A proof development framework combining proofs with computation
690 Project Report

Antonis Stampoulis

Department of Computer Science
Yale University

May 12, 2008

Abstract
In this paper we present the beginnings of the design of a language for large-scale proof development. Our language is based on the CiC logical framework, as implemented in the Coq proof assistant, but departs from it in several important ways. Instead of having a general computational language where CiC is implemented in, and proof-manipulating functions are written in an untyped way, we propose a general computational language that has first-class support for the logical framework. This gives us the benefit of being able to write typeful proof-manipulating functions. Our language can manipulate efficient representations of the objects that the logical framework reasons about, in contrast with Coq which manipulates fairly inefficient representations. We prove several meta-theorems about our language, including type-safety and consistency of logic. We show how an important simplification that we have done in our framework, compared to CiC, doesn't impact the expressivity of our framework, by providing a translation for the missing functionality in our language. We show how this translation can be used to write type-safe tactics that produce proofs of certain propositions that can be derived computationally (e.g. a proof of the primality of a certain number). Last, we see how our language compares to other similar frameworks, and the features that need to be added in order to make it a general proof-development language.

1. Introduction
A serious bottleneck in the development of formally certifiable software systems is the process of proof development in existing proof assistants. As a number of papers demonstrate, the effort required to develop machine-verifiable proofs of correctness of even simple low-level software components is very significant.

Two of the most widely used proof assistants are Coq and Isabelle/HOL. The process of proof development in these is mostly based on creating a proof script for each theorem we need to prove; such proof scripts combine a number of different proof-manipulating functions called tactics in order to produce a valid proof of the current theorem. The tactics that users of such proof assistants employ in their proofs is more-or-less fixed. Advanced users might define their own tactics to simplify the development of a certain kind of proofs.

Whereas proof scripts are written in an interactive surface language, where the current state of the proof is always known, tactics are mostly written in the implementation language of the proof assistant. At that level, propositions and proofs are represented as a normal datatype of the implementation language. A user of the proof assistant who wants to extend the tactics they can use needs therefore to be knowledgeable in the implementation language of the tool and also on the internals of the tool itself. This inhibits the pervasive development of such tactics.

The ultimate goal of our project is to make the proof development process more scalable by permitting more pervasive development of domain-specific tactics and decision procedures. In order to achieve that, we believe that the overhead that is inherrent in having to write tactics in a different language than the one we write proofs in must be eliminated. Thus a unified language that combines general-purpose, side-effectful computation with first class support for a logical framework is needed; this language will be fit both for writing tactics and for combining these tactics into proof scripts in order to prove specific theorems. We believe that this unified language will enable the proof development process to become more modular: the cost of writing a small, reusable tactic to automate a small part of a proof script becomes reasonable, and combining these tactics together into a larger tactic to automate a larger part of a proof is easier than writing this tactic from scratch.

In this paper we will present the beginnings of a proof-development language design that is based on these ideas. Note that our focus is on proof-development for program verification purposes, and not for general mathematical reasoning. We aim at a clean design that allows for extensions, so that this language can eventually replace existing systems, even though at this initial design phase it is still far from this goal.

The rest of this paper is structured as follows. In chapter 2, we present our overall approach motivating our design decisions. In chapters 3 and 4, we present the static and dynamic semantics of our language, and prove a number of metatheory results for it. In chapter 5 we recover the expressivity of similar languages that our design has not explicitly included, while in chapter 6 we compare our language with other existing systems. We present some directions for future research in chapter 7 and conclude in chapter 8.

2. Approach
The main idea behind our language design is to have a general-purpose language like ML with integrated support for a particular logical framework. The reason why we want our language to be general-purpose is apparent when we consider the fact that we ultimately want to code powerful decision procedures that are able to assist users in proving theorems. These decision procedures can be arbitrarily complicated – consider for example the case of a highly optimized SMT solver, as the one described in [2]. Therefore their development should not be constrained by any limitations of the implementation language.

