Towards a Formally Verified Proof Assistant

Abhishek Anand and Vincent Rahli
Cornell University

April 3, 2014

Abstract

This technical report describes our progress towards a formally verified version of the Nuprl Proof Assistant. We define a deep embedding of most of Nuprl into Coq. Among others, it includes a nominal-style definition of the Nuprl language, reduction rules, a coinductively defined computational equivalence, and the curry-style type system where types are defined as partial equivalence relations. Along with the core Martin-Löf dependent types, it includes Nuprl's hierarchy of universes, inductive types and partial types.

Contents

This document is still a work in progress and may contain some mistakes. Please visit `http://www.nuprl.org/html/verification/` for the latest version.

Chapter 1

Introduction

1.1 Trustworthiness of Proof Assistants

When we trust a proof checked by a proof assistant, we are trusting many aspects of it. Most of the popular proof assistants (such as Coq [20], Agda [4, 10], Nuprl [16, 30, 7]) have a relatively small language of "proof terms". Typically, the core of these proof assistants consists of a proof checking machinery which ensures that these proof terms are indeed proofs of the corresponding statements. Proof assistants often also come with a large machinery for proof search (e.g. Coq, Nuprl, ACL2, PVS) [15]. Fortunately, in some of these (Coq, Nuprl), we do *not* have to trust the correctness of the proof search machinery as the generated proof terms have to pass the test of the core proof checking machinery. So, for the remainder of this document, we will only be concerned about the correctness of the core proof checker.

Since most of these provers are based on dependent type theory [32, 33], proof checking is essentially type checking and proof terms are members of the types corresponding to the respective statements. Type checking is usually based on collection of rules about what it means for a term to be in a type. By Gödel's second incompleteness theorem, it is impossible to prove even the consistency of these rules inside the system for these rich logics. Typically, these rules come from a metatheoretical model of what types mean to the designers of the proof assistant. In case of Coq, Nuprl, Agda and other constructive type theories, it is a computational model and the terms are drawn from a suitable extension of lambda calculus. In some cases, this model is made more precise by formally describing *parts* of it in peer-reviewed articles [44, 8, 25], or PhD theses [6, 21, 29], or even books [42]. However, this is not a satisfactory state of affairs because: 1) It is possible to overlook the inconsistencies between the different parts formalized on paper, and 2) Mistakes are possible in these large proofs (often spanning hundreds of pages) which are never done in full detail.

In our experience with the Nuprl proof assistant, we had at least once added an inconsistent rule to its experimental versions even after extensive discussions. Other Proof Assistants also have had close brushes with inconsistency. Bugs in Agda's typechecker that lead to proof(s) of False have been often found and confirmed[1]. Very recently, a bug was found in the termination analysis done by Coq's typechecker[2]. This bug led to a proof that adding the Propositional Extensionality axiom makes Coq inconsistent. However, consistency of Propositional Extensionality is a straightforward consequence of the proof-irrelevant semantics given to the Prop universe in [44]. Less that a month after this bug in Coq was reported, a similar bug was discovered in Agda by trying to make similar definitions[3].

Fortunately, the proof assistant technology has matured enough [31, 24] that we can consider formalizing proof assistants in themselves, or other proof assistants [9]. Most of these theories are stratified into cumulative universes and it is plausible that a universe can be formalized in a higher universe of the same theory. Alternatively, universes of two equally powerful theories can be interleaved so that a round-trip model will result in an increase in the universe level. The latter approach seems likely to catch more mistakes if their theory and implementation are not too correlated to each other. Moreover, this approach is likely to build bridges between user-bases of two proof assistants as they can look at a description of the other system in a language they understand.

1.2 Advantages of a Formalized Metatheory

There are many advantages having and maintaining a formalized meta-theory of a language, especially a proof assistant. The formalized meta-theory can then guide and accelerate innovation. Currently, developers of proof assistants are scared of even making a minor improvement to a proof assistant.

[1]https://lists.chalmers.se/pipermail/agda/2012/003700.html
[2]https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html
[3]https://lists.chalmers.se/pipermail/agda/2014/006252.html

There are many questions to be answered. Is the change going to make the theory inconsistent? Is the change going to break existing developments? If so, is there a way to automatically transform developments of the old theory to the new one while preserving the semantics. The formal meta-theory can be used to confidently answer these questions. Moreover, we would no longer have to sacrifice efficiency for simplicity.

Finally, if the formalization is done as a deep embedding, it would be easier to automatically translate the developments of one theory to another. There is a huge wealth of mechanized mathematics that one cannot use because the mechanization was done in some other theory.

1.3 Formalization of Nuprl in Coq

In this work, we formalize large part of Nuprl in Coq. The two theories appear to be quite different. While Coq is based on an intensional type theory with intrinsic typing, Nuprl is based on an extensional one with extrinsic typing. Also, Nuprl has several expressive types not found in Coq : e.g., quotient types, refinement types, intersection types[29], dependent intersection types [29], union types[29], partial types[17, 41, 18, 21], image types [37]. Partial types enable principled reasoning about partial functions (functions that may not converge on all inputs). This is better than the approach of Agda[4] and Idris [11] where one can disable termination checking and allow oneself to accidentally prove a contradiction.

Following Allen [6] we implement W types (which can be used to encode inductive types [2] in an extensional type theory like Nuprl) instead of Mendler's recursive types, which makes the entire Nuprl metatheory predicatively justifiable. Our Agda model in Sec. 4.1 illustrates this fact. Mendler's recursive types can construct the least fixpoint of an arbitrary monotone endofunction (not necessarily strictly positive) in a Nuprl universe. Formalizing this notion seems to require an impredicative metatheory.

1.4 Organization of this document

This document is organized into several chapters. The contents of each chapter depends on the contents of the previous chapters, as does the code shown

[4]http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.AgdaVsCoq

in each chapter. We almost always present Coq code directly in this document[5]. It is often syntactically colored using the default convention: blue is used for inductive types, dark red for constructors of inductive types, green for defined constants, functions and lemmas, red for keywords and some notations, and purple for (Coq/Agda) variables. Some of the colored items are hyperlinked to the place they are defined, either in this document, in the standard library, or in our publicly available code[6].

Chapter 2 is mainly concerned about the syntax of Nuprl terms. We first show our inductive type NTerm which uniformly represents Nuprl's language. It is parametrized by a collection of operators Opid and makes it possible to add new constructs without changing the core definitions. We show the definitions of our simultaneous substitution lsubst ,alpha equality and statements of several properties that we proved about these. We explain why these proof apply to any instantiation of the type Opid in which equality is decidable.

Chapter 3 explains our formalization of Nuprl's computation system and its properties. The key definition here is a one step computation function compute_step. We then explain our formalization of the coinductive computational approximation relation(approx) that was defined and proved to be a congruence by Howe [26]. We formalize this proof as well as the domain theoretic properties that were used by Crary to justify some typehood rules about partial types [21]. We then define a computational equivalence relation cequiv which plays a key role in the definition of our type system.

Chapter 4 explains the formalization of Nuprl's type system(nuprl). Following Allen's methodology, a Nuprl type is defined as a partial equivalence relation(PER). nuprl determines which closed terms denote equal types and which closed terms are equal members of types. This chapter also shows that using induction-recursion in Agda results in a more intuitive and simple definition of Nuprl's type system. In total, this chapter discusses three different meta-theories that one can use to formalize Nuprl's type system and the pros and cons of each one. Figure 1.1 gives an overview.

Finally, in chapter 4 we define the syntax and semantics of Nuprl's sequents and prove the validity of several inference rules in section 5.2 in the above PER semantics. The above mentioned section is organized into var-

[5]A reader who is not familiar with Coq might want to read a short tutorial at `http://www.cis.upenn.edu/~bcpierce/sf/Basics.html`

[6]Due to some limitations of the coqdoc tool, some items are incorrectly shown in purple.

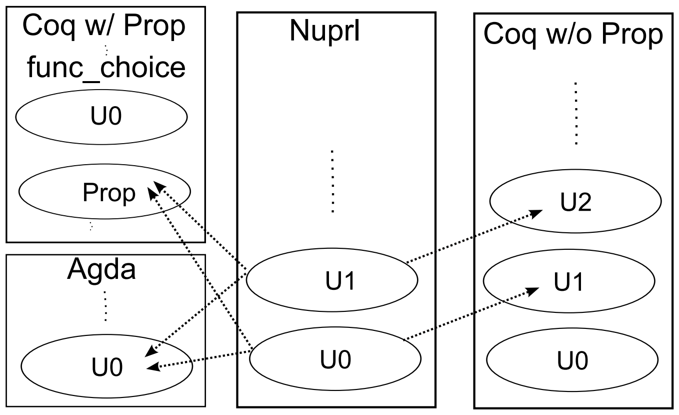

Figure 1.1: Three meta-theories that can formalize (parts of) Nuprl. An arrow from a Nuprl universe A to some universe B means that the PERs of types in A can be defined as relations in the universe B.

ious subsection, each describing to rules corresponding to a particular type constructor of Nuprl. 5.2.10 describes some generic rules. We conclude by showing how all these proved rules can be used to build an Ltac based "refiner" that can serve as a trusted core of Nuprl.

Chapter 2

Terms, Substitution, Alpha Equality

When doing the formalization, I discovered that the core part of the proof . . . is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening

Thorsten Altenkirch in Proceedings of TLCA, 1993

2.1 Preliminaries

For a more concrete understanding of the meaning of Nuprl's constructs, we chose to formalize Nuprl in the predicative part of Coq. This is a significantly weaker meta-theory than the one we get by adding the impredicative universe (`Prop`) [19, 12]. Since most of the Coq's library about proofs lives in

the impredicative world, we had to redefine many of the logical connectives and standard relations on numbers and lists in the `Type` (predicative) world. Infact, we use a notation `[univ]` which currently stands for `Type`. However, most of our development compiles even when `[univ]` is redefined as `Prop`. Details can be found in following files:
————-begin file universe.v ————
————-begin file eq_rel.v ————
————-begin file UsefulTypes.v ————
————-begin file list.v ————

2.1.1 Variables

————-begin file variables.v ————

We define our variables exactly as that in Software Foundations [1] . Basically, variables are wrappers around numbers. We could have defined them as wrappers around any (countably) infinite type with decidable equality.

```
Inductive NVar : Set :=
| nvar : nat → NVar.
```

Here are some examples of variables.

```
Definition nvarx := nvar 0.
Definition nvary := nvar 1.
Definition nvarz := nvar 2.
```

We need decidable equality on variables so that many useful relations on our future definitions of Terms, like equality, alpha equality, etc. will be decidable. Also our substitution function will need this decidable equality to decide whether it needs to come up with fresh variables. Decidable equality on variables is a straightforward consequence of decidable equality on the underlying type.

```
Theorem eq_var_dec: ∀ x y : NVar, {x = y} + {x ≠ y}.
```

Another key requirement for a sensible formalization of variables is to have an unbounded supply of fresh variables. Hence, we prove the following lemma. The notation `{_:_× _}` denotes sigma types (sigT) To those who are unfamiliar with constructive logic, the following lemma might just say that

[1] http://www.cis.upenn.edu/~bcpierce/sf/

that for any *n* and *lv*, there exists a list *lvn* of length *n* of distinct variables such that the members of *lvn* are disjoint from the members of *lv*.

However, because of the propositions as types principle[2], We are actually defining a function fresh_vars that takes a number *n* and a list *lv* of variables and returns a 4-tuple. The first member of that tuple is *lvn*, a list of variables with the above mentioned properties. The next three members are proofs of the desired properties of *lvn*.

An important use *free_vars* of it will be in defining the substitution function.

```
Lemma fresh_vars :
  ∀ (n: nat) (lv : list NVar),
    {lvn : (list NVar) × no_repeats lvn × disjoint lvn lv × length lvn = n}.
```

2.2 Uniform Term Structure

———-begin file terms.v ————

We can now define the terms of the Nuprl language as an inductive type. There are several considerations in choosing the right definition. The definition needs to be general enough so that adding new constructs to the term language does not break proofs about general properties of general operations on terms. For example, the substitution operation and the alpha equality relation only care about the getting access to the variables and do not care about the other operators(constucts) of the language.

Our term definition(similar to [26]) exposes the variables, especially the concept of bound variables in a uniform way so that these and many other operations and proofs work unchanged when adding/removing constructs from the language. These robust proofs run into several thousands of lines and include the many properties about substitution and alpha equality that we need for formalizing all of Nurpl.

Many alternative approaches for variable bindings have been discussed in the literature [38, 14, 15]. Our choice avoided the overhead of translating the paper definitions about Nuprl to some other style of variable bindings.

We will first intuitively explain parts of the definition before showing it. Firstly, we have a constructor (vterm) that builds a term(NTerm) from a variable(NVar)). Variable bindings are made explicit by the concept of a

[2]we are not using Prop.

bound term (BTerm). bterm is the only constructor of BTerm. It takes a list of variables (say *lv*) and a term (say *nt*) and constructs a bound term. Intuitively, a variable that is free in *nt* gets bound to its first occurence in *lv*, if any. For example, the bound term bterm [*nvarx*] (vterm *nvarx*) will be used soon in constructing an identity function($\lambda x.x$).

The rest of our term definition is parametrized by a collection of operators(Opid). Operators take bound terms as input and construct another term. For example, there is an operator that takes [bterm [*nvarx*] (vterm *nvarx*)] and constructs the lambda term $\lambda x.x$. With that in mind, here is the inductive type(NTerm) that represents the terms of Nurpl:

```
Inductive NTerm : Set :=
| vterm: NVar → NTerm
| oterm: Opid → list BTerm → NTerm
with BTerm : Set :=
| bterm: (list NVar) → NTerm → BTerm.
```

It is a mutually inductive definition that simultaneously defines terms and bound terms. As mentioned before, the vterm constructor takes an NVar and constructs an NTerm. The other constructor(oterm) takes an Opid and a list of bound terms (BTerms) and constructs an NTerm. Note that not all members of NTerm are meaningful(well-formed). For example, the Opid corresponding to lambda abstractions must be provided with exactly one bound term as argument. Moreover, that bound term must have exactly one bound variable. So, we have a function OpBindings in type Opid → list *nat* that specifies both the number of arguments and the number of bound variables in each argument(BTerm). We will use it soon to define the subcollection of well-formed terms.

Whenever we talk about the NTerm of a BTerm, this is what we would mean:

```
Definition get_nt (bt: BTerm ) : NTerm :=
 match bt with
 | bterm lv nt ⇒ nt
 end.

Definition get_vars (bt: BTerm ) : list NVar :=
 match bt with
 | bterm lv nt ⇒ lv
 end.

Definition num_bvars (bt : BTerm) := length (get_vars bt).
```

We define a predicate nt_wf on NTerm such that nt_wf *nt* asserts that *nt* is a well-formed term.

```
Inductive nt_wf: NTerm → [univ] :=
| wfvt: ∀ nv : NVar, nt_wf (vterm nv)
| wfot: ∀ (o: Opid) (lnt: list BTerm),
          (∀ l, LIn l lnt → bt_wf l)
           → map (num_bvars) lnt
              = OpBindings o
           → nt_wf (oterm o lnt)
with bt_wf : BTerm → [univ] :=
| wfbt : ∀ (lnv : list NVar) (nt: NTerm),
           nt_wf nt → bt_wf (bterm lnv nt).
```

The only interesting case here is for the oterm case. The wfot constructor requires that the number of bound variables of the bound terms in the list match the signature (OpBindings *o*) of the corresponding operator *o*.

We abstract the Opids into two categories, canonical and noncanonical.

```
    Inductive Opid : Set :=
    | Can : CanonicalOp → Opid
    | NCan : NonCanonicalOp → Opid.
```

This distinction is important from the point of view of computation and simplifies many definitions and properties about computation and also makes them more easily extensible. Nuprl has a lazy computation system and an NTerm is in normal(canonical) form if its outermost Opid is a *CanonicalOp*. No further computation is performed on terms in canonical form. For example, lambda abstraction are constructed by the following Opid :

Can *NLambda*

We have OpBindings (Can *NLambda*) = [1].

On the other hand, an NTerm whose outermost Opid is a *NonCanonicalOp* is not in normal form and can compute to some other term, or to an error. An an example, terms denoting function applications are constructed by the following Opid:

NCan *NApply*

We have OpBindings (NCan *NApply*) = [0,0].

The only restriction in defining *CanonicalOp* and *NonCanonicalOp* is that the equality in these types should be decidable. We will soon show the full-blown definition of the Opids of Nuprl. Before that, we define functions that compute the free variables and bound variables of a term. Note how these functions have just two cases and are robust against addition/deletion of new operators(Opids) to the language. If we had defined NTerm in the usual way(with one constructor for each Opid), these definitions would be of the form of a long pattern match with one case for each Opid. However, these definitions only care about the binding structure. We will reap more benefits of this uniformity when we define substitution and alpha equality in the next subsection.

```
Fixpoint free_vars (t:NTerm) : list NVar :=
  match t with
  | vterm v ⇒ [v]
  | oterm op bts ⇒ flat_map free_vars_bterm bts
  end
 with free_vars_bterm (bt : BTerm ) :=
  match bt with
  | bterm lv nt ⇒ remove_nvars lv (free_vars nt)
  end.

Fixpoint bound_vars (t : NTerm) : list NVar :=
  match t with
  | vterm v ⇒ []
  | oterm op bts ⇒ flat_map bound_vars_bterm bts
  end
 with bound_vars_bterm (bt : BTerm ) :=
  match bt with
  | bterm lv nt ⇒ lv ++ bound_vars nt
  end.

Definition all_vars t := free_vars t ++ bound_vars t.

Definition closed (t : NTerm) := free_vars t = [].
Definition isprogram (t : NTerm) := closed t × nt_wf t.
```

Now, we will describe the Opids of Nuprl and then describe some other useful definitions and lemmas about NTerm.

2.2.1 Opid's of Nuprl

———-begin file opid.v ————

Here are the Canonical Opids of Nuprl:

```
Inductive CanonicalOp : Set :=
```

Firstly, here are the ones that can be thought of data constructors. NInl, NInr, NPair Nint correspond to fairly standard data constructors in Martin Lof's theories. NAxiom is the unique canonical inhabitant logically fundamental types of Nuprl's type system that denote true propositions and for which no evidence is required. NSup is the canonical form representing members of the W types of Nuprl.

```
| NLambda : CanonicalOp
| NAxiom : CanonicalOp
| NInl : CanonicalOp
| NInr : CanonicalOp
| NPair : CanonicalOp
| NSup : CanonicalOp
| Nint : ℤ → CanonicalOp
| NTok : String.string → CanonicalOp
```

Like Martin Lof's theories, types are also first class terms and can be computed with. There is a CanonicalOp for each type constructor. Here are the CanonicalOps that correspond to type constructors. The meanings of most of these types will be formally defined in the Chapter 4 A few of these type-constructors are not currently in use (e.g. \coqdocconstructor{NRec}), but they are there for the possibility of future experimentation.

```
| NUni : nat → CanonicalOp
| NTUni : CanonicalOp
| NEquality : CanonicalOp
| NTEquality : CanonicalOp
| NInt : CanonicalOp
| NAtom : CanonicalOp
| NBase : CanonicalOp
| NFunction : CanonicalOp
| NProduct : CanonicalOp
| NSet : CanonicalOp
| NQuotient : CanonicalOp
| NIsect : CanonicalOp
| NDIsect : CanonicalOp
| NEIsect : CanonicalOp
| NW : CanonicalOp
| NM : CanonicalOp
| NPW : CanonicalOp
| NPM : CanonicalOp
| NEPertype : CanonicalOp
| NIPertype : CanonicalOp
| NSPertype : CanonicalOp
| NPartial : CanonicalOp
| NUnion : CanonicalOp
| NApprox : CanonicalOp
| NCequiv : CanonicalOp
| NCompute : CanonicalOp
| NRec : CanonicalOp
| NImage : CanonicalOp
| NAdmiss : CanonicalOp
| NMono : CanonicalOp.
```

Now we define the binding structure for each CanonicalOp. Recall that the the length of OpBindingsCan c represents the number of arguments($BTerms$) required to form an $NTerm$ using this the CanonicalOp c. The i^{th} element OpBindingsCan c represents the number of bound variables in the i^{th} argument.

```
Definition OpBindingsCan (c : CanonicalOp) : list nat :=
  match c with
  | NLambda ⇒ [1]
  | NAxiom ⇒ []
  | NInl ⇒ [0]
  | NInr ⇒ [0]
  | NPair ⇒ [0,0]
  | NSup ⇒ [0,0]
  | Nint _ ⇒ []
  | NUni _ ⇒ []
  | NTUni ⇒ [0]
  | NTok _ ⇒ []
  | NEquality ⇒ [0,0,0]
  | NTEquality ⇒ [0,0]
  | NInt ⇒ []
  | NBase ⇒ []
  | NAtom ⇒ []
  | NFunction ⇒ [0,1]
  | NProduct ⇒ [0,1]
  | NSet ⇒ [0,1]
  | NIsect ⇒ [0,1]
  | NDIsect ⇒ [0,1]
  | NEIsect ⇒ [0,1]
  | NW ⇒ [0,1]
  | NM ⇒ [0,1]
  | NPW ⇒ [0,1,2,3,0]
  | NPM ⇒ [0,1,2,3,0]
  | NEPertype ⇒ [0]
  | NIPertype ⇒ [0]
  | NSPertype ⇒ [0]
  | NPartial ⇒ [0]
  | NUnion ⇒ [0,0]
  | NApprox ⇒ [0,0]
  | NCequiv ⇒ [0,0]
  | NCompute ⇒ [0,0,0]
  | NRec ⇒ [1]
  | NImage ⇒ [0,0]
  | NQuotient ⇒ [0,2]
  | NAdmiss ⇒ [0]
  | NMono ⇒ [0]
  end.
```

Now, we will define non-canonical Opids of Nuprl. Intuitively, these represent the elimination forms corresponding to some of the canonical terms of Nuprl. For example, NApply represents the elimination form for the CanonicalOp NLambda. Notably, there are no elimination forms for the CanonicalOps that represent type constructors. We also have some arithmetic and comparison operators on numbers. We will define 3 more parameters for defining the type NonCanonicalOp.

```
Inductive ArithOp : Set :=
 | ArithOpAdd : ArithOp
 | ArithOpMul : ArithOp
 | ArithOpSub : ArithOp
 | ArithOpDiv : ArithOp
 | ArithOpRem : ArithOp.
Inductive ComparisonOp : Set :=
 | CompOpLess : ComparisonOp
 | CompOpInteq : ComparisonOp
 | CompOpAtomeq : ComparisonOp.
```

The following type parametrizes some NonCanonicalOps that test the head normal form of a term.

```
Inductive CanonicalTest : Set :=
 | CanIspair : CanonicalTest
 | CanIssup : CanonicalTest
 | CanIsaxiom : CanonicalTest
 | CanIslambda : CanonicalTest
 | CanIsint : CanonicalTest
 | CanIsinl : CanonicalTest
 | CanIsinr : CanonicalTest
 | CanIsatom : CanonicalTest.
```

Finally, here are the noncanonical Opids of Nuprl:

```
Inductive NonCanonicalOp : Set :=
 | NApply : NonCanonicalOp
 | NFix : NonCanonicalOp
 | NSpread : NonCanonicalOp
 | NDsup : NonCanonicalOp
 | NDecide : NonCanonicalOp
 | NCbv : NonCanonicalOp
 | NCompOp : ComparisonOp → NonCanonicalOp
 | NArithOp : ArithOp → NonCanonicalOp
 | NCanTest : CanonicalTest → NonCanonicalOp.
Definition OpBindingsNCan (nc : NonCanonicalOp) : list nat :=
  match nc with
   | NApply ⇒ [0,0]
   | NFix ⇒ [0]
   | NSpread ⇒ [0,2]
   | NDsup ⇒ [0,2]
   | NDecide ⇒ [0,1,1]
   | NCbv ⇒ [0,1]
   | NCompOp _ ⇒ [0,0,0,0]
   | NArithOp _ ⇒ [0,0]
   | NCanTest _ ⇒ [0,0,0]
   end.
```

NFix represents the fixpoint combinator. NSpread is the elimination form for NPair. The first argument of NSpread is supposed to be a *BTerm* with no bound variables such that the underlying *NTerm* converges to something with head Opid as NPair. The second argument is a *BTerm* with two bound variables. These variables indicate the positions where the two components of the pair will be substituted in. The binding structure of the other Non-CanonicalOps can be understood similarly. More details will be available later when we define the computation system in the next chapter.

NDecide is the elimination form for NInl and NInr. NCbv the call-by-value variant of application. It evaluates its first argument down to a value before substituting it into the second argument that is a bound term with one bound variable.

The following function defines the binding structure of any Opid. It was used in the definition of *nt_wf*.

```
Definition OpBindings (op : Opid) : list nat :=
 match op with
 | Can c ⇒ OpBindingsCan c
 | NCan nc ⇒ OpBindingsNCan nc
 end.
```

The only requirement for defining CanonicalOp and NonCanonicalOp is that equality must be decidable in these types. We prove the folowing lemma by straightforward case analysis.

```
Lemma canonical_dec : ∀ x y : CanonicalOp, {x = y} + {x ≠ y}.
Lemma opid_dec : ∀ x y : Opid, {x = y} + {x ≠ y}.
```

2.2.2 Other useful definitions

————-begin file terms2.v ————

Here are some handy definitions that will reduce the verbosity of some of our later definitions

```
Definition mk_lam (v : NVar) (b : NTerm) :=
  oterm (Can NLambda) [bterm [v] b].
Definition nobnd (f:NTerm) := bterm [] f.
Definition mk_apply (f a : NTerm) :=
  oterm (NCan NApply) [nobnd f , nobnd a].
```

We define similar abstractions for other Opids. This document does not show them. As mentioned before, one can click at the hyperlinked filename that is closest above to open a webpage that shows complete contents of this file.

```
Definition isprogram_bt (bt : BTerm) := closed_bt bt × bt_wf bt.
```

Our definition isprog below is is logically equivalent to isprogram, but unlike isprogram, it is easy to prove that for any t, all members(proofs) of isprog t are equal. An interested reader can look at the lemma UIP_dec from that standard library. As mentioned before, clicking on the lemma name in the previous sentence should open the corresponding webpage of the Coq standard library. Instead, one can also look at the lemma isprog_eq below and safely ignore these technicalites.

```
Definition isprog (t : NTerm) := (nullb (free_vars t) && wft t) = true.
Lemma isprog_eq :
  ∀ t, isprog t ⇔ isprogram t.
```

The CTerm type below is useful in compactly stating definitions that are only meaningful for closed terms. A CTerm is a pair of an NTerm t and a proof that t is closed. This CTerm type will be handy in compactly defining the Nuprl type system where types are defined as partial equivalence relations on closed terms.

```
Definition CTerm := { t : NTerm | isprog t }.
Definition get_cterm (t : CTerm) := let (a,_) := t in a.
```

We also define a type of terms that specifies what are the possible free variables of its inhabitants. A term is a (CVTerm vs) term if the set of its free variables is a subset of vs. This type is also useful to define the Nuprl type system. For example, to define a closed family of types such as a closed function type of the form $x{:}A \to B[z\backslash x]$, A has to be closed and the free variables of B can only be z.

```
Definition CVTerm (vs : list NVar) := { t : NTerm | isprog_vars vs t }.
```

Using the CVTerm and CTerm types we can define useful abstraction to build closed versions of the various terms of our computation system. For example, given a variable v and a term in CVTerm $[v]$, we can build a closed lambda abstraction. As an other example, given two closed terms, we can build a closed application term.

```
Definition mkc_lam (v : NVar) (b : CVTerm [v]) : CTerm :=
  let (t,x) := b in
  exist isprog (mk_lam v t) (isprog_lam v t x).

Definition mkc_apply (t1 t2 : CTerm) : CTerm :=
  let (a,x) := t1 in
  let (b,y) := t2 in
    exist isprog (mk_apply a b) (isprog_apply a b x y).
```

2.3 Substitution and Alpha Equality

———-begin file substitution.v ————

The goal of this section is to formalize the notion of simultaneous substitution(lsubst) and alpha equality *alpha_eq*. We needed many properties about substitution and alpha equality to formalize all of Nuprl. Proofs of all these properties run into several thousands of lines and took us several weeks to finish. These proofs are independent of the operators of the language and will work unchanged even if we formalize some different language, e.g. first order logic by merely changing the definition of *Opid*. Thus, we believe that we have accidentally created a fairly general-purpose library for nominal reasoning about virtually any language.

2.3.1 Substitution

The Substitution operation is a key concept in functional languages. Among other things, it is required to define the computation system and the semantics of dependent types. The goal of this subsection is to formalize lsubst, a function that simultaneously substitutes some variables for some terms.

We define a Substitution as a list of pairs:

```
Definition Substitution : Set := list (NVar × NTerm).
```

The function var_ren below provides a way to define the specialized substitutions that are variable renamings (substituting one variable for another). The combine function from the standard library takes two lists and zips them up.

```
Definition var_ren (lvo lvn : list NVar) : Substitution :=
  combine lvo (map vterm lvn).
```

The domain and range of a substitution are defined as follows:

```
Definition dom_sub (sub : Substitution) : list NVar := map (fun x ⇒ fst
x) sub.

Definition range (sub : Substitution) : list NTerm := map (fun x ⇒ snd x) sub.
```

We need to define some helper functions before defining the substitution function that simultaneously substitutes the in the first component(NVars) of the pairs with the second ones(NTerms).

```
Fixpoint sub_find (sub : Substitution) (var : NVar) : option NTerm :=
  match sub with
  | nil ⇒ None
  | (v, t) :: xs ⇒ if beq_var v var then Some t else sub_find xs var
  end.
```

We say that a term *t* is covered by a list of variables *vs* if the free variables of *t* are all in *vs*.

```
Definition covered (t : NTerm) (vs : list NVar) :=
  subvars (free_vars t) vs.

Definition over_vars (vs : list NVar) (sub : CSubstitution) :=
  subvars vs (dom_csub sub).
```

A term *t* is covered by a substitution *sub* if the free variables of *t* are all in the domain of *sub*.

```
Definition cover_vars (t : NTerm) (sub : CSubstitution) :=
  over_vars (free_vars t) sub.
```

We sometimes need the slightly more general definition that expresses that a term *t* is covered by a substitution *sub* up to a set of variables *vs*, meaning that the free variables of *t* have to either be in *vs* or in the domain of *sub*. Such a concept is needed to deal with type families such as function or W types.

```
Definition cover_vars_upto (t : NTerm) (sub : CSub) (vs : list NVar) :=
  subvars (free_vars t) (vs ++ dom_csub sub).

Fixpoint sub_filter (sub : Substitution) (vars : list NVar) : Substitution :=
  match sub with
  | nil ⇒ nil
  | (v, t) :: xs ⇒
      if memvar v vars
      then sub_filter xs vars
      else (v, t) :: sub_filter xs vars
  end.
```

The following function is an auxilliary one that performs a Substitution on an NTerm assuming that its bound variables of are already disjoint from the free variables of the range of the Substitution.

```
Fixpoint lsubst_aux (nt : NTerm) (sub : Substitution) : NTerm :=
  match nt with
  | vterm var ⇒
      match sub_find sub var with
      | Some t ⇒ t
      | None ⇒ nt
      end
  | oterm op bts ⇒ oterm op (map (fun t ⇒ lsubst_bterm_aux t sub) bts)
  end
 with lsubst_bterm_aux (bt : BTerm) (sub : Substitution) : BTerm :=
  match bt with
  | bterm lv nt ⇒ bterm lv (lsubst_aux nt (sub_filter sub lv))
  end.
```

To define the actual substitution function, we just have to pre-process t such that its bound variables have been renamed to avoid the free variables of the range of sub. Here is a function that does that.

```
Fixpoint change_bvars_alpha (lv : list NVar ) (t : NTerm) :=
match t with
| vterm v ⇒ vterm v
| oterm o lbt ⇒ oterm o (map (change_bvars_alphabt lv) lbt)
end
with change_bvars_alphabt lv bt:=
match bt with
```

```
| bterm blv bnt ⇒
    let bnt' := change_bvars_alpha lv bnt in
    match (fresh_vars (length blv) (lv++(all_vars bnt'))) with
    | existT lvn _ ⇒ bterm lvn (lsubst_aux bnt' (var_ren blv lvn))
    end
end.
```

When we define alpha equality in the next section, we prove that change_bvars_alpha returns a term which is alpha equal to the input. Finally, here is the function that safely perfoms a Substitution on an NTerm.

```
Definition lsubst (t : NTerm) (sub : Substitution) : NTerm :=
  let sfr := flat_map free_vars (range sub) in
    if dec_disjointv (bound_vars t) sfr
    then lsubst_aux t sub
    else lsubst_aux (change_bvars_alpha sfr t) sub.
```

The following definition will be useful while defining the computation system.

```
Definition apply_bterm (bt :BTerm) (lnt: list NTerm) : NTerm :=
  lsubst (get_nt bt) (combine (get_vars bt) lnt).
```

The formalization of Nuprl requires many lemmas about lsubst. Because lsubst often renames bound variables, we need alpha equality to state many useful properties of substitution. We will first define alpha equality and then list some useful properties that we proved about lsubst.

2.3.2 Alpha Equality

————-begin file alphaeq.v ————

Most of the operations and relations in functional languages are invariant under renaming of variables. Alpha equality helps us express this property. We define it as follows:

```
Inductive alpha_eq : NTerm → NTerm → Type :=
  | al_vterm : ∀ v, alpha_eq (vterm v) (vterm v)
  | al_oterm : ∀ (op: Opid) (lbt1 lbt2 : list BTerm),
          length lbt1 = length lbt2
          → (∀ n, n < length lbt1
                  → alpha_eq_bterm (selectbt lbt1 n)
                                   (selectbt lbt2 n))
```

```
        → alpha_eq (oterm op lbt1) (oterm op lbt2)
with alpha_eq_bterm : BTerm → BTerm → Type :=
  | al_bterm :
      ∀ (lv1 lv2 lv: list NVar) (nt1 nt2 : NTerm) ,
        disjoint lv (all_vars nt1 ++ all_vars nt2)
        → length lv1 = length lv2
        → length lv1 = length lv
        → no_repeats lv
        → alpha_eq (lsubst nt1 (var_ren lv1 lv))
                     (lsubst nt2 (var_ren lv2 lv))
        → alpha_eq_bterm (bterm lv1 nt1) (bterm lv2 nt2).
```

The interesting case is in the definition of alpha_eq_bterm. We list some useful properties about lsubst and alpha_eq. Just like the above definitions, these proofs are independent of the operators of the language.

2.3.3 Key Properties about Substitution and Alpha Equality

```
Lemma alpha_eq_refl:
  ∀ nt, alpha_eq nt nt.

Lemma alpha_eq_sym:
  ∀ nt1 nt2, alpha_eq nt1 nt2 → alpha_eq nt2 nt1.
Lemma alpha_eq_trans:
  ∀ nt1 nt2 nt3, alpha_eq nt1 nt2 → alpha_eq nt2 nt3 → alpha_eq nt1 nt3.
Lemma alphaeq_preserves_wf:
  ∀ t1 t2, alpha_eq t1 t2 → (nt_wf t2 ⇔ nt_wf t1).
```

The following lemma is the property we promised to prove while definining lsubst.

```
Lemma change_bvars_alpha_spec: ∀ nt lv,
  let nt' := change_bvars_alpha lv nt in
  disjoint lv (bound_vars nt') ×alpha_eq nt nt'.
Theorem alphaeq_preserves_free_vars:
  ∀ t1 t2, alpha_eq t1 t2 →
      (free_vars t1) = (free_vars t2).
Lemma lsubst_wf_iff:
  ∀ sub,
  wf_sub sub
  → ∀ nt, (nt_wf nt ⇔ nt_wf (lsubst nt sub)).
```

In the following theorem, the binary relation (bin_rel_nterm alpha_eq) on list NTerm asserts that the lists have equal length and the members are respectively alpha equal.

```
Theorem lsubst_alpha_congr: ∀ t1 t2 lvi lnt1 lnt2,
  alpha_eq t1 t2
  → length lvi = length lnt1
  → length lvi = length lnt2
  → bin_rel_nterm alpha_eq lnt1 lnt2
  → alpha_eq (lsubst t1 (combine lvi lnt1)) (lsubst t2 (combine lvi lnt2)).

Corollary lsubst_alpha_congr2: ∀ t1 t2 sub,
  alpha_eq t1 t2
  → alpha_eq (lsubst t1 sub) (lsubst t2 sub).
Theorem apply_bterm_alpha_congr: ∀ bt1 bt2 lnt1 lnt2,
  alpha_eq_bterm bt1 bt2
  → bin_rel_nterm alpha_eq lnt1 lnt2
  → length lnt1 = num_bvars bt1
  → alpha_eq (apply_bterm bt1 lnt1) (apply_bterm bt2 lnt2).

Corollary apply_bterm_alpha_congr2: ∀ bt1 bt2 lnt,
  alpha_eq_bterm bt1 bt2
  → length lnt = num_bvars bt1
  → alpha_eq (apply_bterm bt1 lnt) (apply_bterm bt2 lnt).
```

The following lemma characterizes the free variables of the result of a substitution as a set. eqvars is a binary relation on list NVar that asserts that the 2 lists are equal as sets. sub_keep_first *sub* *lv* filters the Substitution *sub* so as to keep only the first occurence of pairs whose domain is in *lv*.

```
Lemma eqvars_free_vars_disjoint :
  ∀ t : NTerm,
  ∀ sub : Substitution,
    eqvars (free_vars (lsubst t sub))
                  (remove_nvars (dom_sub sub) (free_vars t)
                   ++ sub_free_vars (sub_keep_first sub (free_vars t))).
Lemma lsubst_sub_filter_alpha:
  ∀ (t : NTerm) (sub : Substitution) (l : list NVar),
  disjoint (free_vars t) l
  → alpha_eq (lsubst t (sub_filter sub l)) (lsubst t sub).
```

In the following lemma, lsubst_sub is a function such that lsubst_sub *sub1* *sub2* performs the Substitution sub2 on each element of the range of *sub1*.

```
Lemma combine_sub_nest:
  ∀ t sub1 sub2,
    alpha_eq (lsubst (lsubst t sub1) sub2)
                 (lsubst t (lsubst_sub sub1 sub2 ++ sub2)).

Lemma lsubst_nest_same_alpha :
  ∀ t lvi lvio lnt,
  length lvio=length lvi
  → length lvio=length lnt
  → no_repeats lvio
  → disjoint lvio (free_vars t)
  → alpha_eq (lsubst (lsubst t (var_ren lvi lvio)) (combine lvio lnt))
       (lsubst t (combine lvi lnt)).
Lemma lsubst_nest_swap_alpha: ∀ t sub1 sub2,
  let lvi1 := dom_sub sub1 in
  let lvi2 := dom_sub sub2 in
  let lnt1 := range sub1 in
  let lnt2 := range sub2 in
  disjoint lvi1 (flat_map free_vars lnt2)
  → disjoint lvi2 (flat_map free_vars lnt1)
  → disjoint lvi1 lvi2
  → alpha_eq (lsubst(lsubst t sub1) sub2) (lsubst(lsubst t sub2) sub1).
```

Chapter 3

Computation System and Computational Approximation

3.1 Computation System of Nuprl

———-begin file computation.v ———

Here, we define the lazy computation system of Nuprl. Since it is a deterministic one, we can define it as a (partial)function. The computation system of Nuprl is usually specified as a set of one step reduction rules. So, our main goal here is to define the compute_step function. Since a step of computation may result in an error, we first lift NTerm to add more information about the result of one step of computation :

```
Inductive Comput_Result : Set :=
| csuccess : NTerm → Comput_Result
| cfailure : String.string→ NTerm → Comput_Result.
```

For a *NonCanonicalOp*, we say that some of its arguments are principal. Principal arguments are subterms that have to be evaluated to a canonical form before checking whether the term itself is a redex or not. For example, Nuprl's evaluator evaluates the first argument of NApply until it converges (to a λ term). The first argument is principal for all *NonCanonicalOp*s. For NCompOp and NArithOp, even the second argument is canonical.

