Internalising modified realisability in constructive type theory

Erik Palmgren
Department of Mathematics, Uppsala University

November 3, 2004

Abstract

A modified realisability interpretation of infinitary logic is formalised and proved sound in constructive type theory (CTT). The logic considered subsumes first order logic. The interpretation makes it possible to extract programs with simplified types and to incorporate and reason about them in CTT.

1 Modified realisability

Modified realisability interpretation is a well-known method for giving constructive interpretation of some intuitionistic logical system into a simple type structure (Troelstra 1973). The method is used, for instance, in Minlog and Coq for extracting programs from proofs (cf. Schwichtenberg 2004 and Letouzey 2004). These programs are to a large extent free from the computationally irrelevant parts that might be present in programs arising from direct interpretations into constructive type theory. The realisability interpretation requires a separate proof of correctness, which is usually left unformalised.

In this note we present a completely formalised modified realisability interpretation carried out in the proof support system Agda (Coquand 2000). We shall here use what is called *modified realisability with truth* which has the property that anything realised is also true in the system (Theorem 1.2). One difference from usual interpretations as in Minlog is that the logic interpreted goes beyond first order logic: it is a (constructively) infinitary logic, which arises naturally from the type-theoretic notion of universe. Our extension to infinitary logic seems to be a novel result.

Agda is based on Martin-Löf constructive type theory (1972) with an infinite hierarchy of universes #0 = Set, #1 = Type, #2 = Kind, #3, ... Each of these

universes is closed under the formation of generalised inductive data types. We define in Agda an inductive type SP of propositions, so called *simple propositions*, by induction: for each small type A (i.e. a member of Set) an atomic proposition $\mathsf{atom}(A) : \mathsf{SP}$ is introduced; SP contains $\bot$ and is closed under propositional connectives ($\wedge$, $\vee$, $\rightarrow$) and for any small type A and any propositional function $P : A \rightarrow \mathsf{SP}$ the quantified propositions $\forall(A,P)$ and $\exists(A,P)$ belong to SP. There is an obvious homomorphic embedding Tp of SP into the small types defined by $\mathsf{Tp}(\bot) = \emptyset$, $\mathsf{Tp}(\mathsf{atom}(A)) = A$, $\mathsf{Tp}(P \vee Q) = \mathsf{Tp}(P) + \mathsf{Tp}(Q)$, $\mathsf{Tp}(P \wedge Q) = \mathsf{Tp}(P) \times \mathsf{Tp}(Q)$, $\mathsf{Tp}(P \rightarrow Q) = \mathsf{Tp}(P) \rightarrow \mathsf{Tp}(Q)$, $\mathsf{Tp}(\forall(A,P)) = (\Pi x : A)\mathsf{Tp}(P(x))$ and $\mathsf{Tp}(\exists(A,P)) = (\Sigma x : A)\mathsf{Tp}(P(x))$. We shall sometimes write $(\forall x : A)P(x)$ for $\forall(A,P)$ etc.

The simple propositions may be realised by terms from a simplified type structure. All atomic propositions will be realised by the unique element **elt** of the unit type Un. Define another homomorphism Cr (for *crude type*) from SP to small types by letting

$$\begin{aligned}
\mathsf{Cr}(\bot) &= \mathsf{Un} \\
\mathsf{Cr}(\mathsf{atom}(A)) &= \mathsf{Un} \\
\mathsf{Cr}(P \wedge Q) &= \mathsf{Cr}(P) \times \mathsf{Cr}(Q) \\
\mathsf{Cr}(P \vee Q) &= \mathsf{Cr}(P) + \mathsf{Cr}(Q) \\
\mathsf{Cr}(P \rightarrow Q) &= \mathsf{Cr}(P) \rightarrow \mathsf{Cr}(Q) \\
\mathsf{Cr}(\forall(A,P)) &= (\Pi x : A)\mathsf{Cr}(P(x)) \\
\mathsf{Cr}(\exists(A,P)) &= (\Sigma x : A)\mathsf{Cr}(P(x)).
\end{aligned}$$

The only difference from Tp is thus in the translation of absurdity and atoms. Another variant of the crude type map Cr' will be employed in Theorem 1.7 below, which is defined as Cr, except that

$$\mathsf{Cr}'(\exists(A,P)) = \mathsf{Un} + (\Sigma x : A)\mathsf{Cr}'(P(x)).$$

We note that a crude type may still be a dependent type, if the simple proposition is truly infinitary. For example, this is the case with $\mathsf{Cr}(\exists(A,P))$, if $A = N$ and $P(0) = \top$, $P(S(n)) = Q(n) \wedge P(n)$.

The modified realisability $\mathsf{MR}(S,r)$ of a simple proposition $S : \mathsf{SP}$ by an element of crude type $r : \mathsf{Cr}(S)$ is defined as a small proposition (or small type) by the following recursion on S. (We use the identification of propositions and types for small types, so that $\wedge$ and $\vee$ are used interchangeably with $\times$ and $+$, respectively.)

$$
\begin{aligned}
\mathsf{MR}(\bot, r) &= \bot \\
\mathsf{MR}(\mathsf{atom}(P), r) &= P \\
\mathsf{MR}(A \wedge B, r) &= \mathsf{MR}(A, r.1) \wedge \mathsf{MR}(B, r.2) \\
\mathsf{MR}(A \vee B, \mathsf{inl}(s)) &= \mathsf{MR}(A, s) \\
\mathsf{MR}(A \vee B, \mathsf{inr}(t)) &= \mathsf{MR}(B, t) \\
\mathsf{MR}(A \to B, r) &= (\mathsf{Tp}(A) \to \mathsf{Tp}(B)) \\
&\quad \wedge (\Pi s : \mathsf{Cr}(A))(\mathsf{MR}(A, s) \to \mathsf{MR}(B, r(s))) \\
\mathsf{MR}(\forall(A, P), r) &= (\Pi x : A)\mathsf{MR}(P(x), r(x)) \\
\mathsf{MR}(\exists(A, P), r) &= \mathsf{MR}(P(r.1), r.2).
\end{aligned}
$$

Here $r.1$ and $r.2$ denote the first and second projections.

Remark 1.1 The above constructions work in many different type-theoretic settings. What is needed is a type universe U closed under Π, Σ, $+$ and containing basic types Un and $\emptyset$. Moreover the inductive construction SP_U is should be made relative to U instead of Set. Then

$$\mathsf{Tp}_U : \mathsf{SP}_U \to U \qquad \mathsf{Cr}_U : \mathsf{SP}_U \to U$$

are defined by recursion on SP_U similarly to the above, and so is

$$\mathsf{MR}_U : (\Pi s : \mathsf{SP}_U)(\mathsf{Cr}_U(s) \to U).$$

The following correctness, or conservativity, result states that each simple proposition, which is realised, is also true in the standard interpretation.