The most important choice that we are left to make is what the logical framework should look like. We have chosen to base our framework in the Calculus of Inductive Constructions (CiC) [17], as implemented in the Coq proof assistant [8]. The reason is that CiC is a well-understood logical framework with clear metatheory; its trusted kernel –the proof checker that mechanically checks the validity of proofs written in the framework– is fairly small and can therefore be made robust; it has been used in a wide spectrum of verification tasks from compiler verification [9] to mechanized

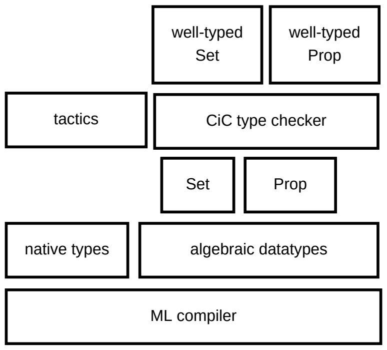

Figure 1. Overview of the structure of the Coq system

proofs of mathematical theorems [5]; and a large body of program verification work has been implemented in it, which we might want to be able to reuse.

We begin with a small simplified overview of the structure of the Coq framework as presented in figure 1, and then arrive at our language design by taking our motivations into account. The core logical language of Coq, the Calculus of Inductive Constructions, is a constructive type theory with inductive definitions. It is composed of two main universes, the Set and the Prop universe. The Set universe is composed of inductive data types such as natural numbers and lists, as well as total higher-order functions between them. The Prop universe is composed of propositions that reason about these objects. Terms of this universe, typed over these propositions, are called proof terms. Under the Curry-Howard isomorphism, constructing a proof term of a certain type that corresponds to a proposition is equivalent to proving the proposition. Essentially, type checking a proof term corresponding to a certain proposition is equivalent to checking the validity of the derivation of a proof of that proposition.

An important part of the Coq system is the implementation of CiC inside a general-purpose language (the O'Caml programming language). In order to achieve this, ML datatypes representing the types and terms of the Set and Prop universes are defined. A type-checking algorithm is employed to check whether the terms that we construct are well-typed under the typing rules of the CiC framework. The code that implements this is of course written in ML too, and is relatively small and straightforward. Taking into account the fact that type-checking proof terms is equivalent to checking the validity of proofs, this means that Coq satisfies the de Bruijn criterion [18]; that is, the validity of proofs in the system is guaranteed by a small checker. The rest of the Coq proof assistant is a set of tactics and an interactive proof development system with a higher-level syntax that is suitable for developing large proofs. We can view tactics as ML functions that take as input a goal that we are trying to prove, and simplify it into simpler goals that need to be proven, while constructing part of the proof term representing the overall proof of the goal.

Coq tactics are manipulating CiC terms as encoded through ML datatype definitions. It is easy to construct such terms that do not correspond to well-typed CiC terms; that's why employing a type checker is necessary. The type checker is only executed at run-time; so statically, we do not have any information about the well-typedness of such terms (under CiC). Equivalently, we do not have any static information or guarantee of the specific CiC type of the encoding of a CiC term. This means that when we are writing tactics that manipulate CiC terms, this manipulation actually takes place in a purely *untyped* manner. Still, the existence of the type-checker and the requirement that the proof term that

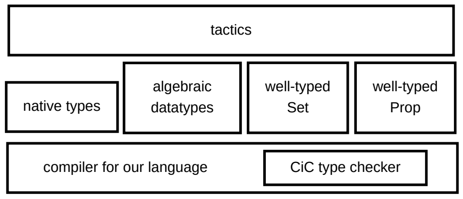

Figure 2. First approximation of our framework

tactics produced is checked before being considered a valid proof of a proposition guarantees the soundness of the approach.

We will now move on to see how our overall language design differs from this picture. In the next two sections, we will see our two key departures from the Coq system.

2.1 First-class support for the logical framework

One of our main motivations in designing a new language for proof development is to make the process of tactic writing more pervasive, by eliminating the overhead that it currently entails; we believe that this will result in better modularity and scalability for the proof development process. In the setting of CiC, we believe that a main hindrance towards that effect is the fact that tactics are manipulating CiC terms in an essentially untyped manner, as we mentioned above. Therefore, we want our general computation language to have explicit support for well-typed CiC terms. In that way, when we are writing a tactic we will have type information for the terms that we are manipulating. We are therefore transforming the process of tactic development from an untyped paradigm to a typed paradigm.

With these in mind, the new picture of our framework is similar to that of figure 2. Compared to figure 1, we see that we have moved the well-typed CiC terms into first-class citizens of our language, incorporating the CiC type checker as part of the type checker of the language itself. Therefore, when programming tactics, we can use static type information about these terms to aid us in their development.

