? *)
| type_falsElim :
   forall e M phi,
   Ptyp e M Fals →
   Ptyp e phi (PSrt star) →
   Ptyp e (FalsElim M phi) phi.
```

The skipped rules are the same as in the definition of typing for CC, i.e. one rule for each syntactic construct of *regular* terms (apart from sorts, which are slightly different in our logic), and one rule for conversion. The remaining rules are grouped around each logical connectives. The first group are rules for the existential quantifier: its formation, introduction, i.e. [_, _], and elimination (`abstract`). The second group concerns the alternative, the third concerns conjunction, and the last one concerns false. Since there is no constructor of type `Fals`, there are only two rules in the last group.

The typing rules are located in file `LambdaP/PTypes.v`.

2.3 Translation to CC and consistency

The translation of our logic to the Calculus of Construction is located entirely in the file `LambdaP/PEmbed.v`.

The translation consists of the following four elements

- translation of the syntax
- correctness of translation wrt reduction
- correctness of translation wrt typing rules
- consistency

2.3.1 Translation of the syntax

Sorts are translated as follows: star is translated to prop, and box to kind.

```
Definition sort_of_Psort (s:Psort) :=
 match s with star ⇒ prop | box ⇒ kind end.
```

The new constructs in terms are translated according to a so called *impredicative encoding*, i.e.

$$\begin{array}{cc} \exists x:\phi.\psi(x) & \forall C.(\forall x:\phi.\psi(x)\to C)\to C \\ \phi\wedge\psi & \forall C.(\phi\to\psi\to C)\to C \\ \phi\vee\psi & \forall C.(\phi\to C)\to(\psi\to C)\to C \\ \bot & \forall C.C \end{array}$$

The introductions and eliminations are translated accordingly.

```
Fixpoint term_of_Pterm (t:Pterm) {struct t} : term :=
 match t with
 | PSrt x ⇒ Srt (sort_of_Psort x)
 | PRef x ⇒ Ref x
 | PAbs x x0 ⇒ Abs (term_of_Pterm x) (term_of_Pterm x0)
 | PApp x x0 ⇒ App (term_of_Pterm x) (term_of_Pterm x0)
 | PProd x x0 ⇒ Prod (term_of_Pterm x) (term_of_Pterm x0)

 | Ex phi1' phi2' ⇒ (* forall (P : Prop), (forall x : phi1, phi2 x → P)→ P *)
   let phi1 := term_of_Pterm phi1' in
```

```
   let phi2 := term_of_Pterm phi2' in
   Prod (Srt prop)
     (Prod (Prod (lift 1 phi1)
             (Prod (Termes.lift_rec 1 phi2 1) (Ref 2)))
       (Ref 1))
| ExCons phi1' phi2' M1' M2' ⇒ (* lambda P f, f M1 M2 *)
   let phi1 := term_of_Pterm phi1' in
   let phi2 := term_of_Pterm phi2' in
   let M1 := term_of_Pterm M1' in
   let M2 := term_of_Pterm M2' in
   Abs (Srt prop)
     (Abs (Prod (lift 1 phi1)
            (Prod (Termes.lift_rec 1 phi2 1) (Ref 2)))
       (App (App (Ref 0) (lift 2 M1)) (lift 2 M2)))
| ExAbs phi1' phi2' psi' M1' M2' ⇒ (* M1 psi M2 *)
   let phi1 := term_of_Pterm phi1' in
   let phi2 := term_of_Pterm phi2' in
   let psi := term_of_Pterm psi' in
   let M1 := term_of_Pterm M1' in
   let M2 := term_of_Pterm M2' in
   App
     (App M1 psi)
      (Abs phi1 (Abs phi2 M2))