Theorem 1.2 *For any* $S : \mathsf{SP}$ *and* $r : \mathsf{Cr}(S)$*, if* $\mathsf{MR}(S, r)$ *then* $\mathsf{Tp}(S)$.

Proof. The proof goes by induction on S. For $S = \bot$ or $S = \mathsf{atom}(A)$ the result is immediate. For $S = A \to B$ we took care to define realisability so that this is direct as well. Here are two examples of the inductive step.

Suppose $\mathsf{MR}(A \vee B, r)$. If $r = \mathsf{inl}(s)$, then $\mathsf{MR}(A, s)$ is true. By the inductive hypothesis, we get $\mathsf{Tp}(A)$ and hence also $\mathsf{Tp}(A \vee B)$. The argument for $r = \mathsf{inr}(t)$ is similar.

Assume $\mathsf{MR}(\forall(A, P), r)$. Let $a \in A$. Then $\mathsf{MR}(P(a), r(a))$, and so by the inductive hypothesis $\mathsf{Tp}(P(a))$. Since a was arbitrary we have actually $\mathsf{Tp}(\forall(A, P))$. □

As a corollary there is an extraction theorem for $\forall\exists$-formulae:

Corollary 1.3 *For small types A and B and a simple proposition $P(x,y)$ where $x:A$ and $y:B$, let*

$$S = (\forall x:A)(\exists y:B)P(x,y).$$

If $\mathsf{MR}(S,r)$ for some r, then there is some $f:A\to B$ such that $\mathsf{Tp}(P(x,f(x)))$ for all $x:A$.

Thereby the program f extracted also satisfies its specification $\mathsf{Tp}(P(x,f(x)))$ within type theory. For $P(x,y)=\mathsf{atom}(R(x,y))$ this is equivalent to $R(x,f(x))$.

Remark 1.4 Note the difference in the $\forall$-case from usual interpretations, which go from theories to theories (Troelstra 1973). It is not required that $\mathsf{Tp}(\Pi(A,P))$ is added to the condition, since this follows from the correctness theorem in the present internalised version.

We present an intuitionistic infinitary propositional logic IPC^{-}_{∞} in type theory in which quantifiers are understood as infinitary versions of conjunction and disjunction. The system has a restriction on the absurdity axiom to atomic formulae.

$$A\vdash A \qquad \frac{A\vdash B \quad B\vdash C}{A\vdash C}$$

$A\vdash \mathsf{atom}(P)$, for any inhabited P

$$A\wedge B\vdash A \qquad A\wedge B\vdash B \qquad \frac{C\vdash A \quad C\vdash B}{C\vdash A\wedge B}$$

$$\bot\vdash \mathsf{atom}(P)$$

$$A\vdash A\vee B \qquad B\vdash A\vee B \qquad \frac{A\vdash C \quad B\vdash C}{A\vee B\vdash C}$$

$$\frac{A\wedge B\vdash C}{A\vdash B\to C} \qquad \frac{A\vdash B\to C}{A\wedge B\vdash C}$$

$$\frac{A\vdash P(t) \quad (t:S)}{A\vdash \forall(S,P)} \qquad \frac{A\vdash \forall(S,P) \quad t:S}{A\vdash P(t)}$$

$$\frac{P(t)\vdash A \quad (t:S)}{\exists(S,P)\vdash A} \qquad \frac{\exists(S,P)\vdash A \quad t:S}{P(t)\vdash A}$$

Remark 1.5 Note in particular that the existential quantifier is of the weak kind, as in first order logic. For $S = \emptyset$ each $\exists(S,P)$ works as absurdity constant. However, if we wish to avoid empty sets as types of realisers, the restricted absurdity axiom $\bot \vdash \mathsf{atom(P)}$ should be used. The full absurdity rule can be derived from the restricted one, for those propositions which do not include quantification over empty sets. By this procedure we can in principle extract simply typed programs as in Minlog.

We say that a sequent $A \vdash B$ is MR-*realised,* if there is some r such that $\mathsf{MR}(A \to B, r)$ is true. A rule is *realised* if whenever all the sequents above the rule bar are realised, then so is the sequent below the bar.

Theorem 1.6 *The axioms and rules of the system* IPC^{-}_{∞} *are* MR*-realised.*

To strengthen the weak absurdity axiom to the full axiom

$$\bot \vdash A$$

where $A : \mathsf{SP}$ may be arbitrary, we use the crude type map Cr' instead and introduce MR'. This is defined recursively as MR apart from the case for the existential quantifier:

$$\begin{array}{rcl} \mathsf{MR}'(\exists(S,P),\mathsf{inl}(s)) & = & \bot \\ \mathsf{MR}'(\exists(S,P),\mathsf{inr}(t)) & = & \mathsf{MR}'(P(t.1),t.2). \end{array}$$

Theorem 1.2 and Corollary 1.3 now go through with MR' and Cr' in place of MR and Cr.

The proof of soundness of the logical rules and axioms is similar as for Theorem 1.6, with the exception for the verification of the absurdity rule, and the left existential rule. This requires a special device. Namely a function which to each $P : \mathsf{SP}$ assigns an element, called $\mathsf{element}(P)$, of $\mathsf{Cr}'(P)$ is necessary. This function is defined straightforwardly by recursion on P. Some key clauses are

$$\begin{array}{rcl} \mathsf{element}(\exists(A,P)) & = & \mathsf{inl}(\mathbf{elt}) \\ \mathsf{element}(\forall(A,P)) & = & \lambda x.\mathsf{element}(P(x)) \\ \mathsf{element}(A \vee B) & = & \mathsf{inl}(\mathsf{element}(A)). \end{array}$$

Observe that no such element need to exist when employing the first definition of Cr, e.g. in the case $\mathsf{Cr}(\exists(\emptyset,P)) = (\Sigma x : \emptyset)\mathsf{Cr}(P(x))$.