Let us note here that our trusted base is not enlarged by this change. In the existing Coq framework, one needs to trust the ML compiler and the CiC type checker, in order to trust that a proof term that passes the type checker represents indeed a valid proof. In our framework, we have merely moved the CiC type checker inside the compiler; the amount of code that we need to trust is essentially the same.

2.2 Removing computation from the Set universe

The next departure from CiC that our framework takes is to strip the Set universe of any notion of computation. To see what this means, we have to revisit the Set universe of the CiC framework. As we have said above, the terms of the Set universe are terms of inductive types like natural numbers and lists, but also total, higher-order functions between these types. Totality is assurred by only providing primitive inductive elimination on the inductive types.

A term of a certain inductive type may therefore be an arbitrarily complex expression involving inductive elimination, lambda-abstractions, and application forms. Beta-reduction and iota-reduction (reduction of inductive elimination forms) are defined and compose the computational aspect of the Set universe. Since all functions are total, it is easy to see that this language is strongly normalizing, so each term can be evaluated to a normal form. The normal form will also be unique by the Church-Rosser theorem. Two terms can be said to be equivalent if their normal form is syntactically equal.

How is this notion of computation used in the actual framework? There are two main answers to this question:

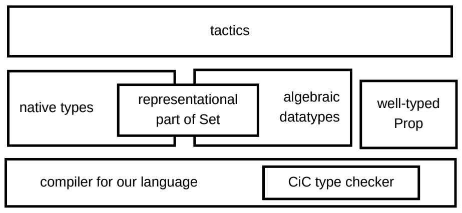

Figure 3. Final overview of our framework

- it is used to identify propositions up to Set-term equivalence. This means that if we need to prove the proposition $P(1+1)$, since the term $1+1$ is equivalent to its normal form 2, a proof of $P(2)$ is enough.
- it is used as actual computational code when CiC is used for program extraction out of proofs

In our setting, we are not concerned with program extraction out of proofs. This is because our ultimate focus is certifying low-level systems code, where program extraction in the style of Coq does not seem to have an immediate benefit. But the first use of this kind of computation is certainly essential. This point also shows why Set-terms should be strongly normalizing, as otherwise, the decidability of the CiC type checker would be compromised.

In our framework, we have limited the Set universe so that it only contains normal forms of primitive inductive datatypes like natural numbers and lists; no form of lambda-abstraction or inductive elimination exists, and there is no arrow type whatsoever in the Set universe. Therefore the Set universe is actually composed only of representation forms of primitive data types, and does not include any notion of computation between them. In order to emphasize this difference from the original CiC Set universe, we call this part of our framework the data-term universe. This new picture of our framework can be seen in figure 3.

To see how we were lead to this decision, consider the overview of our framework presented back in the figure 2. Since we have made the Set universe a first-class citizen of our type system, we have two ways to define an inductive datatype so that we can use it in our language: either as a type in the Set universe, or as a normal algebraic datatype of our computational language. The difference between these two definitions (let's call them types τ_1 and τ_2) is that functions yielding τ_1 must be in the Set universe and therefore can only be defined through a total, pure functional language, while functions yielding τ_2 can use the full power of the computational language. Also, τ_1 can be used both at the propositional and at the computational level; but τ_2 can only be used at the computational level.

This distinction between the two definitions is rather confusing, and we would prefer that a single way to define datatypes is available. We want such types to be used both at the propositional and the computational level, and a single way to define functions involving them that leverages the full power of the computational language. Also, we would prefer if the actual run-time representation of these types is efficient; for example, we would want to use bignums as the runtime representation of the inductive type of natural numbers.

The way to achieve all of these is to merge the inductive definitions at the Set universe with part of the datatype definitions of the computational language, having a single universe of terms that can be used both at the proof and the computational level. Essentially, we remove the computational aspect of the Set universe, and use those terms both for proofs and for programs. Thus, there is a single way to define functions between these terms, at the level of the general computational language. Also, since terms of this universe are always in normal form, we can use an efficient representation for them at run-time. This last point is essentially made possible by the fact that we don't need to use a representation that accounts for lambda-abstractions, elimination forms and beta-redexes.