| Or phi1' phi2' ⇒ (* forall C, (phi1→ C)→ (phi2→ C)→ C *)
   let phi1 := term_of_Pterm phi1' in
   let phi2 := term_of_Pterm phi2' in
   Prod (Srt prop)
     (Prod (Prod (lift 1 phi1) (Ref 1))
       (Prod (Prod (lift 2 phi2) (Ref 2))
         (Ref 2)))
| OrIn1 phi1' phi2' M' ⇒ (* lambda C f g, f M *)
   let phi1 := term_of_Pterm phi1' in
   let phi2 := term_of_Pterm phi2' in
   let M := term_of_Pterm M' in
   Abs (Srt prop)
     (Abs (Prod (lift 1 phi1) (Ref 1))
       (Abs (Prod (lift 2 phi2) (Ref 2))
         (App (Ref 1) (lift 3 M))))
| OrIn2 phi1' phi2' M' ⇒
   let phi1 := term_of_Pterm phi1' in
   let phi2 := term_of_Pterm phi2' in
   let M := term_of_Pterm M' in
   Abs (Srt prop)
     (Abs (Prod (lift 1 phi1) (Ref 1))
       (Abs (Prod (lift 2 phi2) (Ref 2))
         (App (Ref 0) (lift 3 M))))
| OrCase phi1' phi2' psi' M' M1' M2' ⇒
                      (* M psi (lambda x,M1) (lambda x,M2) *)
   let phi1 := term_of_Pterm phi1' in
```

```
    let phi2 := term_of_Pterm phi2' in
    let psi := term_of_Pterm psi' in
    let M := term_of_Pterm M' in
    let M1 := term_of_Pterm M1' in
    let M2 := term_of_Pterm M2' in
    App
      (App (App M psi)
       (Abs phi1 M1))
      (Abs phi2 M2)

| And phi1' phi2' ⇒ (* forall C, (phi1 → phi2 → C) → C *)
    let phi1 := term_of_Pterm phi1' in
    let phi2 := term_of_Pterm phi2' in
    Prod (Srt prop)
      (Prod (Prod (lift 1 phi1)
              (Prod (lift 2 phi2) (Ref 2)))
        (Ref 1))
| AndPair phi1' phi2' M1' M2' ⇒ (* lambda C f, f M1 M2 *)
    let phi1 := term_of_Pterm phi1' in
    let phi2 := term_of_Pterm phi2' in
    let M1 := term_of_Pterm M1' in
    let M2 := term_of_Pterm M2' in
    Abs (Srt prop)
      (Abs (Prod (lift 1 phi1)
             (Prod (lift 2 phi2) (Ref 2)))
        (App (App (Ref 0) (lift 2 M1)) (lift 2 M2)))
| AndPi1 phi1' phi2' M' ⇒ (* M phi1 (lambda x y,x) *)
    let phi1 := term_of_Pterm phi1' in
    let phi2 := term_of_Pterm phi2' in
    let M := term_of_Pterm M' in
    App
      (App M phi1)
      (Abs phi1 (Abs (lift 1 phi2) (Ref 1)))
| AndPi2 phi1' phi2' M' ⇒ (* M phi2 (lambda x y,y) *)
    let phi1 := term_of_Pterm phi1' in
    let phi2 := term_of_Pterm phi2' in
    let M := term_of_Pterm M' in
    App
      (App M phi2)
      (Abs phi1 (Abs (lift 1 phi2) (Ref 0)))

| Fals ⇒ (* forall A, A *)
    Prod (Srt prop) (Ref 0)
| FalsElim M' phi' ⇒
    let M := term_of_Pterm M' in
    let phi := term_of_Pterm phi' in
    App M phi
end.
```

The translation of environments is of course straightforward. One just maps types from our logic to CC.

```
Definition env_of_Penv (e:Penv) := map term_of_Pterm e.
```

2.3.2 Correctness of translation wrt reduction

In order to prove correctness of the above translation, one must first generate a proper induction scheme, which permits us to do mutual inductive proofs: over derivations of types and environments at the same time.