Theorem 1.7 *The axioms and rules of the full system IPC_∞ (IPC_∞^- and the full absurdity axiom) are* MR'*-realised.*

We mention some useful mathematical axioms that are realisable:

Lemma 1.8 *For each propositional function $P : \mathbb{N} \to \mathsf{SP}$ the induction scheme*

$$P(0) \wedge (\forall x : \mathbb{N})[P(x) \to P(S(x))] \to (\forall x : \mathbb{N})P(x)$$

is both MR*-realised and* MR'*-realised.*

Lemma 1.9 *For any binary propositional function $P : A \times B \to \mathsf{SP}$ the type-theoretic choice principle*

$$(\forall x : A)(\exists y : B)P(x,y) \to (\exists g : A \to B)(\forall x : A)P(x,g(x))$$

is MR*-realisable. In case B is inhabited, the principle is* MR'*-realisable as well.*

Proof. The non-trivial part is to prove the second statement. Suppose $b_0 : B$ and $r : \mathsf{Cr}'(S)$ and $p : \mathsf{MR}'(S,r)$, where $S = (\forall x : A)(\exists y : B)\, P(x,y)$. Define an auxiliary operation $f(x,w) : (\Sigma y : B)\mathsf{Cr}'(P(x,y))$ where $x : A$ and $w : \mathsf{Cr}'((\exists y : B)P(x,y))$, by cases

$$\begin{array}{lcl} f(x,\mathsf{inl}(u)) & = & \langle b_0, \mathsf{element}(P(x,b_0))\rangle \\ f(x,\mathsf{inr}(y)) & = & y. \end{array}$$

The realiser k for the implication is now given by

$$k(r) = \langle \lambda x.f(x,r(x)).1, \lambda x.f(x,r(x)).2\rangle$$

To prove it is a realiser, use $\bot$-elimination for the case $r(x) = \mathsf{inl}(u)$. □

The following result is often useful to verify realisability.

Lemma 1.10 *If the* Tp*-translation of the proposition*

$$(\forall x_1 : A_1)\cdots(\forall x_n : A_n)[Q(x_1,\ldots,x_n) \to P(x_1,\ldots,x_n)]$$

is true and P is atomic or $\bot$, then the proposition is MR*-realised as well as* MR'*-realised.*

Proof. The realising function is trivial for such a proposition: $(\lambda x_1)\cdots(\lambda x_n)(\lambda r)\mathbf{elt}$. □

Many stronger "transfer principles" are possible to establish. See Berger, Buchholz and Schwichtenberg (2002) for further results and references.

2 An Example

We test the formalisation and extraction procedure on a simple example, which is due to Berger and Schwichtenberg. The extracted function computes Fibonacci numbers efficiently by "memoization."

A binary predicate G on natural numbers is given. From the axioms

(Ax1) $G(0,0)$

(Ax2) $G(1,1)$

(Ax3) $(\forall m,k,\ell)[G(m,k) \wedge G(S(m),\ell) \rightarrow G(S(S(m)),k+\ell)]$.

one derives by induction and intuitionistic logic the proposition

(P) $(\forall x)(\exists k,\ell)G(x,k) \wedge G(S(x),\ell)$.

Thus there is some realiser f so that

$$\mathsf{MR}(\mathrm{Ax1} \,\&\, \mathrm{Ax2} \,\&\, \mathrm{Ax3} \vdash \mathrm{P}, f).$$

The extracted program p (which is `fib_prog` in the Appendix) for computing the Fibonacci sequence is then given by

$$p(x) = f(nc,x).1$$

where *nc* (`nocontent` in the Appendix) is the trivial realiser for Ax1 & Ax2 & Ax3. After a normalisation process one gets the program:

```
p x =

(case x of {
   (zero) -> t;
   (succ x') ->
     h x'
       g
       (rec
          (\(z::Nat) -> C)
          x'
          t
          (\(x''::Nat) ->
           \(y::C) ->
           h x''
             g
             y));}).1
```

where

```
C = Sigma Nat (\(k::Nat) -> Sigma Nat (\(l::Nat) -> Unit))

h v p q = <q.2.1;
                 <case q.2.1 of {(zero) -> q.1;
                                 (succ u) -> succ (q.1 + u);
                                }
                       ;<q.2.2.2; e>>>

t = <zero; <succ zero; <e;e>>>


g = \(x,y,z::Nat) -> \(h,j::Unit) -> e

e = elt@_
```

Remark 2.1 Note that all truly dependent types have disappeared. The type C is really the type $\mathsf{N} \times (\mathsf{N} \times \mathsf{Un})$.

The normalised program has been computed using the partial normalisation procedure of Agda on selected subexpressions, and was thus not completely automatic. We also introduced the abbreviations C, h, t, g, e by hand. Some syntactical sugar for lambda expressions and pairs is added.

References

U. Berger, W. Buchholz and H. Schwichtenberg. Refined Program Extraction from Classical Proofs, *Annals of Pure and Applied Logic* 114(2002), 3–25.

H. Benl, U. Berger, M. Seisenberger, H. Schwichtenberg and W. Zuber. Proof theory at work: Program development in the Minlog system. In: W. Bibel and P.H. Schmitt (eds.), *Automated Deduction, Vol. II,* Kluwer 1998.

C. Coquand. *The interactive theorem prover Agda.* Chalmers University of Technology, Department of Computer Science and Engineering, 2000
URL: `www.cs.chalmers.se/~catarina/agda/`

P. Letouzey. *Programmation fonctionelle certifiée: L'extraction de programmes dans l'assistant Coq.* Thèse de doctorat, Université de Paris Sud, 2004.

P. Martin-Löf. *An intuitionistic theory of types.* Technical report, Department of Mathematics, University of Stockholm, 1972. Reprinted in (Sambin and Smith 1998).

G. Sambin and J. Smith (eds.). *Twenty-Five Years of Type Theory.* Oxford University Press, 1998.

H. Schwichtenberg. *Minimal Logic for Computable Functions.* Mathematisches Institut der Universität München, Preprint October 30, 2004.

A.S. Troelstra. *Metamathematical Investigation of Intuitionistic Analysis and Arithmetic.* Springer 1973.

A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, vol. I and II.* North-Holland 1988.

Appendix: Formalised theorems

Agda/IAgda version 2003-08-09, with the aid of the Alfa, have been used in this formalisation. The files of this appendix and of the complete formalisation are available at URL:

`www.math.uu.se/~palmgren/modif`

Below is a printout of the file `realisability.agda`

```
--#include "BasicTypeTheory.agda"

-- The simple propositions are defined as an inductive data type

-- where there is an atomic formula atom(P) for

-- each (constructive) proposition P

SP :: Type
  = data absurd |
         atom (P::Set) |
         sand (a::SP) (b::SP) |
         simp (a::SP) (b::SP) |
         sor (a::SP) (b::SP) |
         forall (A::Set) (P::A -> SP) |
         exists (A::Set) (P::A -> SP)

-- translation to  ordinary proposition

Tp (S::SP) :: Set
  = case S of {
       (absurd) -> Absurd;
       (atom P) -> P;
       (sand a b) -> Cart (Tp a) (Tp b);
       (simp a b) -> Tp a -> Tp b;
       (sor a b) -> Sum (Tp a) (Tp b);
       (forall A P) -> Pi A (\(h::A) -> Tp (P h));
       (exists A P) -> Sigma A (\(h::A) -> Tp (P h));}

-- make types for realising terms

Cr (S::SP) :: Set
  = case S of {
       (absurd) -> Unit;
       (atom P) -> Unit;
       (sand a b) -> Cart (Cr a) (Cr b);
       (simp a b) -> Cr a -> Cr b;
       (sor a b) -> Sum (Cr a) (Cr b);
       (forall A P) -> Pi A (\(h::A) -> Cr (P h));
       (exists A P) -> Sigma A (\(h::A) -> Cr (P h));}

-- Modified realisability with truth (see implication case)

MR (S::SP)(r::Cr S) :: Set
  = case S of {
       (absurd) -> Absurd;
       (atom P) -> P;
       (sand a b) -> and (MR a r._1) (MR b r._2);
```

```
      (simp a b) ->
        and ((x::Cr a) -> MR a x -> MR b (r x)) (Tp a -> Tp b);
      (sor a b) ->
        case r of {
          (inl x) -> MR a x;
          (inr y) -> MR b y;};
      (forall A P) -> (t::A) -> MR (P t) (r t);
      (exists A P) -> MR (P r._1) r._2;}

-- Correctness theorem

Correct (S::SP)(r::Cr S)(p::MR S r) :: Tp S
  = case S of {
      (absurd) -> p;
      (atom P) -> p;
      (sand a b) ->
        struct {
          _1 = Correct a r._1 p._1;
          _2 = Correct b r._2 p._2;};
      (simp a b) -> p._2;
      (sor a b) ->
        case r of {
          (inl x) -> inl@_ (Correct a x p);
          (inr y) -> inr@_ (Correct b y p);};
      (forall A P) -> \(x::A) -> Correct (P x) (r x) (p x);
      (exists A P) ->
        struct {
          _1 = r._1;
          _2 = Correct (P _1) r._2 p;};}

-- predicate for realised formulas

Realized (A::SP) :: Set
  = sig{real :: Cr A;
        pf :: MR A real;}

Correct_2 (A::SP)(pf::Realized A) :: Tp A
  = Correct A pf.real pf.pf

-- extraction theorem for AE-formulas

Extract (A::Set)
        (B::Set)
        (P::A -> B -> SP)
        (pfe::Realized (forall@_ A (\(x::A) -> exists@_ B (\(y::B) -> P x y))))
  :: Pi A (\(x::A) -> Sigma B (\(y::B) -> Tp (P x y)))
  = \(x::A) ->
    Correct_2
      (exists@_ B (\(y::B) -> P x y))
      (struct {
         real = pfe.real x;
         pf = pfe.pf x;})

true_impl_realised (A::Set)(B::Set)(p::A -> B)
  :: Realized (simp@_ (atom@_ A) (atom@_ B))
  = struct {
      real = \(h::Cr (atom@_ A)) -> h;
      pf =
        struct {
          _1 = \(x::Cr (atom@_ A)) -> p;
          _2 = p;};}
```

```
-- First order logic rules for realisability

Entails (a::SP)(b::SP) :: SP
  = simp@_ a b

truthaxiom (a::SP)(P::Set)(p::P)
  :: Realized (Entails a (atom@_ P))
  = struct {
      real = \(h::Cr a) -> elt@_;
      pf =
        struct {
          _1 = \(x::Cr a) -> \(h::MR a x) -> p;
          _2 = \(h::Tp a) -> p;};}

-- exfalso restricted to atomic formulae

absurdityaxiom (P::Set) :: Realized (Entails absurd@_ (atom@_ P))
  = struct {
      real = \(h::Cr absurd@_) -> elt@_;
      pf =
        struct {
          _1 =
            \(x::Cr absurd@_) ->
            \(h::MR absurd@_ x) ->
            elempty (MR (atom@_ P) (real x)) h;
          _2 = \(h::Tp absurd@_) -> elempty (Tp (atom@_ P)) h;};}

-- Validity of rules for first order logic

-- system essentially the one often used in categorical logic

idaxiom (a::SP) :: Realized (Entails a a)
  = struct {
      real = \(h::Cr a) -> h;
      pf =
        struct {
          _1 = \(x::Cr a) -> \(h::MR a x) -> h;
          _2 = \(h::Tp a) -> h;};}

cut (a::SP)
    (b::SP)
    (c::SP)
    (p1::Realized (Entails a b))
    (p2::Realized (Entails b c))
  :: Realized (Entails a c)
  = struct {
      real = \(h::Cr a) -> p2.real (p1.real h);
      pf =
        struct {
          _1 =
            \(x::Cr a) ->
            \(h::MR a x) ->
            p2.pf._1 (p1.real x) (p1.pf._1 x h);
          _2 = \(h::Tp a) -> p2.pf._2 (p1.pf._2 h);};}

And_i (a::SP)
      (b::SP)
      (c::SP)
      (p1::Realized (Entails a b))
      (p2::Realized (Entails a c))
  :: Realized (Entails a (sand@_ b c))
  = struct {
```

```
      real =
        \(h::Cr a) ->
        struct {
          _1 = p1.real h;
          _2 = p2.real h;};
      pf =
        struct {
          _1 =
            \(x::Cr a) ->
            \(h::MR a x) ->
            struct {
              _1 = p1.pf._1 x h;
              _2 = p2.pf._1 x h;};
          _2 =
            \(h::Tp a) ->
            struct {
              _1 = p1.pf._2 h;
              _2 = p2.pf._2 h;};};}

And_e1 (a::SP)(b::SP) :: Realized (Entails (sand@_ a b) a)
  = struct {