We present the one step computation function in the next page. Although it refers to many other definitions, we describe it first so that the overall structure of the computation is clear. It is defined by pattern matching on the supplied NTerm. If it was a variable, it results in an error. If it was an oterm with a canonical *Opid*, there is nothing more to be done. Remember that it is a lazy computation system and a term whose outermost *Opid* is canonical is already in normal form.

If the head *Opid ncr* is non-canonical, it does more pattern matching to futher expose the structure of its *BTerm*s. In particular, the first *BTerm* is principal and should have no bound variables and its NTerm should be an oterm (and not a variable). If the *Opid* of that oterm is canonical, some interesting action can be taken. So, there is another (nested) pattern match that invokes the appropriate function, depending on the head *Opid* (*ncr*).

However, if it was non-canonical, the last clause in the top-level pattern match gets invoked and it recursively evaluates the first argument *arg1nt* and (if the recursive calls succeeds,) repackages the result at the same position with the surroundings unchanged.

In the nested pattern match, all but the cases for NCompOp and NArithOp do not need to invoke recursive calls of compute_step. However, in the cases of NCompOp and NArithOp, the first two(instead of one) arguments are principal and need to be evaluated first. Hence, the notations co and ca also take compute_step as an argument.

```
Fixpoint compute_step (t : NTerm) : Comput_Result :=
match t with
  | vterm v ⇒ cfailure "not closed" (vterm v)
  | oterm (Can _) _ ⇒ csuccess t
  | oterm (NCan _) [] ⇒ cfailure "no args supplied" t
  | oterm (NCan _) ((bterm [] (vterm _))::_) ⇒ cfailure "not closed" t
  | oterm (NCan _) ((bterm (h::tl) _)::_) ⇒ cfailure "check 1st arg" t
  | oterm (NCan ncr) ((bterm [] ((oterm (Can arg1c) arg1bts)) as arg1)::btsr) ⇒
      match ncr with
      | NApply ⇒ compute_step_apply arg1c t arg1bts btsr
      | NFix ⇒ compute_step_fix arg1c t arg1bts btsr
      | NSpread ⇒ compute_step_spread arg1c t arg1bts btsr
      | NDsup ⇒ compute_step_dsup arg1c t arg1bts btsr
      | NDecide ⇒ compute_step_decide arg1c t arg1bts btsr
      | NCbv ⇒ compute_step_cbv arg1c t arg1bts btsr
      | NCompOp op ⇒ co btsr t arg1bts arg1c op compute_step arg1 ncr
      | NArithOp op ⇒ ca btsr t arg1bts arg1c op compute_step arg1 ncr
      | NCanTest top ⇒ compute_step_can_test top arg1c t arg1bts btsr
      end

  | oterm (NCan ncr) ((bterm [] arg1nt)::btsr) ⇒
      match (compute_step arg1nt) with
      | csuccess f ⇒ csuccess (oterm (NCan ncr) ((bterm [] f)::btsr))
      | cfailure str ts ⇒ cfailure str ts
      end
end.
```

———-begin file computation1.v ————

Now we define the functions – compute_step_apply, compute_step_fix, etc. – that perform a computation step on each of the noncanonicals forms. We preface each with a reminder of the operator's binding structure and the structure of t, the original term we were computing with when the function was called.

These definitions are used above in the pattern matching(on ncr) that is nested inside in the definition of *compute_step* above. So, while defining these functions, we know that the first argument of the head *Opid* of the original term is already canonical.

3.1.1 *NApply*

We begin with *NApply*.
OpBindings (NCan *NApply*) = [0,0]
t = oterm (NCan *NApply*) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_apply (arg1c:CanonicalOp) (t:NTerm)
                                      (arg1bts btsr: list BTerm) :=
match arg1c with
| NLambda ⇒
    match (arg1bts, btsr) with
    | ([bterm [v] b], [bterm [] arg])
        ⇒ csuccess (apply_bterm (bterm [v] b) [arg])
    | _ ⇒ cfailure "inappropriate args ro apply" t
    end
| _ ⇒ cfailure "bad first arg" t
end.
```

3.1.2 *NFix*

NFix that behaves very much like the Y combinator. Note that because of the way *compute_step* was defined, the following compute_step_fix gets invoked only when the only argument to *NFix* is in canonical form. This informally means that $fix(f)$ reduces to $f\ (fixf)$ only when f is already in canonical (normal) form. Otherwise a step of computation will only evaluate f further. This is unlike the way Crary defined it in [21], where $fix(f)$ always reduces to $f\ (fixf)$ (even if f is not in canonical form). Our change was mainly motivated by the simplicity of *compute_step* definition when the first argument of all noncanonical operators is principal. Our change was mainly motivated by noting that the definition of *compute_step* is simpler if the first argument of all noncanonical operators is principal. One could also argue that our new definition avoids the duplication of computation that results from having to reduce f to canonical form after each unfolding of fix.

It turned out (later) that some Crary's proofs domain theoretic properties of fix critically dependend on $fix(f)$ always reducing to $f\ (fixf)$. With some non-trivial effort, we were able to restore these properties even for our new operational semantics of *NFix*. We will describe these new proofs later in this chapter.
OpBindings (NCan *NFix*) = [0]
t = oterm (NCan *NFix*) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_fix (arg1c:CanonicalOp) (t:NTerm)
                                        (arg1bts btsr: list BTerm) :=
match btsr with
  | [] ⇒ csuccess (mk_apply ((oterm (Can arg1c) arg1bts)) t)
  | _ ⇒ cfailure "inappropriate args to fix " t
end.
```

3.1.3 *NSpread*

NSpread is the elimination form for NPair.
OpBindings (NCan *NSpread*) = [0,2]
t = oterm (NCan *NSpread*) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_spread (arg1c:CanonicalOp) (t:NTerm)
                                        (arg1bts btsr: list BTerm) :=
match arg1c with
| NPair ⇒
      match (arg1bts, btsr) with
        | ([bterm [] a, bterm [] b], [bterm [va,vb] t]) ⇒
          csuccess (apply_bterm (bterm [va,vb] t) [a,b])
        | _ ⇒ cfailure "inappropriate args to spread" t
      end
| _ ⇒ cfailure "bad first arg to spread" t
end.
```

3.1.4 *NDsup*

NDsup behaves exactly like *NSpread* on NSup which is exactly like NPair.
OpBindings (NCan *NDsup*) = [0,2]
t = oterm (NCan *NDsup*) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_dsup (arg1c:CanonicalOp) (t:NTerm)
                                        (arg1bts btsr: list BTerm) :=
match arg1c with
  | NSup ⇒
    match (arg1bts, btsr) with
      | ([bterm [] a, bterm [] b], [bterm [va,vb] t]) ⇒
        csuccess (apply_bterm (bterm [va,vb] t) [a,b])
      | _ ⇒ cfailure "inappropriate args to dsup" t
    end
  | _ ⇒ cfailure "bad first arg to dsup" t
```

```
end.
```

3.1.5 *NDecide*

NDecide is the elimination form for NInl and NInr.
OpBindings (NCan *NDecide*) = [0,1,1]
t = oterm (NCan *NDecide*) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_decide (arg1c:CanonicalOp) (t:NTerm)
                                (arg1bts btsr: list BTerm) :=
match arg1c with
| NInl ⇒
    match (arg1bts, btsr) with
    | ([bterm [] u] , [bterm [v1] t1, bterm [v2] t2])
        ⇒ csuccess (apply_bterm (bterm [v1] t1) [u])
    | _ ⇒ cfailure "bad argsto decide" t
    end
| NInr ⇒
    match (arg1bts, btsr) with
    | ([bterm [] u] , [bterm [v1] t1, bterm [v2] t2])
        ⇒ csuccess (apply_bterm (bterm [v2] t2) [u])
    | _ ⇒ cfailure "inappropriate args to decide for Inr" t
    end
| _ ⇒ cfailure "bad args to decide" t
end.
```

3.1.6 *NCbv*

NCbv that is the call-by-value form of application.
OpBindings (NCan *NCbv*) = [0,1]
t = oterm (NCan *NCbv*) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_cbv (arg1c:CanonicalOp) (t:NTerm)
                             (arg1bts btsr: list BTerm) :=
match btsr with
  | [bterm [vs] t]
      ⇒ csuccess (apply_bterm (bterm [vs] t)
                              [(oterm (Can arg1c) arg1bts)])
  | _ ⇒ cfailure "inappropriate args to cbv " t
end.
```

3.1.7 *NCanTest* _

NCanTest was recently introduced in [39]. We first define the following helper function:

```
Definition canonical_form_test_for (test : CanonicalTest) (op : CanonicalOp) :
bool :=
  match test, op with
    | CanIspair, NPair ⇒ true
    | CanIssup, NSup ⇒ true
    | CanIsaxiom, NAxiom ⇒ true
    | CanIslambda, NLam ⇒ true
    | CanIsint, Nint _ ⇒ true
    | CanIsinl, NInl ⇒ true
    | CanIsinr, NInr ⇒ true
    | CanIsatom, NTok _ ⇒ true
    | _, _ ⇒ false
  end.
```

OpBindings (NCan (*NCanTest* _)) = [0,0,0]
t = oterm (NCan (*NCanTest* _)) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Definition compute_step_can_test top (arg1c:CanonicalOp) (t:NTerm)
                                     (arg1bts btsr: list BTerm) :=
match btsr with
  | [bterm [] arg2nt, bterm [] arg3nt] ⇒
     csuccess (if canonical_form_test_for top arg1c
                then arg2nt
                else arg3nt)
  | _ ⇒ cfailure "bad args to can test " t
end.
```

3.1.8 *NArithOp* _ and *NCompOp* _

The remaining two *NonCanonicalOps* have the first two arguments as pricipal, instead of just the first one. Hence their definitions are a little more complicated as they might need to recursively evaluate their second argument if it is not canonical.

To avoid confusing the termination analysis for these recursive calls, we define these as notations (ca and co). As evident from their use in *compute_step*, the *cstep* argument in these notations represents the (recursive use of) *compute_step*.

The *NonCanonicalOp*s that represent arithmetic operations just call the corresponding coq functions and repackage the result into the corresponding Nuprl Nint. get_int_from_cop below will be used to extract the corresponding Coq numbers out terms that represent Nuprl numbers.

```
Definition get_int_from_cop (c: CanonicalOp) :=
match c with
| Nint n⇒ Some n
| _ ⇒ None
end.
```

get_arith_op returns the corresponding arithmetic operation of Coq.

```
Definition get_arith_op (a :ArithOp) : (Z→Z→Z) :=
match a with
| ArithOpAdd ⇒ Z.add
| ArithOpMul ⇒ Z.mul
| ArithOpSub ⇒ Z.sub
| ArithOpDiv ⇒ Z.div
| ArithOpRem ⇒ Z.rem
end.
```

Here is the function that does one step of computation for *NArithOp* assuming both the principal arguments are canonical
OpBindings (NCan *NArithOp* _) = [0,0]

```
t = oterm (NCan (NArithOp op)) ((bterm [] (oterm (Can arg1c) arg1bts))
                               ::((bterm [] (oterm (Can arg2c) arg2bts)
                               ::btsr)))
```

```
Definition compute_step_arith
            (op : ArithOp) (arg1c arg2c: CanonicalOp)
            (arg1bts arg2bts btsr : list BTerm) (t: NTerm) :=
match (arg1bts,arg2bts,btsr) with
| ([],[],[]) ⇒
    match (get_int_from_cop arg1c , get_int_from_cop arg2c) with
    | (Some n1 , Some n2)
        ⇒ csuccess (oterm (Can (Nint ((get_arith_op op) n1 n2))) [])
    | _ ⇒ cfailure "bad args to arith" t
    end
| _ ⇒ cfailure "bad args to arith" t
```

```
end.
```

Finally, here is the notation that uses the above function if the second argument is also canonical, and evaluates it for one step instead if it was non-canonical. In this notation, *cstep* refers to *compute_step*.

```
t = oterm (NCan (NArithOp op)) ((bterm [] (oterm (Can arg1c) arg1bts))::btsr)
```

```
Notation ca := (fun btsr t arg1bts arg1c op cstep arg1 ncr ⇒
match btsr with
| [] ⇒ cfailure "too few args to arith op" t
| (bterm _ (vterm _)::_) ⇒
      cfailure "cop: not a closed term" t
| (bterm [] (oterm (Can arg2c) arg2bts)::btsr3) ⇒
      compute_step_arith op arg1c arg2c arg1bts arg2bts btsr3 t
| bterm [] (oterm (NCan _) _ as arg2nt) :: btsr3 ⇒
      match (cstep arg2nt) with
      | csuccess f ⇒ csuccess (oterm (NCan ncr)
                              (arg1::(bterm [] f)::btsr3))
      | cfailure str ts ⇒ cfailure str ts
      end
| (bterm _ (oterm _ _)::_) ⇒
      cfailure "cop: malformed 2nd arg" t
end).
```

Finally, we define the case for *NCompOp* in a similar way as *NArithOp*.

```
Definition get_str_from_cop (c: CanonicalOp) :=
 match c with
 | NTok s⇒ Some s
 | _ ⇒ None
 end.
```

```
OpBindings (NCan NCompOp _) = [0,0,0,0]
```

```
t = oterm (NCan (NCompOp op)) ((bterm [] (oterm (Can arg1c) arg1bts))
                                ::((bterm [] (oterm (Can arg2c) arg2bts)
                                ::btsr)))
```

```
Definition compute_step_comp
           (op : ComparisonOp) (arg1c arg2c: CanonicalOp)
           (arg1bts arg2bts btsr : list BTerm) (t: NTerm) :=
match (arg1bts,arg2bts,btsr) with
```

```
| ([],[], [bterm [] arg3nt, bterm [] arg4nt]) ⇒
      match op with
      | CompOpLess ⇒
              match (get_int_from_cop arg1c, get_int_from_cop arg2c) with
              | (Some n1, Some n2)
                   ⇒ csuccess (if (Z.leb n1 n2) then arg3nt else arg4nt)
              | _ ⇒ cfailure "bad args" t
              end
      | CompOpInteq ⇒
              match (get_int_from_cop arg1c, get_int_from_cop arg2c) with
              | (Some n1, Some n2)
                   ⇒ csuccess (if (Z.leb n1 n2) then arg3nt else arg4nt)
              | _ ⇒ cfailure "bad args" t
              end
      | CompAtomeq ⇒
              match (get_str_from_cop arg1c, get_str_from_cop arg2c) with
              | (Some n1, Some n2)
                   ⇒ csuccess (if (String.string_dec n1 n2)
                                   then arg3nt
                                   else arg4nt)
              | _ ⇒ cfailure "bad args" t
              end
      end
| _ ⇒ cfailure "bad args" t
end.
```

$t = $ oterm (NCan (*NCompOp op*)) ((bterm [] (oterm (Can *arg1c*) *arg1bts*))::*btsr*)

```
Notation co := (fun btsr t arg1bts arg1c op cstep arg1 ncr⇒
match btsr with
  | [] ⇒ cfailure "too few args to comparison op" t
  | (bterm _ (vterm _)::_) ⇒
        cfailure "cop: not a closed term" t
  | (bterm [] (oterm (Can arg2c) arg2bts)::btsr3) ⇒
        compute_step_comp op arg1c arg2c arg1bts arg2bts btsr3 t
  | bterm [] (oterm (NCan _) _ as arg2nt) :: btsr3 ⇒
        match (cstep arg2nt) with
        | csuccess f ⇒ csuccess (oterm (NCan ncr)
                                 (arg1::(bterm [] f)::btsr3))
        | cfailure str ts ⇒ cfailure str ts
        end
  | (bterm _ (oterm _ _)::_) ⇒
        cfailure "cop: malformed 2nd arg" t
end).
```

———-begin file computation3.v ———

Now, we will extend the one step computation to multiple steps in a straighforward way.

```
Fixpoint compute_at_most_k_steps
      (k : nat) (t : NTerm) : Comput_Result :=
  match k with
  | 0 ⇒ csuccess t
  | S n ⇒ match compute_at_most_k_steps n t with
            | csuccess tn ⇒ compute_step tn
            | cfailure mesg subterm ⇒ cfailure mesg subterm
            end
  end.

Definition reduces_to (t1 t2 : NTerm) :=
  {k : nat × compute_at_most_k_steps k t1 = csuccess t2}.

Definition computes_to_value (t1 t2 : NTerm) :=
  reduces_to t1 t2 × isvalue t2.

Definition hasvalue (t : NTerm) :=
  {t' : NTerm × computes_to_value t t'}.
Lemma computes_to_value_eq :
  ∀ t v1 v2,
      computes_to_value t v1
    → computes_to_value t v2
    → v1 = v2.
Lemma reduces_to_trans:
 ∀ a b c , reduces_to a b → reduces_to b c → reduces_to a c.

Theorem preserve_compute_step :
  ∀ (t1 t2 :NTerm),
    compute_step t1 = csuccess t2
    → isprogram t1
    → isprogram t2.
Theorem compute_step_alpha :
  ∀ (t1 t2 t1' :NTerm),
    alpha_eq t1 t2
    → compute_step t1 = csuccess t1'
  → { t2':NTerm × compute_step t2 = csuccess t2'
        × alpha_eq t1' t2'}.
```

The following lemma expresses a consequence of the fact the first argument of a NonCanonicalOp is always principal. Hence, if an arbitrary non-canonical term (oterm (NCan *op*) *lbt*) computes to a value in at most S *k* steps, *lbt* must be of the form *lbt* = (bterm [] *la*)::*lbtt* and there is an *m* such that $m \leq k$ and *la* (the principal argument) computes to some canonical term (oterm (Can *c*) *lbtc*) in m and the whole term computes also takes m steps to compute to ((oterm (NCan *op*) ((bterm [] (oterm (Can *c*) *lbtc*))::*lbtt*)))

```
Lemma compute_decompose :
∀ (op : NonCanonicalOp) (k: nat) (lbt : list BTerm) (a : NTerm),
  isprogram (oterm (NCan op) lbt)
  → computes_to_value_in_max_k_steps (S k) (oterm (NCan op) lbt) a
  → {la : NTerm
        × {lbtt, lbtc: (list BTerm)
          × { c : CanonicalOp
            × { m : nat
              × m ≤ k
            × lbt = (bterm [] la)::lbtt
            × computes_to_value_in_max_k_steps m
                                           la
                                          (oterm (Can c) lbtc)
            × compute_at_most_k_steps m (oterm (NCan op) lbt)
               = csuccess ((oterm (NCan op)
                   ((bterm [] (oterm (Can c) lbtc))::lbtt)))}}}}.
```

3.2 Computational Approximation

———-begin file rel_nterm.v ———-

```
Notation "tls {[ m ]}" := (selectbt tls m)
  (at level 99).
```

When we define the type system in the next chapter, we will want it to respect many computational relations. In Agda, Coq and Nuprl, if *t* reduces to *t'*, and *t* is in some type *T*, then *t* and *t'* are equal in *T*. In addition, it is useful to have our types respect a *congruence* that contains the computation relation. For example, Coq has a notion of definitional equality which is a congruence. In Nuprl, we have a computation equivalence ∼ [26], which is a coinductively defined congruence that permits more powerful equational reasoning. For example, all diverging programs are equivalent under $\sim$. The same holds for all programs that generate an infinite stream of zeroes.

Following him, we will first define his computational approximation (*approx*) and prove that it is a congruence. The desired equivalence(*cequiv*) is defined as a conjuction of *approx* with its inverse. In the next chapter, our definition of Nuprl's type system will respect *cequiv* by definition. We often denote *cequiv* by $\sim$.

We will also prove many domain theoretic properties about *approx* in the next section. These properties will be useful in defining partial types.

We begin by describing some key definitions involved in the defining of *approx*. Firstly, olift lifts a binary relation that is meaningful only on programs to to one on well-formed(nt_wf) terms that may have free variables. Recall that a program is a well-formed term that is also closed. the wf_sub predicate on Substitution asserts that the range of the substitution consists of well-formed terms. (One could have used the *CTerm* type to make this definition more compact.) Intuitively, (olift *R*) *s* *t* asserts that for all substitutions that turn *s* and *t* to closed terms *s'* and *t'*, *R* *s'* *t'* holds.

```
Definition olift (R : NTerm → NTerm → [univ]) (x y :NTerm) : [univ] :=
nt_wf x × nt_wf y × ∀ sub: Substitution, wf_sub sub
  → isprogram (lsubst x sub)
  → isprogram (lsubst y sub)
  → R (lsubst x sub) (lsubst y sub).
```

blift and lblift lift a binary relation on NTerms to one on BTerms and one on lists of BTerm respectively.

```
Definition blift (R: NTerm → NTerm → [univ]) (bt1 bt2: BTerm): [univ] :=
{lv: (list NVar) × {nt1 ,nt2 : NTerm × R nt1 nt2
  × alpha_eq_bterm bt1 (bterm lv nt1) × alpha_eq_bterm bt2 (bterm lv nt2) }}.
Definition lblift (R: NTerm → NTerm → [univ])
                              (tls trs: list BTerm): [univ] :=
length tls = length trs
× ∀ n : nat, n < length tls → blift R (tls[n]) (trs[n]).
```

FiXme Note: coqdoc notation messed up above :]} Below, we define some abstractions to denote that a relation respects another relation. In particular, we are interested in proving that many relations respect alpha equaliity

```
Definition respects2_l {T1 T2 : Set} (Rr : T1 → T1 → [univ])
```

```
                                        (R : T1 → T2 → [univ]) :=
  ∀ a b a', Rr a a' → R a b → R a' b.
Definition respects2_r {T1 T2 : Set} (Rr : T2 → T2 → [univ])
                                        (R : T1 → T2 → [univ]) :=
  ∀ a b b', Rr b b' → R a b → R a b'.
Definition respects2 {T : Set} (Rr : T → T → [univ])
                                     (R : T → T → [univ]):=
     (respects2_l Rr R) × (respects2_r Rr R).

Notation respects_alpha := (respects2 alpha_eq).

Lemma respects_alpha_olift: ∀ R,
  respects2 alpha_eq R
  → respects2 alpha_eq (olift R).
```

Because of the way alpha equality is baked into the definition of blift, for any binary relation R on NTerm, blift R respects alpha equality of bound terms.

```
Lemma respects_blift_alphabt: ∀ (R : bin_rel NTerm),
  respects2 alpha_eq_bterm (blift R).
```

We will use the following lemmas later in this document.

```
Lemma blift_numbvars: ∀ R bt1 bt2,
  blift R bt1 bt2
  → num_bvars bt1= num_bvars bt2.
```

———-begin file approx.v ———

We now define close_comput, an endofunction in the type of binary relations on NTerm. approx can be considered the greatest fixpoint of close_comput. However, we define it predicatively using the CoInductive construct of Coq. Howe denotes *close_compute R* as $[R]$.

```
Notation "t1 =v> t2" := (computes_to_value t1 t2)
  (at level 99).

Definition close_comput (R: NTerm → NTerm → [univ]) (tl tr : NTerm): [univ]:=
isprogram tl × isprogram tr × ∀ (c : CanonicalOp) (tl_subterms : list BTerm),
  (tl =v> (oterm (Can c) tl_subterms))
  → {tr_subterms : list BTerm × (tr =v> (oterm (Can c) tr_subterms))
       × lblift (olift R) tl_subterms tr_subterms }.
```

At this point, one could directly define approx as:

```
CoInductive approx_bad :
     NTerm → NTerm → [univ] :=
| approxC: ∀ tl tr,
  close_comput approx_bad tl tr
  → approx_bad tl tr.
```

However, this approach would require using the `cofix` tactic of Coq for proving the the properties of approx. Unfortunately, `cofix` does a very conservative productivity checking and often rejects our legitimate proofs at `Qed` time.

So, we use parametrized coinduction [27] to define it in a slightly indirect way. With this technique, we only need to use `cofix` once.[1]

```
Notation "p =2> q" :=
  (∀ x0 x1 (PR: p x0 x1 : Type), q x0 x1 : Type)
  (at level 50, no associativity).

Notation "p \2/ q" :=
  (fun x0 x1 ⇒ p x0 x1 [+] q x0 x1)
  (at level 50, no associativity).

Definition bot2 (x y : NTerm) := False.

CoInductive approx_aux
  (R : bin_rel NTerm) (tl tr: NTerm): [univ] :=
|approx_fold:
  close_comput (approx_aux R \2/ R) tl tr
                     → approx_aux R tl tr.

Definition approx := approx_aux bot2.
```

The first thing we do is to prove "co-induction principle" for approx using `cofix` and then never ever use `cofix` again. An interested user can walk through the proof of approx_refl below to see this co-induction principle in action.

```
Theorem approx_acc: ∀ (l r0 : bin_rel NTerm)
  (OBG: ∀ (r: bin_rel NTerm)
      (INC: r0 =2> r) (CIH: l =2> r),
         l =2> approx_aux r),
   l =2> approx_aux r0.
```

The following lemma says that for programs a and b, if we have to prove approx a b, the we can assume that a converges. Although, it is trivial

[1]If we allowed ourselves the luxury of impredicativity([Prop]), we would never need [cofix].

to prove it by assuming the law of excluded middle, a constructive proof is almost as easy and follows directly from the definition of approx.

```
Lemma approx_assume_hasvalue :
  ∀ a b,
    isprogram a
    → isprogram b
    → (hasvalue a → approx a b)
    → approx a b.
```

The following is an easy corollary of the above.

```
Corollary bottom_approx_any : ∀ t, isprogram t → approx mk_bottom t.
```

Because in the PER semantics, Nuprl's types are defined as partial equivalence relations on closed terms, we define a closed version of approx as follows:

```
Definition approxc (a b : CTerm) :=
  approx (get_cterm a) (get_cterm b).
```

We formalize a co-inductive proof of reflexivity of approx by using the approx_acc lemma above.

```
Lemma approx_refl :
  ∀ (t : NTerm), isprogram t → approx t t.

Lemma approx_open_refl: ∀ nt: NTerm, (nt_wf nt) → (olift approx) nt nt.

Lemma respects_alpha_closed_comput : ∀ R,
  respects_alpha (close_comput R).

Corollary respects_alpha_approx: respects_alpha approx.

Notation approx_open := (olift approx).
Notation approx_open_bterm := (blift approx_open).
```

Now, we wish to prove that approx is transitive. The proof is essentially a co-indictive argument. However, to get a strong enough co-induction hypothesis when we formalize the proof using the approx_acc lemma above, we have to state it in a more general way. Intuitively, this is because, alpha equality is baked into the definition of blift.

```
Lemma approx_trans_aux :
  ∀ a b c a' c',
    alpha_eq a a'
    → alpha_eq c c'
    → approx a' b
    → approx b c'
    → approx a c.
```

```
Corollary approx_trans :
  ∀ a b c,
    approx a b
    → approx b c
    → approx a c.
```

```
Lemma computes_to_value_implies_approx :
  ∀ t x,
    isprogram t
    → computes_to_value t x
    → approx x t × approx t x.
Lemma hasvalue_approx :
  ∀ t u,
    approx t u
    → hasvalue t
    → hasvalue u.
```

```
Theorem alpha_implies_approx2: ∀ nt1 nt2,
  isprogram nt2
  → alpha_eq nt1 nt2
  → approx nt1 nt2.
```

Given approx_trans, now have to prove that approx_open is transitive. It turned out that our proof of that worked exactly for the following stronger version below.

The proof is not as trivial as suggested by Howe [26] and illustrates a very common pattern where seemingly trivial paper proofs take quite a lot of effort because the concrete proof requires delicate reasoning about details of Substitution. We will describe some of these complications below. The proof begins by introducing the hypothesis and destructing all conjunctions to take care of the nt_wf parts of approx_open. (Recall that approx_open is a notation for olift approx). Then we introduce the additional hypothesis *Hwfs*, *Hispa* and *Hispc*

```
Lemma olift_trans :
  ∀ R,
    trans_rel R
    → respects_alpha R
    → trans_rel (olift R).
```

```
Proof.   intros R Ht Hra a b c Hab Hbc. allunfold olift. all_destruct_ands;
auto.
  clear Hbc0 Hbc1 Hab0. intros sub Hwfs Hispa Hispc.
```

At this point, we have the following proof state:

```
1 subgoals
R : bin_rel NTerm
Ht : trans_rel R
Hra : respects_alpha R
a : NTerm
b : NTerm
c : NTerm
Hab1 : nt_wf b
Hab : ∀ sub : Substitution,
        wf_sub sub →
        isprogram (lsubst a sub) →
        isprogram (lsubst b sub) → R (lsubst a sub) (lsubst b sub)
Hbc : ∀ sub : Substitution,
        wf_sub sub →
        isprogram (lsubst b sub) →
        isprogram (lsubst c sub) → R (lsubst b sub) (lsubst c sub)
sub : Substitution
Hwfs : wf_sub sub
Hispa : isprogram (lsubst a sub)
Hispc : isprogram (lsubst c sub)
______________________________________(1/1)
R (lsubst a sub) (lsubst c sub)
```

We cannot just instantiate *Hab* and *Hbc* with *sub* because there is no reason why (lsubst *b* *sub*) would be a closed term. *b* might have free variables that are disjoint from the ones of *a* and *c*. From *Hispa* and *Hispc*, we can only conclude that the terms that *sub* substitutes for free variables of *a* and *c* are closed. But it might contain substitutions of non-closed terms for variables not free in *a* or *c*.

So, we first filter *sub* to keep only the first pair that applies to a free variable of either *a* or *c*. Let that *Substitution* be *subf*. We then prove that the range of *subf* consists of only closed terms (because of *Hispa* and *Hispc*).

Let *subb* be a substitution that substitutes some(arbitrary) closed terms for the free variables of *b*. We then prove that (lsubst *b* (*subf* ++ *subb*)) is closed and that (lsubst *a* (*subf* ++ *subb*)) and (lsubst *c* (*subf* ++ *subb*)) are alpha equal to (lsubst *a* *sub*) and (lsubst *c* *sub*) respectively. The latter is implied by the definition of *lsubst_aux* because it uses the first pair in the *Substitution* that matches a variable. Then, we can instantiate *Hab* and *Hbc* with (*sub*:=*subf* ++ *subb*) and do some rewriting using alpha equality to finish the proof.

```
Corollary approx_open_trans :
  ∀ a b c,
    approx_open a b
    → approx_open b c
    → approx_open a c.
Proof.
  apply olift_trans.
  - exact approx_trans.
  - exact respects_alpha_approx.
Qed.

Lemma approx_open_lsubst_congr : ∀ ta tb sub,
  wf_sub sub
  → approx_open ta tb
  → approx_open (lsubst ta sub) (lsubst tb sub).
```

We wish to prove that approx is a congruence. However, it is a fairly non-trivial task and the main content of [26]. For now, the following lemmma asserts that approx is a congruence w.r.t canonical *Opid*s. The general proof of congruence is discussed in the next subsection.

```
Lemma approx_canonical_form3 :
  ∀ op bterms1 bterms2,
    isprogram (oterm (Can op) bterms1)
    → isprogram (oterm (Can op) bterms2)
    → lblift (olift approx) bterms1 bterms2
    → approx (oterm (Can op) bterms1) (oterm (Can op) bterms2).

Lemma approx_mk_pair :
  ∀ (t t' a b : NTerm),
    computes_to_value t (mk_pair a b)
    → approx t t'
→ {a', b' : NTerm ×
     computes_to_value t' (mk_pair a' b')
     × approx a a'
     × approx b b'}.
```

3.2.1 Congruence

———–begin file approx_star.v ————

To prove that approx is a cogruence, Howe defined a relation approx_star that contains approx_open and is a congruence by definition. The main challenge then is to prove that approx_star implies approx_open, and hence they are equivalent. Howe reduced this property to a set of conditions(called extensionality) on each non-canonical Opid in the computation system. We managed to mechanize this reduction in Coq and also proved that all the Opids of Nuprl satisfy his extensionality condition.

We begin by defining his approx_star relation. It is a relation on well formed (possibly open) NTerms. So we need not use olift before applying lblift in the apso constructor.

```
Inductive approx_star:
      NTerm → NTerm → [univ] :=
| apsv: ∀ v t2,
      (approx_open (vterm v) t2)
        → (approx_star (vterm v) t2)
| apso: ∀ (op : Opid) (t2: NTerm)
    (lbt1 lbt1' : list BTerm),
     length lbt1 = length lbt1'
     → lblift approx_star lbt1 lbt1'
     → approx_open (oterm op lbt1') t2
     → approx_star (oterm op lbt1) t2.

Notation approx_star_bterm := (blift approx_star).
Notation approx_starbts := (lblift approx_star).

Lemma approx_star_refl : ∀ t, nt_wf t → approx_star t t.

Lemma approx_open_implies_approx_star : ∀ t1 t2, approx_open t1 t2
                                                → approx_star t1 t2.

Lemma approx_star_congruence : ∀ (o: Opid) (lbt1 lbt2 : list BTerm),
  approx_starbts lbt1 lbt2
  → map num_bvars lbt2 = OpBindings o
  → approx_star (oterm o lbt1) (oterm o lbt2).
Lemma approx_star_open_trans: ∀ a b c,
  approx_star a b
  → approx_open b c
  → approx_star a c.
```

The following is a generalization of Howe's lemma 1 [26]. He proved proved (needed) it for substitutions of length one. We need it atleast for substitutions of length upto two because the computation for NSpread performs a simultaneous subsitution for two variables. We prove a more general version instead. Apart from some uninteresting details about substitution, our mechanized proof is essentially the same as Howe's.

```
Lemma lsubst_approx_star_congr: ∀ (t1 t2 : NTerm) (lvi : list NVar) (lnt1 lnt2 :
list NTerm),
  bin_rel_nterm approx_star lnt1 lnt2
  → length lvi = length lnt1
  → length lvi = length lnt2
  → approx_star t1 t2
  → approx_star (lsubst t1 (combine lvi lnt1)) (lsubst t2 (combine lvi lnt2)).
```

Howe implicitly uses the following lemma at least twice in his proofs. It is essentially a more useful way to eliminate (use/destruct) a hypothesis of the form approx_star (oterm *o* *lbt*) *b*. The advantage here is that we additionally obtain the hypothesis isprogram (oterm *o* *lbt'*). The *lbt'* that we obtain by naive inductive destruction of approx_star (oterm *o* *lbt*) *b* need not satisify this property. This additional property simplifies many proofs. For example, in his proof of Lemma 2 (shown below), when Howe says "by Lemma 1 and the definition of $\leq$ on open terms, we may assume that $\theta(\overline{t''})$ is closed", he is essentially using this lemma.

The proof of this lemma involves reasoning like that used in the the proof of approx_open_trans. Essentially, we substitute arbitrary closed terms for free variables in *lbt'* obtained by the inductive destruction so that it becomes closed and show that this substitution has no effect when it gets applied to other terms in the proof.

```
Lemma approx_star_otd : ∀ o lbt b,
approx_star (oterm o lbt) b
→ isprogram b
→ isprogram (oterm o lbt)
→ {lbt' : (list BTerm) × isprogram (oterm o lbt')
        × approx_open (oterm o lbt') b
        × length lbt = length lbt'
        × approx_starbts lbt lbt'}.
```

We now prove Howe's lemma 2 [26]. Using the lemma approx_star_otd above, this proof goes pretty much exactly like Howe describes.

```
Lemma howe_lemma2 :
  ∀ (c : CanonicalOp) (lbt : list BTerm) (b : NTerm),
    let t:= (oterm (Can c) lbt) in
    isprogram t
    → isprogram b
    → approx_star t b
    → {lbt' : (list BTerm) × approx_starbts lbt lbt'
                 × computes_to_value b (oterm (Can c) lbt')}.
```

Informally, howe_lemma2 looks a lot like the definition of close_comput. The only difference is that close_comput was preserved when computation happens on the LHS argument.

Recall the approx can be considered as a greatest fixed point of close_comput. If we could prove that approx_star is preserved when computation happens on the LHS argument, a simple coinductive argument will prove that approx_star implies approx on closed terms. Formally, we only need to prove the following lemma [2] :

```
Lemma howe_lemma3 : ∀ (a a' b : NTerm),
  isprogram a
  → isprogram a'
  → isprogram b
  → computes_to_value a a'
  → approx_star a b
  → approx_star a' b.
```

This proof will proceed by the induction on the number of steps that a took to compute to the value a'. Since variables don't compute to anything, a must be of the form oterm o lbt. The proof then proceeds by case analysis

[2]Howe did not explicitly call it Lemma 3. But he proves it while proving his theorem 1

on *o*. Unlike the previous proofs about approx, which were uniform w.r.t the Opids in the language and only assumed that the computation system was lazy, this proof requires reasoning about each Opid in the language.

Howe abstracts the remainder of the proof of this lemma into the following condition (called extensionality) that has to be proved for each Opid in the language. The last hypothesis (*Hind*, the big one) in this definition is essentially the induction hypothesis in the proof of howe_lemma3.

```
Definition extensional_op (o : Opid) :=
  ∀ (lbt lbt' : list BTerm) (a : NTerm) (k:nat)
  (Hpa : isprogram a)
  (Hpt : isprogram (oterm o lbt))
  (Hpt' : isprogram (oterm o lbt'))
  (Hcv : computes_to_value_in_max_k_steps (S k) (oterm o lbt) a)
  (Has : (lblift approx_star) lbt lbt')
  (Hind : (∀ (u u' v : NTerm),
            isprogram u
            → isprogram u'
            → isprogram v
            → computes_to_value_in_max_k_steps k u u'
            → approx_star u v
            → approx_star u' v)),
  approx_star a (oterm o lbt').
```

It is immediately clear that all the canonical Opids of a lazy computation system are extensional. In this case, we have (oterm *o* *lbt*)= *a* and the conclusion is an immediate consequence of congruence of approx_star.

```
Lemma nuprl_extensional_can : ∀ (cop : CanonicalOp) ,
      extensional_op (Can cop).
```

The next definition is just compact and equivalent restatement of extensional_op for the paper. Please ignore if you are reading the technical report. Sorry!

```
Definition extensional_opc (o : Opid) :=
∀ (lbt lbt' : list BTerm) (a : NTerm) (k:nat),
programs [a,(oterm o lbt),(oterm o lbt')]
→ computes_to_value_in_max_k_steps (S k) (oterm o lbt) a
→ (lblift approx_star) lbt lbt'
→ (∀ (u u' v : NTerm),
        programs [u,u',v]
        → computes_to_value_in_max_k_steps k u u'
```

```
      → approx_star u v
      → approx_star u' v)
→ approx_star a (oterm o lbt').
```

We now begin to prove that the non-canonical Opids of Nuprl are extensional. The following corollary of Howe's lemma 1 (lsubst_approx_star_congr) will be very useful in of the proofs for the Opids NApply, NCbv, NDecide, NSpread.