The other benefit of this approach is that in the future we might want to extend these data-types to also handle higher-order abstract syntax, as used for example in LF [7, 13]. In that case we would need to add an arrow type to this level, which would only account for the parametric functional space (and not for the primitive recursive functional space too). Having this distinction between the two functional spaces is essential to properly support HOAS (see for example [3]).

Now let's consider what are the implications of removing the total computational subpart of the Set universe. First of all, we cannot directly define functions like plus between natural numbers and use them to write propositions like $x+y>z$. This is easily overcome by defining such functions as relations, and writing equivalent relations like $\forall r, \text{plus}\, x\, y\, r \Rightarrow r > z$. Second, the first point that we saw above regarding the use of computations in Coq is made invalid in our setting. Namely, when having to prove a proposition involving a term in non-normal form, we cannot directly say that this is equivalent to proving the same proposition involving the term in normal form. For example, I cannot directly prove the proposition $\forall r, \text{plus}\, 1\, 1\, r \Rightarrow r > 0$ from a proof of the fact that $2 > 0$. This is because plus is not a function anymore, so we do not have a notion of computing its result directly at the static typing level.

Of course, we can still create a proof of the fact that $\forall r, \text{plus}\, 1\, 1\, r \Rightarrow r = 2$, through which we can prove the above proposition given that $2 > 0$. But we are forced to write a proof of a trivial computational fact. This seems counterintuitive, and certainly goes against the Poincaré principle (see [1] and [18]), which states that proofs about computations are uninteresting and should not have to be made explicit. Though this is certainly true, we will see later in the development (specifically in chapter 5) that we can actually create such proofs about computations between dataterms in an automatic way, so that the user of our framework does not have to put any effort into such proofs. So the principle of not having to explicitly provide trivial proofs for computations is maintained.

As a last point, note that this choice of stripping the Set universe of its computational subpart actually makes our trusted base smaller. We are essentially limiting part of the CiC typing rules, so the CiC type-checker that our language needs to implement is somewhat smaller. The missing functionality is regained by expressing it in our computational language, and therefore does not have to be trusted.

3. The Core Language

We will now see our language design in more detail, giving its syntactic categories, its typing rules and its dynamic semantics. Recapping from the previous section, we have the following high-level view of our language design: we have a general purpose computation language, similar in style to ML, with direct support for constructing and manipulating typed proof objects of a fixed logic. There also is a universe of data terms, which are terms of simple inductive types like natural numbers and lists. Propositions and proofs reason about terms of this universe; the computational language also has support for abstracting over such data terms and manipulating them. It therefore serves to view our system as a conservative extension of an existing general-purpose language like ML, in order to enable first-class support of proof terms and the objects they reason about.

Therefore our design is composed of three levels: the level of data terms, the logical framework level, and the general computation language. The data-term level is only composed of normal forms of inductively defined zero-order data types, and also variables of such a type; at this level there is no way in which to form a lambda-abstraction over terms of such types, or define any kind of computation or rewriting relation between them. Since our system is complicated, we have chosen only to support a couple of hard-coded data types, namely natural numbers and lists, instead of permitting general inductively-defined types, so that our presentation is simplified; the points that we want to make hold in this simplified version.

The logical framework that we use is a higher-order logic with inductively-defined predicates, which is heavily based upon the Prop universe of the Calculus of Inductive Constructions. Quantification can be done both over propositions (so the Prop universe is impredicative) and data terms; also the predicates that we define can depend both on propositions and data terms. Predicates over propositions are essentially logical connectives like AND and OR; predicates over data terms are relations such as less-or-equal-than, is-prime, etc. The proof objects are essentially witnesses for the axioms of this logic. Apart from the obvious axioms we have elimination forms over data terms which witness their induction schemas, and similarly elimination forms over inductively defined predicates. It is important to note that no kind of reduction relation like beta- or iota-reduction is defined between proof terms; so proofs do not employ a notion of computation.

This point is more pervasive in our framework. Note that we haven't explicitly talked about total functions over data terms returning data terms, as would be expected in a CiC-inspired framework. Functions between data terms can be defined as inductive predicates in the Prop universe; the logic is expressive enough to prove the mode of such relations in itself. In this way the only notion of computation that exists in our framework is that of the general computational language level. Proof objects and data terms lie outside that level and are thus just representation forms and not reducible terms. Still, a limited notion of computation exists at the propositional level, to identify propositions that are beta- or iota-reducible; but this reducibility relation does not entail any kind of computation between data terms.