      real = \(h::Cr (sand@_ a b)) -> h._1;
      pf =
        struct {
          _1 = \(x::Cr (sand@_ a b)) -> \(h::MR (sand@_ a b) x) -> h._1;
          _2 = \(h::Tp (sand@_ a b)) -> h._1;};}

And_e2 (a::SP)(b::SP) :: Realized (Entails (sand@_ a b) b)
  = struct {
      real = \(h::Cr (sand@_ a b)) -> h._2;
      pf =
        struct {
          _1 = \(x::Cr (sand@_ a b)) -> \(h::MR (sand@_ a b) x) -> h._2;
          _2 = \(h::Tp (sand@_ a b)) -> h._2;};}

Imp_i (a::SP)
      (b::SP)
      (c::SP)
      (p::Realized (Entails (sand@_ a b) c))
  :: Realized (Entails a (simp@_ b c))
  = struct {
      real =
        \(h::Cr a) ->
        \(h'::Cr b) ->
        p.real
          (struct {
             _1 = h;
             _2 = h';});
      pf =
        struct {
          _1 =
            \(x::Cr a) ->
            \(h::MR a x) ->
            struct {
              _1 =
                \(x'::Cr b) ->
                \(h'::MR b x') ->
                p.pf._1
                  (struct {
                     _1 = x;
                     _2 = x';})
                  (struct {
```

```
                    _1 = h;
                    _2 = h';});
              _2 =
                \(h'::Tp b) ->
                p.pf._2
                  (struct {
                     _1 = Correct a x h;
                     _2 = h';});};
          _2 =
            \(h::Tp a) ->
            \(h'::Tp b) ->
            p.pf._2
              (struct {
                 _1 = h;
                 _2 = h';});};}

Or_i1 (a::SP)(b::SP) :: Realized (Entails a (sor@_ a b))
  = struct {
      real = \(h::Cr a) -> inl@_ h;
      pf =
        struct {
          _1 = \(x::Cr a) -> \(h::MR a x) -> h;
          _2 = \(h::Tp a) -> inl@_ h;};}

Or_i2 (a::SP)(b::SP) :: Realized (Entails b (sor@_ a b))
  = struct {
      real = \(h::Cr b) -> inr@_ h;
      pf =
        struct {
          _1 = \(x::Cr b) -> \(h::MR b x) -> h;
          _2 = \(h::Tp b) -> inr@_ h;};}

Or_e (a::SP)
     (b::SP)
     (c::SP)
     (p1::Realized (Entails a c))
     (p2::Realized (Entails b c))
  :: Realized (Entails (sor@_ a b) c)
  = struct {
      real =
        \(h::Cr (sor@_ a b)) ->
        case h of {
          (inl x) -> p1.real x;
          (inr y) -> p2.real y;};
      pf =
        struct {
          _1 =
            \(x::Cr (sor@_ a b)) ->
            case x of {
              (inl x') -> \(h::MR (sor@_ a b) (inl@_ x')) -> p1.pf._1 x' h;
              (inr y) -> \(h::MR (sor@_ a b) (inr@_ y)) -> p2.pf._1 y h;};
          _2 =
            \(h::Tp (sor@_ a b)) ->
            case h of {
              (inl x) -> p1.pf._2 x;
              (inr y) -> p2.pf._2 y;};};}

Imp_e (a::SP)
      (b::SP)
      (c::SP)
      (p::Realized (Entails a (simp@_ b c)))
  :: Realized (Entails (sand@_ a b) c)
```

```
  = struct {
      real = \(h::Cr (sand@_ a b)) -> p.real h._1 h._2;
      pf =
        struct {
          _1 =
            \(x::Cr (sand@_ a b)) ->
            \(h::MR (sand@_ a b) x) ->
            let p1 :: MR (simp@_ b c) (p.real x._1)
                  = p.pf._1 x._1 h._1
            in  p1._1 x._2 h._2;
          _2 =
            \(h::Tp (sand@_ a b)) ->
            let p2 :: (h'::Tp a) -> Tp (simp@_ b c)
                  = p.pf._2
            in  p2 h._1 h._2;};}

All_i (A::Set)
      (a::SP)
      (P::A -> SP)
      (p::(x::A) -> Realized (Entails a (P x)))
  :: Realized (Entails a (forall@_ A P))
  = struct {
      real = \(h::Cr a) -> \(x::A) -> (p x).real h;
      pf =
        struct {
          _1 = \(x::Cr a) -> \(h::MR a x) -> \(t::A) -> (p t).pf._1 x h;
          _2 = \(h::Tp a) -> \(x::A) -> (p x).pf._2 h;};}

All_e (A::Set)
      (a::SP)
      (P::A -> SP)
      (p::Realized (Entails a (forall@_ A P)))
      (x::A)
  :: Realized (Entails a (P x))
  = struct {
      real = \(h::Cr a) -> p.real h x;
      pf =
        let p1 :: MR (Entails a (forall@_ A P)) p.real
              = p.pf
        in  struct {
              _1 = \(x'::Cr a) -> \(h::MR a x') -> p1._1 x' h x;
              _2 = \(h::Tp a) -> p1._2 h x;};}

Ex_i (A::Set)
     (a::SP)
     (P::A -> SP)
     (p::(x::A) -> Realized (Entails (P x) a))
  :: Realized (Entails (exists@_ A P) a)
  = struct {
      real = \(h::Cr (exists@_ A P)) -> (p h._1).real h._2;
      pf =
        struct {
          _1 =
            \(x::Cr (exists@_ A P)) ->
            \(h::MR (exists@_ A P) x) ->
            (p x._1).pf._1 x._2 h;
          _2 = \(h::Tp (exists@_ A P)) -> (p h._1).pf._2 h._2;};}

Ex_e (A::Set)
     (a::SP)
     (P::A -> SP)
     (p::Realized (Entails (exists@_ A P) a))
```

```
    (x::A)
  :: Realized (Entails (P x) a)
  = struct {
      real =
        \(h::Cr (P x)) ->
        p.real
          (struct {
             _1 = x;
             _2 = h;});
      pf =
        struct {
          _1 =
            \(x'::Cr (P x)) ->
            \(h::MR (P x) x') ->
            p.pf._1
              (struct {
                 _1 = x;
                 _2 = x';})
              h;
          _2 =
            \(h::Tp (P x)) ->
            p.pf._2
              (struct {
                 _1 = x;
                 _2 = h;});};}
```