```
Lemma apply_bterm_approx_star_congr: ∀ bt1 bt2 lnt1 lnt2,
  approx_star_bterm bt1 bt2
  → bin_rel_nterm approx_star lnt1 lnt2
  → length lnt1 = num_bvars bt1
  → length lnt1 = length lnt2
  → approx_star (apply_bterm bt1 lnt1) (apply_bterm bt2 lnt2).
```

Howe and Crary prove extensionality of many non-canonical Opids. Our computation system has some new ones and the operational semantics of some earlier ones like NFix is different. We have formally proved that all Opids in our system are extensional. Instead of describing these proofs separately, we will describe the general recipe. A reader who wishes to delve into very concrete details can walk through the coq proof scripts for the extensionality lemmas.

In general, whenever a computation in which an arbitrary non-cononical term (oterm (NCan *op*) *lbt*) computes to a value *a*, *lbt* can be expressed as (map (bterm []) *pnt*)++*bargs*, where *pnt* are the principal arguments of *op*. The length of *pargs* depends on *op*. For NCompOp and NCanTest, it is 2 and it is 1 for the rest. The S *k* steps of computation from (oterm (NCan *op*) (map (bterm []) *pnt* ++ *bargs*)) to the value *a* (see hypothesis *Hcv* in extensional_op) can be split into the following three parts that happen one after the other.

- Each element of *pnt* converges to some canonical NTerm. At the end of this stage, the overall term is of the form (oterm (NCan *op*) ((map (bterm []) *pntc*)++*bargs*)) such that elements of *pnt* converge respectively to canonical elements of *pntc*.

- One interesting step of computation happens by the interaction of the canonical Opids in *pntc* and the corresponding non-canonical Opid *op*. Let the overall term after this step be *t*. Let *llbt* be (*bargs* ++ (flat_map get_bterms *pntc*)). For the proof of extensional_op (NCan *op*), the key property here is that t can always be written as some f $llbt$ such that $\forall$ $lbt1$ $lbt2$, approx_starbts $lbt1$ $lbt2$ $\rightarrow$ approx_star (f $lbt1$) (f $lbt2$). The details of this depend on the specific op. We consider all cases one by one. The reader might want to revisit the definition of $compute\_step$ to understand the claims below. $op=$

  – - NApply : In this case, $pntc$ is of the form [oterm (Can $NLambda$) [(bterm [v] b)]] and $bargs$ is of the form [bterm [] arg] and $t=$ apply_bterm (bterm [v] b) [arg]. For this and the next 4 cases, the required property of f is a direct consequence of the lemma apply_bterm_approx_star_congr above.
  – - NCbv : $pntc$ is of the form [oterm (Can cc) $lbtc$] and $bargs$ is of the form bterm [v] b. $t=$ apply_bterm (bterm [v] b) [(oterm (Can cc) $lbtc$)].
  – - NSpread : $pntc$ is of the form [oterm (Can $NPair$) [bterm [] $pi1$,bterm [] $pi2$]] and $bargs$ is of the form bterm [$v1$,$v2$] b. $t=$ apply_bterm (bterm [$v1$,$v2$] b) [$pi1$,$pi2$]
  – - NDecide : $pntc$ is of the form [oterm (Can $NInl$) [bterm [] arg]] or [oterm (Can $NInr$) [bterm [] arg]] and $bargs$ is of the form [bterm [v] bl,bterm [v] br] and $t=$ apply_bterm (bterm [v] bl) [arg] or $t=$ apply_bterm (bterm [v] br) [arg] depending on $pntc$.
  – - NArithOp : $pntc$ is of the form [oterm (Can ($Nint$ $n1$)) [], oterm (Can ($Nint$ $n2$)) []] and $bargs$ is $\square$. $t=$ oterm (Can ($Nint$ ($n1+n2$))) [] The f in this case does not depend on any BTerms (there aren't any in this case) and is hence a constant function. The same reason applies for the three cases below.
  – - NCompOp : and $bargs$ is of the form $arg3$, $arg4$. t is either $arg3$ or $arg4$ depending only on the head canonical Opids of the NTerms in $pntc$
  – - NCanTest : exactly same as above.

- t converges to a.

One key observation here is that the second part of this 3-way split consumes exactly one step. Hence the the first and last parts consume at most k steps and hence $Hind$ (in the definition of extensional_op) can be applied to both these parts.

To prove extensional_op *op*, we use the hypothesis *Has* to infer that *lbt'* can also be expressed as (map (bterm []) *pnt'*)++*bargs'* (see the lemma blift_numbvars) such that we have *Hasp* : bin_rel_nterm approx_star *pnt pnt'* Applying *Hind* along with *Hasp* to the first stage of computation where *pnt* converges pointwise to *pntc*, and we get *Haspc* : bin_rel_nterm approx_star *pntc pnt'*. Next, we apply howe_lemma2 pointwise to *Haspc*, we get *pntc'* such that elements of *pnt'* converges to *pntc'* respectively such that we have *Haspcc* : bin_rel_nterm approx_star *pntc pntc'*

Next, the second stage of computation happens and we get that oterm (NCan *op*) ((map (bterm []) *pntc'*)++*bargs'*) computes to some *t'* in exactly one step. By the property of this one step function *f* that we described casewise above, we get *Hapt* : approx_star *t t'*.

Finally, we apply *Hind* to the third stage again to get *Hapa* : approx_star *a t'*. Since oterm (NCan *op*) ((map (bterm []) *pnt'*)++*bargs'*) reduced to *t'*, we use the lemma reduces_to_implies_approx_open above to get *Hao* : approx_open *t'* (oterm (NCan *op*) ((map (bterm []) *pnt'*)++*bargs'*)) Now, we can use approx_star_open_trans on *Hapa* and *Hao* to get the desired conclusion of extensional_op *op*.

The concrete Coq proofs of the extensionality lemmas below follow this overall recipe.

Lemma extensional_apply : extensional_op (NCan NApply).

Lemma extensional_fix : extensional_op (NCan NFix).

Lemma extensional_cbv : extensional_op (NCan NCbv).

Lemma extensional_spread : extensional_op (NCan NSpread).

Lemma extensional_dsup : extensional_op (NCan NDsup).

Lemma extensional_decide : extensional_op (NCan NDecide).

Lemma extensional_arith: ∀ *a*, extensional_op (NCan (NArithOp *a*)).

Lemma extensional_ncomp: ∀ *a*, extensional_op (NCan (NCompOp *a*)).

Lemma extensional_cantest : ∀ *a*, extensional_op (NCan (NCanTest *a*)).

```
Theorem nuprl_extensional : ∀ op, extensional_op op.
Proof.
  intro op. destruct op.
  - apply nuprl_extensional_can.
  - destruct n.
    + apply extensional_apply.
    + apply extensional_fix.
    + apply extensional_spread.
    + apply extensional_dsup.
    + apply extensional_decide.
    + apply extensional_cbv.
    + apply extensional_ncomp.
    + apply extensional_arith.
    + apply extensional_cantest.
Qed.
```

As we mentioned above, Howe abstracted the extensionality condition above out of the proof of the following lemma. Hence its proof follows directly from the lemma nuprl_extensional.

```
Lemma howe_lemma3 : ∀ (a a' b : NTerm),
  isprogram a
  → isprogram a'
  → isprogram b
  → computes_to_value a a'
  → approx_star a b
  → approx_star a' b.
```

Now Howe uses a simple coindiuctive argument to show that approx_star implies approx on closed terms.

```
Theorem howetheorem1:
  ∀ a b,
    approx_star a b
    → isprogram a
    → isprogram b
    → approx a b.
```

There are many useful Corollaries of the above theorem.

```
Corollary approx_star_implies_approx_open:
  ∀ t1 t2 : NTerm, approx_star t1 t2 → approx_open t1 t2.

Corollary approx_star_iff_approx_open:
  ∀ t1 t2 : NTerm, approx_star t1 t2 ⇔ approx_open t1 t2.

Corollary approx_open_congruence : ∀ (o : Opid) (lbt1 lbt2 : list BTerm),
  lblift approx_open lbt1 lbt2
  → nt_wf (oterm o lbt2) → approx_open (oterm o lbt1) (oterm o lbt2).

Corollary approx_congruence : ∀ o lbt1 lbt2,
  lblift approx_open lbt1 lbt2
  → isprogram (oterm o lbt1)
  → isprogram (oterm o lbt2)
  → approx (oterm o lbt1) (oterm o lbt2).

Corollary lsubst_approx_congr: ∀ t1 t2 sub1 sub2,
  sub_range_rel approx sub1 sub2
  → approx_open t1 t2
  → isprogram (lsubst t1 sub1)
  → isprogram (lsubst t2 sub2)
  → approx (lsubst t1 sub1) (lsubst t2 sub2).
```

————-begin file cequiv.v ————

Finally, we use approx to define the following equivalence relation on closed terms.

```
Definition cequiv (a b : NTerm) :=
  approx a b × approx b a.
```

We lift cequiv to the CTerm type in the standard way:

```
Definition cequivc (a b : CTerm) :=
  cequiv (get_cterm a) (get_cterm b).
```

The equivalence of cequiv is a trivial consequence of either its symmetrical definition, or the corresponding properties of approx.

```
Lemma cequiv_refl :
  ∀ t,
    isprogram t
    → cequiv t t.
Lemma cequiv_sym :
  ∀ a b,
    cequiv a b ⇔ cequiv b a.
Lemma cequiv_trans :
  ∀ a b c,
    cequiv a b
    → cequiv b c
    → cequiv a c.
```

The following lemma is a straightforward consequence of the corresponding lemmas about approx. It is the holy grail of this section.

Since the type system that we define later will respect cequiv by definition, and cequiv contains the computation relation, we can easily prove, among other things, that reduction at any position inside of a term preserves its type.

```
Lemma cequiv_congruence : ∀ o lbt1 lbt2,
  lblift (olift cequiv) lbt1 lbt2
  → isprogram (oterm o lbt1)
  → isprogram (oterm o lbt2)
  → cequiv (oterm o lbt1) (oterm o lbt2).
Inductive cequiv_subst : Sub → Sub → Type :=
  | ceq_sub_nil : cequiv_subst [] []
  | ceq_sub_cons :
    ∀ v t1 t2 s1 s2,
      cequiv t1 t2
      → cequiv_subst s1 s2
      → cequiv_subst ((v,t1) :: s1) ((v,t2) :: s2).
```

The following useful lemma is also a direct consequence of the corresponding property for approx.

```
Lemma cequiv_lsubst :
  ∀ t sub1 sub2,
    isprogram (lsubst t sub1)
    → isprogram (lsubst t sub2)
    → cequiv_subst sub1 sub2
    → cequiv (lsubst t sub1) (lsubst t sub2).
```

Later, we will show that one can rewrite by cequiv at any place in a hypothesis (*rule_cequiv_subst_hyp* in Sec. 5.2.8) or the conclusion (*rule_cequiv_subst_concl* in Sec. 5.2.8) in a Nuprl proof.

3.3 Some Domain Theory

———-begin file domain_th.v ————

approx is a preorder on closed terms (see lemmas approx_refl and approx_trans above in this section). In this subsection we will formalize some domain theoretic properties of this preorder. We will prove that it has a least element (upto cequiv). We will define finite approximations of a term of the form mk_fix *f* and formalize Crary's [21] proof that mk_fix *f* is the least upperbound of its approximations. We will also formalize his Compactness property which says that if a term of the form mk_fix *f* converges, then one of it's finite approximations also converges. We had to slightly adapt Crary's proofs because our operations semantics for mk_fix f is slightly different. These properties will be used in Sec.5.2.9 to prove rules that talk about membership of terms of the form mk_fix f in partial types.

First we recap some earlier definitions that will be used a lot in this section.

```
Definition mk_fix (f : NTerm) :=
  oterm (NCan NFix) [ bterm [] f ].

Definition mk_id := mk_lam nvarx (vterm nvarx).

Definition mk_lam (v : NVar) (b : NTerm) :=
  oterm (Can NLambda) [bterm [v] b].

Definition mk_bottom := mk_fix mk_id.

Definition mk_bot := mk_bottom.
```

As the name suggests, the following function constructs the n^{th} approximation to mk_fix f.

```
Fixpoint fix_approx (n : nat) (f : NTerm) : NTerm :=
match n with
| 0 ⇒ mk_bottom
| S n ⇒ mk_apply f (fix_approx n f)
end.

Lemma fix_approx_program: ∀ f,
  isprogram f
  → ∀ n, isprogram (fix_approx n f).

Definition is_chain {T : Type} (R : @bin_rel T) (tf : nat → T) : [univ] :=
  ∀ n, R (tf n) (tf (S n)).

Definition is_upper_bound {T : Type} (R : @bin_rel T) (tf : nat → T) (t: T)
: [univ] :=
  ∀ n, R (tf n) t.

Notation is_approx_ub := (is_upper_bound approx).
Notation is_approx_chain := (is_chain approx).
```

The following lemma is an easy consequence of the congruence of approx

```
Lemma is_approx_chain_fix_aprox : ∀ (f : NTerm),
  isprogram f
  → is_approx_chain (fun n ⇒ fix_approx n f).
```

The following lemma says that mk_fix f is an upper bound of its approximations.

```
Lemma fix_approx_ub : ∀ f,
  isprogram f
  → is_approx_ub (fun n ⇒ fix_approx n f) (mk_fix f).
```

One of the main goals of this section is to prove that it is infact the least upper bound (upto cequiv). We will hereby call it the LUB principle(theorem 5.9 in [21]). The proof is similar in spirit to the one in [21]. However, our definition of the operational semantics(compute_step) of mk_fix is different from his. As mentioned before in the section 3.1.2, in our computation system, mk_fix f computes to mk_apply f (mk_fix f) only if f is canonical. In other words, unlike in his system, the first argument if a *NFix* is always principal, It evaluates f further otherwise.

When we made this innocuous looking change (months before), we did not think that these proofs will require some non-trivial changes. Intead of reverting to the old semantics by fixing our previous Coq proofs that were simplified because the first argument is now uniformly principal for all *NonCanonicalOp*s, we decided hunt for a proof of the LUB principle that holds in this new operational semantics. We were successful in this endeavour.

To prove the LUB principle, Crary first proves 2 lemmas (5.7 and 5.8) Although both the proof and statement of his lemma 5.7 holds for our system, lemma 5.8 probably does not hold for our system (TODO : prove a counterexample). However, a weaker version of it holds. To prove the LUB principle for our system we proved a strengthened version of his lemma 5.7, and a weak version of his lemma 5.8. Along with some other new lemmas about our computation system, we could then prove Crary's LUB principle (and the compactness principle below).
To begin, here is a sneak peek at the statements of the LUB principle and Compactness principle which we wish to prove along the lines of Crary.

```
Theorem crary5_9_LUB: ∀ f t e,
  isprogram f
  → is_approx_ub (fun n ⇒ apply_bterm (bterm [nvarx] e) [fix_approx n f]) t
  → approx (apply_bterm (bterm [nvarx] e) [mk_fix f]) t.
```

Abort.

Compactness says that if a term with a subterm of the form mk_fix f then there (constructively) exists a number n such that replacing that subterm by its n^{th} finite approximation also conveges.

```
Theorem fix_compactness: ∀ f e,
  let tf := (apply_bterm (bterm [nvarx] e) [mk_fix f]) in
  isprogram f
  → isprogram tf
  → hasvalue tf
  → {n : nat × let tfa := (apply_bterm (bterm [nvarx] e) [fix_approx n f]) in
            hasvalue tfa}.
Abort.
```

A key use of Compactness will be in proving for certain types T (called admissible types), that if a f is in the function type *mk_partial* $T \rightarrow$ *mk_partial* T, then mk_fix f is in the type *mk_partial* T. Intuitively, $t \in$ *mk_partial* T iff (*hasvaue* $t \rightarrow t \in T$). We defer the proof of this rule till section 5.2.9 where we will we have a formal definition of our type system. The goal of the rest of this section to formalize the proofs of Compactness and the LUB property.

The following lemma is an (at least apparently) stronger version of Crary's lemma 5.7. Although its statement might seem overly complicated. it is expressing a very intuitive property of our(and Crary's) comutation system that closed, noncanonical terms that lie within a term being evaluated are not destructed. They either are moved or copied around unchanged (the first case in the disjunction in the lemma's conclusion) or are evaluated in place with the surrounding term left unchanged (the second case).

More concretely, in the first case, the behaviour remains unchanged even when we replace that closed noncanonical subterm with some other term. The second case might look like a complicated way to express the property above. Intuitively, it says that some of the occurences of that non-canonical term get evaluated in place. These occurences are denoted by y. It turns out from the proof that exactly 1 of the occurences gets evaluated in a step. Crary's original statement can be derived as a corollary (see crary5_7_original below). Our strengthening only affects the second case where we additionally require that if that closed non-canonical subterm is replaced by any other closed non-canonical subterm, the behaviour will remain unchanged. It is not immediately clear if this strenghening puts additional restrictions on the kind of computation systems for this this property holds. The first case in

Crary's(and our) statement already forces the computation system to behave in the same way if the noncanonical subterm is replaced by some other subterm.

```
Lemma crary5_7 : ∀ vx vy e1 e2 no lbt,
let t := (oterm (NCan no) lbt) in
let tl := subst e1 vx t in
vx ≠ vy
→ isprogram t
→ isprogram tl
→ computes_in_1step tl e2
→ {e2' : NTerm × ∀ t', isprogram t'
                → computes_in_1step_alpha (subst e1 vx t')
                                          (subst e2' vx t')}
                              [+]
   {e1',t' : NTerm × alpha_eq e1 (subst e1' vy (vterm vx))
        × computes_in_1step t t'
        × ∀ t'' td td',
              isprogram t''
              → isprogram td
              → (computes_in_1step td td'
                     → computes_in_1step (lsubst e1' [(vx,t''),(vy,td)])
                                         (lsubst e1' [(vx,t''),(vy,td')]))
                × (computes_to_error td
                     → computes_to_error
                         (lsubst e1' [(vx,t''),(vy,td)]))}.

Corollary crary5_7_original : ∀ e1 e2 no lbt,
let t := (oterm (NCan no) lbt) in
let tl := subst e1 nvarx t in
isprogram t
→ isprogram tl
→ computes_in_1step tl e2
→ {e2' : NTerm × ∀ t', isprogram t'
          → computes_in_1step_alpha (subst e1 nvarx t')
                                    (subst e2' nvarx t')}
                              [+]
   {e1',t' : NTerm × alpha_eq e1 (subst e1' nvary (vterm nvarx))
        × computes_in_1step t t'
        × ∀ t'', isprogram t''
           → computes_in_1step (lsubst e1' [(nvarx,t''),(nvary,t)])
```

```
                                        (lsubst e1' [(nvarx,t''),(nvary,t')])}.
```

Here is a weaker version of Crary's 5.8 that certainly holds for our system. In the place of the hypothesis isvalue f, Crary only had the hypothesis isprogram f. In his proof, Crary uses the fact that mk_fix f evaluates to mk_apply f (mk_fix f) in his computation system. In our system, f needs to be a value for that to be true.

```
Lemma weaker_crary_5_8_aux2 : ∀ k (f e1 e2 : NTerm),
let bt1:= bterm [nvarx] e1 in
compute_at_most_k_steps k (apply_bterm bt1 [(mk_fix f)]) = csuccess e2
→ isvalue f
→ isprogram_bt bt1
→ {e2' : NTerm × let bt2 := (bterm [nvarx] e2') in
        alpha_eq e2 (apply_bterm bt2 [(mk_fix f)])
        × {j : nat × ∀ k,
                k ≥ j
                → approx (apply_bterm bt2 [(fix_approx (k-j) f)])
                         (apply_bterm bt1 [(fix_approx (k) f)])
                      }}.

Corollary weaker_crary_5_8 : ∀ (f e1 e2 : NTerm),
let bt1:= bterm [nvarx] e1 in
computes_to_value (apply_bterm bt1 [(mk_fix f)]) e2
→ isvalue f
→ isprogram_bt bt1
→ {e2' : NTerm × disjoint (bound_vars e2') [nvarx]
             ×let bt2 := (bterm [nvarx] e2') in
                alpha_eq e2 (apply_bterm bt2 [(mk_fix f)])
                × {j : nat × ∀ k,
                        k ≥ j
                        → approx (apply_bterm bt2 [(fix_approx (k-j) f)])
                                 (apply_bterm bt1 [(fix_approx (k) f)])
                              }}.
```

Crary's constructive proofs of both the LUB property and compactness use his lemma 5.8. How do we deal with the fact that we only have that when the f in those theorems (shown above) is a value. Intuitively, both these properties are about approx and since it is a congruence and includes the reduces_to relation, we are free to replace f at any place with whatever it reduces. So, if f does reduce to a value (say fv), we we can replace f with fv everywhere and get away with just the weaker version of 5.8 which we have.

The following 3 lemmas justify the legitimacy of that replacement. Recall that cequiv *a* *b* stands for approx *a* *b* × approx *b* *a*.

```
Lemma reduces_to_subst_fix: ∀ f fv e v,
    reduces_to f fv
    → isprogram f
    → isprogram_bt (bterm [v] e)
    → cequiv (lsubst e [(v, mk_fix f)]) (lsubst e [(v, mk_fix fv)]).
Lemma reduces_to_subst_fix_aprrox: ∀ n f fv e v,
    reduces_to f fv
    → isprogram f
    → isprogram_bt (bterm [v] e)
    → cequiv (lsubst e [(v, fix_approx n f)]) (lsubst e [(v, fix_approx n fv)]).
Lemma reduces_to_computes_to_value_rw : ∀ f fv e v c lbt,
    reduces_to f fv
    → isprogram f
    → isprogram (apply_bterm (bterm [v] e) [mk_fix f])
    → computes_to_value (apply_bterm (bterm [v] e) [mk_fix f])
          (oterm (Can c) lbt)
    → {lbtv : list BTerm × computes_to_value (apply_bterm (bterm [v] e)
[mk_fix fv])
          (oterm (Can c) lbtv)}.
```

Since we have a constructive meta-theory here and Nuprl's language is turing complete, there is no way to determine if *f* computes to a value. So, unlike a classical logician, we cannot separately handle the cases when it converges to a value and the one where it doesn't. However, we have a weaker dichotomy that is sufficient for both the proofs (of Compactness and LUB theorems). Near the beginning of both these proofs, we will have a hypothesis of the form computes_to_value (apply_bterm *bt* [(mk_fix *f*)]) *v* (recall the definitions of approx and *close_comput*).

By using lemma 5.7 at each step of this computation, we can determine if the *f* was ever evaluated in this computation (second case of lemma 5.7). Our strengthening of 5.7 lets us prove that if *f* was evaluated even for 1 step in the above computation, it must have been evaluated to a value in order for the computation of the overall term to converge to value. This case corresponds to the first case in the disjunction in the conclusion of the lemma fix_value2 below.

However, if the first case of lemma 5.7 holds for all these steps of computation, then (mk_fix *f*) can be replaced with anything and we will still get the same behaviour. The predicate dummy_context below formalizes this property. computes_to_alpha_value a b asserts that a converges to a value that is alpha equal to b.

```
Definition dummy_context (v : NVar) (e : NTerm) :=
  let bt := bterm [v] e in
    {vc : NTerm × let btv := bterm [v] vc in
          ∀ t, isprogram t→
            computes_to_alpha_value
              (apply_bterm bt [t])
              (apply_bterm btv [t])}.
```

```
Lemma fix_value2 : ∀ f v e, let bt := bterm [nvarx] e in
  isprogram f
  → isprogram (apply_bterm bt [(mk_fix f)])
  → computes_to_value (apply_bterm bt [(mk_fix f)]) v
  → {fv : NTerm × computes_to_value f fv}
          [+]
      dummy_context nvarx e.
```

The following lemma gives us something like Crary's 5.8 (with j=0) when the second case of fix_value2 above holds.

```
Lemma dummy_context_crary_5_8_aux : ∀ (f e1 e2 : NTerm),
  let bt1:= bterm [nvarx] e1 in
  dummy_context nvarx e1
  → isprogram f
  → isprogram_bt bt1
  → computes_to_value (apply_bterm bt1 [(mk_fix f)]) e2
  → {e2' : NTerm ×
              let bt2 := (bterm [nvarx] e2') in
                alpha_eq e2 (apply_bterm bt2 [(mk_fix f)])
                × ∀ k, approx (apply_bterm bt2 [(fix_approx k f)])
                                           (apply_bterm bt1 [(fix_approx k f)])
                                         }.
```

```
Corollary dummy_context_crary_5_8 : ∀ (f e1 e2 : NTerm),
let bt1:= bterm [nvarx] e1 in
dummy_context nvarx e1
→ isprogram f
→ isprogram_bt bt1
→ computes_to_value (apply_bterm bt1 [(mk_fix f)]) e2
→ {e2' : NTerm × disjoint (bound_vars e2') [nvarx]
        × let bt2 := (bterm [nvarx] e2') in
```

```
            alpha_eq e2 (apply_bterm bt2 [(mk_fix f)])
            × {j : nat × ∀ k,
                    k ≥ j
                    → approx (apply_bterm bt2 [(fix_approx (k-j) f)])
                               (apply_bterm bt1 [(fix_approx (k) f)])
                         }}.
```

Finally, we have enough lemmas to prove the compactness theorem. Crary shows a classical proof that uses the LUB property. He also gives enough hints to derive a constructive proof that he thinks is less elegant. Here is a less elegant, but constructive proof that is better suited for a constructive meta-theory like the predicative fragment of Coq.

```
Theorem fix_compactness: ∀ f e,
  let tf := (apply_bterm (bterm [nvarx] e) [mk_fix f]) in
  isprogram f
  → isprogram tf
  → hasvalue tf
  → {n : nat × let tfa :=
        (apply_bterm (bterm [nvarx] e) [fix_approx n f]) in
          hasvalue tfa}.
Proof.
  simpl. intros f e Hpf Hprt Hcv.
  unfold hasvalue in Hcv.
  destruct_all_exists. rename Hcv0 into Hcv.
  rename t' into vv.
```

After some straightforward intial steps, here is the proof state:

```
1 subgoals
f : NTerm
e : NTerm
Hpf : isprogram f
Hprt : isprogram (apply_bterm (bterm [nvarx] e) [mk_fix f])
vv : NTerm
Hcv : apply_bterm (bterm [nvarx] e) [mk_fix f] =v> vv
--------------------------------------(1/1)
{n : nat × hasvalue (apply_bterm (bterm [nvarx] e) [fix_approx n f])}
```

Now apply fix_value2 to get the dichotomy in its conclusion.

- In the left case, we get that *f* reduces to a value *fv*. In this case, we replace f by fv everywhere using the lemmas reduces_to_subst_fix, *reduces_to_subst_fix_approx* and reduces_to_computes_to_value_rw above. Then we can apply weaker_crary_5_8 to Hcv. The remaining proof is same even for the other case. So, for uniformity of both cases, we then rename fv into f and also rename the replacement for vv in Hcv into vv.

- In the right case, we get $Hdum$: dummy_context e. then we can apply dummy_context_crary_5_8 to get to a proof state which is virtually same as that of the previous case.

In both cases, we have that there is some *e2'* such that vv is alpha equal to (apply_bterm (bterm [nvarx] *e2'*) [mk_fix f]). Note that *e2'* must be of the form oterm [Can _] _. If it were a variable, or of the form oterm [NCan _] _, we wont get a value(something of the form oterm [Can _] _) when substituting nvarx by mk_fix f.
We also have that there is a j such that

```
∀ k : nat,
      k ≥ j →
      approx (apply_bterm (bterm [nvarx] e2') [fix_approx (k - j) f])
        (apply_bterm (bterm [nvarx] e) [fix_approx k f])
```

We then instantiate above with k:=j. to get:

```
  approx (apply_bterm (bterm [nvarx] e2') [mk_bottom])
        (apply_bterm (bterm [nvarx] e) [fix_approx j f])]
```

Then LHS argument of approx in above must be a value because of what we mentioned above. So, by the lemma hasvalue_approx of the previous section, we have that the RHS argument of approx in above also is a value. Now, we instantiate our goal with n:=j to finish the proof.

```
Corollary fix_compactness_no_context: ∀ f,
  isprogram f
  → hasvalue (mk_fix f)
  → {n : nat × hasvalue (fix_approx n f)}.

Corollary fix_compactness_apply: ∀ f G,
  isprogram f
  → isprogram G
  → hasvalue (mk_apply G (mk_fix f))
  → {n : nat × hasvalue (mk_apply G (fix_approx n f))}.
```

We can now prove the LUB property similar to the way Crary did. Only the initial part of the proof is different. We have to deal with the dichotomy of fix_value2 exactly the way we did in the proof of Compactness.

After dealing with the dichotomy, the remaning argument is essentially a coinductive one. For these domain theoretic proofs, Like Crary, we make use of the fact of the fact that the computation system of Nuprl is deterministic. Howe showed that for such determinstic lazy computation systems, approx is equivalent of sqle (defined below). Hence, this proof can be done more conveniently by induction on natural numbers.

```
Lemma crary5_9: ∀ f t e,
  isprogram f
  → is_approx_ub (fun n ⇒ apply_bterm (bterm [nvarx] e) [fix_approx n f]) t
  → approx (apply_bterm (bterm [nvarx] e) [mk_fix f]) t.
```

```
Lemma crary5_9_no_context: ∀ f t,
  isprogram f
  → is_approx_ub (fun n ⇒ fix_approx n f) t
  → approx (mk_fix f) t.
```

————-begin file sqle.v ————

For a deterministic computation system, approx is equivalent to sqle. Since sqle is defined by induction on natural numbers, the proofs about approx that would otherwise need coinduction can be done more conveniently by induction on natural numbers.

```
Inductive sqle_n: nat → NTerm → NTerm → [univ] :=
 | sql0 : ∀ tl tr, isprogram tl → isprogram tr → sqle_n 0 tl tr
 | sql_add1 : ∀ n tl tr, (close_comput (sqle_n n)) tl tr
                                          → sqle_n (S n) tl tr.
```

```
Definition sqle (tl tr :NTerm) :=
  ∀ n, sqle_n n tl tr.
Lemma approx_sqle :
  ∀ a b,
    approx a b ⇔ sqle a b.
```

Chapter 4

Type System

In his thesis, Stuart Allen provided a semantics of Nuprl where types are defined as Partial Equivalence Relations (PER) [5, 6, 21]. A partial equivalence relation is a relation that is symmetric and transitive, but not necessarily reflexive. The partiality is required because the domain of these relations is the entire collection of terms, and not just the members of the type. The membership predicate for a type can be derived by just using the corresponding PER with same same elements.

This PER semantics was later extended to handle various new types that were added to Nuprl type theory since its first version CTT86 [16]. For example, the theory has been extended with intersection and union types [29], dependent intersection types [29], image types [37], partial types [17, 41, 18, 21], recursive types [35] and W types [6].

As mentioned by Allen, induction mechanisms such as the one of Coq are not enough to provide a straightforward definition of the semantics of Nuprl, where one would define typehood and member equality by mutual induction [6, Section 4.2,page 49]. One issue is that in the inductive clause that defines the dependent function types (called dependent products in Coq), the equality predicate occurs at a non-strictly-positive position. Allen suggests that the definition should however be valid, and calls the corresponding principle "half-positivity". This is what induction-recursion, as implemented in Agda, achieves [22, 23]. Instead of making that induction-recursion formal, the approach taken by Allen was to define ternary relations between types and equalities instead of simultaneously inductively defining the two notions of types and equalities.

This trick of translating a mutually inductive-recursive definition to a single inductive definition has been formally studied by Capretta [13]. He observes that this translation is problematic when the return type of the function mentions a predicative universe. Indeed, we experienced the same problem while formalizing Allen's definition in a precise meta-theory like Coq. For a fixed number n, we can only formalize n universes of Nuprl in $n+1$ universes of Coq. This is not surprising given the results of Setzer that intensional and extensional versions of various dependent type theories have same proof theoretic strength [40]. Also, because Coq does not (officially) have universe polymorphism yet, we have to replicate many of definitions for each universe. So, we verified this translation only for $n = 3$. Sec. 4.3 discusses this meta-theory.

Although we prefer to keep our definitions predicative, an alternate solution is to use `Prop`, the impredicative universe of Coq. Sec. 4.2 mainly discusses this meta-theory.

———–-begin file type_system_intro.v ————

This chapter presents our formalization of Nuprl's type system. We will first explain how induction-recursion can be used to define the type system in an intuitive way. We will then explain why such a simple definition cannot be achieved in Coq either by pure mutual induction(using the `Inductive` construct) or by pure mutual recursion(using the `Fixpoint` construct). As mentioned above, this problem is well known, and we are just adapting it to our context with a slightly different explanation. In one sentence, the problem is that the intuitive definition the evidence of typehood of an *NTerm* is an inductive one, but equality can be best understood as function that is structurally recursive on the evidence of typehood.

As a proof of concept, we show how to use induction-recursion to define the entire predicative hierarchy of universes of Nuprl in the first universe of Agda's predicative hierarchy. The use of induction-recursion to define shallow embeddings of hierarchies of Martin Lof universes have been often described in the literature [22, 34]. However, since we have a deep embedding, we have to use parametrized induction-recursion and our definitions are parametrized over (pairs of) *NTerm*. This deep approach is required for our goals of extracting an independent, correct by construction proof assistant.

Also for Nuprl, we have to simultaneously define the equality of types and equality of members in types, unlike other works where just typehood and membership are defined simultaneously. Note that two terms that are not related by cequiv can represent the same type. For example, $\lambda x.((x+1) -$

1) $=_{\mathbb{N}\to\mathbb{N}} \lambda x.x$ and $\lambda x.x =_{\mathbb{N}\to\mathbb{N}} \lambda x.((x+1)-1)$ are equal equality types, but the two terms are not related by cequiv. On the other hand, types that have the same PER are not always equal. For example, as we will see later, the equality types $0 =_{\mathbb{N}} 0$ and $1 =_{\mathbb{N}} 1$ are not equal types. Hence, the equality of types in a Nuprl universe is non-trivial and our the inductive part defines the pairs of *NTerm*s that represent equal types, instead of just defining the *NTerm*s that denote types.

Although the inductive-recursive definition is easier to understand and would enable a predicative formalization of all of Nuprl, Agda does not have a tactic language, which we heavily depend on to automate many otherwise tedious parts of proofs. So we had to accept the lack of induction-recursion in Coq and we finally settled on using Allen's trick to define the type system of Nuprl in Coq. At first, this purely inductive definition in section 4.2 might seem overly complicated. However, it can be understood as applying the generalized recipe of [13] to the inductive-recursive definition. One might want to revisit Fig. 1.1 for a summary our deep embeddings.

4.1 An Inductive Recursive Definition of Nuprl's Type System

Just for this section, we will pretend that all our definitions so far are in Agda. Given the theoretical slimilarity between Coq and Agda[1], and the fact that all our definitions are predicative, we think that it should be fairly straightforward to convert our definitions to Agda. In fact, all the definitions so far can be defined in `Set`, the first predicate universe in both Coq and Agda.

We will first show the construction of just one universe of Nuprl in Agda and then define the whole hierarchy of universes. We only define the following 3 representative types to illustrate the idea:

- the integer type
- dependent functions
- partial types
- W types

[1] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.AgdaVsCoq`

Our Agda definition is just intended as a proof of concept. It illustrates the elegance and the consistency strength of induction-recursion. As mentioned before, we have defined all the types of Nuprl in Coq because the powerful tactic machinery of Coq is critical for automating our tedious proofs.

We will assume that we had the following definitions(among others) in Agda instead of Coq.

```
Definition subst (t : NTerm) (v : NVar) (u : NTerm) : NTerm :=
    lsubst t [(v,u)].
Definition mk_lam (v : NVar) (b : NTerm) : NTerm :=
    oterm (Can NLambda) [bterm [v] b].
Definition mk_apply (f a : NTerm) : NTerm :=
    oterm (NCan NApply) [bterm [] f , bterm [] a].
Definition mk_int (z : Z ) := oterm (Can (Nint z)) [].
Definition mk_Int := oterm (Can NInt) [].
Definition mk_Uni n := oterm (Can (NUni n)) [].
Definition mk_Function (T1 : NTerm) (v : NVar) (T2 : NTerm) :=
  oterm (Can NFunction) [bterm [] T1, bterm [v] T2].
Definition mk_W (T1 : NTerm) (v : NVar) (T2 : NTerm) :=
    oterm (Can NW) [bterm [] T1, bterm [v] T2].
```

Now, we are in the Agda world. Agda is very similar to Coq. The `data` is used to define inductive types, just like the `Inductive` in Coq.

Since the type used to represent PERs would be used quite often, we first abstract it as a definition. Ideally, one would bake symmetry and transitivity into this definition, but skip that here for simplicity.

```
PER : Set₁
PER = NTerm → NTerm → Set
```

Weq wil be used later to define the equality of terms in a W type. The reader can skip this definition now and come back to it later when it is used.

```
data Weq (Aeq : PER)
    (Beq : ∀ (a a' : NTerm) (p : Aeq a a') → PER)
    (t1 t2 : NTerm) : Set where
  weq : (a1 f1 a2 f2 : NTerm)
```

```
→ (e : Aeq a1 a2)
→ (computes_to_value t1 (mk_sup a1 f1))
→ (computes_to_value t2 (mk_sup a2 f2))
→ (∀ (b1 b2 : NTerm) → Beq a1 a2 e (mk_apply f1 b1) (mk_apply f2 b2))
→ Weq Aeq Beq t1 t2
```

Among others, our definition uses the modules Data.Vec and Data.Fin of the standard library. For a number m, Fin m represents the collection of numbers less that m. The functions fromℕ and toℕ provide ways to go back and forth between Fin m and the corresponding members in ℕ. Also, given an element k of Fin m and a vector v of length m, lookup k v returns the k^{th} element of v.

We now use induction-recursion to define all universes of Nuprl. Below, three concepts are simultaneously defined using the mutual keyword. equalType inductively defines which types are equal. equalInType recursively defines which terms are equal in a type[2]. The function type-family is just a notational convenience and one could inline its definition wherever it is used in equalType. Note that there are no recursive calls in the body of type-family.

These definitions are parametrized by a number n and $iUnivs$, a vector of PERs of size n. The idea is that we have already defined the first n universes by now and the PER defined by (equalType n $iUnivs$) will serve as the equality of types in the next universe. The m^{th} member (where $m < n$) of $iUnivs$ is the already constructed PER that determines which two terms denote equal types in the m^{th} universe. Given an evidence that $T1$ and $T2$ are equal types in this universe, equalInType returns the PER of this type. Note that equalInType is structurally recursive on its argument teq.

Inuitively, the teqUNIV clause says that the types in the n^{th} universe can additionally mention the constants denoting all the universes upto (but not including) n. The corresponding clause in the definition of the function equalInType just returns the appropriate PER that was already provided in the input $iUnivs$.

```
mutual
  data equalType (n : ℕ) (iUnivs : Vec PER n) (T1 T2 : NTerm) : Set where
    teqINT : { _ : computes_to_value T1 mk_Int }
      { _ : computes_to_value T2 mk_Int}
      → (equalType n iUnivs T1 T2)
```

[2]We ignore the issue of closedness of terms here.