Our computational language is a higher-order impure functional language; the only special constructs it has are related to data terms and proofs. It permits named abstraction over data-terms and propositions; since propositions can depend on data-term variables and proposition variables, we could say that our language is dependently-typed over those universes. Also, it has a special existential package type that bundles a data-term with a proof object that proves that a certain proposition holds for that data-term. Any kind of construct that we want to support can be added to this language; for demonstration purposes we have added a fixpoint-operator, which is clearly a construct that cannot directly be incorporated as part of a logical framework (as it leads immediately to logical paradoxes). It is easy to add other language features, e.g. references, that would reinforce our claim that general side-effecting computation is permitted.

An important point to make here is that we can create (potentially) non-terminating terms in this language, that produce an existential package of a data-term and a proof object. Our type system together with type safety ensures that if the evaluation of the term does terminate, the produced proof object will be a valid proof of the proposition that it claims it proves. In this way, proof-producing procedures can be written in a general purpose language, while maintaining the validity of the resulting proofs. The practical benefit is that we don't have to guarantee totality of the functions that produce proofs (as we would need to do if these were developed purely inside the logical framework), and this leads to a more natural programming style.

$$
\begin{array}{rrl}
(\textit{Datatypes})\ \tau & ::= & \mathbf{1} \mid \mathbf{nat} \mid \mathbf{list} \\
(\textit{Dataterms})\ \mathrm{w} & ::= & u \mid \mathbf{unit} \mid \mathbf{zero} \mid \mathbf{succ}\,\mathrm{w} \mid \mathbf{nil} \mid \mathbf{cons}(\mathrm{w}_1, \mathrm{w}_2) \\
(\textit{Dataterm env})\ \Gamma & ::= & \cdot \mid \Gamma, u : \tau
\end{array}
$$

Figure 4. Syntax of datatype language

$$
\begin{array}{rrl}
(\textit{Prop kinds})\ \mathcal{K} & ::= & \mathrm{prop} \mid \tau \to \mathcal{K} \mid \mathcal{K} \to \mathcal{K}' \\
(\textit{Propositions})\ P & ::= & U \mid P \Rightarrow P' \mid \forall u : \tau.P \mid \forall U : \mathcal{K}.P \mid \lambda u : \tau.P \\
 & & \mid P\,\mathrm{w} \mid \lambda U : \mathcal{K}.P \mid P_1\,P_2 \mid \mathrm{IndRel}(U : \mathcal{K})\{\vec{P}\} \\
 & & \mid \mathrm{ElimN}\ \mathrm{w}\,P_1\,P_2 \mid \mathrm{ElimL}\ \mathrm{w}\,P_1\,P_2 \\
(\textit{Prop env})\ \Phi & ::= & \cdot \mid \Phi, U : \mathcal{K} \\
(\textit{Proof objects})\ M & ::= & X \mid \lambda X : P.M \mid M\,M' \mid \lambda u : \tau.M \mid M\,\mathrm{w} \\
 & & \mid \lambda U : \mathcal{K}.M \mid M\,P \mid \mathrm{elimN} \mid \mathrm{elimL} \mid \mathrm{ctorRel}_P\ i \\
 & & \mid \mathrm{elimRel}_P \\
(\textit{Proof env})\ \Psi & ::= & \cdot \mid \Psi, X : P
\end{array}
$$

Figure 5. Syntax of logical framework

$$
\begin{array}{rrl}
(\textit{Comp types})\ \sigma & ::= & \mathsf{D}[\tau, P] \mid \sigma_1 \to \sigma_2 \mid \forall u : \tau.\sigma \mid \forall U : \mathcal{K}.\sigma \\
(\textit{Comp terms})\ e & ::= & x \mid \lambda x : \sigma.e \mid e_1\,e_2 \mid \mathsf{fix}\,x : \sigma.e \mid \langle \mathrm{w}, M \rangle \\
 & & \mid \mathsf{let}\ (u, X) = e_1\ \mathsf{in}\ e_2 \mid \lambda u : \tau.e \mid e_1\,[\mathrm{w}_2] \\
 & & \mid \lambda U : \mathcal{K}.e \mid e\,[P] \mid \mathsf{caseN}(\mathrm{w}_1, e_2, e_3) \\
 & & \mid \mathsf{caseL}(\mathrm{w}_1, e_2, e_3) \\
(\textit{Term env})\ \Sigma & ::= & \cdot \mid \Sigma, x : \sigma
\end{array}
$$