The Example

File: `ex_realisability.agda`

```
--#include "realisability.agda"

Ex_i2 (A::Set)(P::A -> SP)(x::A)
  :: Realized (Entails (P x) (exists@_ A P))
  = Ex_e A (exists@_ A P) P (idaxiom (exists@_ A P)) x

All_e2 (A::Set)(P::A -> SP)(x::A)
  :: Realized (Entails (forall@_ A P) (P x))
  = All_e A (forall@_ A P) P (idaxiom (forall@_ A P)) x

modus_ponens (a::SP)(b::SP)
  :: Realized (Entails (sand@_ (simp@_ a b) a) b)
  = Imp_e (simp@_ a b) a b (idaxiom (simp@_ a b))

prop_lemma (a::SP)(b::SP)(c::SP)
  :: Realized
       (Entails (simp@_ a (simp@_ b c)) (simp@_ (sand@_ a b) (sand@_ b c)))
  = Imp_i
      (simp@_ a (simp@_ b c))
      (sand@_ a b)
      (sand@_ b c)
      (And_i
         (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
         b
         c
         (cut
            (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
```

```
            (sand@_ a b)
            b
            (And_e2 (simp@_ a (simp@_ b c)) (sand@_ a b))
            (And_e2 a b))
         (cut
            (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
            (sand@_ (simp@_ b c) b)
            c
            (And_i
               (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
               (simp@_ b c)
               b
               (cut
                  (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
                  (sand@_ (simp@_ a (simp@_ b c)) a)
                  (simp@_ b c)
                  (And_i
                     (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
                     (simp@_ a (simp@_ b c))
                     a
                     (And_e1 (simp@_ a (simp@_ b c)) (sand@_ a b))
                     (cut
                        (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
                        (sand@_ a b)
                        a
                        (And_e2 (simp@_ a (simp@_ b c)) (sand@_ a b))
                        (And_e1 a b)))
                  (modus_ponens a (simp@_ b c)))
               (cut
                  (sand@_ (simp@_ a (simp@_ b c)) (sand@_ a b))
                  (sand@_ a b)
                  b
                  (And_e2 (simp@_ a (simp@_ b c)) (sand@_ a b))
                  (And_e2 a b)))
            (modus_ponens b c)))

pred_lemma (b::SP)(A::Set)(P::A -> SP)
  :: Realized
       (Entails
          (forall@_ A (\(x::A) -> simp@_ (P x) b))
          (simp@_ (exists@_ A P) b))
  = Imp_i
      (forall@_ A (\(x::A) -> simp@_ (P x) b))
      (exists@_ A P)
      b
      (cut
         (sand@_ (forall@_ A (\(x::A) -> simp@_ (P x) b)) (exists@_ A P))
         (sand@_ (exists@_ A P) (forall@_ A (\(x::A) -> simp@_ (P x) b)))
         b
         (And_i
            (sand@_ (forall@_ A (\(x::A) -> simp@_ (P x) b)) (exists@_ A P))
            (exists@_ A P)
            (forall@_ A (\(x::A) -> simp@_ (P x) b))
            (And_e2 (forall@_ A (\(x::A) -> simp@_ (P x) b)) (exists@_ A P))
            (And_e1 (forall@_ A (\(x::A) -> simp@_ (P x) b)) (exists@_ A P)))
         (Imp_e
            (exists@_ A P)
            (forall@_ A (\(x::A) -> simp@_ (P x) b))
            b
            (Ex_i
               A
               (simp@_ (forall@_ A (\(x::A) -> simp@_ (P x) b)) b)
```

```
              P
              (\(x::A) ->
               Imp_i
                 (P x)
                 (forall@_ A (\(x'::A) -> simp@_ (P x') b))
                 b
                 (cut
                    (sand@_ (P x) (forall@_ A (\(x'::A) -> simp@_ (P x') b)))
                    (sand@_ (simp@_ (P x) b) (P x))
                    b
                    (And_i
                       (sand@_ (P x) (forall@_ A (\(x'::A) -> simp@_ (P x') b)))
                       (simp@_ (P x) b)
                       (P x)
                       (cut
                          (sand@_
                             (P x)
                             (forall@_ A (\(x'::A) -> simp@_ (P x') b)))
                          (forall@_ A (\(x'::A) -> simp@_ (P x') b))
                          (simp@_ (P x) b)
                          (And_e2
                             (P x)
                             (forall@_ A (\(x'::A) -> simp@_ (P x') b)))
                          (All_e2 A (\(x'::A) -> simp@_ (P x') b) x))
                       (And_e1 (P x) (forall@_ A (\(x'::A) -> simp@_ (P x') b))))
                    (modus_ponens (P x) b))))))

(+) (m::Nat)(n::Nat) :: Nat
  = case n of {
      (zero) -> m;
      (succ x) -> succ@_ (m + x);}

ind (P::Nat -> SP)
  :: Realized
       (Entails
          (sand@_
             (P zero@_)
             (forall@_ Nat (\(n::Nat) -> simp@_ (P n) (P (succ@_ n)))))
          (forall@_ Nat (\(n::Nat) -> P n)))
  = struct {
      real =
        \(h::Cr
               (sand@_
                  (P zero@_)
                  (forall@_ Nat (\(n::Nat) -> simp@_ (P n) (P (succ@_ n)))))) ->
        \(x::Nat) ->
        rec
          (\(z::Nat) -> Cr (P z))
          x
          h._1
          (\(x'::Nat) -> \(y::Cr (P x')) -> h._2 x' y);
      pf =
        struct {
          _1 =
            \(h::Cr
                   (sand@_
                      (P zero@_)
                      (forall@_ Nat (\(n::Nat) -> simp@_ (P n) (P (succ@_ n)))))) ->
            \(r::MR
                   (sand@_
                      (P zero@_)
                      (forall@_ Nat (\(n::Nat) -> simp@_ (P n) (P (succ@_ n))))))
```

```
                h) ->
          \(n::Nat) ->
          rec
            (\(z::Nat) -> MR (P z) (real h z))
            n
            r._1
            (\(x::Nat) ->
             \(y::MR (P x) (real h x)) ->
             let p2 :: MR (simp@_ (P x) (P (succ@_ x))) (h._2 x)
                   = r._2 x
             in  p2._1
                   (rec
                      (\(z::Nat) -> Cr (P z))
                      x
                      h._1
                      (\(x'::Nat) -> \(y'::Cr (P x')) -> h._2 x' y'))
                 y);
        _2 =
          \(h::Tp
                 (sand@_
                    (P zero@_)
                    (forall@_ Nat (\(n::Nat) -> simp@_ (P n) (P (succ@_ n)))))) ->
          \(x::Nat) ->
          rec
            (\(z::Nat) -> Tp (P z))
            x
            h._1
            (\(x'::Nat) -> \(y::Tp (P x')) -> h._2 x' y);};}

postulate G :: Nat -> Nat -> Set

Ax1 :: SP
  = atom@_ (G zero@_ zero@_)

Ax2 :: SP
  = atom@_ (G (succ@_ zero@_) (succ@_ zero@_))

Ax3 :: SP
  = forall@_
      Nat
      (\(n::Nat) ->
       forall@_
         Nat
         (\(k::Nat) ->
          forall@_
            Nat
            (\(l::Nat) ->
             simp@_
               (atom@_ (G n k))
               (simp@_
                  (atom@_ (G (succ@_ n) l))
                  (atom@_ (G (succ@_ (succ@_ n)) (k + l)))))))

Q (n::Nat)(k::Nat)(l::Nat) :: SP
  = sand@_ (atom@_ (G n k)) (atom@_ (G (succ@_ n) l))

F (n::Nat) :: SP
  = exists@_ Nat (\(k::Nat) -> exists@_ Nat (\(l::Nat) -> Q n k l))

fib_lm (x::Nat)
  :: Realized (Entails Ax3 (simp@_ (F x) (F (succ@_ x))))
  = cut
```

```
Ax3
(forall@_
   Nat
   (\(k::Nat) ->
    forall@_ Nat (\(l::Nat) -> simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))))
(simp@_ (F x) (F (succ@_ x)))
(cut
   Ax3
   (forall@_
      Nat
      (\(k::Nat) ->
       forall@_
         Nat
         (\(l::Nat) ->
          simp@_
            (atom@_ (G x k))
            (simp@_
               (atom@_ (G (succ@_ x) l))
               (atom@_ (G (succ@_ (succ@_ x)) (k + l)))))))
   (forall@_
      Nat
      (\(k::Nat) ->
       forall@_
         Nat
         (\(l::Nat) -> simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))))
   (All_e2
      Nat
      (\(n::Nat) ->
       forall@_
         Nat
         (\(k::Nat) ->
          forall@_
            Nat
            (\(l::Nat) ->
             simp@_
               (atom@_ (G n k))
               (simp@_
                  (atom@_ (G (succ@_ n) l))
                  (atom@_ (G (succ@_ (succ@_ n)) (k + l)))))))
      x)
   (All_i
      Nat
      (forall@_
         Nat
         (\(k::Nat) ->