```
teqFUNCTION :      {A1 B1 A2 B2 : NTerm } {v1 v2 : NVar}
            { _ : computes_to_value T1 (mk_Fun A1 v1 B1)}
            { _ : computes_to_value T2 (mk_Fun A2 v2 B2)}
            (PA : equalType n iUnivs A1 A2)
            (PB : type-family n iUnivs PA v1 v2 B1 B2)
              →   (equalType n iUnivs T1 T2)

teqW : {A1 B1 A2 B2 : NTerm} {v1 v2 : NVar}
            { _ : computes_to_value T1 (mk_W A1 v1 B1)}
            { _ : computes_to_value T2 (mk_W A2 v2 B2)}
            (PA : equalType n iUnivs A1 A2)
            (PB : type-family n iUnivs PA v1 v2 B1 B2)
              → (equalType n iUnivs T1 T2)

teqPART : {A1 A2 : NTerm}
            { _ : computes_to_value T1 (mk_Part A1)}
            { _ : computes_to_value T2 (mk_Part A2)}
            (PA : equalType n iUnivs A1 A2)
            { _ : (t : NTerm)
              → equalInType n iUnivs PA t t
              → hasvalue t}
              → (equalType n iUnivs T1 T2)

teqUNIV : (m :        Fin n)
            { _ : computes_to_value T1 (mk_Univ (toℕ m))}
            { _ : (computes_to_value T2 (mk_Univ (toℕ m)))}
              → (equalType n iUnivs T1 T2)

type-family : {A1 A2 : NTerm} → (n : ℕ) (iUnivs : Vec PER n)
                → (eqA : equalType n iUnivs A1 A2) →
                (v1 v2 : NVar) (B1 B2 : NTerm) → Set
type-family n iUnivs eqA v1 v2 B1 B2 =
  (∀ (a1 a2 : NTerm)
    → equalInType n iUnivs eqA a1 a2
    → equalType n iUnivs (subst B1 v1 a1)
      (subst B2 v2 a2))
```

```
equalInType : {T1 T2 : NTerm}   (n : ℕ) (iUnivs : Vec PER n)
                       (teq : equalType n iUnivs T1 T2)
                       →   PER

equalInType n iUnivs teqINT   t1 t2 =
  ∑ ℤ (λ n
  → (computes_to_value t1 (mk_int n))
      × computes_to_value t2 (mk_int n))

equalInType n iUnivs (teqFUNCTION PA PB)   t1 t2 =
  ∑ NVar (λ v1
  → ∑ NTerm (λ b1
  → ∑ NVar (λ v2
  → ∑ NTerm (λ b2
  → computes_to_value t1 (mk_lam v1 b1)
     × computes_to_value t2 (mk_lam v2 b2)
     × (∀ (a1 a2 : NTerm)
                → (pa : equalInType n iUnivs PA a1 a2)
                → equalInType n iUnivs (PB _ _ pa) (subst b1 v1 a1)
                   (subst b2 v2 a2))))))

equalInType n iUnivs (teqW PA PB) t1 t2 =
  Weq (equalInType n iUnivs PA)
     (λ a1 a2 e → equalInType n iUnivs (PB a1 a2 e))
        t1
        t2

equalInType n iUnivs (teqPART PA) t1 t2 =
  hasvalue t1 ⇔ hasvalue t2
  × (hasvalue t1 → equalInType n iUnivs PA t1 t2)

equalInType n iUnivs (teqUNIV m) T1 T2 = lookup m iUnivs T1 T2
```

Note that once we unfold the occurence of type-family in teqFUNCTION, the second clause of equalType that defines dependent functions, it is easy to see that equalInType occurs at a negative position. Hence this definition does not fit into the schema of mutual inductive definitions. The same is true for the teqPART that constructs partial types. (The type mkPart *T* can only be constructed if the type T only has converging elements.)

Also, note that equalType cannot be defined by structural recursion. Although, *A1* and *B1* are structural subterms of mk_Fun *A1 v1 B1*), T1 is not so. In general, the size of a term can increase on comutation.

Hence, this simple definition cannot be achieved by either mutual induction, or by mutual recursion. It needs a delicate mix of induction and recursion. Also, note that the return type of both the partial equivalence relations is Set.

When one converts this definition to a single inductive definition using the recipe in [13], only finitely many universes can be defined. Informally, an inductive-recursive definition can be converted to an inductive definition by paraetrizing the inductive part over the return type of the function part. This often results in a qualtification on this return type in the constructors of the inductive type. When this return type mentions a universe (as is the case here), the quantification causes bumping up a level due to predicativity. So, (it seems that) a universe of Nuprl can only be defined in the next universe of Coq.

Now, it is easy to construct a function nuprl-univ that takes a number n and returns a PER that determines which pairs of NTerms represent equal types in the n^{th} universe of Nuprl.

We need two simple helper functions. snoc is just like the cons operator on lists, except it adds the new element at the end of a vector. Then, nuprl-univs n consructs PERs of the first n universes.

```
snoc : {A : Set₁}{n : ℕ} -> Vec A n -> A -> Vec A (Data.Nat.suc n)
snoc [] e = e :: []
snoc (x :: xs) e = x :: (snoc xs e)

nuprl-univs : (n : ℕ) → Vec PER n
nuprl-univs 0 = []
nuprl-univs (Data.Nat.suc n) = snoc (nuprl-univs n) (equalType n (nuprl-univs n))

nuprl-univ   : (n : ℕ) → PER
nuprl-univ n = lookup (fromℕ n) (nuprl-univs (Data.Nat.suc n))
```

Given that two (possibly) same types represent equal types in some universe, one can that use equalInType to get the PER (equality) of that type.

Note how were able to define all the the universes of Nuprl in just one universe of Agda. Although we did not define all the types of Nuprl, it seems extremely likely that they can be added to this definition in a similar way. This is not surprising, given that induction recursion is known to increase the proof theoretic strength of many type theories [23]

4.2 Inductive Formulation

4.2.1 Types

———-begin file per.v ————

After that brief excursion to Agda, we come back to the Coq world where there is no induction-recursion, but there is a powerful tactic language to probably more than offset the inconveniences caused by the lack of induction-recursion.

We follow Allen's PER semantics method [6] and use pure induction to define Nuprl's types as partial equivalence relations on closed terms.

A type is a partial equivalence relation that specifies which pairs of closed terms are equal in the types. Therefore, a partial equivalence relation can be seen as a type and vice-versa.

`Notation` per := (CTerm → CTerm → `[U]`).

Note that $[U]$ here can either be `Prop` or `Type`. This allows us to have a uniform notation and to easily switch between our Nuprl impredicative meta-theory that uses Coq's `Prop` where we can define Nuprl's full hierarchy of universes, and our predicative meta-theory that uses Coq's `Type` hierarchy of types to define n Nuprl universes using n+1 Coq `Type` levels. This issue will be further discussed below when defining Nuprl's universes of types. Also, because everything so far has been defined using `Type`, in the case where $[U]$ is `Prop`, we need to cast into `Prop` the types we need in order to define the Nuprl type system. For example capproxc *t1 t2* is *Cast* (approxc *t1 t2*) in the `Prop` case and approxc *t1 t2* in the `Type` case, where *Cast* casts types in `Type` into propositions in `Prop`, and where the first *c* stands for "cast". Note that we do not currently use the same type system definitions for both our meta-theories because of Coq's lack of universe polymorphism and the fact that we currently have to duplicate our code at each level of our hierarchy of Nuprl types in our predicative meta-theory. Only the sequents and rules definitions and lemmas are shared by both meta-theories. However, we anticipate that we will be able to share this code when Coq will have universe polymorphism.

We also introduce the following useful notation:

Notation "$a \preceq b$" := (capproxc a b) (at level 0).
Notation "$a \sim b$" := (ccequivc a b) (at level 0).
Notation "$a \Downarrow b$" := (ccomputes_to_valc a b) (at level 0).

We say that two term equalities are equal if they define the same relation.

```
Definition eq_term_equals (eq1 eq2 : per) :=
  ∀ t1 t2 : CTerm, eq1 t1 t2 ⇔ eq2 t1 t2.
```

```
Notation "eq1 <=2=> eq2" := (eq_term_equals eq1 eq2) (at level 0).
```

A candidate type system is a ternary relation that, intuitively, holds of the triple (T_1, T_2, R) if T_1 and T_2 are equal types and R is their equality relation. As we explain below, a type system is a *candidate-type-system* (or cts for short) that satisfies properties such as symmetry and transitivity.

```
Notation "candidate-type-system" :=
  (CTerm → CTerm → per → [U]).
```

```
Notation cts := (CTerm → CTerm → per → [U]).
```

For a cts *c*, *c T1 T2 eq* asserts that *T1* and *T2* are equal types in the type system *c* and that *eq* is the PER of this type.

For each type constructor of the form *mkc_TyCons*, we define a monotone operator *per_TyCons* on candidate type systems. Intuitively, each constructor *per_TyCons* takes a candidate type system *ts* and returns a new candidate type system where all the types are of the form *TyCons*(*T1*,...,*Tn*) where *T1*, ..., *Tn* are types from *ts* (n could be 0, as in the case of Integer type).

Once we have defined all the type constructors of the theory, we can define an operator that takes a candidate type system *ts* and builds the least fixed point of the closure of *ts* under the various type constructors of the theory. This operator is the close operator defined below, which is the μ operator defined by Allen in his thesis [6, section 4.2,page 52] and the "CLOSE" operator defined by Crary in his thesis [21, section 4.8,page 52]

The integer type is defined as follows: The two types have to compute to the integer type $\mathbb{Z}$ and two terms are equal in the integer type if they both compute to the same integer int *n* for some Coq integer n.

```
Definition per_int (ts : cts) T1 T2 (eq : per) : [U] :=
  T1 ⇓ ℤ
  × T2 ⇓ ℤ
  × ∀ t t',
      eq t t' ⇔ {n : Z , t ⇓ (int n) × t' ⇓ (int n) }.
```

The Base type as suggested by Howe in 1989 [26] is defined as follows: The two types have to compute to the closed canonical form `Base` and two terms are equal in the Base type if they are computationally equivalent.

```
Definition per_base (ts : cts) T1 T2 (eq : per) : [U] :=
  T1 ⇓ Base
  × T2 ⇓ Base
  × ∀ t t', eq t t' ⇔ t ∼ t'.
```

The approximation type was introduced in Nuprl in 2013 in order to reason about computations [39]. It is a way to reason directly in the type theory about the approximation meta-relation introduced by Howe [26] directly in the type theory. It is defined as follows: The two types have to compute to closed canonical forms of the form mkc_approx a b and mkc_approx c d where $a \preceq b$ iff $c \preceq d$, and two terms are equal in that type if they both compute to `Ax` and $a \preceq b$ is true; so these types have no computational content.

Therefore, two approximation types are equal if they are either both true or both false. We need this for technical reasons: In the end, we want to be able to prove that mkc_approx a a is true in any context. Because of the way the truth of the Nuprl sequents is formulated, this means that for any two substitutions $s1$ and $s2$ over the free variables of a, the types mkc_approx $(s1(a))$ $(s1(a))$ and mkc_approx $(s2(a))$ $(s2(a))$ have to be equal. Because our context can be arbitrary, $s1$ and $s2$ can substitute anything at all for the free variables of a. By stipulating that any two true approximation types are equal allows us to prove such equalities. The same is true about the computational equivalence type presented below.

```
Definition per_approx
             (ts : cts)
             (T1 T2 : CTerm)
             (eq : per) : [U] :=
  {a, b, c, d : CTerm
   , T1 ⇓ (mkc_approx a b)
   × T2 ⇓ (mkc_approx c d)
   × (a ⪯ b ⇔ c ⪯ d)
   × (∀ t t',
        eq t t' ⇔ (t ⇓ Ax
                   × t' ⇓ Ax
                   × a ≼ b)) }.
```

The computational equivalence type was also recently added to Nuprl, as a way to reason in the type theory about the computational equivalence meta-relation introduced by Howe [26]. It is defined as follows: The two types have to compute to closed canonical forms of the form mkc_cequiv a b and mkc_cequiv c d where $a \sim b$ iff $c \sim d$, and two terms are equal in that type if they both compute to `Ax` and $a \sim b$ is true.

```
Definition per_cequiv
            (ts : cts)
            (T1 T2 : CTerm)
            (eq : per) : [U] :=
  {a, b, c, d : CTerm
   , T1 ⇓ (mkc_cequiv a b)
   × T2 ⇓ (mkc_cequiv c d)
   × (a ~ b ⇔ c ~ d)
   × (∀ t t',
        eq t t' ⇔ (t ⇓ Ax
                   × t' ⇓ Ax
                   × a ~ b)) }.
```

We define the eqorceq eq binary relation as being either eq or $\sim$.

```
Definition eqorceq (eq : per) a b : [U] := eq a b {+} a ~ b.
```

The equality type allows one to express when two term are equal in a type directly in the type theory. The equality type is defined as follows:

- The two types have to compute to closed canonical forms of the

form mkc_equality $a1$ $a2$ A and mkc_equality $b1$ $b2$ B

- such A and B are equal in the candidate type system ts with

equality eqa,

- $a1$ and $b1$ are either equal in A or computationally

equivalent,

- $a2$ and $b2$ are also either equal in A or computationally equivalent,

- and two terms are equal in that type if they both compute to

Ax and *a1* is equal to *a2* in *A* is true.

These types have recently changed: eqorceq *eqa a1 b1* and eqorceq *eqa a2 b2* used to be *eqa a1 b1* and *eqa a2 b2*. Therefore an equality type could be well-formed only if it was true. This issue is for example discussed in section 4.1.4 of Crary's thesis [21]. This change had several interesting repercussions. For example, $\sqsubseteq$, the relations saying that one type is a subtype of another, originally had to be a primitive of the system, because it could not be defined. One could not, in general, prove that $\forall x : A.\ x \in B$ was a proposition. Now it can be defined as follows (as suggested by Nogin [36, pp. 51]): $\lambda x.x \in A \rightarrow B$, because $\lambda x.x$ is a closed term.

An another example, for any term $t \in T$, we can define the concept of begin greater than T w.r.t. approx as follows: $u \in \texttt{Base}$ is greater than t if $\exists z : \texttt{Base}.\ z = t \in T\ \wedge\ z \preceq u$, where the $t_1 \preceq t_2$ is a type only if t_1 and t_2 are in Base. This definition is especially useful to reason about partial types by allowing us to define Crary's mono() type [21].

```
Definition per_eq (ts : cts) T1 T2 (eq : per) : [U] :=
  {A, B, a1, a2, b1, b2 : CTerm
   , {eqa : per
      , T1 ⇓ (mkc_equality a1 a2 A)
      × T2 ⇓ (mkc_equality b1 b2 B)
      × ts A B eqa
      × eqorceq eqa a1 b1
      × eqorceq eqa a2 b2
      × (∀ t t',
            eq t t' ⇔ (t ⇓ Ax × t' ⇓ Ax × eqa a1 a2)) }}.
```

per_teq is an experiment. Can we have a type that represents equality between types (*tequality* defined later) as we have mkc_equality a type which allows one to reason about equalities of types.

```
Definition true_per (t t' : CTerm) := True.
```

```
Definition per_teq (ts : cts) T1 T2 (eq : per) : [U] :=
  {a1, a2, b1, b2 : CTerm
   , {eqa : per
      , T1 ⇓ (mkc_tequality a1 a2)
      × T2 ⇓ (mkc_tequality b1 b2)
      × ts a1 b1 eqa
      × ts a2 b2 eqa
      × ts a1 a2 eqa
      × eq ⇐2⇒ true_per}}.
```

We now define the concept of a type family. This is going to be useful for defining several types such as dependent functions and products (called dependents products and sums in Coq). In that definition *C* is a type constructor. It takes a closed terms which is the domain of the type, a variable *v*, and a term that does not have any free variable except *v*, i.e., it is a type family constructor. The equality *eqa* is the equality of the domain and *eqb* is the equality of the co-domain.

```
Notation "per-fam ( eqa )" :=
  (∀ a a' (p : eqa a a'), per) (at level 0).

Definition type_family TyCon (ts : cts) T1 T2 eqa eqb : [U]:=
  {A, A' : CTerm , {v, v' : NVar , {B : CVTerm [v] , {B' : CVTerm [v'] ,
     T1 ⇓ (TyCon A v B)
     × T2 ⇓ (TyCon A' v' B')
     × ts A A' eqa
     × (∀ a a', ∀ e : eqa a a',
          ts (B[v\a]) (B'[v'\a']) (eqb a a' e))}}}}.
```

Intersection types were introduced in Nuprl by Kopylov in 2004 [29]. An intersection type is a type family. Two terms *t* and *t'* are equal in an intersection type of the form $\cap x{:}A.B[z\backslash x]$ if for any two elements *a* and *a'* equal in A, *t* and *t'* are equal in $B[z\backslash a]$.

```
Definition per_isect
                 (ts : cts)
                 (T1 T2 : CTerm)
                 (eq : per) : [U] :=
  {eqa : per
   , {eqb : per-fam(eqa)
      , type_family mkc_isect ts T1 T2 eqa eqb
      × (∀ t t',
           eq t t'
             ⇔
             (∀ a a', ∀ e : eqa a a', eqb a a' e t t')) }}.
```

Dependent intersection types were invented by Kopylov in 2003 [28, 29], and were used among other things to define records, structures, and signatures. A dependent intersection type is a type family. Two terms *t* and *t'* are equal in a dependent intersection type of the form $⋒x : A.B[z\backslash x]$ if *t* and *t'* are equal in A and also in $B[z\backslash t]$.

```
Definition per_disect
             (ts : cts)
             (T1 T2 : CTerm)
             (eq : per) : [U] :=
  {eqa : per
   , {eqb : per-fam(eqa)
      , type_family mkc_disect ts T1 T2 eqa eqb
      × (∀ t t', eq t t' ⇔ {e : eqa t t' , eqb t t' e t t'}) }}.
```

Function types are also type families. Two terms *t* and *t'* are equal in a function type of the form $x{:}A \rightarrow B[z\backslash x]$ if for all *a* and *a'* equal in A, (mkc_apply *t* *a*) and (mkc_apply *t' a'*) are equal in $B[z\backslash a]$. Therefore, members of function types do not have to be lambda terms. For example, function types that have an empty domain are all inhabited by diverging terms.

```
Definition per_func (ts : cts) T1 T2 (eq : per) : [U] :=
  {eqa : per
   , {eqb : per-fam(eqa)
      , type_family mkc_function ts T1 T2 eqa eqb
      × (∀ t t',
            eq t t'
            ⇔
            (∀ a a' (e : eqa a a'),
               (eqb a a' e) (mkc_apply t a) (mkc_apply t' a')))}}.
```

We call products the types of pairs. The product type is defined as follows:

```
Definition per_product_eq
             (eqa : per)
             (eqb : per-fam(eqa))
             t t' : [U] :=
  {a, a', b, b' : CTerm
   , {e : eqa a a'
      , t ⇓ (mkc_pair a b)
      × t' ⇓ (mkc_pair a' b')
      × eqa a a'
      × eqb a a' e b b'}}.

Definition per_product
             (ts : cts)
```

```
                 (T1 T2 : CTerm)
                 (eq : per) : [U] :=
  {eqa : per
   , {eqb : per-fam(eqa)
      , type_family mkc_product ts T1 T2 eqa eqb
      × eq <=2=> (per_product_eq eqa eqb)}}.
```

Disjoint unions are defined as follows:

```
Definition per_union_eq_L (eq : per) t t' : [U] :=
  {x, y : CTerm
   , t ⇓ (mkc_inl x)
     × t' ⇓ (mkc_inl y)
     × eq x y}.

Definition per_union_eq_R (eq : per) t t' : [U] :=
  {x, y : CTerm
   , t ⇓ (mkc_inr x)
     × t' ⇓ (mkc_inr y)
     × eq x y}.

Definition per_union_eq (eqa eqb : per) (t t' : CTerm) : [U] :=
  per_union_eq_L eqa t t' {+} per_union_eq_R eqb t t'.

Definition per_union
                 (ts : cts)
                 (T1 T2 : CTerm)
                 (eq : per) : [U] :=
  {eqa, eqb : per
   , {A1, A2, B1, B2 : CTerm
      , T1 ⇓ (mkc_union A1 B1)
      × T2 ⇓ (mkc_union A2 B2)
      × ts A1 A2 eqa
      × ts B1 B2 eqb
      × eq <=2=> (per_union_eq eqa eqb)}}.
```

Image types were introduced by Kopylov and Nogin in 2006 [37]. It turns out that refinement and union types can be defined using image types. An image is defined by a type A and a function f with A as its domain, and contains any term computationally equivalent to mkc_apply f a for some a in A. Its equality is defined as the smallest partial equivalence relation such that f a and f b are equals in mkc_image A f whenever a anb b are equal in A that respects the computational equivalence relation.

```
Inductive per_image_eq (eqa : per) (f t1 t2 : CTerm) : [U] :=
| image_eq_cl :
    ∀ t,
      per_image_eq eqa f t1 t
      → per_image_eq eqa f t t2
      → per_image_eq eqa f t1 t2
| image_eq_eq :
    ∀ a1 a2,
      eqa a1 a2
      → t1 ~ (mkc_apply f a1)
      → t2 ~ (mkc_apply f a2)
      → per_image_eq eqa f t1 t2.
```

```
Definition per_image
             (ts : cts)
             (T1 T2 : CTerm)
             (eq : per) : [U] :=
  {eqa : per
   , {A1 , A2 , f1 , f2 : CTerm
      , T1 ⇓ (mkc_image A1 f1)
      × T2 ⇓ (mkc_image A2 f2)
      × ts A1 A2 eqa
      × f1 ~ f2
      × eq <=2=> (per_image_eq eqa f1)}}.
```

We introduce the concept of an extensional type family. The difference with a type family is that domains do not have to be equal anymore. Two families *C A v B* and *C A' v' B'* are equal if forall *a* in *A* there exists an *a'* in *A'* such that substc *a v B* and substc *a' v' B'* are equal, and vice-versa. This is useful to define intersection types that are slightly more extensional than the ones defined above. Extensional intersection types can be used, for example, to define computational equivalence types from approximation types.

```
Notation "per-efam ( eqa , eqa' )" :=
  (∀ a a' (e : eqa a a) (e' : eqa' a' a'), per) (at level 0).
```

```
Definition eisect_eq (eqa eqa' : per) (eqb : per-efam(eqa,eqa')) : per :=
  fun t t' ⇒
    ∀ a a' (e : eqa a a) (e' : eqa' a' a'),
      eqb a a' e e' t t'.
```

term selector `Notation "t-sel ( eqa )" :=`

```
  (∀ a (e : eqa a a), CTerm) (at level 0).
   equality selector  Notation "e-sel ( eqa , eqa' , f )" :=
  (∀ a (e : eqa a a), eqa' (f a e) (f a e)) (at level 0).
Definition etype_family
           (C : CTerm → ∀ (v : NVar), CVTerm [v] → CTerm)
           (ts : cts)
           (T1 T2 : CTerm)
           (eqa eqa' : per)
           (eqb : per-efam(eqa,eqa')) : [U] :=
  {A, A' : CTerm
   , {v, v' : NVar
   , {B : CVTerm [v]
   , {B' : CVTerm [v']
   , {f : t-sel(eqa)
   , {g : e-sel(eqa,eqa',f)
   , {f' : t-sel(eqa')
   , {g' : e-sel(eqa',eqa,f')
      , T1 ⇓ (C A v B)
      × T2 ⇓ (C A' v' B')
      × ts A A eqa
      × ts A' A' eqa'
      × (∀ a (e : eqa a a),
           ts (substc a v B)
              (substc (f a e) v' B')
              (eqb a (f a e) e (g a e)))
      × (∀ a' (e' : eqa' a' a'),
           ts (substc (f' a' e') v B)
              (substc a' v' B')
              (eqb (f' a' e') a' (g' a' e') e')) }}}}}}}}.
```

Extensional intersection types are similar to intersection types but they use the concept of extensional type families instead of type families.

An intuitive way of defining computational equivalence types from approximation types would be to define them using intersection types as follows: $a \sim b = a \preceq b \cap b \preceq a$. We need the equality between computational equivalence types to be extensional, i.e., $a \sim b$ is equal to $c \sim d$ iff $a \sim b \iff c \sim d$, i.e., $(a \preceq b \wedge b \preceq a) \iff (c \preceq d \wedge d \preceq c)$. This comes from the fact that we wish to have a rule that says that $a \sim a$ is provable in any context and from the semantics of Nuprl's sequents as discussed above. Now, if we were to define $a \sim b$ as $a \preceq b \cap b \preceq a$ then we would obtain that $a \sim b$ is equal to $c \sim d$

iff $a \preceq b \cap b \preceq a$ is equal to $c \preceq d \cap d \preceq c$. By definition of the intersection types, we would obtain that $a \sim b$ is equal to $c \sim d$ iff $a \preceq b$ is equal to $c \preceq d$ and $b \preceq a$ is equal to $d \preceq c$. For example, we want to have $a \sim b$ and $b \sim a$ be equal types. With our definition using intersection types, we would have to prove that $a \preceq b$ is equal to $b \preceq a$ and $c \preceq d$ is equal to $d \preceq c$ which are not provable in general. Therefore, we cannot define computation equivalence types using intersection types.

However, if we were to define computational equivalence types using our new extensional intersection types, we would have that $a \sim b$ is equal to $c \sim d$ iff $a \preceq b \Cap b \preceq a$ is equal to $c \preceq d \Cap d \preceq c$ (where $\Cap$ is the symbol we use for extensional intersection types, as opposed to $\cap$ which we use for "intensional intersection types"). By definition of the extensional intersection types, we would obtain that $a \sim b$ is equal to $c \sim d$ iff $a \preceq b$ is equal to $c \preceq d$ or $d \preceq c$ and $b \preceq a$ is equal to $c \preceq d$ or $d \preceq c$. With that definition, we can now easily prove, e.g., that $a \preceq b$ is equal to $b \preceq a$.

```
Definition per_eisect
             (ts : cts)
             (T1 T2 : CTerm)
             (eq : per) : [U] :=
  {eqa, eqa' : per , {eqb : per-efam(eqa,eqa')
     , etype_family mkc_eisect ts T1 T2 eqa eqa' eqb
     × eq <=2=> (eisect_eq eqa eqa' eqb) }}.
```

We now define the PER type constructor. PER types have been added to Nuprl in 2013 as a way to minimize the number of primitive type constructors needed to express the Nuprl type theory by directly defining new types as partial equivalence relations in the type theory (and not the meta-theory). Because the Nuprl types are partial equivalence relations on closed terms, the idea is that most of Nuprl's types can be defined as partial equivalence relations on Base using the PER type constructor.

The idea of introducing such a PER type constructor in the theory is not new. Stuart Allen mentions in his thesis [6, page 15] that "The set type and quotient type constructors could have been unified in a single constructor $x, y \in A/E_{x,y}$ which is like quotient except that, rather than requiring (the inhabitation of) $E_{x,y}$ to be an equivalence relation, we require only that it be transitive and symmetric over A, i.e., its restriction to A should be a partial equivalence relation. The equal members are the members of A that make $E_{x,y}$ inhabited."

We say that a term equality *R* is inhabited if there is at least one term *t* such that *R t t*. If a type *T* has equality *R* and *R* is inhabited, this means that *T* is not empty.

As specified by is_per, in the type system, a relation on terms is a partial equivalence relation if it is symmetric and transitive.

```
Definition inhabited (R : per) := { t : CTerm , R t t }.

Definition is_per (R : CTerm → CTerm → per) :=
  (∀ x y, inhabited (R x y) → inhabited (R y x))
   × (∀ x y z, inhabited (R x y)
                            → inhabited (R y z)
                            → inhabited (R x z)).
```

A PER type is defined by a binary relation (i.e., a function from terms to terms to types) on terms. Two PER types are equal if the two corresponding relations are equivalent partial equivalence relations. Two terms *t* and *t'* are equal in a PER type if they are in the corresponding relation.

```
Definition per_pertype
             (ts : cts)
             (T1 T2 : CTerm)
             (eq : per) : [U] :=
  {R1 , R2 : CTerm
   , {eq1 , eq2 : CTerm → CTerm → per
     , T1 ⇓ (mkc_pertype R1)
     × T2 ⇓ (mkc_pertype R2)
     × (∀ x y, ts (mkc_apply2 R1 x y)
                     (mkc_apply2 R1 x y)
                     (eq1 x y))
     × (∀ x y, ts (mkc_apply2 R2 x y)
                     (mkc_apply2 R2 x y)
                     (eq2 x y))
     × (∀ x y, inhabited (eq1 x y) ⇔ inhabited (eq2 x y))
     × is_per eq1
     × ∀ t t', eq t t' ⇔ inhabited (eq1 t t') }}.
```

An intensional version of per_pertype:

```
Definition pertype_eq (eq : CTerm → CTerm → per) t t' := inhabited (eq t t').

Definition per_ipertype (ts : cts) (T1 T2 : CTerm) (eq : per) :=
  {R1 , R2 : CTerm
   , {eq1 : CTerm → CTerm → per
     , ccomputes_to_valc T1 (mkc_ipertype R1)
```

```
        × ccomputes_to_valc T2 (mkc_ipertype R2)
        × (∀ x y, ts (mkc_apply2 R1 x y)
                        (mkc_apply2 R2 x y)
                        (eq1 x y))
        × is_per eq1
        × eq <=2=> (pertype_eq eq1) }}.
```

Yet another intensional version of per_pertype:

```
Definition per_spertype (ts : cts) (T1 T2 : CTerm) (eq : per) :=
  {R1, R2 : CTerm
   , {eq1 : CTerm → CTerm → per
      , ccomputes_to_valc T1 (mkc_spertype R1)
      × ccomputes_to_valc T2 (mkc_spertype R2)
      × (∀ x y,
           ts (mkc_apply2 R1 x y) (mkc_apply2 R2 x y) (eq1 x y))
      × (∀ x y z,
           inhabited (eq1 x z)
           → ts (mkc_apply2 R1 x y) (mkc_apply2 R1 z y) (eq1 x y))
      × (∀ x y z,
           inhabited (eq1 y z)
           → ts (mkc_apply2 R1 x y) (mkc_apply2 R1 x z) (eq1 x y))
      × is_per eq1
      × eq <=2=> (pertype_eq eq1) }}.
```

We define quotient types as follows:

```
Definition quot_rel_eq (eqa : per) :=
  ∀ (a1 a2 : CTerm)
        (ea : eqa a1 a2)
        (b1 b2 : CTerm)
        (eb : eqa b1 b2),
    per.

Definition per_quotient_eq
             (eqa : per)
             (eqe : quot_rel_eq eqa)
             (t t' : CTerm) :=
  {e : eqa t t , {e' : eqa t' t' , inhabited (eqe t t e t' t' e')}}.

Definition per_quotient
               (ts : cts)
               (T1 T2 : CTerm)
               (eq : per) : [U] :=
  {A1 , A2 : CTerm
   , {x1 , x2 , y1 , y2 : NVar
   , {E1 : CVTerm [x1 ,y1]
   , {E2 : CVTerm [x2 ,y2]
   , {eqa : per
   , {eqe1 , eqe2 : quot_rel_eq eqa
      , T1 ⇓ (mkc_quotient A1 x1 y1 E1)
      × T2 ⇓ (mkc_quotient A2 x2 y2 E2)
      × ts A1 A2 eqa
      × (∀ a1 a2 b1 b2 (ea : eqa a1 a2) (eb : eqa b1 b2),
           ts (lsubstc2 x1 a1 y1 b1 E1)
              (lsubstc2 x1 a2 y1 b2 E1)
              (eqe1 a1 a2 ea b1 b2 eb))
      × (∀ a1 a2 b1 b2 (ea : eqa a1 a2) (eb : eqa b1 b2),
           ts (lsubstc2 x2 a1 y2 b1 E2)
              (lsubstc2 x2 a2 y2 b2 E2)
              (eqe2 a1 a2 ea b1 b2 eb))
      × (∀ a1 a2 b1 b2 (ea : eqa a1 a2) (eb : eqa b1 b2),
           inhabited (eqe1 a1 a2 ea b1 b2 eb)
           ⇔ inhabited (eqe2 a1 a2 ea b1 b2 eb))
      × (∀ a1 a2 (ea : eqa a1 a2) (ea1 : eqa a1 a1) (ea2 : eqa a2 a2),
           inhabited (eqe1 a1 a1 ea1 a2 a2 ea2))
      × (∀ a1 a2 (ea1 : eqa a1 a1) (ea2 : eqa a2 a2),
           inhabited (eqe1 a1 a1 ea1 a2 a2 ea2)
           → inhabited (eqe1 a2 a2 ea2 a1 a1 ea1))
      × (∀ a1 a2 a3 (ea1 : eqa a1 a1) (ea2 : eqa a2 a2) (ea3 : eqa a3 a3),
           inhabited (eqe1 a1 a1 ea1 a2 a2 ea2)
           → inhabited (eqe1 a2 a2 ea2 a3 a3 ea3)
           → inhabited (eqe1 a1 a1 ea1 a3 a3 ea3))
      × eq ⇐2⇒ (per_quotient_eq eqa eqe1) }}}}}}.
```

Refinement types (also called set or subset types) are defined as follows (note that Crary [21] as a slightly more extensional definition, but here we chose to define refinment types as they are currently implemented in Nuprl):

```
Definition per_set
                (ts : cts)
                (T1 T2 : CTerm)
                (eq : per) : [U] :=
  {eqa : per
   , {eqb : per-fam(eqa)
```

```
      , type_family mkc_set ts T1 T2 eqa eqb
      × (∀ t t', eq t t' ⇔ {e : eqa t t' , inhabited (eqb t t' e)})}}.
```

We define partial types as they are introduced in Crary's thesis [21]. Partial types were invented to provide meaning to partial functions. Intuitively, if *eqa* is the equality of the type *A* per_partial_eq *eqa* gives the equality of the partial type *mk_partial T*

```
Definition per_partial_eq (eqa : per) (t t' : CTerm) : [U] :=
  (chaltsc t ⇔ chaltsc t')
  × (chaltsc t → eqa t t').

Definition per_partial (ts : cts) T1 T2 (eq : per) : [U] :=
  {A1 , A2 : CTerm , {eqa : per
      , T1 ⇓ (mkc_partial A1)
      × T2 ⇓ (mkc_partial A2)
      × ts A1 A2 eqa
      × (∀ a, eqa a a → chaltsc a)
      × eq ⇐2⇒ (per_partial_eq eqa) }}.
```

One of the main ways to prove membership in a partial type is the fixpoint induction rule. Intuitively, it says that *mk_fix f* is is a member of *mk_partial T* whenever *f* is in *mk_partial T* → *mk_partial T* . This rules only does not hold for all types. Crary introduced 2 sufficient conditions to characterize the types for which this property holds. He reflects these conditions to the type theory by defining two types *mk_mono T* and *mk_admiss T* which assert that the type *T* has the above mentioned property. *mk_mono T* is more intuitive, but weaker. We will prove that *mk_mono T* implies *mk_admiss T*. We will first define the closed version of finite approximations of mkc_fix *f*. fix_approxc is similar to *fix_approx* that we defined in 3.3

```
Fixpoint fix_approxc (n : nat) (f : CTerm) : CTerm :=
match n with
| 0 ⇒ mkc_bot
| S n ⇒ mkc_apply f (fix_approxc n f)
end.

Definition subst_fapproxc {v:NVar}
  (e : CVTerm [v]) (f : CTerm) (n : nat) : CTerm :=
substc (fix_approxc n f) v e.

Definition subst_fixc {v:NVar}
  (e : CVTerm [v]) (f : CTerm) : CTerm :=
substc (mkc_fix f) v e.
```

First, we define the Crary's more inuitive Mono-hood property. Intuitively, a type T is a Mono type if its equality satisfies the following condition. The following predicate charecterizes the PERs of types that are Mono

```
Definition mono_equality (eq : per) :=
  ∀ (t t' : CTerm),
    eq t t
    → approxc t t'
    → eq t t'.
```

Crary the defined a stronger condition called admissibility. It is defined below

```
Definition per_mono_eq (eqa : per) (t t' : CTerm) : [U] :=
  t ⇓ Ax
  × t' ⇓ Ax
  × mono_equality eqa.

Definition per_mono
                (ts : cts)
                (T1 T2 : CTerm)
                (eq : per) : [U] :=
  {A1 , A2 : CTerm ,
    {eqa : per ,
        T1 ⇓ (mkc_mono A1)
        × T2 ⇓ (mkc_mono A2)
        × ts A1 A2 eqa
        × eq ⇐2⇒ (per_mono_eq eqa)}}.

Definition cofinite_subst_fapprox_eqc
               (eq : per)
               {v: NVar}
               (e e' : CVTerm [v])
               (f : CTerm) :=
    {j : nat
     , ∀ (k :nat),
          k>j → eq (subst_fapproxc e f k) (subst_fapproxc e' f k)}.

Definition subst_fix_eqc
               (eq : per)
               {v: NVar}
               (e e' : CVTerm [v])
               (f : CTerm) :=
  eq (subst_fixc e f) (subst_fixc e' f).
```

```
Definition admissible_equality (eq : per) :=
  ∀ v (e e' : CVTerm [v]) (f : CTerm),
    cofinite_subst_fapprox_eqc eq e e' f
    → subst_fix_eqc eq e e' f.

Definition per_admiss_eq (eqa : per) (t t' : CTerm) : [U] :=
  t ⇓ Ax
  × t' ⇓ Ax
  × admissible_equality eqa.

Definition per_admiss
             (ts : cts)
             (T1 T2 : CTerm)
             (eq : per) : [U] :=
  {A1 , A2 : CTerm ,
    {eqa : per ,
        T1 ⇓ (mkc_admiss A1)
      × T2 ⇓ (mkc_admiss A2)
      × ts A1 A2 eqa
      × eq <=2=> (per_admiss_eq eqa)}}.
```

W types are defined using the `Inductive` feature of Coq.

```
Inductive weq
            (eqa : per)
            (eqb : per-fam(eqa))
            (t t' : CTerm) : [U] :=
| weq_cons :
    ∀ a f a' f' : CTerm,
    ∀ e : eqa a a',
      t ⇓ (mkc_sup a f)
      → t' ⇓ (mkc_sup a' f')
      → (∀ b b',
            eqb a a' e b b'
            → weq eqa eqb (mkc_apply f b) (mkc_apply f' b'))
      → weq eqa eqb t t'.

Definition per_w (ts : cts) T1 T2 (eq : per) : [U] :=
  {eqa : per , {eqb : per-fam(eqa) ,
   type_family mkc_w ts T1 T2 eqa eqb
   × eq <=2=> (weq eqa eqb)}}.
```

M types [43, 3] are defined using the `CoInductive` feature of Coq.

```
CoInductive meq
```

```
                (eqa : per)
                (eqb : per-fam(eqa))
                (t t' : CTerm) : [U] :=
| meq_cons :
     ∀ a f a' f' : CTerm,
     ∀ e : eqa a a',
        t ⇓ (mkc_sup a f)
        → t' ⇓ (mkc_sup a' f')
        → (∀ b b',
                eqb a a' e b b'
                → meq eqa eqb (mkc_apply f b) (mkc_apply f' b'))
        → meq eqa eqb t t'.