```
Scheme Ptyp_Pwf_ind := Induction for Ptyp Sort Prop
with Pwf_Ptyp_ind := Induction for Pwf Sort Prop.
```

Then one needs a couple of auxiliary lemmas, for example a substitution lemma, like this one:

```
Lemma subst_of_Psubst : forall M N,
  term_of_Pterm (Psubst N M) =
    subst (term_of_Pterm N) (term_of_Pterm M).
```

Now, in order to prove that the beta reduction relation holds between terms that are a produced in the translation, one needs a couple of tactics, written in Ltac, it order to simulate beta reductions. The `rstep` tactic helps perform one beta reduction of a topmost leftmost redex, the tactic `slsimpl`, *executes* and cleans up substitutions and de Bruijn lift operations. In the end the tactic `simul`, *simulates* a sequence of beta reductions.

```
Ltac rstep := eapply red_trans; [
  solve [ repeat (solve [ eapply Termes.beta ]
                  || eapply app_red_l) ]
| unfold subst; simpl; fold lift_rec ].

Ltac slsimpl :=
  repeat rewrite simpl_subst by auto with arith;
  repeat rewrite simpl_subst_rec by auto with arith;
  repeat rewrite lift0;
  repeat rewrite lift_rec0.

Ltac simul :=
  repeat rstep;
  slsimpl;
  auto with coc.
```

In the end, the proof of the key lemma can be done with a one line tactic.

```
Lemma red_of_Beta : forall M N,
  Beta M N → red (term_of_Pterm M) (term_of_Pterm N).
Proof.
  induction 1; simpl; autorewrite with coc;
```

```
   [eauto with coc | simul..].
Qed.
```

Even simpler are now the following three lemmas, which establish correctness of translation with respect to reduction and conversion. These lemmas, basically use induction and automatic search tactic.

```
Lemma red_of_Pred1 : forall M N,
 Pred1 M N → red (term_of_Pterm M) (term_of_Pterm N).

Lemma red_of_Pred : forall M N,
 Pred M N → red (term_of_Pterm M) (term_of_Pterm N).

Lemma conv_of_Pconv : forall M N,
 Pconv M N → conv (term_of_Pterm M) (term_of_Pterm N).
```

2.3.3 Correctness of translation wrt typing rules

Once we have established the correctness of reduction and conversion we can go on to correctness of typing. We will do a large inductive proof, involving many cases and complex terms. To cope well with these obstacles, we need a number of tactics. The first tactic, aiming mainly in proving goals of the form `typ e t T`, is called `typbum`. It has 23 cases, depending on the goal itself, and the state of the context. One of the cases consists in choosing one of two hypothesis of a given type (and properly instantiating existential variables in other goals) and that is why one needs a parameter `ch`.

```
Ltac typbum ch := match goal with
| [ |- typ _ (Srt _) _ ] ⇒ eapply type_prop
| [ |- typ _ (Srt _) _ ] ⇒ eapply type_set
[...]
| [ H1 : (typ _ _ (Srt _)),
     H2 : (typ _ _ (Srt _))
     |- (typ _ _ (Srt _)) ] ⇒ choose ch H1 H2
[...]
| [ |- item_lift _ _ _ ] ⇒ econstructor
| [ |- (Srt _) = (lift _ _) ] ⇒ eapply lift_sort
| [ |- _ = (lift _ _) ] ⇒ unfold lift; simpl
[...]
end.
```

The next two tactics wrap `typbum` with resolution of some trivial goals, and instantiate it with 1, i.e. at the decision point choose the first hypothesis.

```
Ltac chbum ch :=
 try assumption; typbum ch || eassumption; try reflexivity.

Ltac bum := chbum 1.
```

The next tactic, similar to `slsimpl`, is used to simplify and partially resolve goals concerning lifts and substitutions.