          forall@_
            Nat
            (\(l::Nat) ->
             simp@_
               (atom@_ (G x k))
               (simp@_
                  (atom@_ (G (succ@_ x) l))
                  (atom@_ (G (succ@_ (succ@_ x)) (k + l)))))))
      (\(k::Nat) ->
       forall@_
         Nat
         (\(l::Nat) -> simp@_ (Q x k l) (Q (succ@_ x) l (k + l))))
      (\(k::Nat) ->
       All_i
         Nat
         (forall@_
            Nat
```

```
   (\(k'::Nat) ->
    forall@_
      Nat
      (\(l::Nat) ->
       simp@_
         (atom@_ (G x k'))
         (simp@_
            (atom@_ (G (succ@_ x) l))
            (atom@_ (G (succ@_ (succ@_ x)) (k' + l)))))))
(\(l::Nat) -> simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))
(\(l::Nat) ->
 cut
   (forall@_
      Nat
      (\(k'::Nat) ->
       forall@_
         Nat
         (\(l'::Nat) ->
          simp@_
            (atom@_ (G x k'))
            (simp@_
               (atom@_ (G (succ@_ x) l'))
               (atom@_ (G (succ@_ (succ@_ x)) (k' + l')))))))
   (forall@_
      Nat
      (\(l'::Nat) ->
       simp@_
         (atom@_ (G x k))
         (simp@_
            (atom@_ (G (succ@_ x) l'))
            (atom@_ (G (succ@_ (succ@_ x)) (k + l'))))))
   (simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))
   (All_e2
      Nat
      (\(k'::Nat) ->
       forall@_
         Nat
         (\(l'::Nat) ->
          simp@_
            (atom@_ (G x k'))
            (simp@_
               (atom@_ (G (succ@_ x) l'))
               (atom@_ (G (succ@_ (succ@_ x)) (k' + l'))))))
      k)
   (cut
      (forall@_
         Nat
         (\(l'::Nat) ->
          simp@_
            (atom@_ (G x k))
            (simp@_
               (atom@_ (G (succ@_ x) l'))
               (atom@_ (G (succ@_ (succ@_ x)) (k + l'))))))
      (simp@_
         (atom@_ (G x k))
         (simp@_
            (atom@_ (G (succ@_ x) l))
            (atom@_ (G (succ@_ (succ@_ x)) (k + l))))))
      (simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))
      (All_e2
         Nat
         (\(l'::Nat) ->
```

```
                  simp@_
                    (atom@_ (G x k))
                    (simp@_
                       (atom@_ (G (succ@_ x) l'))
                       (atom@_ (G (succ@_ (succ@_ x)) (k + l')))))
                 l)
              (prop_lemma
                 (atom@_ (G x k))
                 (atom@_ (G (succ@_ x) l))
                 (atom@_ (G (succ@_ (succ@_ x)) (k + l)))))))))
(cut
   (forall@_
      Nat
      (\(k::Nat) ->
       forall@_
         Nat
         (\(l::Nat) -> simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))))
   (forall@_
      Nat
      (\(x'::Nat) ->
       simp@_ (exists@_ Nat (\(l::Nat) -> Q x x' l)) (F (succ@_ x))))
   (simp@_ (F x) (F (succ@_ x)))
   (All_i
      Nat
      (forall@_
         Nat
         (\(k::Nat) ->
          forall@_
            Nat
            (\(l::Nat) -> simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))))
      (\(x'::Nat) ->
       simp@_ (exists@_ Nat (\(l::Nat) -> Q x x' l)) (F (succ@_ x)))
      (\(k::Nat) ->
       cut
         (forall@_
            Nat
            (\(k'::Nat) ->
             forall@_
               Nat
               (\(l::Nat) -> simp@_ (Q x k' l) (Q (succ@_ x) l (k' + l)))))
         (forall@_ Nat (\(x'::Nat) -> simp@_ (Q x k x') (F (succ@_ x))))
         (simp@_ (exists@_ Nat (\(l::Nat) -> Q x k l)) (F (succ@_ x)))
         (All_i
            Nat
            (forall@_
               Nat
               (\(k'::Nat) ->
                forall@_
                  Nat
                  (\(l::Nat) ->
                   simp@_ (Q x k' l) (Q (succ@_ x) l (k' + l)))))
            (\(x'::Nat) -> simp@_ (Q x k x') (F (succ@_ x)))
            (\(l::Nat) ->
             cut
               (forall@_
                  Nat
                  (\(k'::Nat) ->
                   forall@_
                     Nat
                     (\(l'::Nat) ->
                      simp@_ (Q x k' l') (Q (succ@_ x) l' (k' + l')))))
               (forall@_
```

```
                  Nat
                  (\(l'::Nat) ->
                   simp@_ (Q x k l') (Q (succ@_ x) l' (k + l'))))
               (simp@_ (Q x k l) (F (succ@_ x)))
               (All_e2
                  Nat
                  (\(k'::Nat) ->
                   forall@_
                     Nat
                     (\(l'::Nat) ->
                      simp@_ (Q x k' l') (Q (succ@_ x) l' (k' + l'))))
                  k)
               (cut
                  (forall@_
                     Nat
                     (\(l'::Nat) ->
                      simp@_ (Q x k l') (Q (succ@_ x) l' (k + l'))))
                  (simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))
                  (simp@_ (Q x k l) (F (succ@_ x)))
                  (All_e2
                     Nat
                     (\(l'::Nat) ->
                      simp@_ (Q x k l') (Q (succ@_ x) l' (k + l')))
                     l)
                  (Imp_i
                     (simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))
                     (Q x k l)
                     (F (succ@_ x))
                     (cut
                        (sand@_
                           (simp@_ (Q x k l) (Q (succ@_ x) l (k + l)))
                           (Q x k l))
                        (Q (succ@_ x) l (k + l))
                        (F (succ@_ x))
                        (modus_ponens (Q x k l) (Q (succ@_ x) l (k + l)))
                        (cut
                           (Q (succ@_ x) l (k + l))
                           (exists@_ Nat (\(l'::Nat) -> Q (succ@_ x) l l'))
                           (F (succ@_ x))
                           (Ex_i2
                              Nat
                              (\(l'::Nat) -> Q (succ@_ x) l l')
                              (k + l))
                           (Ex_i2
                              Nat
                              (\(k'::Nat) ->
                               exists@_
                                 Nat
                                 (\(l'::Nat) -> Q (succ@_ x) k' l'))
                              l)))))))
            (pred_lemma (F (succ@_ x)) Nat (\(l::Nat) -> Q x k l))))
      (pred_lemma
         (F (succ@_ x))
         Nat
         (\(k::Nat) -> exists@_ Nat (\(l::Nat) -> Q x k l))))

fib_thm
  :: Realized
       (Entails
          (sand@_ (sand@_ Ax1 Ax2) Ax3)
          (forall@_ Nat (\(n::Nat) -> F n)))
  = cut
```

```
      (sand@_ (sand@_ Ax1 Ax2) Ax3)
      (sand@_
         (F zero@_)
         (forall@_ Nat (\(n::Nat) -> simp@_ (F n) (F (succ@_ n)))))
      (forall@_ Nat (\(n::Nat) -> F n))
      (And_i
         (sand@_ (sand@_ Ax1 Ax2) Ax3)
         (F zero@_)
         (forall@_ Nat (\(n::Nat) -> simp@_ (F n) (F (succ@_ n))))
         (cut
            (sand@_ (sand@_ Ax1 Ax2) Ax3)
            (sand@_ Ax1 Ax2)
            (F zero@_)
            (And_e1 (sand@_ Ax1 Ax2) Ax3)
            (cut
               (sand@_ Ax1 Ax2)
               (exists@_ Nat (\(l::Nat) -> Q zero@_ zero@_ l))
               (F zero@_)
               (Ex_i2 Nat (\(l::Nat) -> Q zero@_ zero@_ l) (succ@_ zero@_))
               (Ex_i2
                  Nat
                  (\(k::Nat) -> exists@_ Nat (\(l::Nat) -> Q zero@_ k l))
                  zero@_)))
         (cut
            (sand@_ (sand@_ Ax1 Ax2) Ax3)
            Ax3
            (forall@_ Nat (\(n::Nat) -> simp@_ (F n) (F (succ@_ n))))
            (And_e2 (sand@_ Ax1 Ax2) Ax3)
            (All_i
               Nat
               Ax3
               (\(n::Nat) -> simp@_ (F n) (F (succ@_ n)))
               (\(x::Nat) -> fib_lm x))))
      (ind F)


nocontent :: Cr (sand@_ (sand@_ Ax1 Ax2) Ax3)
  = struct {
      _1 =
        struct {
          _1 = elt@_;
          _2 = elt@_;};
      _2 =
        \(x::Nat) ->
        \(x'::Nat) ->
        \(x0::Nat) ->
        \(h::Cr (atom@_ (G x x'))) ->
        \(h'::Cr (atom@_ (G (succ@_ x) x0))) ->
        elt@_;}


fib_prog (x::Nat) :: Nat
  = (fib_thm.real nocontent x)._1
```