Definition per_m
                    (ts : cts)
                    (T1 T2 : CTerm)
                    (eq : per) : [U] :=
   {eqa : per
    , {eqb : per-fam(eqa)
        , type_family mkc_m ts T1 T2 eqa eqb
        × eq ⇐2⇒ (meq eqa eqb)}}.
```

We now define the concept of parameterized type families, which we use to define parameterized W and M types. In a parameterized W type of the form mkc_pw *P ap A bp ba B cp ca cb B p*: *P* is the parameter type; *p* is the initial parameter; the *A* describes the non inductive parts of the constructors of the corresponding inductive type, and might depend on the parameter (*ap*); the *B* part describes the inductive parts of the constructors of the corresponding inductive type, and might depend on the parameter (*bp*) and a *A* (*ba*); finally, *C* describes how to build the parameters for each of the inductive calls, and might depend on a *P*, an *A*, and a *B*.

```
Definition pfam_type :=
   ∀ P : CTerm,
   ∀ ap : NVar,
   ∀ A : CVTerm [ap],
   ∀ bp ba : NVar,
   ∀ B : CVTerm [bp,ba],
   ∀ cp ca cb : NVar,
   ∀ C : CVTerm [cp,ca,cb],
   ∀ p : CTerm,
      CTerm.
```

```
Notation "per-fam-fam ( eqp , eqa )" :=
  (∀ p p' : CTerm,
   ∀ ep : eqp p p',
   ∀ a a' : CTerm,
   ∀ ea : eqa p p' ep a a',
     per)
    (at level 0).

Definition type_pfamily
             (Cons : pfam_type)
             (ts : cts)
             (T1 T2 : CTerm)
             (eqp : per)
             (eqa : per-fam(eqp))
             (eqb : per-fam-fam(eqp,eqa))
             (cp1 ca1 cb1 : NVar)
             (C1 : CVTerm [cp1;ca1;cb1])
             (cp2 ca2 cb2 : NVar)
             (C2 : CVTerm [cp2;ca2;cb2])
             (p1 p2 : CTerm): [U] :=
  {P1, P2 : CTerm
   , {ap1, ap2 : NVar
   , {A1 : CVTerm [ap1]
   , {A2 : CVTerm [ap2]
   , {bp1, bp2 : NVar
   , {ba1, ba2 : NVar
   , {B1 : CVTerm [bp1, ba1]
   , {B2 : CVTerm [bp2, ba2]
      , T1 ⇓ (Cons P1 ap1 A1 bp1 ba1 B1 cp1 ca1 cb1 C1 p1)
      × T2 ⇓ (Cons P2 ap2 A2 bp2 ba2 B2 cp2 ca2 cb2 C2 p2)
      × ts P1 P2 eqp
      × (∀ p1 p2,
          ∀ ep : eqp p1 p2,
            ts (substc p1 ap1 A1) (substc p2 ap2 A2) (eqa p1 p2 ep))
      × (∀ p1 p2,
          ∀ ep : eqp p1 p2,
          ∀ a1 a2,
          ∀ ea : eqa p1 p2 ep a1 a2,
            ts (lsubstc2 bp1 p1 ba1 a1 B1)
               (lsubstc2 bp2 p2 ba2 a2 B2)
               (eqb p1 p2 ep a1 a2 ea))
```

```
        × (∀ p1 p2,
           ∀ ep : eqp p1 p2,
           ∀ a1 a2,
           ∀ ea : eqa p1 p2 ep a1 a2,
           ∀ b1 b2,
           ∀ eb : eqb p1 p2 ep a1 a2 ea b1 b2,
             eqp (lsubstc3 cp1 p1 ca1 a1 cb1 b1 C1)
                 (lsubstc3 cp2 p2 ca2 a2 cb2 b2 C2))
        × eqp p1 p2
         }}}}}}}}.
```

Using the concept of parameterized type families, we define parameterized W types as follows:

```
Inductive pweq
                (eqp : per)
                (eqa : per-fam(eqp))
                (eqb : per-fam-fam(eqp,eqa))
                (cp ca cb : NVar)
                (C : CVTerm [cp;ca;cb])
                (p : CTerm)
                (t1 t2 : CTerm) : [U] :=
| pweq_cons :
    ∀ ep : eqp p p,
    ∀ a1 f1 a2 f2 : CTerm,
    ∀ ea : eqa p p ep a1 a2,
      t1 ⇓ (mkc_sup a1 f1)
      → t2 ⇓ (mkc_sup a2 f2)
      → (∀ b1 b2,
            eqb p p ep a1 a2 ea b1 b2
            → pweq eqp eqa eqb
                       cp ca cb C
                       (lsubstc3 cp p ca a1 cb b1 C)
                       (mkc_apply f1 b1)
                       (mkc_apply f2 b2))
      → pweq eqp eqa eqb cp ca cb C p t1 t2.

Definition per_pw
                  (ts : cts)
                  (T1 T2 : CTerm)
                  (eq : per) : [U] :=
  {eqp : per
, {eqa : per-fam(eqp)
, {eqb : per-fam-fam(eqp,eqa)
, {p1 , p2 : CTerm
, {cp1 , cp2 , ca1 , ca2 , cb1 , cb2 : NVar
, {C1 : CVTerm [cp1 , ca1 , cb1]
, {C2 : CVTerm [cp2 , ca2 , cb2]
   , type_pfamily mkc_pw ts T1 T2 eqp eqa eqb
                    cp1 ca1 cb1
                    C1
                    cp2 ca2 cb2
                    C2
                    p1 p2
   × eq ⇐2⇒ (pweq eqp eqa eqb cp1 ca1 cb1 C1 p1)
    }}}}}}}.
```

Let us now illustrate how one can use parametrized W types to define parametrized inductive types such as the following vector type:

```
Inductive vec (T : Type) (n : nat) :=
| vnil : n = 0 → vec T n
| vcons : 0 < n → T → vec T (n - 1) → vec T n.
```

The P part is the type of parameters. For our vec example, it is the product of a type and a nat.

```
Definition pw_vec_P i := mk_prod (mk_uni i) mk_tnat.
```

The A part is the type of labels. For our vec example, a label is either (1) a proof that the natural number in the current parameter is 0, or (2) a pair of a proof that the natural number in the current parameter is less than 0 and the type in the current parameter.

```
Definition pw_vec_A (p : NVar) :=
  mk_spread
    (mk_var p) nvara nvarb
    (mk_union (mk_equality (mk_var nvarb) mk_zero mk_tnat)
                   (mk_prod (mk_less_than mk_zero (mk_var nvarb))
                                (mk_var nvara))).
```

The B part is the part that specifies the recursive calls. For our vec example, in the vnil branch there is no recursive calls therefore we return mk_void, and in the vcons branch there is one recursive call therefore we return mk_unit.

```
Definition pw_vec_B (p a : NVar) :=
  mk_decide (mk_var a) nvarx mk_void nvary mk_unit.
```

The C part provide the parameters from the recursive calls. For our vec example, in the vnil branch because there is no recursive calls, we can return anything, and in the vcons branch, we have to build one parameter from the current parameter: the type does not change and we subtract 1 to the nat.

```
Definition pw_vec_C (p a b : NVar) :=
  mk_spread
    (mk_var p) nvara nvarb
    (mk_decide
       (mk_var a)
       nvarx (mk_var p)
       nvary (mk_pair (mk_var nvara) (mk_sub (mk_var nvarb) mk_one))
    ).
```

We now define vec as follow where i is T's level.

```
Definition pw_vec i T n :=
  mk_pw
    (pw_vec_P i)
    nvarx (pw_vec_A nvarx)
    nvary nvarz (pw_vec_B nvary nvarz)
    nvara nvarb nvarc (pw_vec_C nvara nvarb nvarc)
    (mk_pair T n).
```

Using the concept of parameterized type families, we define parameterized M types as follows:

```
CoInductive pmeq
              (eqp : per)
              (eqa : per-fam(eqp))
              (eqb : per-fam-fam(eqp,eqa))
              (cp ca cb : NVar)
              (C : CVTerm [cp;ca;cb])
              (p : CTerm)
              (t1 t2 : CTerm) : [U] :=
| pmeq_cons :
    ∀ ep : eqp p p,
    ∀ a1 f1 a2 f2 : CTerm,
    ∀ ea : eqa p p ep a1 a2,
      t1 ⇓ (mkc_sup a1 f1)
      → t2 ⇓ (mkc_sup a2 f2)
      → (∀ b1 b2,
               eqb p p ep a1 a2 ea b1 b2
               → pmeq eqp eqa eqb
                       cp ca cb C
                       (lsubstc3 cp p ca a1 cb b1 C)
                       (mkc_apply f1 b1)
                       (mkc_apply f2 b2))
          → pmeq eqp eqa eqb cp ca cb C p t1 t2.
```

```
Definition per_pm
               (ts : cts)
               (T1 T2 : CTerm)
               (eq : per) : [U] :=
  {eqp : per
   , {eqa : per-fam(eqp)
   , {eqb : per-fam-fam(eqp,eqa)
   , {p1 , p2 : CTerm
   , {cp1 , cp2 , ca1 , ca2 , cb1 , cb2 : NVar
   , {C1 : CVTerm [cp1 , ca1 , cb1]
   , {C2 : CVTerm [cp2 , ca2 , cb2]
      , type_pfamily mkc_pm ts T1 T2 eqp eqa eqb
                     cp1 ca1 cb1
                     C1
                     cp2 ca2 cb2
                     C2
                     p1 p2
      × eq ⇐2⇒ (pmeq eqp eqa eqb cp1 ca1 cb1 C1 p1)
      }}}}}}}.
```

The close operator takes a candidate type system and returns a candidate type system closed under the type constructor defined above (except the ones for extensional intersection types and quotient types which are not yet in the system because we have not yet had time to prove that these types preserve the type system properties presented below).

```
Inductive close (ts : cts) (T T' : CTerm) (eq : per) : [U] :=
  | CL_init : ts T T' eq → close ts T T' eq
  | CL_int : per_int (close ts) T T' eq → close ts T T' eq
  | CL_base : per_base (close ts) T T' eq → close ts T T' eq
  | CL_approx : per_approx (close ts) T T' eq → close ts T T' eq
  | CL_cequiv : per_cequiv (close ts) T T' eq → close ts T T' eq
  | CL_eq : per_eq (close ts) T T' eq → close ts T T' eq
  | CL_teq : per_teq (close ts) T T' eq → close ts T T' eq
  | CL_isect : per_isect (close ts) T T' eq → close ts T T' eq
  | CL_func : per_func (close ts) T T' eq → close ts T T' eq
  | CL_disect : per_disect (close ts) T T' eq → close ts T T' eq
  | CL_pertype : per_pertype (close ts) T T' eq → close ts T T' eq
  | CL_ipertype : per_ipertype (close ts) T T' eq → close ts T T' eq
  | CL_spertype : per_spertype (close ts) T T' eq → close ts T T' eq
  | CL_w : per_w (close ts) T T' eq → close ts T T' eq
  | CL_m : per_m (close ts) T T' eq → close ts T T' eq
  | CL_pw : per_pw (close ts) T T' eq → close ts T T' eq
  | CL_pm : per_pm (close ts) T T' eq → close ts T T' eq
  | CL_union : per_union (close ts) T T' eq → close ts T T' eq
  | CL_image : per_image (close ts) T T' eq → close ts T T' eq

  | CL_partial : per_partial (close ts) T T' eq → close ts T T' eq
  | CL_admiss : per_admiss (close ts) T T' eq → close ts T T' eq
  | CL_mono : per_mono (close ts) T T' eq → close ts T T' eq
  | CL_set : per_set (close ts) T T' eq → close ts T T' eq
  | CL_product : per_product (close ts) T T' eq → close ts T T' eq.
```

4.2.2 Universes

———-begin file nuprl.v ————

We now define Nuprl's universes of types, the Nuprl type system and various useful abstractions such as the equality meta-theoretical relation which expresses when two terms are equal in a type.

univi *i* is a candidate type system that contains all the types at level i. univi 0 is the empty type system. univi 1 contains all the types that do not mention universes. univi 2 contains all the types of univi 1 plus all the types that mention universes no higher than $\mathbb{U}$ 0.... Two types *A* and *A'* are equal in a universe at level S *i* if there exists an equality *eqa* such that *A* and *A'* are equal with PER *eqa* in the closed type system close (univi *i*). For simplicity, the universe univi (S *i*) is denoted by $\mathbb{U}$ *i*.

```
Fixpoint univi (i : nat) (T T' : CTerm) (eq : per) : [U] :=
  match i with
  | 0 ⇒ False
  | S n ⇒
    (T ⇓ (𝕌 n)
    × T' ⇓ (𝕌 n)
    × ∀ A A',
        eq A A' ⇔ {eqa : per , close (univi n) A A' eqa})
  {+} univi n T T' eq
end.
```

Even though we can define univi in Type as a fixpoint, this definition is not useful to define more than one universe. As a matter of fact, we can prove prove that 𝕌 0 is a member of 𝕌 1, but we cannot prove that 𝕌 *i* is a member of 𝕌 (S *i*) when *i* is greater than 0. Intuitively this is because we cannot fit all the universes of Nuprl into one universe of Coq, and univi *i* cannot be at different levels for different *i*s.

More precisely, the problem in univi (S *i*)'s definition is that if *eqa* is a relation at the Coq level l then univi (S *i*) is at least at level $l+1$, and in general to be able to prove the double implication, *eq* must also be at level $l+1$. However, because *eqa* is at level l, in the recursive call, in the case where close (univi *i*) *A A' eqa* is univi *i A A' eqa*, we end up having to prove that our relation at level l is equivalent to a relation at level $l+1$, which in general is not possible.

One alternative approach is to put univi in Prop instead of Type. Another approach is to define instead a finite number of universes and add more as needed.

The univ operator is a candidate type system that contains all the universes.

```
Definition univ (T T' : CTerm) (eq : per) :=
  {i : nat , univi i T T' eq}.
```

We say that a candidate type system defines only universes if all its types are of the forms 𝕌 *i*. For example, we can prove that univ defines only universes.

```
Definition defines_only_universes (ts : candidate-type-system) :=
  ∀ T eq, ts T T eq → {i : nat , T ⇓ (𝕌 i)}.
```

Finally, the Nuprl type system is defined by closing the univ candidate type system.

```
Definition nuprl := close univ.
```

The predicate tequality *T1 T2* is true if *T1* and *T2* are equal types of the Nuprl type system.

```
Definition tequality T1 T2 :=
  { eq : per , nuprl T1 T2 eq }.
```

The predicate type T is true if T is a type of the Nuprl type system.

```
Definition type T := tequality T T.
```

The predicate equality $t1$ $t2$ T is true if $t1$ and $t2$ are equal members of the type T.

```
Definition equality t1 t2 T :=
  { eq : per , nuprl T T eq × eq t1 t2 }.
```

member t T is true if t is a member of the type T.

```
Definition member t T := equality t t T.
```

The predicate tequalityi i $T1$ $T2$ is true if $T1$ and $T2$ are equal types of the Nuprl type system at level i.

```
Definition tequalityi i T1 T2 := equality T1 T2 (𝕌 i).
```

The predicate typei i T is true if T is a type of the Nuprl type system at level i.

```
Definition typei i T := tequalityi i T T.
```

Because it is sometime useful to remember the levels of types, we define the following abstractions. When we do not want to remember levels we will use tequality and when we want to remember levels, we will use tequalityi. Therefore, we use eqtypes nolvl when we do not to remember levels and we use eqtypes (atlvl i) we we do want to remember levels.

```
Inductive lvlop : Type :=
| nolvl : lvlop
| atlvl : nat → lvlop.

Definition eqtypes lvl T1 T2 :=
  match lvl with
    | nolvl ⇒ tequality T1 T2
    | atlvl l ⇒ tequalityi l T1 T2
  end.

Definition ltype l T := eqtypes l T T.
```

4.2.3 Type system

————-begin file type_sys.v ————

A term equality is a partial equivalence relation if it is symmetric and transitive. It also has to respect computation.

```
Definition term_equality_symmetric (eq : per) :=
  ∀ t1 t2, eq t1 t2 → eq t2 t1.

Definition term_equality_transitive (eq : per) :=
  ∀ t1 t2 t3, eq t1 t2 → eq t2 t3 → eq t1 t3.

Definition term_equality_respecting (eq : per) :=
  ∀ t t', eq t t → t ∼ t' → eq t t'.
```

We now define the concept of a type system. A candidate type system is a type system if it satisfies the following properties:

- uniquely valued
- type extensionality
- type symmetric
- type transitive
- type value respecting
- term symmetric
- term transitive
- term value respecting

One difference with the way, e.g., the way Crary defines the value respecting properties [21, section 4.4.1,definition 4.8,page 52] is that he uses the computation relation while we have to use the computational equivalence relation.

Here is the reason: In the case of equality types, we have to prove that close cts satisfies the type system properties. Assuming that *per_eq* cts *T* *T'* *eq* is true, $T \Downarrow$ (mkc_equality *a1* *a2* *A*), $T' \Downarrow$ (mkc_equality *b1* *b2* *B*), and *eqa* is the equality relation of A, and such that close cts satisfies the type system properties on A, B, and *eqa*.

In the type_symmetric case we have to prove *per_eq* cts *T'* *T* *eq*. We can trivially prove most of the clauses of *per_eq*. The non-trivial one is proving the equivalence between *eq* *t* *t'* and $t \Downarrow \texttt{Ax} \times t' \Downarrow \texttt{Ax} \times$ *eqa* *b1* *b2* for all *t* and *t'*, which we have to prove using the clause we get from *per_eq* cts *T T'* *eq*, i.e., that *eq t t'* is equivalence to and $t \Downarrow$ `Ax` $\times$ $t' \Downarrow$ `Ax` $\times$ *eqa a1 a2* for all *t* and *t'*. Therefore we have to prove that *eqa a1 a2* iff *eqa b1 b2* knowing that eqorceq *eqa a1 b1* and eqorceq *eqa a2 b2*. If *a1* $\sim$ *b1* or *a2* $\sim$ *b2* then to prove that double implication, we need to know that *A* and *B* are closed not only under computation but also under computational equivalence.

```
Definition uniquely_valued (ts : cts) :=
  ∀ T T' eq eq',
        ts T T' eq → ts T T' eq' → eq ⇐2⇒ eq'.

Definition type_extensionality (ts : cts) :=
  ∀ T T' eq eq', ts T T' eq → eq ⇐2⇒ eq' → ts T T' eq'.

Definition type_symmetric (ts : cts) :=
  ∀ T T' eq, ts T T' eq → ts T' T eq.

Definition type_transitive (ts : cts) :=
  ∀ T1 T2 T3 eq, ts T1 T2 eq → ts T2 T3 eq → ts T1 T3 eq.

Definition type_value_respecting (ts : cts) :=
  ∀ T T' eq, ts T T eq → cequivc T T' → ts T T' eq.

Definition term_symmetric (ts : cts) :=
  ∀ T T' eq, ts T T' eq → term_equality_symmetric eq.

Definition term_transitive (ts : cts) :=
  ∀ T T' eq, ts T T' eq → term_equality_transitive eq.

Definition term_value_respecting (ts : cts) :=
  ∀ T eq, ts T T eq → term_equality_respecting eq.

Definition type_system (ts : cts) : Type :=
  uniquely_valued ts
   × type_extensionality ts
   × type_symmetric ts
   × type_transitive ts
   × type_value_respecting ts
   × term_symmetric ts
   × term_transitive ts
   × term_value_respecting ts.
```

————-begin file close_type_sys.v ————

It is then tedious but fairly straightforward to prove that close preserves the type_system property. We currently have prove that the integer, base, approximation, computational equivalence, equality, intersection, function, dependent intersection, PER, W, parameterized W, M, refinement, partial, admissibility, disjoint union, image, and product types preserve the type system properties. It remains to prove that the parameterized M and extensional intersection types preserves the type system properties. This means that we cannot use these types yet. This is why we have not yet added these types to the type system.

```
Lemma close_type_system :
  ∀ ts : candidate-type-system,
    type_system ts
    → defines_only_universes ts
    → type_system (close ts).
```

———-begin file nuprl_type_sys.v ———

We prove that all the Nuprl universes satisfy the type system properties.

```
Lemma univi_type_system :
  ∀ i : nat, type_system (univi i).
```

We prove that that univ satisfies the type system properties.

```
Lemma univ_type_system : type_system univ.
```

Finally, we prove that that nuprl satisfies the type system properties.

```
Lemma nuprl_type_system : type_system nuprl.
```

———-begin file per_props.v ———

Using the type definitions we can prove various useful equivalences that are true about the Nuprl type system nuprl. These equivalences provide characterizations of our types. For example, we can prove that two terms *t1* and *t2* are equal in a type *T* if and only if the type mkc_equality *t1* *t2* *T* is inhabited by `Ax`. This equivalence shows the relations between the equality meta-relation and the mkc_equality type.

```
Lemma member_equality_iff :
  ∀ t1 t2 T,
    equality t1 t2 T
    ⇔ member Ax (mkc_equality t1 t2 T).
```

As another example, we can prove that two terms *f* and *g* are equal in the function type mkc_function *A* *v* *B* if and only if *A* is a type, *B* is functional over A, and forall a and a' equal in A, mkc_apply f a and mkc_apply g a' are equals in substc a v B.

This is one of the lemmas where we need the *FunctionalChoice_on* axiom. Let us explain that issue. Let us assume that we want to prove the left-hand-side of ⇔ from its right-hand-side. To prove that f and g are equal in mkc_function A v B, we have to provide the equality of the function type, and in particular, we have to provide the equality of its co-domain. We obtain that equality from the third clause in the right-hand-side of the ⇔. However, in our current `Prop` metatheory, that clause is (roughly) a ∀ of a propositional ∃. From such a formula, we need to build a per-*fam* function (the equality of the co-domain), which is exactly what *FunctionalChoice_on* gives us. This axiom is necessary because in general we cannot access the witness of a propositional existential.

```
Lemma equality_in_function :
  ∀ f g A v B,
    equality f g (mkc_function A v B)
    ⇔
    (type A
     × (∀ a a', equality a a' A → tequality (substc a v B) (substc a' v B))
     × (∀ a a',
           equality a a' A
           → equality (mkc_apply f a) (mkc_apply g a') (substc a v B))).
```

As we proved a relation between the equality relation and the mkc_equality type, we can prove the followig result that relates the computational equality relation to the mkc_cequiv type.

```
Lemma member_cequiv_iff :
  ∀ t1 t2,
    ccequivc t1 t2
    ⇔ member Ax (mkc_cequiv t1 t2).
```

Using the Coq induction principle we obtain for `weq`, we can prove the following induction principle for our W types. The then use this principle to prove the *rule_w_induction* rule below.

```
Lemma w_ind_eq :
  ∀ A va B (Q : CTerm → CTerm → [U]),
    (∀ t1 t2 t3 t4, cequivc t1 t3 → cequivc t2 t4 → Q t1 t2 → Q t3 t4)
    → (∀ a1 a2 f1 f2,
           equality a1 a2 A
```

```
          → equality f1 f2 (mkc_fun (substc a1 va B) (mkc_w A va B))
          → (∀ b1 b2,
                equality b1 b2 (substc a1 va B)
                → Q (mkc_apply f1 b1) (mkc_apply f2 b2))
          → Q (mkc_sup a1 f1) (mkc_sup a2 f2))
    → (∀ w1 w2, equality w1 w2 (mkc_w A va B) → Q w1 w2).
```

4.3 Predicative Metatheory

———-begin file nuprl_fin.v ————

Let us now describe our predicative metatheory. In this metatheory, we define the PERs in `Type` instead of `Prop`. One limitation is that now we can only model a finite model of universes because each time we define a new Nuprl universe it makes the Coq universe in which it lives in rise w.r.t. the previous universe.

As in our `Prop` metatheory, we define a per as a binary relation on CTerms, where $[U]$ is now `Type` instead of `Prop`.

We also define the concepts of PERs of families and families of families.

```
Notation "term-equality" := (CTerm → CTerm → [U]).

Notation per := (CTerm → CTerm → [U]).

Notation "term-equality-fam ( eqa )" :=
  (∀ (a a' : CTerm) (p : eqa a a'), term-equality) (at level 0).

Notation "per-fam ( eqa )" :=
  (∀ (a a' : CTerm) (p : eqa a a'), per) (at level 0).

Notation "term-equality-fam-fam ( eqp , eqa )" :=
  (∀ (p p' : CTerm) (ep : eqp p p')
            (a a' : CTerm) (ea : eqa p p' ep a a'), term-equality)
    (at level 0).

Notation "per-fam-fam ( eqp , eqa )" :=
  (∀ (p p' : CTerm) (ep : eqp p p')
            (a a' : CTerm) (ea : eqa p p' ep a a'), per)
    (at level 0).
```

A candidate type system is defined as in the `Prop` metatheory, where $[U]$ is now `Type` instead of `Prop`.

```
Notation "candidate-type-system" := (CTerm → CTerm → per → [U]).
Notation cts := (CTerm → CTerm → per → [U]).
```

We now define the PERs of the types of our type system. Unfortunately, because of the lack of universe polymorphism in Coq, we have to duplicate each PER definition for each level. We do not show these definitions here as they are the same as the ones we use in our Prop metatheory.

We now define 3 levels of universes of types. We start by defining univ0 the empty type system. We close that universe using close0 (a copy of the *close* operator we use in our Prop metatheory, where we removed a few types in order for the code duplication to be slightly more manageable: we removed parametrized W, union, image, partial, admissible and mono types) to obtain our first universe of types. This first universe of types cannot mention universes.

```
Definition univ0 (T T' : CTerm) (eq : per) : Type := False.
Definition cuniv0 := close0 univ0.
```

Then, we define univ_eq1, the PER of our first universe. Using this PER, we define at_univ_def1, the PER of mkc_uni 0 which is a type that denotes our first universe. Next, we define univ1 which is nothing more than a simple wrapper around at_univ_def1. It is defined as either at_univ_def1 or univ0. In this case, because univ0 is empty, we could have omitted the second branch. However, such a second branch is crucial in higher universes to model the cumulativity of universes. Therefore, in order to use the same scheme to define all our universes, we keep univ1's second branch. The lifting operator used in that second branch, will be better explained at the next level. We close univ1 using close1 (yet another copy of the *close* operator we use in our Prop metatheory) to obtain our second universe of types. This second universe of types can only mention the universe mkc_uni 0.

```
Inductive univ_eq1 (A A' : CTerm) :=
| ueq1 : ∀ eqa : per, cuniv0 A A' eqa → univ_eq1 A A'.
Inductive at_univ_def1 (T T' : CTerm) (eq : per) :=
| aud1 :
  computes_to_valc T (mkc_uni 0)
  → computes_to_valc T' (mkc_uni 0)
  → (∀ A A', eq A A' ⇔ univ_eq1 A A')
  → at_univ_def1 T T' eq.
Inductive Lift0 : per → per :=
| lift0 : ∀ (eq : per) t1 t2, eq t1 t2 → Lift0 eq t1 t2.
Inductive univ1 : CTerm → CTerm → per → Type :=
| u1_1 : ∀ T T' eq, at_univ_def1 T T' eq → univ1 T T' eq
| u1_0 : ∀ T T' eq eq', eq_term_equals1 eq (Lift0 eq') → univ0 T T' eq' → univ1
T T' eq.
Definition cuniv1 := close1 univ1.
```

Then we define univ_eq2, the PER of our second universe. Using this PER, we define at_univ_def2, the PER of mkc_uni 1 which is a type that denotes our second universe. Next, we define univ2 which is defined as either at_univ_def2 or as univ1. As mentioned above, this is necessary to model the cumulativity of Nuprl's universes of types. Let us now explain the lifting operator used in univ2's second branch. The issue is that because of the constraint on *eq* in at_univ_def2, the level of *eq* must be at least at one level higher than the level of *eqa* in univ_eq2, i.e., it must be at least at one level higher than PERs of types in cuniv1. We then close univ2 using close2 (yet another copy of the *close* operator we use in our Prop metatheory) to obtain our third universe of types. This third universe of types can only mention the universes mkc_uni 0 and mkc_uni 1.

```
Inductive univ_eq2 (A A' : CTerm) :=
| ueq2 : ∀ eqa : per, cuniv1 A A' eqa → univ_eq2 A A'.
Inductive at_univ_def2 (T T' : CTerm) (eq : per) :=
| aud2 :
  computes_to_valc T (mkc_uni 1)
  → computes_to_valc T' (mkc_uni 1)
  → (∀ A A', eq A A' ⇔ univ_eq2 A A')
  → at_univ_def2 T T' eq.
Inductive Lift1 : per → per :=
| lift1 : ∀ (eq : per) t1 t2, eq t1 t2 → Lift1 eq t1 t2.
Inductive univ2 : CTerm → CTerm → per → Type :=
| u2_2 : ∀ T T' eq, at_univ_def2 T T' eq → univ2 T T' eq
| u2_1 : ∀ T T' eq eq', eq_term_equals2 eq (Lift1 eq') → univ1 T T' eq' → univ2
T T' eq.
Definition cuniv2 := close2 univ2.
```

As an illustrative example on how to define more universes of types, the following definitions show how to define a fourth universe of types. However, because of the lack of universe polymorphism and the amount of code duplication currently required to avoid universe inconsistency errors, we stop at cuniv2.

```
Inductive univ_eq3 (A A' : CTerm) :=
| ueq3 : ∀ eqa : per, cuniv2 A A' eqa → univ_eq3 A A'.
```

```
Inductive at_univ_def3 (T T' : CTerm) (eq : per) :=
| aud3 :
  computes_to_valc T (mkc_uni 2)
  → computes_to_valc T' (mkc_uni 2)
  → (∀ A A', eq A A' ⇔ univ_eq3 A A')
  → at_univ_def3 T T' eq.
Inductive Lift2 : per → per :=
| lift2 : ∀ (eq : per) t1 t2, eq t1 t2 → Lift2 eq t1 t2.
Inductive univ3 : CTerm → CTerm → per → Type :=
| u3_3 : ∀ T T' eq, at_univ_def3 T T' eq → univ3 T T' eq
| u3_2 : ∀ T T' eq eq', eq_term_equals3 eq (Lift2 eq') → univ2 T T' eq' → univ3
T T' eq.
Definition cuniv3 := close3 univ3.
```

As mentioned above, we stop at cuniv2, and define nuprl as cuniv2.

```
Definition nuprl := cuniv2.
```

Equality and membership in a type are defined in the same way as in our Prop metatheory.

```
Inductive equality t1 t2 T :=
| nueq : ∀ eq : per,
              nuprl T T eq
              → eq t1 t2
              → equality t1 t2 T.
Definition member t T := equality t t T.
```

Equality between types is also defined in a similar way as in our Prop metatheory.

```
Definition tequality T1 T2 := { eq : per , nuprl T1 T2 eq }.
Definition type T := tequality T T.
Definition tequalityi i T1 T2 := equality T1 T2 (mkc_uni i).
Definition typei i T := tequalityi i T T.
```

However, we define nuprli differently because we have to account for the fact that PERs of types have different Coq levels for types at different Nuprl levels. As before we use our Lifting operators. nuprl i returns the universe of types at the Nuprl level i.

```
Inductive nuprli : nat → cts :=
| ni0 : ∀ T T' eq eq', eq_term_equals2 eq (Lift1 (Lift0 eq')) → cuniv0 T T' eq' →
nuprli 0 T T' eq
```

| ni1 : ∀ *T T' eq eq'*, eq_term_equals2 *eq* (Lift1 *eq'*) → cuniv1 *T T' eq'* → nuprli 1 *T T' eq*
| ni2 : ∀ *i T T' eq*, *i* ≥2 → cuniv2 *T T' eq* → nuprli *i T T' eq*.

We can then prove the following lemmas that shows that equal types cuniv0 are also equal in cuniv1. Note that their PERs have to be lifted.

```
Lemma cuniv0_implies_cuniv1 :
  ∀ a b eq,
    cuniv0 a b eq
    → cuniv1 a b (Lift0 eq).
```

Similarity, we prove the following lemmas that shows that equal types cuniv1 are also equal in cuniv2. Again, we have to lift their PERs.

```
Lemma cuniv1_implies_cuniv2 :
  ∀ a b eq,
    cuniv1 a b eq
    → cuniv2 a b (Lift1 eq).
```

Using theses definitions we proved that our various *close* operator preserve the type system properties as in our Prop metatheory. We then proved that nuprl is a type system and we proved various useful properties about nuprl. We do not show all theses lemmas here because the statements are the same as the ones in our Prop metatheory. Only the proofs change. When proving a property of nuprl we now have to prove that it is true for each of its universes when in our Prop metatheory we can generally go by induction on the level. One difference with our Prop development is that we sometimes have to add an extra hypothesis to our lemmas when proving a property true about all levels, which says that the level is indeed a level, i.e., one of our three levels.

Once we have proved these properties, the definitions and properties of sequents and rules do not have to change. We then checked that we can prove the exact same rules as the ones we proved in Prop, using the exact same proofs. We currently only did that for function types.

Chapter 5

Sequents and Rules

5.1 Syntax and Semantics

————-begin file sequents.v ————

Let us now define the syntax and semantics of sequents and rules. As we did for terms, we first define the concept of a "bare" sequent, and we then define a sequent as a "bare" sequent that is well-formed.

A sequent is defined as a pair of a list of hypotheses and a conclusion. A sequent is then true if the conclusion is a true statement in the context of the given hypotheses. Therefore, we first define our syntax for "bare" and well-formed hypotheses; for "bare" and well-formed conclusions; and finally, for "bare" and well-formed sequents.

An hypothesis is a variable hvar (also called the name of the hypothesis), a type htyp, a Boolean hidden which says whether or not the hypothesis is hidden, and a level annotation that can provide the level of the type. The hiding mechanism is used to make it explicit in a sequent which variables can occur in the extract of the sequent.

```
Record hypothesis :=
  {hvar : NVar; hidden : bool; htyp : NTerm; lvl : lvlop}.
```

A barehypotheses is a list of hypotheses. We say that such a list is "bare" because it is not yet accompanied with a proof that the list is well-formed (see below).

```
Definition barehypotheses := list hypothesis.
```

The following function extracts the variables from a list of hypotheses.

```
Definition vars_hyps (hs : barehypotheses) := map hvar hs.
```

The following predicate states when hypotheses are well-formed. For a "bare" list of hypotheses of the form $(x_1 : T1, \ldots, x_n : T_n)$ to be well-formed, each free variable x occurring in a T_i has to be one of the variables among $x_1, \ldots, x_{i-1}$. Also, all the $x_1, \ldots, x_n$ have to be different from each other.

```
Inductive wf_hypotheses : barehypotheses → Type :=
  | hyps_nil : wf_hypotheses nil
  | hyps_cons :
      ∀ h : hypothesis,
      ∀ hs : barehypotheses,
        isprog_vars (vars_hyps hs) (htyp h)
        → ! LIn (hvar h) (vars_hyps hs)
        → wf_hypotheses hs
        → wf_hypotheses (snoc hs h).
```

A well-formed list of hypotheses is a "bare" list of hypotheses along with a proof that these hypotheses are well-formed.

```
Definition hypotheses := { hs : barehypotheses & wf_hypotheses hs }.
```

We have two forms of sequents. Some sequents have extracts and some do not. If a sequent with an extract is true then the extract provides an evidence of the truth of the type, and the type is a type of the Nuprl type system. If a sequent without an extract is true then the type is a type of the Nuprl type system. The current Nuprl version has only one kind of sequent. However, we feel that it is sometimes useful to prove that types are indeed types without having to deal with levels. Currently in Nuprl, to prove that a type T is a type one has to prove that mkc_member T (*mkc_uni* i) is true for some level i.

```
Inductive conclusion :=
| concl_ext : ∀ (ctype : NTerm) (extract : NTerm), conclusion
| concl_typ : ∀ (ctype : NTerm), conclusion.
```

The following ctype function returns the type of a conclusion.

```
Definition ctype c :=
  match c with
    | concl_ext t e ⇒ t
    | concl_typ t ⇒ t
  end.
```

The following extract function returns the extract of a conclusion whenever it exists. We then define several operations that are simple maps on options.

```
Definition extract c :=
  match c with
    | concl_ext t e ⇒ Some e
    | concl_typ t ⇒ None
  end.

Definition wf_term_op top :=
  match top with
    | Some t ⇒ wf_term t
    | None ⇒ true = true
  end.

Definition covered_op top vs :=
  match top with
    | Some t ⇒ covered t vs
    | None ⇒ true = true
  end.

Definition cover_vars_op top sub :=
  match top with
    | Some t ⇒ cover_vars t sub
    | None ⇒ true = true
  end.
```

We say that a conclusion is well-formed if its type and extract are well-formed.

```
Definition wf_concl (c : conclusion) : Type :=
  wf_term (ctype c) × wf_term_op (extract c).
```

A conclusion depends on hypotheses. For a conclusion to be closed w.r.t. a list of hypotheses, means that the variables of the type have to be covered by the hypotheses.

```
Definition closed_type (hs : barehypotheses) (c : conclusion) : Type :=
  covered (ctype c) (vars_hyps hs).
```

In addition the variables of the extract also have to be covered by the non-hidden hypotheses.

```
Definition closed_extract (hs : barehypotheses) (c : conclusion) : Type :=
  covered_op (extract c) (nh_vars_hyps hs).
```

We can now define sequents. A "bare" sequent is a pair of a "bare" list of hypotheses and a "bare" conclusion (which are just called conclusions here).

```
Record baresequent := {hyps : barehypotheses; concl : conclusion}.
```

A sequent is well-formed if its hypotheses and conclusion are well-formed.

```
Definition wf_sequent (S : baresequent) :=
  wf_hypotheses (hyps S)
  × wf_concl (concl S).
```

A sequent is a "bare" sequent along with a proof that it is well-formed.

```
Definition sequent := { s : baresequent & wf_sequent s}.
```

Let us now define a few concepts that are useful to define various degrees of well-formedness of sequents. A "bare" sequent is said to be closed if its conclusion is closed w.r.t. to the hypotheses. First we define the notion of a ctsequent which is a sequent where its type is closed. Then, we define the notion of a csequent (closed sequent) which is a ctsequent where its extract is closed.

```
Definition closed_type_baresequent (s : baresequent) :=
  closed_type (hyps s) (concl s).

Definition closed_type_sequent (s : sequent) :=
  closed_type_baresequent (projT1 s).

Definition ctsequent := { s : sequent & closed_type_sequent s}.

Definition closed_extract_baresequent (S : baresequent) :=
  closed_extract (hyps S) (concl S).

Definition closed_extract_sequent (s : sequent) :=
  closed_extract_baresequent (projT1 s).

Definition closed_extract_ctsequent (s : ctsequent) :=
  closed_extract_sequent (projT1 s).

Definition csequent := {s : ctsequent & closed_extract_ctsequent s}.
```

We now define rules. A non "bare" rule (simply called a rule here) is a main sequent called goal, a list of subgoals and a list of arguments or side-conditions. A side-condition can either be a condition on a variable v, which means that the variable v has to be fresh (we have not had to use these yet), or a condition on a term t, which means that the free variables of t have to be declared as non-hidden in the hypotheses of the main goal of the rule.