```
Ltac lssimpl :=
  repeat (unfold subst; unfold lift;
          simpl; autorewrite with coc; simpl);
  auto with coc.
```

Now, with this equipment, we can attack the main proof:

```
Lemma typ_of_Ptyp : forall e t T, Ptyp e t T →
  typ (env_of_Penv e) (term_of_Pterm t) (term_of_Pterm T).
Proof.
  intros e t T.
  induction 1 using Ptyp_Pwf_ind with
    (P:=fun e t T H ⇒ typ (env_of_Penv e)
                         (term_of_Pterm t)
                         (term_of_Pterm T))
    (P0:=fun e H ⇒ wf (env_of_Penv e));
  simpl in *; autorewrite with coc in * |- *.
[...]
```

The proof is by structural induction on the judgement and because it has to be done by mutual induction with environment formation rules, one must give the induction predicate by hand.

Fortunately, when tactic development advanced, we managed to reduce all cases to just a few lines. The longest case is the `type_exCons` rules, when one has to provide a number of hints by hand, before going automatic (see below).

```
* (* type_exCons *)
assert (wf (env_of_Penv e)) by eauto with coc.

assert (typ (env_of_Penv e)
         (subst (term_of_Pterm M1) (term_of_Pterm phi2))
         (Srt prop)).
change (Srt prop) with
         (subst (term_of_Pterm M1) (Srt prop)).
eapply substitution; eauto.

repeat (repeat bum; lssimpl).
[...]
```

With the proof of correctness of translation, we may attack the proof of consistency, which now becomes very easy.

2.3.4 Consistency

The consistency theorem consists in showing, that there is no proof of `Fals` in the empty environment. Since our constant `Fals` is translated directly to the term used to

encode falsity in the consistency proof for the calculus of construction, our proof simply translates the hypothetical judgement proving false to CC and uses the consistency result from there.

```
Require Import CoqInCoq.Consistency.

Theorem consistency : forall t, ~ Ptyp nil t Fals.
intro.
intro.
apply typ_of_Ptyp in H.
apply coc_consistency in H.
trivial.
Qed.
```

3 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
        | 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

3.1 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

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

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.

3.2 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
```

3.3 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
```

4 Verification of C code

Important part of the project is to develop a verified version of the typechecker tool. Initial research revealed that a good platform to express properties of the C code is Frama-C [3] code assistance toolset with ACSL specification language [2].

Two small case studies were conducted to check the possibilities of the Frama-C toolset:

1. A simple abstract data structure of sets of strings realised in an array was checked for absence of runtime errors and for its basic functional properties.
2. A simple open source TFTP (Trivial File Transfer Protocol) server was verified for absence of runtime errors.

The goal of the studies was to check, which is the appropriate configuration of the tool to make a reasonable verification task. Several issues were discovered in the process that influence the further process of the typechecker verification.

4.1 Technical issues

There are two plugins of Frama-C that make it possible to do functional verification of programs. One is called *Jessie* and it translates the control flow of C programs into

a control flow of programs in the native language of Why3. The main advantage of this approach is that the intermediate representation retains the structure of the control flow and as a result the process of verification is more comprehensive. Unfortunately, the support for memory management in this plugin is very limited and as a result it is not possible to use it for a realistic project in C programming language as the standard practice in such projects is to use dynamic memory allocation. The plugin is also unable to handle some of the existing specifications for the C standard library.

Therefore, the first project decision was to focus attention on the *WP* plugin of Frama-C, which decomposes a program in C with specifications in ACSL into a set of sentences in a combination of first-order theories that describe the internal state of a program in its crucial places (usually borders of blocks). The Why3 tool is used here largly to manage the access to multiple proving backends, Z3, CVC4, Coq etc.

An important feature of the verification effort is the model in which the verification is performed. There are two kinds of models available models. The memory models describe how the memory is in the *WP* plugin of Frama-C: memory models and arithmetic modelled by the logical formulae while the arithmetic models describe analogously the way numbers are modelled. For the purpose of our case studies we took for the memory model the model "Typed" with possible casts "+casts". This model offers the possibility to deal with casts and at the same time it offers significant flexifility in proving.

One more technical issue that should be mentioned here is the problem with expanding macros that define constants. Many constants (e.g. `INT_MAX`) are defined in C through `#define` directives. These definitions are expanded by the C preprocessor before compilation starts. However, contemporary preprocessors do not expand those constants when they are used in comments. This is precisely scenario that is employed by many specifications in Frama-C. As a result one must use a separate preprocessor that is able to expand macros in comments. We use `gcc` with special arguments