Basic Type Theory

File: `BasicTypeTheory.agda`

```
-- Families of sets

Fam (A::Set) :: Type
  = A -> Set

--  Syntactic sugar for the $\Pi$-construction

Pi (A::Set)(B::Fam A) :: Set
  = (x::A) -> B x

ForAll (A::Set)(B::Fam A) :: Set
  = (x::A) -> B x

-- Syntactic sugar relating to the $\Sigma$-construction

Sigma (A::Set)(B::Fam A) :: Set
  = sig{_1 :: A;
        _2 :: B _1;}

Exists (A::Set)(B::Fam A) :: Set
  = sig{_1 :: A;
        _2 :: B _1;}

pair (A::Set)(B::Fam A)(a::A)(b::B a) :: Sigma A B
  = struct {
      _1 = a;
      _2 = b;}

pairExists (A::Set)(B::Fam A)(a::A)(b::B a) :: Exists A B
  = struct {
      _1 = a;
      _2 = b;}

split (A::Set)
      (B::Fam A)
      (C::Fam (Sigma A B))
      (c::Sigma A B)
      (d::(x::A) -> (y::B x) -> C (pair A B x y))
  :: C c
  = d c._1 c._2

splitExists (A::Set)
            (B::Fam A)
            (C::Fam (Exists A B))
            (c::Exists A B)
            (d::(x::A) -> (y::B x) -> C (pairExists A B x y))
  :: C c
  = d c._1 c._2

Cart (A::Set)(B::Set) :: Set
  = sig{_1 :: A;
        _2 :: B;}
```

```
pairCart (A::Set)(B::Set)(a::A)(b::B) :: Cart A B
  = struct {
      _1 = a;
      _2 = b;}

proj1 (A::Set)(B::Set)(c::Cart A B) :: A
  = c._1

proj2 (A::Set)(B::Set)(c::Cart A B) :: B
  = c._2

and (A::Set)(B::Set) :: Set
  = Cart A B

--  Disjoint binary sums

Sum (A::Set)(B::Set) :: Set
  = data inl (x::A) | inr (y::B)

or (A::Set)(B::Set) :: Set
  = Sum A B

when (A::Set)
     (B::Set)
     (C::Fam (Sum A B))
     (c::Sum A B)
     (d::(x::A) -> C (inl@_ x))
     (e::(y::B) -> C (inr@_ y))
  :: C c
  = case c of {
      (inl x) -> d x;
      (inr y) -> e y;}

-- If and only if

iff (A::Set)(B::Set) :: Set
  = Cart (A -> B) (B -> A)

-- The empty set

empty :: Set
  = data

Absurd :: Set
  = empty

elempty (A::Set)(x::empty) :: A
  = case x of { }

not (A::Set) :: Set
  = A -> Absurd

-- A unit set

Unit :: Set
  = data elt

True :: Set
  = Unit

-- Booleans
```

```
Bool :: Set
  = data ff | tt

-- Natural numbers

Nat :: Set
  = data zero | succ (x::Nat)

rec (C::(z::Nat) -> Set)
    (t::Nat)
    (f::C zero@_)
    (g::(x::Nat) -> (y::C x) -> C (succ@_ x))
  :: C t
  = case t of {
      (zero) -> f;
      (succ x) -> g x (rec C x f g);}

-- A basic dependent type

L (z::Nat) :: Set
  = case z of {
      (zero) -> empty;
      (succ x) -> Nat;}

-- Deriving first projection from split

pr1 (A::Set)(B::Fam A)(c::Sigma A B) :: A
  = split A B (\(h::Sigma A B) -> A) c (\(x::A) -> \(y::B x) -> x)

pr2 (A::Set)(B::Fam A)(c::Sigma A B) :: B (pr1 A B c)
  = split
      A
      B
      (\(h::Sigma A B) -> B (pr1 A B h))
      c
      (\(x::A) -> \(y::B x) -> y)
```