```
Inductive sarg :=
  | sarg_var : NVar → sarg
  | sarg_term : NTerm → sarg.

Record rule :=
  {goal : baresequent
   ; subgoals : list baresequent
   ; sargs : list sarg
  }.

Definition mk_rule
                  (goal : baresequent)
                  (subgoals : list baresequent)
                  (args : list sarg) :=
  {| goal := goal; subgoals := subgoals; sargs := args |}.
```

We now provide the definition of what it means for sequents and rules to be true. There have been several definitions over the years. We present four versions: one that is presented in the Nuprl book [7], one that is presented by Karl Crary in his thesis [21], one that is presented by Aleksey Nogin in his thesis [36], and finally one which we obtained by simplifying Nogin's definition. We then show that all these definitions are equivalent. We are then free to use the one we want. Crary's definition is similar to the one provided in Nuprl's Book. The main difference is that Crary uses substitutions while the definition from the Book uses lists of terms (substitutions are then built from lists of terms and lists of hypotheses). Nogin's definition differs by having not a pointwise requirement on each hypothesis but instead a more global pointwise requirement on the entire list of hypotheses.

In order to introduce these definitions, we first need various abstractions on hypotheses:

- hyps_true_at *H* *ts* which is used by the Book's definition and should be read: "the list of hypotheses *H* is true at the list of terms *ts*".
- equal_terms_in_hyps *ts1* *ts2* *H* which is also use by the Book's definition and should be read: "the two lists of terms *ts1* and *ts2* are equal in the list of hypotheses *H*. This can be seen as a lifting operation of the equality relation from types to lists of hypotheses.
- pw_assign_eq *s1* *s2* *H* which is used in Crary's definition and should be read: "the two substitutions *s1* and *s2* are pointwise equal in the list of hypotheses *H*".

- similarity *s1 s2 H* which is used in both Crary and Nogin's definitions and should be read: "the two substitutions *s1* and *s2* are similar in *H*". This can be seen as a simple lifting operation of the equality relation from types to lists of hypotheses.

- eq_hyps *s1 s2 H* which is used in Nogin's definition and is a simple lifting operation of the tequality relation from types to lists of hypotheses.

- hyps_functionality *s H* which is also used in Nogin's definition and should be read: "the hypotheses *H* are functional w.r.t. the substitution *s*".

Along the way we also define a few other useful abstractions. For example, mk_hs_subst is used in the Book's definition to build a substitution from a list of terms and a list of hypotheses.

```
Fixpoint mk_hs_subst
              (ts : list CTerm)
              (hs : barehypotheses) : CSubstitution :=
  match ts, hs with
  | nil, _ ⇒ nil
  | _, nil ⇒ nil
  | t :: ts, h :: hs ⇒ (hvar h, t) :: mk_hs_subst ts hs
  end.
```

As mentioned above, Crary (as well as Nogin) defines a relation called "assignment similarity" [21, page 56] which we call similarity. The Nuprl book defines a similar relation that we call equal_terms_in_hyps.

Two substitutions *s1* and *s2* are similar in a list of hypotheses *H* if *s1* is of the form [(*x1*,*t1*),...,(*xn*,*tn*)], *s2* is of the form [(*x1*,*u1*),...,(*xn*,*un*)], *H* is of the form [(*x1*,*T1*),...,(*xn*,*Tn*)], and *t1* = *u1* in *T1*, *t2* = *u2* in *T2*[(*x1*,*t1*)], *t3* = *u3* in *T3*[(*x1*,*t1*),(*x2*,*t2*)],...

```
Inductive similarity : CSubstitution
                             → CSubstitution
                             → barehypotheses → [U] :=
  | sim_nil : similarity nil nil nil
  | sim_cons :
      ∀ t1 t2 : CTerm,
      ∀ s1 s2 : CSubstitution,
      ∀ h : hypothesis,
      ∀ hs : barehypotheses,
      ∀ w : wf_term (htyp h),
      ∀ p : cover_vars (htyp h) s1,
      ∀ e : equality t1 t2 (lsubstc (htyp h) w s1 p),
      ∀ sim : similarity s1 s2 hs,
        similarity (snoc s1 (hvar h, t1))
                   (snoc s2 (hvar h, t2))
                   (snoc hs h).
```

The equal_terms_in_hyps relation is used in the Book's definition and is similar to the similarity relation except that it is defined on lists of terms rather than on substitutions.

```
Inductive equal_terms_in_hyps : list CTerm
                                → list CTerm
                                → barehypotheses → [U] :=
  | EqInHyps_nil : equal_terms_in_hyps nil nil nil
  | EqInHyps_cons :
        ∀ t1 t2 : CTerm,
        ∀ ts1 ts2 : list CTerm,
        ∀ h : hypothesis,
        ∀ hs : barehypotheses,
        ∀ w : wf_term (htyp h),
        ∀ p : cover_vars (htyp h) (mk_hs_subst ts1 hs),
        ∀ e : equality t1 t2 (lsubstc (htyp h) w (mk_hs_subst ts1 hs) p),
        ∀ eqt : equal_terms_in_hyps ts1 ts2 hs,
          equal_terms_in_hyps (snoc ts1 t1) (snoc ts2 t2) (snoc hs h).
```

This is the "Assignment equality with pointwise functionality" relation that Crary defines on page 57 of his thesis. This is different from the hyps_true_at relation defined in the Nuprl book. The relation defined in the Nuprl book relates a list of terms to a list of hypotheses while Crary's relation related a pairs of lists of terms to a list of hypotheses.

This relation is closer to the similarity relation. However, it is more complicated because it has a "pointwise" condition that says that each hypothesis has to be functional w.r.t. to the preceding hypotheses.

```
Inductive pw_assign_eq : CSubstitution
                         → CSubstitution
                         → barehypotheses → [U] :=
  | pw_eq_nil : pw_assign_eq [] [] []
  | pw_eq_cons :
        ∀ t1 t2 : CTerm,
        ∀ s1 s2 : CSubstitution,
        ∀ h : hypothesis,
        ∀ hs : barehypotheses,
        ∀ w : wf_term (htyp h),
        ∀ p : cover_vars (htyp h) s1,
        ∀ e : equality t1 t2 (lsubstc (htyp h) w s1 p),
        ∀ hf : (∀ s' : CSubstitution,
                      ∀ p' : cover_vars (htyp h) s',
                        similarity s1 s' hs
                        → eqtypes (lvl h)
                                     (lsubstc (htyp h) w s1 p)
                                     (lsubstc (htyp h) w s' p')),
        ∀ pw : pw_assign_eq s1 s2 hs,
          pw_assign_eq (snoc s1 (hvar h, t1))
                       (snoc s2 (hvar h, t2))
                       (snoc hs h).
```

Nogin can do without pw_assign_eq and simply use similarity because he "lifts" the functionality requirement so that it is not required to be proved for every single hypothesis but instead for the entire list of hypotheses. This is achieved by the eq_hyps and hyps_functionality abstractions. Having a more "global" pointwise functionality requirement greatly simplifies the proofs.

```
Inductive eq_hyps : CSubstitution
                    → CSubstitution
                    → barehypotheses → [U] :=
  | eq_hyps_nil : eq_hyps [] [] []
  | eq_hyps_cons :
      ∀ t1 t2 : CTerm,
      ∀ s1 s2 : CSubstitution,
      ∀ h : hypothesis,
      ∀ hs : barehypotheses,
      ∀ w : wf_term (htyp h),
      ∀ p1 : cover_vars (htyp h) s1,
      ∀ p2 : cover_vars (htyp h) s2,
      ∀ eqt : eqtypes (lvl h)
                      (lsubstc (htyp h) w s1 p1)
                      (lsubstc (htyp h) w s2 p2),
      ∀ eqh : eq_hyps s1 s2 hs,
        eq_hyps (snoc s1 (hvar h, t1)) (snoc s2 (hvar h, t2)) (snoc hs h).
Definition hyps_functionality (s : CSubstitution) (H : barehypotheses) :=
  ∀ s' : CSubstitution,
    similarity s s' H → eq_hyps s s' H.
```

This hyps_true_at abstraction is used in the Book's definition. It is similar to pw_assign_eq but uses lists of terms instead of substitutions, and does not relate two substitutions—this is dealt with using equal_terms_in_hyps which is presented above.

```
Inductive hyps_true_at : barehypotheses → list CTerm → [U] :=
  | InHyp_nil : hyps_true_at [] []
  | InHyp_cons :
      ∀ t : CTerm,
      ∀ ts : list CTerm,
      ∀ h : hypothesis,
      ∀ hs : barehypotheses,
      ∀ w : wf_term (htyp h),
      ∀ p : cover_vars (htyp h) (mk_hs_subst ts hs),
      ∀ m : member t (lsubstc (htyp h) w (mk_hs_subst ts hs) p),
      ∀ hf :
              (∀ ts' : list CTerm,
               ∀ p' : cover_vars (htyp h) (mk_hs_subst ts' hs),
                 equal_terms_in_hyps ts ts' hs
                 → eqtypes (lvl h)
                             (lsubstc (htyp h) w (mk_hs_subst ts hs) p)
                             (lsubstc (htyp h) w (mk_hs_subst ts' hs) p')),
      ∀ hta : hyps_true_at hs ts,
        hyps_true_at (snoc hs h) (snoc ts t).
```

The Nuprl book defines the truth of a sequent as follows: A sequent S with hypotheses H, type T, and extract *ext*, is true if for all list of terms *ts* that cover H (i.e., cover_csequent *ts* S), and for all list of term *ts'*, if H is true at *ts* (i.e., hyps_true_at *ts* H), and the two lists of terms are equal in H (i.e., equal_terms_in_hyps *ts ts'* H), and *s1* and *s2* are two substitutions built from *ts*, *ts'*, and H, then $s1(T)$ is equal to $s2(T)$ and $s1(ext)$ and $s2(ext)$ are equal in $s1(T)$.

```
Definition sequent_true_at (S : csequent) (ts : list CTerm) : Type :=
  ∀ ts' : list CTerm,
    match destruct_csequent S with
      | cseq_comps hs T wh wt ct ec ⇒
          ∀ p : hyps_true_at hs ts,
          ∀ e : equal_terms_in_hyps ts ts' hs,
            let sub1 := mk_hs_subst ts hs in
            let sub2 := mk_hs_subst ts' hs in
              (
                tequality
                  (lsubstc T wt sub1 (seq_cover_typ1 T ts ts' hs ct e))
                  (lsubstc T wt sub2 (seq_cover_typ2 T ts ts' hs ct e))
                ×
                match ec with
                  | Some (existT ext (we, ce)) ⇒
                      equality
                        (lsubstc ext we sub1 (seq_cover_ex1 ext ts ts' hs ce e))
                        (lsubstc ext we sub2 (seq_cover_ex2 ext ts ts' hs ce e))
                        (lsubstc T wt sub1 (seq_cover_typ1 T ts ts' hs ct e))
                  | None ⇒ True
                end
              )
    end.
Definition sequent_true (S : csequent) : Type :=
  ∀ ts : list CTerm,
    cover_csequent ts S
    → sequent_true_at S ts.
```

Karl Crary defines the truth of a sequent as follows: A sequent S with hypotheses H, type T, and extract ext, is true if for all substitutions $s1$ and $s2$, if the two substitutions are pointwise equal in H, i.e., pw_assign_eq $s1$ $s2$ H, then $s1(T)$ is equal to $s2(T)$ and $s1(ext)$ and $s2(ext)$ are equal in $s1(T)$.

```
Definition KC_sequent_true (S : csequent) : Type :=
  ∀ s1 s2 : CSubstitution,
    match destruct_csequent S with
      | cseq_comps H T wh wt ct ec ⇒
          ∀ p : pw_assign_eq s1 s2 H,
            tequality (lsubstc T wt s1 (scover_typ1 T s1 s2 H ct p))
                      (lsubstc T wt s2 (scover_typ2 T s1 s2 H ct p))
            ×
            match ec with
              | Some (existT ext (we, ce)) ⇒
                  equality (lsubstc ext we s1 (scover_ex1 ext s1 s2 H ce p))
                           (lsubstc ext we s2 (scover_ex2 ext s1 s2 H ce p))
                           (lsubstc T wt s1 (scover_typ1 T s1 s2 H ct p))
              | None ⇒ True
            end
      end.
```

Aleksey Nogin defines the truth of a sequent as follows: A sequent *S* with hypotheses *H*, type *T*, and extract *ext*, is true if for all substitution *s1*, if similarity *s1* *s1* *H*, and hyps_functionality *s1* *H*, then for all substitution *s2*, if similarity *s1* *s2* *H* then *s1*(*T*) is equal to *s2*(*T*) and *s1*(*ext*) and *s2*(*ext*) are equal in *s1*(*T*).

```
Definition AN_sequent_true (S : csequent) : Type :=
  ∀ s1,
    match destruct_csequent S with
      | cseq_comps H T wh wt ct ec ⇒
          similarity s1 s1 H
          → hyps_functionality s1 H
          → ∀ s2,
             ∀ p : similarity s1 s2 H,
               tequality
                 (lsubstc T wt s1 (s_cover_typ1 T s1 s2 H ct p))
                 (lsubstc T wt s2 (s_cover_typ2 T s1 s2 H ct p))
               ×
               match ec with
                 | Some (existT ext (we, ce)) ⇒
                     equality
                       (lsubstc ext we s1 (s_cover_ex1 ext s1 s2 H ce p))
                       (lsubstc ext we s2 (s_cover_ex2 ext s1 s2 H ce p))
                       (lsubstc T wt s1 (s_cover_typ1 T s1 s2 H ct p))
                 | None ⇒ True
               end
  end.
```

We slightly simplify Nogin's definition of a truth of a sequent by removing the similarity *s1* *s1* *H*, which is redundant. We say that a sequent *S* with hypotheses *H*, type *T*, and extract *ext*, is true if for all substitutions *s1* and *s2*, if similarity *s1* *s2* *H* and hyps_functionality *s1* *H* are true then *s1*(*T*) is equal to *s2*(*T*) and *s1*(*ext*) and *s2*(*ext*) are equal in *s1*(*T*).

```
Definition VR_sequent_true (S : csequent) : Type :=
  ∀ s1 s2,
    match destruct_csequent S with
      | cseq_comps H T wh wt ct ec ⇒
```

```
            ∀ p : similarity s1 s2 H,
              hyps_functionality s1 H
              → tequality
                    (lsubstc T wt s1 (s_cover_typ1 T s1 s2 H ct p))
                    (lsubstc T wt s2 (s_cover_typ2 T s1 s2 H ct p))
                 × match ec with
                      | Some (existT ext (we, ce)) ⇒
                           equality
                              (lsubstc ext we s1 (s_cover_ex1 ext s1 s2 H ce p))
                              (lsubstc ext we s2 (s_cover_ex2 ext s1 s2 H ce p))
                              (lsubstc T wt s1 (s_cover_typ1 T s1 s2 H ct p))
                      | None ⇒ True
                    end
   end.
```

It is straightforward to prove that Nogin's definition is equivalent to ours.

```
Lemma AN_sequent_true_eq_VR :
  ∀ S : csequent,
   AN_sequent_true S ⇔ VR_sequent_true S.
```

We now prove that Crary's notion of truth of a sequent is equivalent to the one defined in the Nuprl book.

```
Lemma sequent_true_eq_KC :
  ∀ S, sequent_true S ⇔ KC_sequent_true S.
```

Finally, we prove that Nogin's notion of truth of a sequent is equivalent to the one defined by Crary.

```
Lemma sequent_true_KC_eq_AN :
  ∀ S, KC_sequent_true S ⇔ AN_sequent_true S.
```

We are now ready to define what it means for a rule to be valid. A rule is valid if assuming that the main goal is well-formed (but not necessarily closed), and its type is closed w.r.t. the hypotheses, and each subgoal is well-formed, closed (both types and extracts), and true, then we can prove that the extract of the main goal is closed w.r.t. the non-hidden hypotheses of the sequent, and that the main goal is true.

```
Definition rule_true (R : rule) : Type :=
  ∀ wg : wf_sequent (goal R),
  ∀ cg : closed_type_baresequent (goal R),
  ∀ cargs: args_constraints (sargs R) (hyps (goal R)),
  ∀ hyps : (∀ s : baresequent,
                  LIn s (subgoals R)
                  → {c : wf_csequent s & sequent_true (mk_wcseq s c)}),
    {c : closed_extract_baresequent (goal R)
     & sequent_true (mk_wcseq (goal R) (ext_wf_cseq (goal R) wg cg c))}.
```

Equivalently, we say that a rule is valid if it satisfies rule_true2 below. The reason for having two definitions is that we had already proved several rules at the time we stated the simpler rule_true2 definition. The difference with rule_true is that in rule_true2 we use an abstraction (sequent_true2) for the pair of proofs that a sequent is well-formed (wf_csequent) and true (sequent_true).

```
Definition sequent_true2 s :=
  {c : wf_csequent s & sequent_true (mk_wcseq s c)}.
Definition pwf_sequent s := wf_sequent s × closed_type_baresequent s.
```

```
Definition rule_true2 (R : rule) : Type :=
  ∀ pwf : pwf_sequent (goal R),
  ∀ cargs : args_constraints (sargs R) (hyps (goal R)),
  ∀ hyps : (∀ s, LIn s (subgoals R) → sequent_true2 s),
    sequent_true2 (goal R).
```

The proof that the two definitions are equal is trivial.

```
Lemma rule_true_iff_rule_true2 :
  ∀ R, rule_true R ⇔ rule_true2 R.
```

Let us now define the following nuprove abstraction, which we use to prove Nuprl lemmas in Coq using the proofs that the Nuprl rules are valid and the definition of validity of rules. It says that we have to prove that the provided goal is a true Nuprl type.

```
Definition nuprove goal :=
  {t : NTerm
   & pwf_sequent (mk_baresequent [] (mk_concl goal t))
   × sequent_true2 (mk_baresequent [] (mk_concl goal t))}.
```

Because to prove nuprove expressions we also have to prove the well-formedness of sequents and because often, in a sequent, the subgoals are well-formed if the main goal is well-formed, we define the following wf_rule abstraction, which we use to simplify the process of proving Nuprl lemmas in Coq.

```
Definition wf_subgoals R :=
  ∀ s, LIn s (subgoals R) → pwf_sequent s.
```

```
Lemma fold_wf_subgoals :
  ∀ R,
    (∀ s, LIn s (subgoals R) → pwf_sequent s) = wf_subgoals R.
Definition wf_rule (R : rule) :=
  pwf_sequent (goal R) → wf_subgoals R.
```

Using our definition of mk_false and the meaning of sequents, we can prove that the following sequent is not true, is this for any extract t:

```
      |- False ext t
```

```
Lemma weak_consistency :
  ∀ t,
    wf_term t
    → !rule_true (mk_rule (mk_baresequent [] (mk_concl mk_false t)) [] []).
```

5.2 Verifying Nuprl's rules

5.2.1 Function type

———–begin file rules_function.v ————

The Nuprl manual [30] presents various useful rules to reason about function types. This section prove the truth of several of these rules.

The following rule is called the function elimination rules. It says that if we can prove that a is in the domain A then we can assume that $f\ a$ is in $B[x\backslash a]$. As for the other rules, first we show an informal statement of this rule, then we show its formal statement (a Definition), and finally we show the statement that the rule is true (a Lemma). In such informal statements, we write "C ext e" to indicate that e is the extract of the sequent, i.e., the evidence that C is true. We omit the "ext e" part when e is mk_axiom.

```
   H, f : x:A->B[x], J |- C ext e[z\axiom]

     By perFunctionElimination s z

     H, f : x:A->B[x], J |- a in A
     H, f : x:A->B[x], J, z : (f a) in B[a] |- C ext e
```

```
Definition rule_function_elimination
             (A B C a e : NTerm)
             (f x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp f (mk_function A x B)) ++ J)
       (mk_concl C (subst e z mk_axiom)))
    [ mk_baresequent
        (snoc H (mk_hyp f (mk_function A x B)) ++ J)
        (mk_conclax (mk_member a A)),
      mk_baresequent
        (snoc (snoc H (mk_hyp f (mk_function A x B)) ++ J)
              (mk_hyp z (mk_member (mk_apply (mk_var f) a)
                                   (subst B x a))))
        (mk_concl C e)
    ]
    [sarg_term a, sarg_var z].

Lemma rule_function_elimination_true :
  ∀ A B C a e : NTerm,
  ∀ f x z : NVar,
  ∀ H J : barehypotheses,
  ∀ bc1 : x ≠ f,
  ∀ bc2 : disjoint (vars_hyps H) (bound_vars B),
  ∀ bc3 : disjoint (vars_hyps J) (bound_vars B),
  ∀ bc4 : !LIn f (bound_vars B),
    rule_true (rule_function_elimination
                 A B C a e
                 f x z
                 H J).
```

The following rule is called the lambda formation rules. It says that to prove that a function type is true, it is enough to prove that given a element z in its domain A, we can prove that its codomain $B[x\backslash z]$ is true. We also have to prove that its domain $A\}$ *is a well-formed type.x* « $H \vdash x{:}A \to B$ *ext* $\backslash z.b$

By lambdaFormation lvl(i) z ()

H z : $A \vdash B[x\backslash z]$ *ext* b $H \vdash A = A$ in Type(i) »

```
Definition rule_lambda_formation
```

```
                (A B b : NTerm)
                (x z : NVar)
                (i : nat)
                (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_concl (mk_function A x B) (mk_lam z b)))
    [ mk_baresequent (snoc H (mk_hyp z A))
                     (mk_concl (subst B x (mk_var z)) b),
      mk_baresequent H (mk_conclax (mk_equality A A (mk_uni i))) ]
    [sarg_var z].

Lemma rule_lambda_formation_true :
  ∀ (A B b : NTerm)
         (x z : NVar)
         (i : nat)
         (H : barehypotheses)
         (bc1 : !LIn z (bound_vars B)),
    rule_true (rule_lambda_formation A B b x z i H).
```

The following rule is called the function equality rule. It says that to prove that a function type is well-formed, we have to prove that its domain is well-formed and that its co-domain is functional over its domain.

```
  H |- x1:a1 -> b1 = x2:a2 -> b2 in Type(i)

    By functionEquality y ()

    H |- a1 = a2 in Type(i)
    H y : a1 |- subst b1 x1 y = subst b2 x2 y in Type(i)
```

```
Definition rule_function_equality
                (a1 a2 b1 b2 : NTerm)
                (x1 x2 y : NVar)
                (i : nat)
                (H : barehypotheses) :=
  mk_rule
    (mk_baresequent
       H
       (mk_conclax (mk_equality
                      (mk_function a1 x1 b1)
                      (mk_function a2 x2 b2)
```

```
                        (mk_uni i))))
    [ mk_baresequent
        H
        (mk_conclax (mk_equality a1 a2 (mk_uni i))),
      mk_baresequent
        (snoc H (mk_hyp y a1))
        (mk_conclax (mk_equality
                          (subst b1 x1 (mk_var y))
                          (subst b2 x2 (mk_var y))
                          (mk_uni i)))
    ]
    [ sarg_var y ].

Lemma rule_function_equality_true :
  ∀ a1 a2 b1 b2 : NTerm,
  ∀ x1 x2 y : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
  ∀ bc1 : !LIn y (bound_vars b1),
  ∀ bc2 : !LIn y (bound_vars b2),
    rule_true (rule_function_equality
                 a1 a2 b1 b2
                 x1 x2 y
                 i
                 H).
```

The following rule is called the lambda equality rule. It allows one to prove that lambda-abstractions are well-typed.

```
H |- \x1.t1 = \x2.t2 in x:A->B

  By lambdaEquality lvl(i) z ()

  H z : A |- t1[x1\z] = t2[x2\z] in B[x\z]
  H |- A = A in Type(i)
```

```
Definition rule_lambda_equality
             (A B t1 t2 : NTerm)
             (x1 x2 x z : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent
       H
       (mk_conclax (mk_equality
                      (mk_lam x1 t1)
                      (mk_lam x2 t2)
                      (mk_function A x B))))
    [ mk_baresequent
        (snoc H (mk_hyp z A))
        (mk_conclax (mk_equality
                       (subst t1 x1 (mk_var z))
                       (subst t2 x2 (mk_var z))
                       (subst B x (mk_var z)))),
      mk_baresequent
        H
        (mk_conclax (mk_equality A A (mk_uni i))) ]
    [sarg_var z].

Lemma rule_lambda_equality_true :
  ∀ (A B t1 t2 : NTerm)
         (x1 x2 x z : NVar)
         (i : nat)
         (H : barehypotheses)
         (bc1 : !LIn z (bound_vars B))
         (bc2 : !LIn z (bound_vars t1))
         (bc3 : !LIn z (bound_vars t2)),
    rule_true (rule_lambda_equality A B t1 t2 x1 x2 x z i H).
```

We following rule called “apply equality“ allows one to prove that applications are well-typed.

```
  H |- f1 t1 = f2 t2 in B[x\t1]

    By applyEquality ()

    H |- f1 = f2 in x:A->B
    H |- t1 = t2 in A
```

```
Definition rule_apply_equality
             (A B f1 f2 t1 t2 : NTerm)
             (x : NVar)
```

```
                (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality
                                           (mk_apply f1 t1)
                                           (mk_apply f2 t2)
                                           (subst B x t1))))
    [ mk_baresequent H (mk_conclax (mk_equality f1 f2 (mk_function A x B))),
      mk_baresequent H (mk_conclax (mk_equality t1 t2 A))
    ]
    [].

Lemma rule_apply_equality_true :
  ∀ (A B f1 f2 t1 t2 : NTerm)
           (x : NVar)
           (i : nat)
           (H : barehypotheses)
           (bc1 : disjoint (free_vars t1) (bound_vars B))
           (bc2 : disjoint (free_vars t2) (bound_vars B)),
    rule_true (rule_apply_equality A B f1 f2 t1 t2 x H).
```

The following rule called the "apply reduce" rule is the computation rule for applications.

```
H |- (\x.t)s = u in T

  By applyReduce ()

  H |- t[x\s] = u in T
```

```
Definition rule_apply_reduce
                (T t s u : NTerm)
                (x : NVar)
                (H : barehypotheses) :=
  mk_rule
    (mk_baresequent
       H
       (mk_conclax (mk_equality (mk_apply (mk_lam x t) s) u T)))
    [ mk_baresequent
        H
        (mk_conclax (mk_equality (subst t x s) u T))
    ]
    [].
Lemma rule_apply_reduce_true :
  ∀ (T t s u : NTerm)
          (x : NVar)
          (H : barehypotheses)
          (bc1 : disjoint (free_vars s) (bound_vars t)),
    rule_true (rule_apply_reduce T t s u x H).
```

The following rule called the "function extensionality rule" states that the equality between functions in Nuprl is extensional.

```
H |- f1 = f2 in x:A->B

  By funcionExtensionality lvl(i) z ()

  H z : A |- f1 z = f2 z in B[x\z]
  H |- A = A in Type(i)
```

```
Definition rule_function_extensionality
             (A B f1 f2 : NTerm)
             (x z : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent
       H
       (mk_conclax (mk_equality f1 f2 (mk_function A x B))))
    [ mk_baresequent
        (snoc H (mk_hyp z A))
        (mk_conclax (mk_equality
                          (mk_apply f1 (mk_var z))
                          (mk_apply f2 (mk_var z))
                          (subst B x (mk_var z)))),
      mk_baresequent
        H
        (mk_conclax (mk_equality A A (mk_uni i))) ]
    [sarg_var z].
Lemma rule_function_extensionality_true :
  ∀ (A B f1 f2 : NTerm)
          (x z : NVar)
          (i : nat)
          (H : barehypotheses)
          (bc1 : !LIn z (bound_vars B)),
    rule_true (rule_function_extensionality A B f1 f2 x z i H).
```

5.2.2 W type

————-begin file rules_w.v ————

We now prove the truth of several rules about the W type. These are the standard elimination and introduction rules for W types. For the elimination (or induction) rule we only deal with goals that have a trivial extract (*mk_axiom*) as it is the case for equality types. Therefore, this rule can be used to prove memberships.

The W induction rule can be stated as follows:

```
H, w : W(A;v.B) |- Q ext Ax

  By Winduction a f z b

  H, a : A
     f : B(a) -> W(A;v.B)
     z : b:B(a) -> Q[w\f(b)]
   |- Q[w\sup(a,f)] ext Ax
```

```
Definition rule_w_induction
               (H : barehypotheses)
               (Q A B : NTerm)
               (v w a f z b : NVar) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp w (mk_w A v B)))
       (mk_conclax Q))
    [ mk_baresequent
        (snoc (snoc (snoc H (mk_hyp a A))
                    (mk_hyp f (mk_function
                                 (lsubst B [(v,mk_var a)])
                                 b
                                 (mk_w A v B))))
```

```
                 (mk_hyp z (mk_function
                             (lsubst B [(v,mk_var a)])
                             b
                             (lsubst Q [(w,mk_apply (mk_var f) (mk_var b))])))))
              (mk_conclax (lsubst Q [(w,mk_sup (mk_var a) (mk_var f))]))
        ]
        [sarg_var a, sarg_var f, sarg_var z, sarg_var b].

Lemma rule_w_induction_true :
  ∀ (H : barehypotheses)
         (Q A B : NTerm)
         (v w a f z b : NVar)
         (bc1 : !LIn b (w :: a :: f :: z :: v :: vars_hyps H))
         (bc2 : !LIn w [a,f,z])
         (bc3 : !LIn a (bound_vars B))
         (bc4 : !LIn b (bound_vars Q))
         (bc5 : !LIn f (bound_vars Q))
         (bc6 : !LIn a (bound_vars Q)),
    rule_true (rule_w_induction H Q A B v w a f z b).
```

We state the W equality rules as follows:

```
H |- W(a1,x1.b1) = W(a2,x2.b2) in Type(i)

  By WEquality y ()

  H |- a1 = a2 in Type(i)
  H y : a1 |- subst b1 x1 y = subst b2 x2 y in Type(i)
```

```
Definition rule_w_equality
             (a1 a2 b1 b2 : NTerm)
             (x1 x2 y : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent
       H
       (mk_conclax (mk_equality
                      (mk_w a1 x1 b1)
                      (mk_w a2 x2 b2)
                      (mk_uni i))))
    [ mk_baresequent
        H
        (mk_conclax (mk_equality a1 a2 (mk_uni i))),
      mk_baresequent
        (snoc H (mk_hyp y a1))
        (mk_conclax (mk_equality
                       (subst b1 x1 (mk_var y))
                       (subst b2 x2 (mk_var y))
                       (mk_uni i)))
    ]
    [ sarg_var y ].

Lemma rule_w_equality_true :
  ∀ a1 a2 b1 b2 : NTerm,
  ∀ x1 x2 y : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
  ∀ bc1 : !LIn y (bound_vars b1),
  ∀ bc2 : !LIn y (bound_vars b2),
    rule_true (rule_w_equality
                 a1 a2 b1 b2
                 x1 x2 y
                 i
                 H).
```

We state the W introduction rule as follows:

```
H |- sup(a1,f1) = sup(a2,f2) in W(A,v.B)

  By WIntroduction ()

  H |- a1 = a2 in A
  H |- f1 = f2 in B[v\a1] -> W(A,v.B)
  H, a : A |- B[v\a] in Type(i)
```

```
Definition rule_w_introduction
             (a1 a2 f1 f2 A B : NTerm)
             (i : nat)
             (a v b : NVar)
             (H : barehypotheses) :=
  mk_rule
```

```
    (mk_baresequent
       H
       (mk_conclax (mk_equality
                      (mk_sup a1 f1)
                      (mk_sup a2 f2)
                      (mk_w A v B))))
    [ mk_baresequent
        H
        (mk_conclax (mk_equality a1 a2 A)),
      mk_baresequent
        H
        (mk_conclax (mk_equality
                       f1
                       f2
                       (mk_function (subst B v a1) b (mk_w A v B)))),
      mk_baresequent
        (snoc H (mk_hyp a A))
        (mk_conclax (mk_member (subst B v (mk_var a)) (mk_uni i)))
    ]
    [].

Lemma rule_w_introduction_true :
  ∀ a1 a2 f1 f2 A B : NTerm,
  ∀ i : nat,
  ∀ a v b : NVar,
  ∀ H : barehypotheses,
  ∀ bc1 : !LIn b (v :: vars_hyps H),
  ∀ bc2 : !LIn a (bound_vars B),
  ∀ bc3 : disjoint (free_vars a1) (bound_vars B),
    rule_true (rule_w_introduction
                 a1 a2 f1 f2 A B
                 i
                 a v b
                 H).
```

5.2.3 parametrized W type

————-begin file rules_pw3.v ————

Let us now prove the truth of rules about the parametrized W type.

The parametrized W induction rule can be stated as follows:

```
H, w : pW(P;ap.A;bp,ba.B;cp,ca,cb.C;p) |- Q(p)

  By pWinduction a f v

  H, q : P
     a : A(q)
     f : b:B(q,a) -> pW(P;ap.A;bp,ba.B;cp,ca,cb.C;C(q,a,b))
     v : b:B(q,a) -> Q[w\f(b)](C(q,a,b))
   |- Q[w\sup(a,f)](q)
```

```
Definition rule_pw_induction
            (H : barehypotheses)
            (Q P A B C p : NTerm)
            (i : nat)
            (ap bp ba cp ca cb w b a f v q : NVar) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp w (mk_pw P ap A bp ba B cp ca cb C
p)))
                    (mk_conclax (mk_apply Q p)))
    [ mk_baresequent
        (snoc (snoc (snoc (snoc H (mk_hyp q P))
                              (mk_hyp a (lsubst A [(ap,mk_var q)])))
                        (mk_hyp f (mk_function
                                     (lsubst B [(bp,mk_var q),(ba,mk_var a)])
                                     b
                                     (mk_pw P ap A bp ba B cp ca cb C
                                            (lsubst C [(cp,mk_var q),(ca,mk_var
a),(cb,mk_var b)])))))
                 (mk_hyp v (mk_function
                              (lsubst B [(bp,mk_var q),(ba,mk_var a)])
                              b
                              (mk_apply
                                 (lsubst Q [(w,mk_apply (mk_var f) (mk_var
b))])
                                 (lsubst C [(cp,mk_var q),(ca,mk_var a),(cb,mk_var
b)])))))
        (mk_conclax (mk_apply (lsubst Q [(w,mk_sup (mk_var a) (mk_var f))])
(mk_var q)))
```

```
    ]
    [sarg_var a, sarg_var f, sarg_var v].
Lemma rule_pw_induction_true :
  ∀ (H : barehypotheses)
         (Q P A B C p : NTerm)
         (i : nat)
         (ap bp ba cp ca cb w b a f v q : NVar)
         (bc1 : disjoint (free_vars p) (bound_vars A))
         (bc2 : disjoint (free_vars p) (bound_vars B))
         (bc3 : !LIn a (bound_vars B))
         (bv4 : !(ba = bp))
         (bc5 : !LIn q (bound_vars A))
         (bc6 : !LIn q (bound_vars B))
         (bc7 : !LIn b (w :: q :: a :: f :: v :: vars_hyps H))
         (bc8 : !LIn q (bound_vars C))
         (bc9 : !LIn a (bound_vars C))
         (bc10 : !LIn b (bound_vars C))
         (bc11 : !LIn b (bound_vars Q))
         (bc12 : !LIn f (bound_vars Q))
         (bc13 : !LIn q (bound_vars Q))
         (bc14 : !LIn w [q,a,f,v])
         (bc15 : !LIn a (bound_vars Q)),
    rule_true (rule_pw_induction H Q P A B C p i ap bp ba cp ca cb w b a f v
q).
```

5.2.4 Extensional PER type

———-begin file rules_pertype.v ————

We now prove the truth of several rules about the PER type.

The following rule is called the "pertype member equality" rule. It allows one to prove that terms are well-formed partial equivalence relations, i.e., members of a "pertype" type.

```
H |- t1 = t2 in pertype(R)

  By pertypeMemberEquality i

  H |- pertype(R) in Type(i)
```

```
    H |- R t1 t2
    H |- t1 in Base
```

```
Definition rule_pertype_member_equality
           (t1 t2 R e : NTerm)
           (i : nat)
           (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality t1 t2 (mk_pertype R))))
    [ mk_baresequent H (mk_conclax (mk_member (mk_pertype R) (mk_uni
i))),
      mk_baresequent H (mk_concl (mk_apply2 R t1 t2) e),
      mk_baresequent H (mk_conclax (mk_member t1 mk_base))
    ]
    [].
Lemma rule_pertype_member_equality_true :
  ∀ t1 t2 R e : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_pertype_member_equality
                 t1 t2 R e
                 i
                 H).
```

We can state the pertype elimination rule as follows:

```
H, x : t1 = t2 in pertype(R), J |- C ext e

  By pertypeElimination i z

  H, x : t1 = t2 in pertype(R), [z : R t1 t2], J |- C ext e
  H |- R t1 t2 in Type(i)
```

```
Definition rule_pertype_elimination
           (R t1 t2 C e : NTerm)
           (x z : NVar)
           (i : nat)
           (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
        (snoc H (mk_hyp x (mk_equality t1 t2 (mk_pertype R))) ++ J)
        (mk_concl C e))
    [ mk_baresequent
         (snoc (snoc H (mk_hyp x (mk_equality t1 t2 (mk_pertype R))))
               (mk_hhyp z (mk_apply2 R t1 t2))
               ++ J)
         (mk_concl C e),
      mk_baresequent
         H
         (mk_conclax (mk_member (mk_apply2 R t1 t2) (mk_uni i)))
    ]
    [sarg_var z].
Lemma rule_pertype_elimination_true :
  ∀ R t1 t2 C e : NTerm,
  ∀ x z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_pertype_elimination
                 R t1 t2 C e
                 x z
                 i
                 H J).
```

This is a variant of rule_pertype_elimination but where the well-formedness goal is stated using a “type” sequent. Also it allows one to use all the hypotheses:

```
   H, x : t1 = t2 in pertype(R), J |- C ext e

     By pertypeElimination i z

     H, x : t1 = t2 in pertype(R), [z : R t1 t2], J |- C ext e
     H, x : t1 = t2 in pertype(R), J |- R t1 t2 is a type
```

```
Definition rule_pertype_elimination_t
             (R t1 t2 C e : NTerm)
             (x z : NVar)
             (i : nat)
             (H J : barehypotheses) :=
  mk_rule
```

```
    (mk_baresequent
       (snoc H (mk_hyp x (mk_equality t1 t2 (mk_pertype R))) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_equality t1 t2 (mk_pertype R))))
              (mk_hhyp z (mk_apply2 R t1 t2))
              ++ J)
        (mk_concl C e),
      mk_baresequent
        (snoc H (mk_hyp x (mk_equality t1 t2 (mk_pertype R))) ++ J)
        (mk_concl_t (mk_apply2 R t1 t2))
    ]
    [sarg_var z].