```
-traditional-cpp
```

Still, this solution is not fully functional as only macros in one-line comments are expanded and

```
/*@ assert i <= INT_MAX; */
```

will not work as needed. Sometimes one has to introduce `ghost` variables to make some constant available in many-line specifications:

```
//@ ghost __spec_max_int = INT_MAX;

...

/*@
  @     i == __spec_max_int;
  @*/
```

4.2 Dynamic memory management

In the memory model we decided for not all features are available. In particular the default specification of the `malloc` does not fully work.

```
/*@ allocates \result;
  @ assigns __fc_heap_status \from size, __fc_heap_status;
  @ assigns \result \from size, __fc_heap_status;
  @ behavior allocation:
  @   assumes is_allocable(size);
  @   assigns __fc_heap_status \from size, __fc_heap_status;
  @   assigns \result \from size, __fc_heap_status;
  @   ensures \fresh(\result,size);
  @ behavior no_allocation:
  @   assumes !is_allocable(size);
  @   assigns \result \from \nothing;
  @   allocates \nothing;
  @   ensures \result==\null;
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
void *malloc(size_t size);
```

In particular the predicate `fresh` and clause `allocates` are not fully operational. Therefore, we decided to change the specification to explicitely use the separation predicate. For this we introduced explicitely the array of allocated pointers `__allocated` together with its size `__size_allocated`

```
/*@ ghost extern char* __allocated[]; */
/*@ ghost extern int  __size_allocated; */
```

This is supplemented with a set of axioms that describe the invariants associated with the array

```
  @ axiom sep__allocated{L}:
  @   \forall int i; 0 <= i < __size_allocated ==>
  @       \separated(__allocated,__allocated[i]);
  @ axiom sep__allocated_between{L}:
  @   \forall int i, j; 0 <= i < j < __size_allocated ==>
  @       \separated(__allocated[i],__allocated[j]);
```

The first one says that all elements of the array `__allocated` do not overlap with the array itself. This is a natural axiom, which guarantees that our additional data structure is independent of the allocation mechanism. The second one says that different elements of the array do not overlap one with the other. This is the basic functional property we require from allocator: it allocates only fresh memory.

In addition we introduced two free predicates

```
  @ predicate is_allocable{L}(size_t n);
  @ predicate is_allocable_after{L}(size_t n, size_t m);
```

that express if the allocator is able to allocate new memory. The second predicate, `is_allocable_after`, is necessary when we want to express that a function allocates two times.

Now, we can express the requirements for the memory allocation.