Lemma rule_pertype_elimination_t_true :
  ∀ R t1 t2 C e : NTerm,
  ∀ x z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_pertype_elimination_t
                 R t1 t2 C e
                 x z
                 i
                 H J).
```

We can state the pertype equality rule as follows:

```
H |- pertype(R1) = pertype(R2) in Type(i)

  By pertypeMemberEquality x y z u v

  H, x : Base, y : Base |- R1 x y in Type(i)
  H, x : Base, y : Base |- R2 x y in Type(i)
  H, x : Base, y : Base, z : R1 x y |- R2 x y
  H, x : Base, y : Base, z : R2 x y |- R1 x y
  H, x : Base, y : Base, z : R1 x y |- R1 y x
  H, x : Base, y : Base, z : Base, u : R1 z y, v : R1 y z |- R1 x z
```

```
Definition rule_pertype_equality
             (R1 R2 e1 e2 e3 e4 : NTerm)
             (x y z u v : NVar)
```

```
                 (i : nat)
                 (H : barehypotheses) :=
   mk_rule
      (mk_baresequent H (mk_conclax (mk_equality (mk_pertype R1) (mk_pertype
R2) (mk_uni i))))
      [ mk_baresequent
            (snoc (snoc H (mk_hyp x mk_base)) (mk_hyp y mk_base))
            (mk_conclax (mk_member
                              (mk_apply2 R1 (mk_var x) (mk_var y))
                              (mk_uni i))),
         mk_baresequent
            (snoc (snoc H (mk_hyp x mk_base)) (mk_hyp y mk_base))
            (mk_conclax (mk_member
                              (mk_apply2 R2 (mk_var x) (mk_var y))
                              (mk_uni i))),
         mk_baresequent
            (snoc (snoc (snoc H (mk_hyp x mk_base))
                             (mk_hyp y mk_base))
                     (mk_hyp z (mk_apply2 R1 (mk_var x) (mk_var y))))
            (mk_concl (mk_apply2 R2 (mk_var x) (mk_var y)) e1),
         mk_baresequent
            (snoc (snoc (snoc H (mk_hyp x mk_base))
                             (mk_hyp y mk_base))
                     (mk_hyp z (mk_apply2 R2 (mk_var x) (mk_var y))))
            (mk_concl (mk_apply2 R1 (mk_var x) (mk_var y)) e2),
         mk_baresequent
            (snoc (snoc (snoc H (mk_hyp x mk_base))
                             (mk_hyp y mk_base))
                     (mk_hyp z (mk_apply2 R1 (mk_var x) (mk_var y))))
            (mk_concl (mk_apply2 R1 (mk_var y) (mk_var x)) e3),
         mk_baresequent
            (snoc (snoc (snoc (snoc (snoc H (mk_hyp x mk_base))
                                           (mk_hyp y mk_base))
                                   (mk_hyp z mk_base))
                             (mk_hyp u (mk_apply2 R1 (mk_var x) (mk_var y))))
                     (mk_hyp v (mk_apply2 R1 (mk_var y) (mk_var z))))
            (mk_concl (mk_apply2 R1 (mk_var x) (mk_var z)) e4)
      ]
      [].

Lemma rule_pertype_equality_true :
```

```
∀ R1 R2 e1 e2 e3 e4 : NTerm,
∀ x y z u v : NVar,
∀ i : nat,
∀ H : barehypotheses,
  rule_true (rule_pertype_equality
               R1 R2 e1 e2 e3 e4
               x y z u v
               i
               H).
```

Let us now provide another pertype elimination rule. This version is stated in terms of a pertype hypothesis and not an equality in a pertype:

```
  H, x : pertype(R), J |- C ext e

    By pertypeElimination i z

    H, x : pertype(R), [z : R x x], J |- C ext e
    H, x : pertype(R) |- R x x in Type(i)
```

```
Definition rule_pertype_elimination2
             (R C e : NTerm)
             (x z : NVar)
             (i : nat)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x (mk_pertype R)) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_pertype R)))
              (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var x)))
              ++ J)
        (mk_concl C e),
      mk_baresequent
        (snoc H (mk_hyp x (mk_pertype R)))
        (mk_conclax (mk_member
                       (mk_apply2 R (mk_var x) (mk_var x))
                       (mk_uni i)))
    ]
    [sarg_var z].
Lemma rule_pertype_elimination2_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_pertype_elimination2
                 R C e
                 x z
                 i
                 H J).
```

We state yet another pertype elimination rule. This one is similar to the second one presented above but does not require us to provide a level for R x x:

```
   H, x : pertype(R), J |- C ext e

     By pertypeElimination i z

     H, x : pertype(R), [z : R x x], J |- C ext e
     H, x : pertype(R) |- (R x x) is a type
```

```
Definition rule_pertype_elimination3
             (R C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp x (mk_pertype R)) ++ J)
                    (mk_concl C e))
    [ mk_baresequent (snoc (snoc H (mk_hyp x (mk_pertype R)))
                               (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var x)))
                               ++ J)
                    (mk_concl C e),
      mk_baresequent (snoc H (mk_hyp x (mk_pertype R)))
                     (mk_concl_t (mk_apply2 R (mk_var x) (mk_var x)))
    ]
    [sarg_var z].
Lemma rule_pertype_elimination3_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_pertype_elimination3
                 R C e
                 x z
                 H J).
```

We state yet another pertype elimination rule. This one is similar to the third one presented above but $R\ x\ x$ is now the last hypothesis in the first subgoal:

```
  H, x : pertype(R), J |- C ext e

    By pertypeElimination i z

    H, x : pertype(R), [z : R x x], J |- C ext e
    H, x : pertype(R) |- (R x x) is a type
```

```
Definition rule_pertype_elimination4
             (R C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp x (mk_pertype R)) ++ J)
                    (mk_concl C e))
    [ mk_baresequent (snoc ((snoc H (mk_hyp x (mk_pertype R))) ++ J)
                           (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var
x))))
                    (mk_concl C e),
      mk_baresequent (snoc H (mk_hyp x (mk_pertype R)))
                     (mk_concl_t (mk_apply2 R (mk_var x) (mk_var x)))
    ]
    [sarg_var z].

Lemma rule_pertype_elimination4_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_pertype_elimination4
                  R C e
                  x z
                  H J).
```

5.2.5 Intensional PER type

———-begin file rules_ipertype.v ————

We now prove the truth of several rules about the intensional PER type.

The following rule is called the "ipertype member equality" rule. It allows one to prove that terms are well-formed partial equivalence relations, i.e., members of a "ipertype" type.

```
   H |- t1 = t2 in ipertype(R)

     By ipertypeMemberEquality i

     H |- ipertype(R) in Type(i)
     H |- R t1 t2
     H |- t1 in Base
```

```
Definition rule_ipertype_member_equality
             (t1 t2 R e : NTerm)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality t1 t2 (mk_ipertype R))))
    [ mk_baresequent H (mk_conclax (mk_member (mk_ipertype R) (mk_uni
i))),
      mk_baresequent H (mk_concl (mk_apply2 R t1 t2) e),
      mk_baresequent H (mk_conclax (mk_member t1 mk_base))
    ]
    [].

Lemma rule_ipertype_member_equality_true :
  ∀ t1 t2 R e : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_ipertype_member_equality
                  t1 t2 R e
```

```
                    i
                    H).
```

We can state the ipertype elimination rule as follows:

```
H, x : t1 = t2 in ipertype(R), J |- C ext e

  By ipertypeElimination i z

  H, x : t1 = t2 in ipertype(R), [z : R t1 t2], J |- C ext e
  H |- R t1 t2 in Type(i)
```

```
Definition rule_ipertype_elimination
           (R t1 t2 C e : NTerm)
           (x z : NVar)
           (i : nat)
           (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x (mk_equality t1 t2 (mk_ipertype R))) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_equality t1 t2 (mk_ipertype R))))
              (mk_hhyp z (mk_apply2 R t1 t2))
              ++ J)
        (mk_concl C e),
      mk_baresequent
        H
        (mk_conclax (mk_member (mk_apply2 R t1 t2) (mk_uni i)))
    ]
    [sarg_var z].

Lemma rule_ipertype_elimination_true :
  ∀ R t1 t2 C e : NTerm,
  ∀ x z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_ipertype_elimination
                 R t1 t2 C e
                 x z
                 i
```

```
                    H J).
```

This is a variant of rule_ipertype_elimination but where the well-formedness goal is stated using a "type" sequent. Also it allows one to use all the hypotheses:

```
H, x : t1 = t2 in ipertype(R), J |- C ext e

  By ipertypeElimination i z

  H, x : t1 = t2 in ipertype(R), [z : R t1 t2], J |- C ext e
  H, x : t1 = t2 in ipertype(R), J |- R t1 t2 is a type
```

```
Definition rule_ipertype_elimination_t
           (R t1 t2 C e : NTerm)
           (x z : NVar)
           (i : nat)
           (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x (mk_equality t1 t2 (mk_ipertype R))) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_equality t1 t2 (mk_ipertype R))))
              (mk_hhyp z (mk_apply2 R t1 t2))
              ++ J)
        (mk_concl C e),
      mk_baresequent
        (snoc H (mk_hyp x (mk_equality t1 t2 (mk_ipertype R))) ++ J)
        (mk_concl_t (mk_apply2 R t1 t2))
    ]
    [sarg_var z].

Lemma rule_ipertype_elimination_t_true :
  ∀ R t1 t2 C e : NTerm,
  ∀ x z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_ipertype_elimination_t
                 R t1 t2 C e
                 x z
```

```
                    i
                    H J).
```

We can state the ipertype equality rule as follows:

```
H |- ipertype(R1) = ipertype(R2) in Type(i)

  By ipertypeMemberEquality x y z u v

  H, x : Base, y : Base |- R1 x y = R2 x y in Type(i)
  H, x : Base, y : Base, z : R1 x y |- R1 y x
  H, x : Base, y : Base, z : Base, u : R1 z y, v : R1 y z |- R1 x z
```

```
Definition rule_ipertype_equality
             (R1 R2 e1 e2 : NTerm)
             (x y z u v : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality (mk_ipertype R1) (mk_ipertype
R2) (mk_uni i))))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x mk_base)) (mk_hyp y mk_base))
        (mk_conclax (mk_equality
                       (mk_apply2 R1 (mk_var x) (mk_var y))
                       (mk_apply2 R2 (mk_var x) (mk_var y))
                       (mk_uni i))),
      mk_baresequent
        (snoc (snoc (snoc H (mk_hyp x mk_base))
                    (mk_hyp y mk_base))
              (mk_hyp z (mk_apply2 R1 (mk_var x) (mk_var y))))
        (mk_concl (mk_apply2 R1 (mk_var y) (mk_var x)) e1),
      mk_baresequent
        (snoc (snoc (snoc (snoc (snoc H (mk_hyp x mk_base))
                                      (mk_hyp y mk_base))
                                (mk_hyp z mk_base))
                          (mk_hyp u (mk_apply2 R1 (mk_var x) (mk_var y))))
              (mk_hyp v (mk_apply2 R1 (mk_var y) (mk_var z))))
        (mk_concl (mk_apply2 R1 (mk_var x) (mk_var z)) e2)
    ]
    [].

Lemma rule_ipertype_equality_true :
  ∀ R1 R2 e1 e2 : NTerm,
  ∀ x y z u v : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_ipertype_equality
                 R1 R2 e1 e2
                 x y z u v
                 i
                 H).
```

Let us now provide another ipertype elimination rule. This version is stated in terms of a ipertype hypothesis and not an equality in a ipertype:

```
H, x : ipertype(R), J |- C ext e

  By ipertypeElimination i z

  H, x : ipertype(R), [z : R x x], J |- C ext e
  H, x : ipertype(R) |- R x x in Type(i)
```

```
Definition rule_ipertype_elimination2
             (R C e : NTerm)
             (x z : NVar)
             (i : nat)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x (mk_ipertype R)) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_ipertype R)))
              (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var x)))
              ++ J)
        (mk_concl C e),
      mk_baresequent
        (snoc H (mk_hyp x (mk_ipertype R)))
        (mk_conclax (mk_member
                          (mk_apply2 R (mk_var x) (mk_var x))
                          (mk_uni i)))
    ]
    [sarg_var z].
```

```
Lemma rule_ipertype_elimination2_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_ipertype_elimination2
                 R C e
                 x z
                 i
                 H J).
```

We state yet another ipertype elimination rule. This one is similar to the second one presented above but does not require us to provide a level for $R\ x\ x$:

```
   H, x : ipertype(R), J |- C ext e

     By ipertypeElimination i z

     H, x : ipertype(R), [z : R x x], J |- C ext e
     H, x : ipertype(R) |- (R x x) is a type
```

```
Definition rule_ipertype_elimination3
             (R C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp x (mk_ipertype R)) ++ J)
                    (mk_concl C e))
    [ mk_baresequent (snoc (snoc H (mk_hyp x (mk_ipertype R)))
                                (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var x)))
                                ++ J)
                    (mk_concl C e),
      mk_baresequent (snoc H (mk_hyp x (mk_ipertype R)))
                     (mk_concl_t (mk_apply2 R (mk_var x) (mk_var x)))
    ]
    [sarg_var z].
Lemma rule_ipertype_elimination3_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_ipertype_elimination3
                 R C e
                 x z
                 H J).
```

We state yet another ipertype elimination rule. This one is similar to the third one presented above but $R\ x\ x$ is now the last hypothesis in the first subgoal:

```
  H, x : ipertype(R), J |- C ext e

    By ipertypeElimination i z

    H, x : ipertype(R), J, [z : R x x] |- C ext e
    H, x : ipertype(R) |- (R x x) is a type
```

```
Definition rule_ipertype_elimination4
             (R C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp x (mk_ipertype R)) ++ J)
                    (mk_concl C e))
    [ mk_baresequent (snoc ((snoc H (mk_hyp x (mk_ipertype R))) ++ J)
                           (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var
x))))
                    (mk_concl C e),
      mk_baresequent (snoc H (mk_hyp x (mk_ipertype R)))
                     (mk_concl_t (mk_apply2 R (mk_var x) (mk_var x)))
    ]
    [sarg_var z].
Lemma rule_ipertype_elimination4_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_ipertype_elimination4
                    R C e
                    x z
                    H J).
```

5.2.6 Yet Another Intensional PER type

————-begin file rules_spertype.v ————

We now prove the truth of several rules about the intensional PER type.

The following rule is called the “spertype member equality” rule. It allows one to prove that terms are well-formed partial equivalence relations, i.e., members of a “spertype” type.

```
H |- t1 = t2 in spertype(R)

  By spertypeMemberEquality i

  H |- spertype(R) in Type(i)
  H |- R t1 t2
  H |- t1 in Base
```

```
Definition rule_spertype_member_equality
           (t1 t2 R e : NTerm)
           (i : nat)
           (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality t1 t2 (mk_spertype R))))
    [ mk_baresequent H (mk_conclax (mk_member (mk_spertype R) (mk_uni
i))),
      mk_baresequent H (mk_concl (mk_apply2 R t1 t2) e),
      mk_baresequent H (mk_conclax (mk_member t1 mk_base))
    ]
    [].

Lemma rule_spertype_member_equality_true :
  ∀ t1 t2 R e : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_spertype_member_equality
                 t1 t2 R e
                 i
                 H).
```

We can state the spertype elimination rule as follows:

```
H, x : t1 = t2 in spertype(R), J |- C ext e

  By spertypeElimination z

  H, x : t1 = t2 in spertype(R), [z : R t1 t2], J |- C ext e
```

```
Definition rule_spertype_elimination
             (R t1 t2 C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x (mk_equality t1 t2 (mk_spertype R))) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_equality t1 t2 (mk_spertype R))))
              (mk_hhyp z (mk_apply2 R t1 t2))
              ++ J)
        (mk_concl C e)
    ]
    [sarg_var z].

Lemma rule_spertype_elimination_true :
  ∀ R t1 t2 C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_spertype_elimination
                 R t1 t2 C e
                 x z
                 H J).
```

We can state the spertype equality rule as follows:

```
H |- spertype(R1) = spertype(R2) in Type(i)

  By spertypeMemberEquality x y z u v

  H, x : Base, y : Base |- R1 x y = R2 x y in Type(i)
  H, x : Base, y : Base, z : R1 x y |- R1 y x
  H, x : Base, y : Base, z : Base, u : R1 z y, v : R1 y z |- R1 x z
  H, x : Base, y : Base, z : Base, u : R1 x z |- R1 x y = R1 z y in Type(i)
  H, x : Base, y : Base, z : Base, u : R1 y z |- R1 x y = R1 x z in Type(i)
```

```
Definition rule_spertype_equality
           (R1 R2 e1 e2 : NTerm)
           (x y z u v : NVar)
           (i : nat)
           (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality (mk_spertype R1) (mk_spertype
R2) (mk_uni i))))
    [ mk_bseq
        (snoc (snoc H (mk_hyp x mk_base)) (mk_hyp y mk_base))
        (mk_conclax (mk_equality
                       (mk_apply2 R1 (mk_var x) (mk_var y))
                       (mk_apply2 R2 (mk_var x) (mk_var y))
                       (mk_uni i))),
      mk_bseq
        (snoc (snoc (snoc H (mk_hyp x mk_base))
                    (mk_hyp y mk_base))
              (mk_hyp z (mk_apply2 R1 (mk_var x) (mk_var y))))
        (mk_concl (mk_apply2 R1 (mk_var y) (mk_var x)) e1),
      mk_bseq
        (snoc (snoc (snoc (snoc (snoc H (mk_hyp x mk_base))
                                (mk_hyp y mk_base))
                          (mk_hyp z mk_base))
                    (mk_hyp u (mk_apply2 R1 (mk_var x) (mk_var y))))
              (mk_hyp v (mk_apply2 R1 (mk_var y) (mk_var z))))
        (mk_concl (mk_apply2 R1 (mk_var x) (mk_var z)) e2),
      mk_bseq
        (snoc (snoc (snoc (snoc H (mk_hyp x mk_base))
                          (mk_hyp y mk_base))
                    (mk_hyp z mk_base))
              (mk_hyp u (mk_apply2 R1 (mk_var x) (mk_var z))))
          (mk_conclax (mk_equality
                          (mk_apply2 R1 (mk_var x) (mk_var y))
                          (mk_apply2 R1 (mk_var z) (mk_var y))
                          (mk_uni i))),
        mk_bseq
          (snoc (snoc (snoc (snoc H (mk_hyp x mk_base))
                                (mk_hyp y mk_base))
                          (mk_hyp z mk_base))
              (mk_hyp u (mk_apply2 R1 (mk_var y) (mk_var z))))
          (mk_conclax (mk_equality
                          (mk_apply2 R1 (mk_var x) (mk_var y))
                          (mk_apply2 R1 (mk_var x) (mk_var z))
                          (mk_uni i)))
      ]
      [].

Lemma rule_spertype_equality_true :
  ∀ R1 R2 e1 e2 : NTerm,
  ∀ x y z u v : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_spertype_equality
                 R1 R2 e1 e2
                 x y z u v
                 i
                 H).
```

Let us now provide another spertype elimination rule. This version is stated in terms of a spertype hypothesis and not an equality in a spertype:

```
H, x : spertype(R), J |- C ext e

  By spertypeElimination i z

  H, x : spertype(R), [z : R x x], J |- C ext e
```

```
Definition rule_spertype_elimination2
             (R C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x (mk_spertype R)) ++ J)
       (mk_concl C e))
    [ mk_baresequent
        (snoc (snoc H (mk_hyp x (mk_spertype R)))
              (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var x)))
              ++ J)
        (mk_concl C e)
    ]
    [sarg_var z].

Lemma rule_spertype_elimination2_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_spertype_elimination2
                 R C e
                 x z
                 H J).
```

We state yet another spertype elimination rule. This one is similar to rule_spertype_elimination2 but $R\ x\ x$ is now the last hypothesis in the first subgoal:

```
   H, x : spertype(R), J |- C ext e

     By spertypeElimination i z

     H, x : spertype(R), J, [z : R x x] |- C ext e
```

```
Definition rule_spertype_elimination4
             (R C e : NTerm)
             (x z : NVar)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp x (mk_spertype R)) ++ J)
                    (mk_concl C e))
    [ mk_baresequent (snoc ((snoc H (mk_hyp x (mk_spertype R))) ++ J)
                           (mk_hhyp z (mk_apply2 R (mk_var x) (mk_var
x))))
```

```
                          (mk_concl C e)
      ]
      [sarg_var z].
```

```
Lemma rule_spertype_elimination4_true :
  ∀ R C e : NTerm,
  ∀ x z : NVar,
  ∀ H J : barehypotheses,
    rule_true (rule_spertype_elimination4
                  R C e
                  x z
                  H J).
```

5.2.7 Intersection type

———-begin file rules_isect.v ————

We now prove the truth of several rules about the intersection type.

We can state the intersection member formation rule as follows:

```
H |- isect x:A. B ext b

  By isect_memberFormation lvl(i) z ()

  H [z : A] |- subst B x z ext b
  H |- A = A in Type(i)
```

```
Definition rule_isect_member_formation
             (A B b : NTerm)
             (x z : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_concl (mk_isect A x B) b))
    [ mk_baresequent
         (snoc H (mk_hhyp z A))
         (mk_concl (subst B x (mk_var z)) b),
      mk_baresequent H (mk_conclax (mk_equality A A (mk_uni i))) ]
    [sarg_var z].
```

```
Lemma rule_isect_member_formation_true :
```

```
∀ (A B b : NTerm)
        (x z : NVar)
        (i : nat)
        (H : barehypotheses)
        (bc1 : !LIn z (bound_vars B)),
  rule_true (rule_isect_member_formation A B b x z i H).
```

We can state the intersection equality rule as follows:

```
H |- isect x1:a1. b1 = isect x2:a2.b2 in Type(i)

  By isectEquality y ()

  H |- a1 = a2 in Type(i)
  H y : a1 |- subst b1 x1 y = subst b2 x2 y in Type(i)
```

```
Definition rule_isect_equality
             (a1 a2 b1 b2 : NTerm)
             (x1 x2 y : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent
       H
       (mk_conclax (mk_equality
                      (mk_isect a1 x1 b1)
                      (mk_isect a2 x2 b2)
                      (mk_uni i))))
    [ mk_baresequent
        H
        (mk_conclax (mk_equality a1 a2 (mk_uni i))),
      mk_baresequent
        (snoc H (mk_hyp y a1))
        (mk_conclax (mk_equality
                       (subst b1 x1 (mk_var y))
                       (subst b2 x2 (mk_var y))
                       (mk_uni i)))
    ]
    [ sarg_var y ].

Lemma rule_isect_equality_true :
  ∀ a1 a2 b1 b2 : NTerm,
  ∀ x1 x2 y : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
  ∀ bc1 : !LIn y (bound_vars b1),
  ∀ bc2 : !LIn y (bound_vars b2),
    rule_true (rule_isect_equality
                 a1 a2 b1 b2
                 x1 x2 y
                 i
                 H).
```

————-begin file rules_isect2.v ————

We state the intersection member equality rule as follows:

```
H |- b1 = b2 in isect x:A. B

  By isect_memberEquality lvl(i) z ()

  H, z : A |- b1 = b2 in subst B x z
  H |- A = A in Type(i)
```

```
Definition rule_isect_member_equality
             (A B b1 b2 : NTerm)
             (x z : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality b1 b2 (mk_isect A x B))))
    [ mk_baresequent
        (snoc H (mk_hyp z A))
        (mk_conclax (mk_equality b1 b2 (subst B x (mk_var z)))),
      mk_baresequent H (mk_conclax (mk_member A (mk_uni i)))
    ]
    [sarg_var z].

Lemma rule_isect_member_equality_true :
  ∀ (A B b1 b2 : NTerm)
           (x z : NVar)
           (i : nat)
           (H : barehypotheses)
```

```
          (bc1 : !LIn z (bound_vars B)),
      rule_true (rule_isect_member_equality A B b1 b2 x z i H).
```

5.2.8 Computational approximation and equivalence types

———-begin file rules_squiggle.v ————

We now prove the truth of several rules about the approximation and computational equivalence types.

We can state the computational equivalence reflexivity rule as follows:

```
H |- a ~ a
  no subgoals
```

```
Definition rule_cequiv_refl
             (H : barehypotheses)
             (a : NTerm) :=
  mk_rule (mk_baresequent H (mk_conclax (mk_cequiv a a))) [] [].

Lemma rule_cequiv_refl_true :
  ∀ H : barehypotheses,
  ∀ a : NTerm,
    rule_true (rule_cequiv_refl H a).
```

We can state the approximation reflexivity rule as follows:

```
H |- a <= a
  no subgoals
```

```
Definition rule_approx_refl
             (H : barehypotheses)
             (a : NTerm) :=
  mk_rule (mk_baresequent H (mk_conclax (mk_approx a a))) [] [].

Lemma rule_approx_refl_true :
  ∀ H : barehypotheses,
  ∀ a : NTerm,
    rule_true (rule_approx_refl H a).
```

We can state the computational equivalence intensional equality rule as follows:

```
|- a ~ b = a ~ b in Type(i)

  By cequivIntensionalEquality (simple)
```

```
Definition rule_cequiv_intensional_equality_simple
           (i : nat)
           (a b : NTerm) :=
  mk_rule
    (mk_baresequent []
                    (mk_conclax (mk_equality (mk_cequiv a b)
                                             (mk_cequiv a b)
                                             (mk_uni i))))
    []
    [].

Lemma rule_cequiv_intensional_equality_simple_true :
  ∀ (i : nat)
         (a b : NTerm),
    rule_true (rule_cequiv_intensional_equality_simple i a b).
```

We can state the approximation intensional equality rule as follows:

```
|- a <= b = a <= b in Type(i)

  By approxIntensionalEquality (simple)
```

```
Definition rule_approx_intensional_equality_simple
           (i : nat)
           (a b : NTerm) :=
  mk_rule
    (mk_baresequent []
                    (mk_conclax (mk_equality (mk_approx a b)
                                             (mk_approx a b)
                                             (mk_uni i))))
    []
    [].

Lemma rule_approx_intensional_equality_simple_true :
  ∀ (i : nat)
         (a b : NTerm),
    rule_true (rule_approx_intensional_equality_simple i a b).
```

The following rule says that to prove that a is computationally equivalent to b it is enough to prove that they are equal in *Base*:

```
H |- a ~ b
  H |- a = b in Base
```

```
Definition rule_cequiv_base
             (H : barehypotheses)
             (a b : NTerm) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_cequiv a b)))
    [ mk_baresequent H (mk_conclax (mk_equality a b mk_base))
    ]
    [].
```

```
Lemma rule_cequiv_base_true :
  ∀ H : barehypotheses,
  ∀ a b : NTerm,
    rule_true (rule_cequiv_base H a b).
```

The following rule says that to prove that a is computationally equivalent to b it is enough to prove that a is an approximation of b and vice versa:

```
H |- a ~ b
  H |- a <= b
  H |- b <= a
```

```
Definition rule_cequiv_approx
             (H : barehypotheses)
             (a b : NTerm) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_cequiv a b)))
    [ mk_baresequent H (mk_conclax (mk_approx a b)),
      mk_baresequent H (mk_conclax (mk_approx b a))
    ]
    [].
```

```
Lemma rule_cequiv_approx_true :
  ∀ H : barehypotheses,
  ∀ a b : NTerm,
    rule_true (rule_cequiv_approx H a b).
```

————-begin file rules_squiggle2.v ————

The following rule states that we can always replace an *a* for a *b* in the conclusion of a sequent if *a* and *b* are computationally equivalent:

```
H |- C[a] ext t

  By cequivSubstConcl

  H |- C[b] ext t
  H |- a ~ b
```

```
Definition rule_cequiv_subst_concl
           (H : barehypotheses)
           (x : NVar)
           (C a b t : NTerm) :=
  mk_rule
    (mk_baresequent H (mk_concl (subst C x a) t))
    [ mk_baresequent H (mk_concl (subst C x b) t),
      mk_baresequent H (mk_conclax (mk_cequiv a b))
    ]
    [].

Lemma rule_cequiv_subst_concl_true :
  ∀ (H : barehypotheses)
         (x : NVar)
         (C a b t : NTerm)
         (bc1 : disjoint (free_vars a) (bound_vars C))
         (bc2 : disjoint (free_vars b) (bound_vars C)),
    rule_true (rule_cequiv_subst_concl H x C a b t).
```

The following rule states that we can always replace an *a* for a *b* in an hypothesis of a sequent if *a* and *b* are computationally equivalent:

```
H, z : T[a], J |- C ext t

  By cequivSubstHyp

  H, z : T[b], J |- C ext t
  H, z : T[a], J |- a ~ b
```

```
Definition rule_cequiv_subst_hyp
           (H J : barehypotheses)
```

```
              (x z : NVar)
              (C T a b t : NTerm) :=
  mk_rule
    (mk_baresequent (snoc H (mk_hyp z (subst T x a)) ++ J) (mk_concl C t))
    [ mk_baresequent (snoc H (mk_hyp z (subst T x b)) ++ J) (mk_concl C t),
      mk_baresequent (snoc H (mk_hyp z (subst T x a)) ++ J) (mk_conclax
(mk_cequiv a b))
    ]
    [].
```

```
Lemma rule_cequiv_subst_hyp_true :
  ∀ (H J : barehypotheses)
           (x z : NVar)
           (C T a b t : NTerm)
           (bc1 : disjoint (free_vars a) (bound_vars T))
           (bc2 : disjoint (free_vars b) (bound_vars T)),
    rule_true (rule_cequiv_subst_hyp H J x z C T a b t).
```

5.2.9 Partial type

————-begin file rules_partial.v ————

In this section we prove rules about partial types. We assume that the reader has seen the definitions of per_partial, fix_approxc, per_mono and per_admiss in 4.2.1 We will also use the domain theoretic properties we proved in Sec. 3.3.

We first characterize the central content of the formation rule for partial types. The proof is a straightforward consequence of the definition of per_partial. Like Crary[21], we can form the partial type mkc_partial T only if the T is a total type. A type is a total type if all its members converge to a value. chaltsc is a lifted version of hasvalue, i.e., chaltsc t asserts t converges to a cononical form.

```
Lemma tequality_mkc_partial :
  ∀ T1 T2,
  tequality (mkc_partial T1) (mkc_partial T2)
  ⇔ (tequality T1 T2 × (∀ t, member t T1 → chaltsc t)).
```

Now, We characterize when two elements are equal in a partial type. The proof is a straightforward consequence of the definition of per_partial_eq that is used in per_partial.

```
Lemma equality_in_mkc_partial :
  ∀ t t' T,
    equality t t' (mkc_partial T)
    ⇔ (type (mkc_partial T)
         × (chaltsc t ⇔ chaltsc t')
         × (chaltsc t → equality t t' T)).
```

The following rule states when two partial types are equal.

```
H |- partial(A) = partial(B) in Type(i)

  By partialEquality a ()

  H |- A = B in Type(i)
  H a : A |- halts(a)
```

```
Definition rule_partial_equality
             (A B : NTerm)
             (a : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality (mk_partial A) (mk_partial
B) (mk_uni i))))
    [ mk_baresequent H (mk_conclax (mk_equality A B (mk_uni i))),
      mk_baresequent
        (snoc H (mk_hyp a A))
        (mk_conclax (mk_halts (mk_var a)))
    ]
    [ sarg_var a ].

Lemma rule_partial_equality_true :
  ∀ A B : NTerm,
  ∀ a : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_partial_equality A B a i H).
```

To prove that two terms are equal in a partial(A), it is enough to prove that they are equal in A:

```
H |- a = b in partial(A)
```

```
     By partialInclusion (i)

     H |- a = b in A
     H |- partial(A) in Type(i)
```

```
Definition rule_partial_inclusion
             (a b A : NTerm)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality a b (mk_partial A))))
    [ mk_baresequent H (mk_conclax (mk_equality a b A)),
      mk_baresequent H (mk_conclax (mk_member (mk_partial A) (mk_uni i)))
    ]
    [].
```

```
Lemma rule_partial_inclusion_true :
  ∀ a b A : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_partial_inclusion a b A i H).
```

To prove that two terms a and b are equal in a partial(A), it is enough to prove that they are equal in T assuming that a has a value; that a and b have the same convergence behavior; and that partial(A) is a type:

```
  H |- a = b in partial(A)

     By partialInclusion z i

     H, z : halts(a) |- a = b in A
     H |- halts(a) <=> halts(b)
     H |- partial(A) in Type(i)
```

```
Definition rule_partial_member_equality
             (a b A e : NTerm)
             (z : NVar)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
```

```
      (mk_baresequent H (mk_conclax (mk_equality a b (mk_partial A))))
      [ mk_baresequent (snoc H (mk_hyp z (mk_halts a))) (mk_conclax (mk_equality
a b A)),
        mk_baresequent H (mk_concl (mk_iff (mk_halts a) (mk_halts b)) e),
        mk_baresequent H (mk_conclax (mk_member (mk_partial A) (mk_uni i)))
      ]
      [].

Lemma rule_partial_member_equality_true :
  ∀ a b A e : NTerm,
  ∀ z : NVar,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_partial_member_equality a b A e z i H).
```

To prove that two terms a and b are equal in a type A, it is enough to prove that a converges and that they are equal in mk_partial(A):

```
  H |- a = b in A

    By termination

    H |- halts(a)
    H |- a = b in partial(A)
```

```
Definition rule_termination
             (a b A : NTerm)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality a b A)))
    [ mk_baresequent H (mk_conclax (mk_halts a)),
      mk_baresequent H (mk_conclax (mk_equality a b (mk_partial A)))
    ]
    [].

Lemma rule_termination_true :
  ∀ a b A : NTerm,
  ∀ H : barehypotheses,
    rule_true (rule_termination a b A H).
```

To prove that two "halts" terms mk_halts(a) and mk_halts(b) are equal types, it is enough to prove that a and b are equal in some partial type mk_partial(A):

```
H |- halts(a) = halts(b) in Type(i)

  By HasValueEquality A

  H |- a = b in partial(A)
```

```
Definition rule_hasvalue_equality
           (a b A : NTerm)
           (i : nat)
           (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_equality (mk_halts a) (mk_halts b)
(mk_uni i))))
    [ mk_baresequent H (mk_conclax (mk_equality a b (mk_partial A)))
    ]
    [].
```

```
Lemma rule_hasvalue_equality_true :
  ∀ a b A : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_hasvalue_equality a b A i H).
```

5.2.9.1 Typing the Fixpoint Combinator

The fix operator of the Nuprl language is a key way to build interesting members of partial types. Crary proved several rules (called fixpoint induction rules) that allow one to prove equality of terms containing the fix operator in a partial type. Here is a simple one. Note that the second subgoal in the rule cannot be removed. In other words, in a powerful dependent type theory like Nuprl, there are fairly intuitive dependent types T for which we have some f in mk_partial $T \rightarrow$ mk_partial T, but mk_fix f is not a member of mk_partial T. See [21, Theorem 5.1]. Monohood and Admissibility are two predicates on types which ensure that this principle holds. Monohood implies Admissibility. However, monohood is easier to understand.

The following lemma expresses the key property of Mono types. Intuitively a type T is mono (i.e. mkc_mono T is inhabited) if, whenever we have some a in that type, anything above it in the approx ordering (say b) is equal to it.

```
Lemma mono_inhabited_implies: ∀ T,
member Ax (mkc_mono T)
→ (∀ (a b : CTerm),
        member a T
        → approxc a b
        → equality a b T).
```

Fixpoint principle for Mono types:

```
H |- fix(f1) = fix(f2) in partial(T)

  By fixpointPrinciple

  H |- f1 = f2 in partial(T)->partial(T)
  H |- Mono(T)
```

```
Definition rule_fixpoint_principle_mono
              (f1 f2 T : NTerm)
              (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax
        (mk_equality (mk_fix f1) (mk_fix f2) (mk_partial T))))
    [ mk_baresequent H
        (mk_conclax (mk_equality f1 f2
            (mk_fun (mk_partial T) (mk_partial T)))),
      mk_baresequent H (mk_conclax (mk_mono T))
    ]
    [].
```

In order to prove the above rule, we first prove some lemmas.

```
Lemma fix_approx_hasvalue: ∀ f n,
  isprogram f
  → hasvalue (fix_approx n f)
  → hasvalue (mk_fix f).

Lemma fix_approx_hasvalue_context : ∀ f G n,
  isprogram f
  → isprogram G
  → hasvalue (mk_apply G (fix_approx n f))
  → hasvalue (mk_apply G (mk_fix f)).

Lemma fapprox_hasvalue_higher: ∀ f n,
  isprogram f
  → hasvalue (fix_approx n f)
  → ∀ m, hasvalue (fix_approx (m+n) f).
Lemma equal_chains_equal_fun_gen : ∀ fa fb T,
  equality fa fb (mkc_fun T T)
  → member mkc_bot T
  → ∀ n, equality (fix_approxc n fa) (fix_approxc n fb) T.
Corollary equal_chains_equal_fun : ∀ fa fb T,
  equality fa fb (mkc_fun (mkc_partial T) (mkc_partial T))
  → ∀ n, equality (fix_approxc n fa) (fix_approxc n fb) (mkc_partial T).
Lemma fix_approxc_removec : ∀ f (fp: isprog f) n,
 (get_cterm (fix_approxc n (exist _ f fp))) = (fix_approx n f).
```

To recap the compactness theorem *fix_compactness* from 3.3, here is a trivial restatement of it which that lifts it to the CTerm type.

```
Lemma fixc_compactness: ∀ (f G : CTerm),
  hasvaluec (mkc_apply G (mkc_fix f))
  → {n : nat × hasvaluec (mkc_apply G (fix_approxc n f))}.
Lemma fixc_compactness_no_context: ∀ (f : CTerm),
  hasvaluec (mkc_fix f)
  → {n : nat × hasvaluec (fix_approxc n f)}.
Lemma fix_converges_equal_fun_iff_context : ∀ fa fb Ua Ub T S,
  member mkc_bot T
  → equality fa fb (mkc_fun T T)
  → equality Ua Ub (mkc_fun T (mkc_partial S))
  → (chaltsc (mkc_apply Ua (mkc_fix fa))
     ⇔ chaltsc (mkc_apply Ub (mkc_fix fb))).
Lemma fix_converges_equal_fun_iff : ∀ fa fb T,
  equality fa fb (mkc_fun (mkc_partial T) (mkc_partial T))
  → (chaltsc (mkc_fix fa) ⇔ chaltsc (mkc_fix fb)).
Lemma fapproxc_hasvalue_higher: ∀ f n,
  hasvaluec (fix_approxc n f)
  → ∀ m, hasvaluec (fix_approxc (m+n) f).
Definition is_nice_per (eq : term-equality) :=
  term_equality_symmetric eq
  × term_equality_transitive eq
  × term_equality_respecting eq.
Lemma nuprl_pers_are_nice:
  ∀ (T T': CTerm) (eq: term-equality),
  nuprl T T' eq
  → is_nice_per eq.
```

We will show that if the PER of a type satisfies the following fixpoint_induction_suff condition, then the fixpoint induction principle holds for it. The next lemma shows that PERs of Mono types satisfy this condition.

```
Definition fixpoint_induction_suff (eq : term-equality) :=
  ∀ (f f' : CTerm),
    {j: nat, ∀ k : nat, k > j → eq (fix_approxc k f) (fix_approxc k f') }
    → eq (mkc_fix f) (mkc_fix f').

Lemma fixpoint_induction_suff_if_mono : ∀ (eq : term-equality),
  is_nice_per eq
  → mono_equality eq
  → fixpoint_induction_suff eq.
```

The following lemma distills out the key idea in the proofs of the rules rule_fixpoint_principle_mono (above) and rule_fixpoint_principle_admiss (below). Due to functionality requirements of sequents, it is actually used three times in the proofs of these rules. The proof uses the compactness property fixc_compactness_no_context.

We apply the lemma equality_in_mkc_partial. So we have to prove 3 things. Firstly, we have to prove that mkc_partial T is a type. This is true because our second hypothesis implies that (mk_fun (mk_partial T) (mk_partial T)) is a type. Secondly, we have to prove (chaltsc (mkc_fix fa) ⇔ chaltsc (mkc_fix fb)). This is a trivial consequence of the lemma fix_converges_equal_fun_iff above. Finally, we have to assume that (mkc_fix fa) converges and prove that (mkc_fix fa) and (mkc_fix fb) are equal in the type T.

By compactness and because (mkc_fix fa) converges, one of its partial approximations(say (fix_approxc n fa)) converges, and is hence a member of the type T. By the lemma fapprox_hasvalue_higher, we have that (fix_approxc m fa) converges for all $m \geq n$.

By lemma equal_chains_equal_fun, we have that for all $m \geq n$, (fix_approxc m fb) is equal to (fix_approxc m fa) in mkc_partial T. So, by the lemma *equality_mkc_partial*, fix_approxc m fb converges too and is equal to (fix_approxc m fa) in the type T.

The rest of the proof is a straightforward consequence of thet hypothesis that the PER of the type T satisfies the fixpoint_induction_suff condition,

```
Lemma fixpoint_induction_general : ∀ fa fb T eqT,
  nuprl T T eqT
  → fixpoint_induction_suff eqT
  → equality fa fb (mkc_fun (mkc_partial T) (mkc_partial T))
  → equality (mkc_fix fa) (mkc_fix fb) (mkc_partial T).

Corollary fixpoint_induction_helper_mono : ∀ fa fb T,
    member Ax (mkc_mono T)
  → equality fa fb (mkc_fun (mkc_partial T) (mkc_partial T))
  → equality (mkc_fix fa) (mkc_fix fb) (mkc_partial T).

Corollary rule_fixpoint_principle_mono_true :
  ∀ f1 f2 T : NTerm,
  ∀ H : barehypotheses,
     rule_true (rule_fixpoint_principle_mono
                    f1 f2 T
                    H).
```

The above rule is not very useful. For notational brevity, we will often denote mk_partial T by $\overline{T}$. Intuitively, to prove typehood of partial functions from numbers to numbers using the above rule, we would have to show that the type $\mathbb{N} \to \overline{\mathbb{N}}$ is as mono type. However, this is not true as we will see below.

As a concrete example, suppose we wish to prove that the 3x+1 function is in the type $\mathbb{N} \to \overline{\mathbb{N}}$. The function can be represented in Nuprl as $fix(f3x)$, where $f3x$ is $\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ if\ n\ mod\ 2 = 0\ then\ f\ \frac{n}{2}\ else\ f\ (3 \times n + 1)$ Since $fix(f3x)$ reduces to a lambda expression (value) after one step of computation, by the termination rule it suffices to show that it is in the type $\overline{\mathbb{N} \to \overline{\mathbb{N}}}$. However, the type $\mathbb{N} \to \overline{\mathbb{N}}$ is not Mono. Monohood (see mono_inhabited_implies above) is not closed under the partial type constructor. $\lambda n.\bot$ is clearly a member of the type $\mathbb{N} \to \overline{\mathbb{N}}$, and we have approx $(\lambda n.\bot)$ $(\lambda n.Ax)$, but $\lambda n.Ax$ is not even a member of the type $\mathbb{N} \to \overline{\mathbb{N}}$ because Ax is a value that is not a number. Next, we will discuss two approaches to get around this issue.

We were able to prove a stronger version of the fixpoint induction principle (rule_fixpoint_principle_context below) which can be stated just in terms of the Mono property. This rule is enough to prove the typehood of many partial functions like the one above. So far, this stronger version sufficed for all our uses. Monohood is much easier to understand than admissibility.

Crary introduced a weaker predicate on types, called admissibility to deal with these cases. Unlike Monohood, for any type T, mkc_admiss T implies mkc_admiss (mk_partial T). Below, We will prove another flavour of the fixpoint induction rule where mkc_mono T is replaced by mkc_admiss T.

Here is a version of fixpoint induction rule that makes it easy to prove typehood of partial functions. Unlike the previous rule, mk_fix *f* now occurs in a context. *U* denotes this context. This rule is stated in terms of the more intuitive monohood property.

Fixpoint principle w/ context:

```
H |- U fix(f) = U fix(g) in partial S

  By fixpointPrinciple_context

  H |- f = g in T -> T
  H |- Mono S
  H |- mk_bot = mk_bot in T
  H |- U = U in T -> partial S
```

```
Definition rule_fixpoint_principle_context
           (f g U T S: NTerm)
           (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax
       (mk_equality
          (mk_apply U (mk_fix f))
          (mk_apply U (mk_fix g))
          (mk_partial S))))
    [
      mk_baresequent H (mk_conclax (mk_equality f g (mk_fun T T))),
      mk_baresequent H (mk_conclax (mk_mono S)),
      mk_baresequent H (mk_conclax (mk_equality mk_bot mk_bot T)),
      mk_baresequent H (mk_conclax
                         (mk_equality U U (mk_fun T (mk_partial S))))
    ]
    [].

Lemma fixpoint_induction_context_helper : ∀ Ua Ub fa fb T S,
  member Ax (mkc_mono S)
  → member mkc_bot T
  → equality fa fb (mkc_fun T T)
  → equality Ua Ub (mkc_fun T (mkc_partial S))
  → equality (mkc_apply Ua (mkc_fix fa))
             (mkc_apply Ub (mkc_fix fb))
             (mkc_partial S).
```

```
Theorem rule_fixpoint_principle_context_true :
  ∀ f g U T S: NTerm,
  ∀ H : barehypotheses,
    rule_true ( rule_fixpoint_principle_context
                    f g U T S
                    H).
```

Now, we discuss the second solution which was given by Crary. The type (mkc_admiss A) is inhabited (by `Ax`) if the PER of the type A satisfies the admissible_equality condition. The definition of admissible_equality can be found in Sec. 4.2.1 if you are reading the technical report.

```
Lemma equality_in_mkc_admiss :
  ∀ A t1 t2,
    equality t1 t2 (mkc_admiss A)
    ⇔ {eqa : term_equality , t1 ⇓ Ax × t2 ⇓ Ax
      × close univ A A eqa × admissible_equality eqa}.
```

Fixpoint principle for Admissible types:

```
H |- fix(f1) = fix(f2) in partial(T)

  By fixpointPrinciple

  H |- f1 = f2 in partial(T)->partial(T)
  H |- Admiss(T)
```

```
Definition rule_fixpoint_principle_admiss
              (f1 f2 T : NTerm)
              (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax
        (mk_equality (mk_fix f1) (mk_fix f2) (mk_partial T))))
    [ mk_baresequent H
        (mk_conclax (mk_equality f1 f2
            (mk_fun (mk_partial T) (mk_partial T)))),
      mk_baresequent H (mk_conclax (mk_admiss T))
    ]
    [].
```

In order to use the lemma fixpoint_induction_general to prove the above rule, we need to prove the following lemma. The proof is tricky and uses the least upper bound principle (crary5_9_no_context). Intuitively, the difficulty is that fixpoint_induction_suff is about partial approximations of two functions, while admissible_equality is about partial approximations of a single function in two contexts. The idea behind this proof is due to Crary.

```
Theorem fixpoint_induction_suff_if2 :
  ∀ (eq : term-equality),
  respects2 cequivc eq
  → admissible_equality eq
  → fixpoint_induction_suff eq.
```

The following Corrollary corresponds to Crary's Theorem 5.14 [21]

```
Corollary fixpoint_induction_helper_admiss : ∀ fa fb T,
    member Ax (mkc_admiss T)
  → equality fa fb (mkc_fun (mkc_partial T) (mkc_partial T))
  → equality (mkc_fix fa) (mkc_fix fb) (mkc_partial T).
Proof.
  introv Hm Hfeq.
  apply equality_in_mkc_admiss in Hm.
  exrepnd.
  apply fixpoint_induction_general with (eqT:=eqa); auto.
  apply fixpoint_induction_suff_if2; auto.
  apply nice_per_respects_cequivc.
  eapply nuprl_pers_are_nice;eauto.
Qed.
```

```
Corollary rule_fixpoint_principle_admiss_true :
  ∀ f1 f2 T : NTerm,
  ∀ H : barehypotheses,
    rule_true (rule_fixpoint_principle_admiss
                     f1 f2 T
                     H).
```

Crary proved the admissibility and monohood of many types (see [21, Sec. 5.3]). The collection of admissible types includes Base, $\mathbb{Z}$. Admissibility is also closed under many type constructors: e.g. dependent functions, independent pairs, partial types. Unlike above, the type Base is not a Mono type. Also, Monohood is not closed under the partial type constructor, but is closed under dependent pairs. The next two subsubsections deal with the characterization of Mono and Admissible types, respectively.

5.2.9.2 Characterizing Mono Types

———-begin file rules_mono.v ————

This section is a work in progress. We plan to prove the rules that characterize the class of Mono types, i.e. types T such that the type mkc_mono *T* is inhabited. Crary proved monohood of many types (see [21, Sec. 5.3.2]). Here is what we have proved so far:

The class of Mono types include :

- mkc_int (see rule_mono_int below)

Monohood is closed under the following type constructors.

- mkc_function (see rule_mono_function below)
- mkc_product (see rule_mono_product below)

We will keep updating this list as we prove more rules.

```
H |- Mono Z

  By int_mono
  no subgoals
```

```
Definition rule_mono_int
            (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_mono mk_int) ))
    []
    [].
```

```
Lemma rule_mono_int_true : ∀ (H : barehypotheses),
  rule_true (rule_mono_int H).
```

```
H |- Mono ((x:A) -> B)

  By function_mono
  H |- Mono A
  H, x:A |- Mono B
```

```
Definition rule_mono_function
    (A B : NTerm) (x : NVar)
            (H : barehypotheses) :=
  mk_rule
```

```
    (mk_baresequent H (mk_conclax
        (mk_mono (mk_function A x B)) ))
    [ (mk_baresequent H (mk_conclax (mk_mono A) )),
        (mk_baresequent (snoc H (mk_hyp x A)) (mk_conclax (mk_mono B)))
    ]
    [].

Lemma approxc_mkc_apply_congr:
  ∀ f g a b : CTerm,
    approxc f g
    → approxc a b
    → approxc (mkc_apply f a) (mkc_apply g b).

Lemma rule_mono_function_true : ∀ (H : barehypotheses)
  (A B : NTerm) (x: NVar),
    rule_true (rule_mono_function A B x H).
```

```
H |- Mono ((x:A) × B)

  By product_mono
  H |- Mono A
  H, x:A |- Mono B
```

```
Definition rule_mono_product
    (A B : NTerm) (x : NVar)
           (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax
        (mk_mono (mk_product A x B)) ))
    [ (mk_baresequent H (mk_conclax (mk_mono A) )),
        (mk_baresequent (snoc H (mk_hyp x A)) (mk_conclax (mk_mono B)))
    ]
    [].
```

The following lemma lifts approx_mk_pair of Sec. 3.2 to the CTerm type

```
Lemma approxc_mkc_pair :
  ∀ (t t' a b : CTerm),
    computes_to_valc t (mkc_pair a b)
    → approxc t t'
    → {a', b' : CTerm $
          computes_to_valc t' (mkc_pair a' b')
```

```
            # approxc a a'
            # approxc b b'}.
Lemma rule_mono_product_true : ∀ (H : barehypotheses)
  (A B : NTerm) (x: NVar),
    rule_true (rule_mono_product A B x H).
```

5.2.9.3 Characterizing Admissible Types

———-begin file rules_admiss.v ————

This section is a work in progress. We plan to prove the rules that characterize the class of Admissible types, i.e. types T such that the type *mkc_admiss T* is inhabited.

5.2.10 Structural rules

———-begin file rules_struct.v ————

We now prove the truth of several structural rules.

The following rule says that we can always thin any tail of a list of hypotheses:

```
H, J |- C ext t

  By thinHyps ()

  H |- C ext t
```

```
Definition rule_thin_hyps
             (H J : barehypotheses)
             (C t : NTerm) :=
  mk_rule
    (mk_baresequent (H ++ J) (mk_concl C t))
    [ mk_baresequent H (mk_concl C t) ]
    [].
Lemma rule_thin_hyps_true :
  ∀ (H J : barehypotheses)
          (C t : NTerm),
    rule_true (rule_thin_hyps H J C t).
```

The following rule says that we can always unhide an hypothesis if the conclusion is an equality (in general this is true if the conclusion has a trivial extract):

```
H [x : A] J |- t1 = t2 in C

  By equalityUnhide hyp(i) ()

  H x : A J |- t1 = t2 in C
```

```
Definition rule_unhide_equality
             (H J : barehypotheses)
             (A C t1 t2 : NTerm)
             (x : NVar) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hhyp x A) ++ J)
       (mk_conclax (mk_equality t1 t2 C)))
    [ mk_baresequent
        (snoc H (mk_hyp x A) ++ J)
        (mk_conclax (mk_equality t1 t2 C)) ]
    [].
```

```
Lemma rule_unhide_equality_true :
  ∀ (H J : barehypotheses)
       (A C t1 t2 : NTerm)
       (x : NVar),
    rule_true (rule_unhide_equality H J A C t1 t2 x).
```

The following rule is the standard “hypothesis” rule:

```
G, x : A, J |- x = x in A

  By hypothesisEquality hyp(i) ()

  no subgoals
```

```
Definition rule_hypothesis_equality
               (G J : barehypotheses)
               (A : NTerm)
               (x : NVar) :=
  mk_rule
    (mk_baresequent
       (snoc G (mk_hyp x A) ++ J)
       (mk_conclax (mk_equality (mk_var x) (mk_var x) A)))
    []
    [].
Lemma rule_hypothesis_equality_true :
  ∀ G J : barehypotheses,
  ∀ A : NTerm,
  ∀ x : NVar,
    rule_true (rule_hypothesis_equality G J A x).
```

The following rule says that to prove a conclusion C one can always provide an evidence t for that type and prove instead that t is a member of C:

```
H |- C ext t

  By introduction t

  H |- t = t in C
```

```
Definition rule_introduction
             (H : barehypotheses)
             (C t : NTerm) :=
  mk_rule
    (mk_baresequent H (mk_concl C t))
    [ mk_baresequent H (mk_conclax (mk_equality t t C)) ]
    [ sarg_term t ].
Lemma rule_introduction_true :
  ∀
    (H : barehypotheses)
    (C t : NTerm),
    rule_true (rule_introduction H C t).
```

The following rule is another form of the standard “hypothesis” rule:

```
G, x : A, J |- A ext x

  By hypothsis hyp(i) ()
```

```
     no subgoals
```

```
Definition rule_hypothesis
             (G J : barehypotheses)
             (A : NTerm)
             (x : NVar) :=
  mk_rule
    (mk_baresequent (snoc G (mk_hyp x A) ++ J) (mk_concl A (mk_var x)))
    []
    [].

Lemma rule_hypothesis_true :
  ∀ G J : barehypotheses,
  ∀ A : NTerm,
  ∀ x : NVar,
    rule_true (rule_hypothesis G J A x).
```

The following rule says that one can always delete (or thin) an hypothesis (as long as J does not depend on x, because H, J has to be well-formed):

```
   H, x : A, J |- C ext t

     By thin hyp(i) ()

     H, J |- C ext t
```

```
Definition rule_thin
           (G J : barehypotheses)
           (A C t : NTerm)
           (x : NVar) :=
  mk_rule
    (mk_baresequent (snoc G (mk_hyp x A) ++ J) (mk_concl C t))
    [ mk_baresequent (G ++ J) (mk_concl C t) ]
    [].

Lemma rule_thin_true :
  ∀ (G J : barehypotheses)
        (A C t : NTerm)
        (x : NVar),
    rule_true (rule_thin G J A C t x).
```

The following rule state that if we are trying to prove a goal under the assumption that x has type T, then it suffices to prove the goal under the hypothesis that x has type U, as long as we can prove that T is a subtype of U, and T respects the equality of U on the elements of T:

```
H, x : T, J |- C ext t

    By widening y z i

    H, x : U, J |- C ext t
    H, x : T, y : U, z : x = y in U |- x = y in T
    H, x : T |- x in U
```

```
Definition rule_widening
             (T U C t : NTerm)
             (x y z : NVar)
             (i : nat)
             (H J : barehypotheses) :=
  mk_rule
    (mk_baresequent
       (snoc H (mk_hyp x T) ++ J)
       (mk_concl C t))
    [ mk_baresequent (snoc H (mk_hyp x U) ++ J)
                     (mk_concl C t),
      mk_baresequent (snoc (snoc (snoc H (mk_hyp x T))
                                 (mk_hyp y U))
                           (mk_hyp z (mk_equality (mk_var x) (mk_var y) U)))
                     (mk_conclax (mk_equality (mk_var x) (mk_var y) T)),
      mk_baresequent (snoc H (mk_hyp x T))
                     (mk_conclax (mk_member (mk_var x) U))
    ]
    [sarg_var y, sarg_var z].

Lemma rule_widening_true :
  ∀ T U C t : NTerm,
  ∀ x y z : NVar,
  ∀ i : nat,
  ∀ H J : barehypotheses,
    rule_true (rule_widening
                 T U C t
```

```
                      x y z
                      i
                      H J).
```

The following rule is the standard cut rule:

```
H |- C ext t[x\u]

  By cut x B

  H |- B ext u
  H, x : B |- C ext t
```

```
Definition rule_cut
             (H : barehypotheses)
             (B C t u : NTerm)
             (x : NVar) :=
  mk_rule
    (mk_baresequent H (mk_concl C (subst t x u)))
    [ mk_baresequent H (mk_concl B u),
      mk_baresequent (snoc H (mk_hyp x B)) (mk_concl C t)
    ]
    [sarg_var x].
```

```
Lemma rule_cut_true :
  ∀
    (H : barehypotheses)
    (B C t u : NTerm)
    (x : NVar)
    (bc : disjoint (free_vars u) (bound_vars t)),
    rule_true (rule_cut H B C t u x).
```

This rule is similar to the cut rule, but is valid only if x is not free in the extract:

```
H |- C ext t

  By cutH x B

  H |- B ext u
  H, [x : B] |- C ext t
```

```
Definition rule_cutH
             (H : barehypotheses)
             (B C t u : NTerm)
             (x : NVar) :=
  mk_rule
    (mk_baresequent H (mk_concl C t))
    [ mk_baresequent H (mk_concl B u),
      mk_baresequent (snoc H (mk_hhyp x B)) (mk_concl C t)
    ]
    [sarg_var x].

Lemma rule_cutH_true :
  ∀
    (H : barehypotheses)
    (B C t u : NTerm)
    (x : NVar),
    rule_true (rule_cutH H B C t u x).
```

This is the functionality rule that exposes the functionality part of the meaning of sequents:

```
   G, x : A, J |- C ext t

     By functionality y z

     G, [x : Base], [z : x in A], J |- C ext t
     G, x : Base, y : Base, z : x = y in A, J |- C = C[x\y]
```

```
Definition rule_functionality
           (G J : barehypotheses)
           (A C t : NTerm)
           (x y z : NVar)
           (i : nat) :=
  mk_rule
    (mk_baresequent (snoc G (mk_hyp x A) ++ J) (mk_concl C t))
    [ mk_baresequent
        (snoc (snoc G (mk_hhyp x mk_base))
              (mk_hhyp z (mk_member (mk_var x) A))
              ++ J)
        (mk_concl C t),
      mk_baresequent
      (snoc (snoc (snoc G (mk_hyp x mk_base))
                  (mk_hyp y mk_base))
            (mk_hyp z (mk_equality (mk_var x) (mk_var y) A))
            ++ J)
      (mk_conclax (mk_equality C (subst C x (mk_var y)) (mk_uni i)))
  ]
  [].

Lemma rule_functionaliy_true :
  ∀ G J : barehypotheses,
  ∀ A C t : NTerm,
  ∀ x y z : NVar,
  ∀ i : nat,
  ∀ bc : !LIn y (bound_vars C),
    rule_true (rule_functionality G J A C t x y z i).
```

5.2.11 Classical logic

———-begin file rules_classical.v ———–

Using the axiom of excluded middle from Classical_Prop, we can easily prove the following squashed excluded middle rule:

```
H |- squash(P \/ ~P) ext Ax

  By SquashedEM

  H |- P in Type(i) ext Ax
```

where mk_squash is defined as `Definition` mk_squash T := *mk_image* T (*mk_lam nvarx mk_axiom*), i.e., we map all the elements of T to *mk_axiom*. The only inhabitant of mk_squash T is *mk_axiom*, and we can prove that `Ax` is a member of *mkc_squash* T iff T is inhabited. However, in the theory, there is in general no way to know which terms inhabit T.

```
Definition rule_squashed_excluded_middle
             (P : NTerm)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_squash (mk_or P (mk_not P)))))
    [ mk_baresequent H (mk_conclax (mk_member P (mk_uni i)))
    ]
    [].
```

```
Lemma rule_squashed_excluded_middle_true :
  ∀ P : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_squashed_excluded_middle P i H).
```

5.2.12 Canonical form tests

————-begin file rules_cft.v ————

We now prove some rules involving our canonical form tests.

We state the ispairCases rule as follows:

```
H |- C ext ispair(t,ea,eb)[x\Ax]

  By ispairCases x z t u v

  H |- halts(t)
  H |- t in Base
  H x : t ~ <fst(t),snd(t)> |- C ext ea
  H x : (uall u v : Base, ispair(z,u,v) ~ v)[z\t] |- C ext eb
```

This rule allows one to do a proof by case on whether or not a term is a pair. Therefore, we need to know that the term computes to a value. Also, because on this rule, the t is provided by the user and is used in the final extract, we have to ensure that t only depends on the non-hidden variables of H. This is done by the side-condition sarg_term t

```
Definition rule_ispair_cases
             (C t ea eb : NTerm)
             (x z u v : NVar)
             (H : barehypotheses) :=
  mk_rule
    (mk_bsequent H (mk_concl C (subst (mk_ispair t ea eb) x mk_axiom)))
    [ mk_bsequent H (mk_conclax (mk_halts t)),
      mk_bsequent H (mk_conclax (mk_member t mk_base)),
      mk_bsequent (snoc H (mk_hyp x (mk_cequiv t (mk_eta_pair t))))
                  (mk_concl C ea),
      mk_bsequent (snoc H (mk_hyp x (mk_not_pair u v z t)))
                  (mk_concl C eb)
    ]
    [sarg_term t].

Lemma rule_ispair_cases_true :
  ∀ C t ea eb : NTerm,
  ∀ x z u v : NVar,
  ∀ H : barehypotheses,
  ∀ sc1 : !LIn v (vars_hyps H),
  ∀ sc2 : !LIn u (vars_hyps H),
  ∀ sc3 : z ≠ u,
  ∀ sc4 : z ≠ v,
  ∀ sc5 : v ≠ u,
    rule_true (rule_ispair_cases
                 C t ea eb
                 x z u v
                 H).
```

5.2.13 Convergence

———-begin file rules_hasvalue.v ———–

The following rule states that if mk_spread p x y b computes to a value then p must compute to a pair:

```
  H |- p in Top x Top

    By haltSpread

    H |- halts(mk_spread p x y b)
```

```
Definition rule_halt_spread
             (H : barehypotheses)
             (x y : NVar)
             (p b : NTerm) :=
  mk_rule
    (mk_baresequent H (mk_conclax (mk_member p (mk_prod mk_top mk_top))))
    [ mk_baresequent H (mk_conclax (mk_halts (mk_spread p x y b)))
    ]
    [].
```

```
Lemma rule_halt_spread_true :
  ∀ (H : barehypotheses)
           (x y : NVar)
           (p b : NTerm),
    rule_true (rule_halt_spread H x y p b).
```

5.2.14 Type equality

———-begin file rules_tequality.v ————

The following rule is used to prove that *mkc_tequality* terms are types (equal types).

```
  H |- (mkc_tequality T1 T2) = (mkc_tequality U1 U2) in Type(i)

    By tequalityEquality ()

    H |- T1 = U1 in Type(i)
    H |- T2 = U2 in Type(i)
    H |- T1 = T2 in Type(i)
```

```
Definition rule_tequality_equality
             (T1 T2 U1 U2 : NTerm)
             (i : nat)
             (H : barehypotheses) :=
  mk_rule
    (mk_bseq
       H
       (mk_conclax (mk_equality
                      (mk_tequality T1 T2)
                      (mk_tequality U1 U2)
                      (mk_uni i))))
    [ mk_bseq H (mk_conclax (mk_equality T1 U1 (mk_uni i))),
      mk_bseq H (mk_conclax (mk_equality T2 U2 (mk_uni i))),
      mk_bseq H (mk_conclax (mk_equality T1 T2 (mk_uni i)))
    ]
    [ ].

Lemma rule_tequality_equality_true :
  ∀ T1 T2 U1 U2 : NTerm,
  ∀ i : nat,
  ∀ H : barehypotheses,
    rule_true (rule_tequality_equality
                 T1 T2 U1 U2
                 i
                 H).
```

This rule says that in an equality, a type can be replaced by an equal types (this rule could be generalized):

```
H |- a = b in T

  By tequalitySubstEqual ()

  H |- (T = U)
  H |- a = b in U
```

```
Definition rule_tequality_subst_equal
             (a b T U : NTerm)
             (H : barehypotheses) :=
  mk_rule
    (mk_bseq H (mk_conclax (mk_equality a b T)))
    [ mk_bseq H (mk_conclax (mk_tequality T U)),
      mk_bseq H (mk_conclax (mk_equality a b U))
    ]
    [ ].

Lemma rule_tequality_subst_equal_true :
  ∀ a b T U : NTerm,
  ∀ H : barehypotheses,
    rule_true (rule_tequality_subst_equal
                 a b T U
                 H).
```

The following rule says that tequality types do not have any computational content:

```
H |- a = b in (T = U)

  By tequalityMember ()

  H |- (T = U)
```

```
Definition rule_tequality_member
             (a b T U : NTerm)
             (H : barehypotheses) :=
  mk_rule
    (mk_bseq H (mk_conclax (mk_equality a b (mk_tequality T U))))
    [ mk_bseq H (mk_conclax (mk_tequality T U))
    ]
    [ ].

Lemma rule_tequality_member_true :
  ∀ a b T U : NTerm,
  ∀ H : barehypotheses,
    rule_true (rule_tequality_member
                 a b T U
                 H).
```

The following rule says that tequality types are well-formed only when they're true:

```
  H |- (T = U)

    By tequalityWf ()

    H |- (T = U) is a type
```

```
Definition rule_tequality_type
             (T U : NTerm)
             (H : barehypotheses) :=
  mk_rule
    (mk_bseq H (mk_conclax (mk_tequality T U)))
    [ mk_bseq H (mk_concl_t (mk_tequality T U))
    ]
    [ ].

Lemma rule_tequality_type_true :
  ∀ T U : NTerm,
  ∀ H : barehypotheses,
    rule_true (rule_tequality_type
                 T U
                 H).
```

The following rule says that tequality types are symmetric:

```
  H |- (T = U)

    By tequalitySym ()

    H |- (U = T)
```

```
Definition rule_tequality_sym
             (T U : NTerm)
             (H : barehypotheses) :=
  mk_rule
    (mk_bseq H (mk_conclax (mk_tequality T U)))
    [ mk_bseq H (mk_conclax (mk_tequality U T))
    ]
    [ ].
```

```
Lemma rule_tequality_sym_true :
  ∀ T U : NTerm,
  ∀ H : barehypotheses,
    rule_true (rule_tequality_sym
                 T U
                 H).
```

5.3 A trusted core

———-begin file nuprl_lemmas2.v ————

Let us now illustrate how we can use the proofs of the validity of Nuprl's rules and the meaning of rules to build a simple refiner of Nuprl proofs directly in Coq using Ltac.

We first define the rule inductive type that contains the list of rules that we allow ourselves to use. This list does not contain yet all the rules we have proved to be valid.

```
Notation lvl := nat.
```

```
Inductive rule :=
| isect_equality : NVar → lvl → rule
| isect_member_formation : lvl → NVar → rule
| approx_intensional_equality_simple : lvl → rule
| cequiv_refl : rule
| thin_hyps : rule.
```

A lemma will be stated as follows: nuprove *statement*.

We now define a few useful rules to simulate a proof environment. First, the *start_proof* tactic is used when starting a proof. It uses `eexists` in order to build extracts bottom-up while doing a proof top-down. Then we prove the well-formedness of the given goal and the remaining subgoal is a statement that says that the goal is true in an empty environment.

```
Ltac start_proof :=
  let pwf := fresh "pwf" in
    eexists;
  prove_and pwf;
  [ unfold pwf_sequent; wfseq
  | idtac
  ].
```

When using *start_proof* we are also left with a subgoal where we have to prove that the extract is well-formed. However, we cannot prove that just yet because we do not know yet what the extract is. Extracts are built bottom-up, therefore, the extract will be fully built only once the proof is finished. Therefore, we use *end_proof* at the end of a proof to prove the well-formedness of the extract.

```
Ltac end_proof := sp.
```

We use *use_lemma* to use a Nuprl lemma already proven and get its extract.

```
Ltac use_lemma lem :=
  let l := fresh "l" in
  let hl := fresh "hl" in
  let r := fresh "r" in
  let hr := fresh "hr" in
  let e := fresh "e" in
  let t := fresh "t" in
  let w := fresh "w" in
    remember lem as l eqn:hl;
  remember (projT1 l) as r eqn:hr;
  dup hr as e;
  rewrite hl in e;
  rewrite hr in e;
  simpl in e;
  clear hl hr;
  unfold nuprove in l;
  destruct l as [t l];
  destruct l as [w l];
  simpl in e;
  subst;
  try (apply l).
```

The following few tactics are used to prove simple side-conditions of rules.

```
Ltac inveq :=
  match goal with
    | [ H : _ = _ ⊢ _ ] ⇒ complete inversion H
  end.

Ltac prove_side_condition := simpl; sp; try inveq; try wfseq.
Ltac tc_prove_side_condition := try (complete prove_side_condition).
```

The next tactic is our most important tactic, which refines a goal using a rule. It applies the corresponding lemmas that states that the rule is valid. It also uses the facts that most of our rules satisfy the wf_rule predicate to prove the well-formedness of sequents.

```
Ltac nuprl_refine R :=
  let r := fresh "r" in
    match R with
      | isect_equality ?v ?n ⇒
          generalize (rule_isect_equality_true2 v n);
          intro r;
          apply r;
          clear r;
          [ tc_prove_side_condition
          | tc_prove_side_condition
          | tc_prove_side_condition
          | tc_prove_side_condition
          | let w := fresh "w" in
            let i := fresh "i" in
              generalize (rule_isect_equality_wf v n);
              intro w;
              unfold wf_rule in w;
              simpl in w;
              match goal with
```

```
                    [ H : pwf_sequent _ ⊢ _ ] ⇒
                      apply w in H;
                    clear w;
                    [ introv i; simpl in i; repdors; subst;
                      unfold wf_subgoals in H; simpl in H;
                      [ apply in_lst_1_out_of_2 in H
                      | apply in_lst_2_out_of_2 in H
                      | complete sp
                      ]
                    | tc_prove_side_condition
                    | tc_prove_side_condition
                    | tc_prove_side_condition
                    ]
               end
          ]

     | isect_member_formation ?l ?v ⇒
          generalize (rule_isect_member_formation_true2 l v);
          intro r;
          apply r;
          clear r;
          [ tc_prove_side_condition
          | tc_prove_side_condition
          | tc_prove_side_condition
          | let w := fresh "w" in
            let i := fresh "i" in
               generalize (rule_isect_member_formation_wf l v);
               intro w;
               unfold wf_rule in w;
               simpl in w;
               match goal with
                    [ H : pwf_sequent _ ⊢ _ ] ⇒
                      apply w in H;
                    clear w;
                    [ introv i; simpl in i; repdors; subst;
                      unfold wf_subgoals in H; simpl in H;
                      [ apply in_lst_1_out_of_2 in H
                      | apply in_lst_2_out_of_2 in H
                      | complete sp
                      ]
```

```
                | tc_prove_side_condition
                | tc_prove_side_condition
                ]
           end
      ]

| approx_intensional_equality_simple ?n ⇒
      generalize (rule_approx_intensional_equality_simple_true2 n);
      intro r;
      apply r;
      clear r;
      [ tc_prove_side_condition
      | tc_prove_side_condition
      | complete (simpl; sp)
      ]

| cequiv_refl ⇒
      generalize (rule_cequiv_refl_true2);
      intro r;
      apply r;
      clear r;
      [ tc_prove_side_condition
      | tc_prove_side_condition
      | complete (simpl; sp)
      ]

| thin_hyps ⇒
      generalize (rule_thin_hyps_true2);
      intro r;
      apply r;
      clear r;
      [ tc_prove_side_condition
      | tc_prove_side_condition
      | let w := fresh "w" in
        let i := fresh "i" in
          generalize rule_thin_hyps_wf;
          intro w;
          unfold wf_rule in w;
          simpl in w;
          match goal with
```

```
                      [ H : pwf_sequent _ ⊢ _ ] ⇒
                        apply w in H;
                      clear w;
                      [ introv i; simpl in i; repdors; subst;
                        unfold wf_subgoals in H; simpl in H;
                        [ apply in_lst_1_out_of_1 in H
                        | complete sp
                        ]
                      | tc_prove_side_condition
                      ]
                  end
              ]
      end.
```

Let us now prove a few simple lemmas. First, we prove that mk_false is a type, where mk_false is defined as *mk_approx mk_axiom mk_bot*, where *mk_bot* is defined as *mk_fix mk_id*.

```
Lemma false_in_type :
  nuprove (mk_member mk_false (mk_uni 0)).
Proof.
  start_proof.
  Focus 2.

  nuprl_tree;
    [ nuprl_refine (approx_intensional_equality_simple 0)
    ].

  end_proof.
Defined.
```

Next, we prove that mk_top is a type, where mk_top is defined as *mk_isect* mk_false nvarx mk_false. This proof contains a few extra steps, which eventually will disappear once we have built enough tactics.

```
Lemma top_in_type :
  nuprove (mk_member mk_top (mk_uni 0)).
Proof.
  start_proof.
  Focus 2.

  nuprl_tree;
    [ nuprl_refine (isect_equality nvary 0);
```

```
        [ use_lemma false_in_type

        | rw subst_mk_false; try prove_side_condition;
          rw subst_mk_false in pwf; try prove_side_condition;
          assert ([mk_hyp nvary mk_false] = [] ++ [mk_hyp nvary mk_false]) as
eq by (simpl; sp);
          rw eq; rw eq in pwf; clear eq;

          nuprl_refine thin_hyps;
          [ use_lemma false_in_type
          ]
        ]
      ].
  end_proof.
Defined.
```

The following lemma proves that for all x in mk_top, x is computationally equivalent to x. Once again this proof contains a few extra steps which eventually will disappear once we have built enough tactics.

```
Lemma cequiv_refl_top :
  nuprove (mk_uall mk_top nvarx (mk_cequiv (mk_var nvarx) (mk_var nvarx))).
Proof.
  start_proof.
  Focus 2.

  nuprl_tree;
    [ nuprl_refine (isect_member_formation 0 nvary);
        [ unfold subst, lsubst; simpl; rw fold_nobnd; rw fold_cequiv;
          unfold subst, lsubst in pwf; simpl in pwf; rw fold_nobnd in pwf; rw
fold_cequiv in pwf;
              nuprl_refine cequiv_refl
        | use_lemma top_in_type
        ]
    ].
  end_proof.
Defined.
```

Bibliography

[1] *Interactive Theorem Proving - 4th Int'l Conf.*, volume 7998 of *LNCS*. Springer, 2013.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using w-types. In *Automata, Languages and Programming: 31st Int'l Colloq., ICALP 2004*, volume 3142 of *LNCS*, pages 59–71. Springer, 2004.

[3] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005.

[4] The Agda wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[5] Stuart F. Allen. A non-type-theoretic definition of martin-löf's types. In *LICS*, pages 215–221. IEEE Computer Society, 1987.

[6] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[7] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. http://www.nuprl.org/.

[8] Bruno Barras. Sets in Coq, Coq in sets. *Journal of Fromalized Reasoning*, 3(1):29–48, 2010.

[9] Bruno Barras and Benjamin Werner. Coq in Coq. Technical report, INRIA Rocquencourt, 1997.

[10] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd Int'l Conf.*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[11] Edwin Brady. Idris —: systems programming meets full dependent types. In *5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011*, pages 43–54. ACM, 2011.

[12] Wilfried Buchholz, Solomon Feferman, Wolfram Pohlers, and Wilfried Sieg. *Iterated inductive definitions and subsystems of analysis: recent proof-theoretical studies*. Springer-Verlag Berlin, Heidelberg, New York, 1981.

[13] Venanzio Capretta. A polymorphic representation of induction-recursion. www.cs.ru.nl/~venanzio/publications/induction_recursion.ps, 2004.

[14] Venanzio Capretta and Amy P Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *Types for Proofs and Programs*, pages 63–77. Springer Berlin Heidelberg, 2007.

[15] Adam Chlipala. Certified programming with dependent types, 2011.

[16] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[17] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987.

[18] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121(1&2):89–112, 1993.

[19] Thierry Coquand. Completeness theorems and λ-calculus. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461

of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin Heidelberg, 2005.

[20] The Coq Proof Assistant. http://coq.inria.fr/.

[21] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

[22] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

[23] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Logic*, 124(1-3):1–47, 2003.

[24] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *ITP 2013* [1], pages 163–179.

[25] Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14(1):71–84, 1992.

[26] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, 1989.

[27] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *The 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 2013*, pages 193–206. ACM, 2013.

[28] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *LICS*, pages 86–95. IEEE Computer Society, 2003.

[29] Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, Ithaca, NY, 2004.

[30] Christoph Kreitz. *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 2002. www.nuprl.org/html/02cucs-NuprlManual.pdf.

[31] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 42–54. ACM, 2006.

[32] Martin-Löf. Constructive mathematics and computer programming. In *6th INternational Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, 1982.

[33] Per Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[34] Conor McBride. Hier soir, an OTT hierarchy, 2011. http://sneezy.cs.nott.ac.uk/epilogue/?p=1098.

[35] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[36] Aleksey Nogin. *Theory and Implementings of an Efficient Tactic-Based Logical Framework*. PhD thesis, Cornell University, 2002.

[37] Aleksey Nogin and Alexei Kopylov. Formalizing type operations using the "image" type constructor. *Electr. Notes Theor. Comput. Sci.*, 165:121–132, 2006.

[38] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208. ACM, 1988.

[39] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. In *ITP 2013* [1], pages 261–278.

[40] Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*. PhD thesis, Ludwig Maximilian University of Munich, 1993.

[41] Scott F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.

[42] I.A.S. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations, 2013.

[43] Benno van den Berg and Federico De Marchi. Non-well-founded trees in categories. *Ann. Pure Appl. Logic*, 146(1):40–59, 2007.

[44] Benjamin Werner. Sets in types, types in sets. In *TACS*, volume 1281 of *LNCS*, pages 530–546. Springer, 1997.