```
/*@
  @ terminates \true;
  @ assigns \result,__allocated[0..__size_allocated];
  @ behavior allocation:
  @   assumes is_allocable{Pre}(size);
  @   assigns \result,__allocated[0..__size_allocated];
  @   ensures \result!=\null;
  @   ensures \valid((char*)\result+(0..size-1));
  @   ensures \block_length{Post}((char*)\result)==size;
  @   ensures \exists int i; __allocated[i] == \result &&
  @     \forall int j; i != j && 0 <= j < __size_allocated ==>
  @        \separated((char*)__allocated[j],(char*)\result);
  @   ensures \separated(__allocated,(char*)\result);
  @   ensures \forall size_t ssize;
  @        is_allocable_after{Pre}(size, ssize) ==>
  @        is_allocable{Post}(ssize);
  @ behavior no_allocation:
  @   assumes !is_allocable{Pre}(size);
  @   assigns \result;
  @   ensures \result==\null;
  @   ensures !is_allocable{Post}(size);
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
void *malloc(size_t size);
```

The specification for the `malloc` function has two cases: either required block can be allocated (`allocation`) or cannot (`no_allocation`). Each of the cases assumes appropriate status of the predicate `is_allocable`. In case the allocation is possible we can guarantee, respectively, that

- the resulting pointer is not null,
- the resulting pointer points to a valid buffer of the size given in the argument to `malloc`,
- the array `__allocated` contains a position that is allocated,
- each element of the array `__allocated` contains a pointer that is separated from the one freshly allocated,
- the array `__allocated` is separated from the resulting buffer,
- if a block of `ssize` size was asserted to be allocable through the predicate `is_allocable_after` before the call then the block is indeed allocable after the call.

In case the allocation is not possible we can guarantee that

- the result is null and
- the block in question remains impossible to allocate.

The requirements for freeing the block are as follows.

```
/*@ ensures \block_length{Post}((char*)p)==0;
  @ assigns  __allocated[0..];
  @ behavior deallocation:
  @   assumes  \exists size_t i;__allocated[i]==p;
  @   requires freeable: \exists size_t i;
  @              \valid((char*)p+(0..i));
  @   assigns  __allocated[0..];
  @   ensures \block_length{Post}((char*)p)==0;
  @   ensures \forall int j; 0 <= j < __size_allocated ==>
  @      (char*)__allocated[j]!=(char*)p;
  @ behavior no_deallocation:
  @   assumes  p==\null || \forall size_t i;__allocated[i]!=p;
  @   assigns  \nothing;
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
```

First of all we guarantee unconditionally that the block that is freed is no longer allocated. We again have two cases: when deallocation is possible and when deallocation is impossible.

The first (allocation) case is only possible when the given block is valid and is recorded as allocated in the `__allocated` array. As a result we guarantee that the block in the argument is no longer allocated and has no position in the `__allocated` array.

The second (non-allocation) case is possible when the argument is either null or it does not occur in the `__allocated` array. In that case nothing happens.

4.3 Handling of local arrays

In case a local array is defined in the frame of a function it is not possible to assert that its memory is separated from the arguments of the array as well as global buffers. This assumption is not automatically provided by the system, but it is necessary in some proofs. This can be remedied (for non-recursive functions) by definition of the local array as a global one. In that case the necessary separation condition can be expressed as a precondition to the function in question. For instance a local array `opts` of length 12 can be made global and then separation from the global array `optarg` that `getopt` function operates on can be expressed as

```
  @ requires \separated(opts+(0..12),optarg+(0..));
```

4.4 Assumptions for the `main` function

The function `main` is handled in a special, but not fully sound, way by the *WP* plugin. Consider the following standard form of the function

```
int main(int argc, char *argv[]) { /* ... */ }
```

The standard C99 [4] prescribes that the parameter `argc` should be nonnegative. However, this occurs nowhere in the assumptions available in the function. In particular for the code

```
int main(int argc, char** argv) {
  //@ assert argc >= 0;
}
```

the assert cannot be established to hold. Of course, some programs will depend on the correctness of the arguments to the `main` function. We established [4] that at least the following preconditions should be assumed by it:

```
/*@ requires argc >= 0;
  @ requires argv[argc] == 0;
  @ requires argc > 0 ==> \valid(argv+(0..argc+1));
  @ requires \forall int i; 0 <= i < argc ==>
  @    valid_string(argv[i]);
  @*/
```

That means respectively that

- the parameter `argc` is nonnegative,
- the array `argv` is terminated with 0,
- if `argc` is nonzero `argv` is allocated so that it can hold `argc+1` positions (including the terminating 0),
- all elements of the array `argv` are pointers to valid strings.

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. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. of SEFM'12*, volume 7504 of *LNCS*. Springer, 2012.

[4] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

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

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