Figure 6. Syntax of computational language

$$
(\textit{Values})\ v\ ::= \langle \mathrm{w}, M \rangle \mid \lambda x : \sigma.e \mid \lambda u : \tau.e \mid \lambda U : \mathcal{K}.e
$$

Figure 7. Values

The three levels of the language are defined in figures 4, 5 and 6. The typing rules are given in figures 9, 10, 11 and 12. The operational semantics are shown in figure 8. These rules depend on definitions of the free variables set for each level of the language, plus all the possible kinds of substitutions; these definitions are straightforward.

Let us note a few things about the terms of the computation language. We have different terms for abstraction and function application, and that is because abstraction over a data-term is actually dependent and makes the variable representing the term available in the type level (so essentially this is quantification over data-terms). The same thing holds for abstraction over propositions. Existential packages of dataterms plus proofs are destructed through a let-notation.

4. Meta-theory proofs

In this section we present the proof of the type safety of our language, and argue about the consistency of our logic. Last we sketch the proof of a result regarding proof-erasure semantics for our language.

4.1 Type safety

First we formulate a number of substitution lemmas which will be very important in proving the preservation theorem.

Lemma 4.1 *If* $\Gamma, u_0 : \tau_0 \vdash w_1 : \tau_1$ *and* $\Gamma \vdash w_0 : \tau_0$*, then* $\Gamma \vdash w_1[w_0/u_0] : \tau_1$.

Lemma 4.2 *If* $\Gamma, u_0 : \tau_0; \Phi \vdash P : \mathcal{K}$ *and* $\Gamma \vdash w_0 : \tau_0$*, then* $\Gamma; \Phi \vdash P[w_0/u_0] : \mathcal{K}$.

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\;\textsc{Op-CLApp1} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\, e_2 \longrightarrow v_1\, e_2'}\;\textsc{Op-CLApp2}$$

$$\frac{v_1 = \lambda x : \sigma . e}{v_1\, v_2 \longrightarrow e[v_2/x]}\;\textsc{Op-CLApp3} \qquad \frac{e' = e[\mathsf{fix}\, x : \sigma . e/x]}{\mathsf{fix}\, x : \sigma . e \longrightarrow e'}\;\textsc{Op-Fix}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\,[\mathrm{w}_2] \longrightarrow e_1'\,[\mathrm{w}_2]}\;\textsc{Op-DTApp1}$$

$$\frac{v_1 = \lambda u : \tau . e}{v_1\,[\mathrm{w}_2] \longrightarrow e[\mathrm{w}_2/u]}\;\textsc{Op-DTApp2}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\,[P_2] \longrightarrow e_1'\,[P_2]}\;\textsc{Op-PRApp1}$$

$$\frac{v_1 = \lambda U : \mathcal{K} . e}{v_1\,[P_2] \longrightarrow e[P_2/U]}\;\textsc{Op-PRApp2}$$

$$\frac{e_1 \longrightarrow e_1'}{\mathsf{let}\,(u,X) = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\,(u,X) = e_1'\ \mathsf{in}\ e_2}\;\textsc{Op-Let1}$$

$$\frac{v_1 = \langle \mathrm{w}, M \rangle}{\mathsf{let}\,(u,X) = v_1\ \mathsf{in}\ e_2 \longrightarrow e_2[\mathrm{w}/u][M/X]}\;\textsc{Op-Let2}$$

$$\frac{\mathrm{w} = \mathbf{zero}}{\mathsf{caseN}(\mathrm{w}, e_1, e_2) \longrightarrow e_1}\;\textsc{Op-CaseN1}$$

$$\frac{\mathrm{w} = \mathbf{succ}\,\mathrm{w}'}{\mathsf{caseN}(\mathrm{w}, e_1, e_2) \longrightarrow e_2\,[\mathrm{w}']}\;\textsc{Op-CaseN2}$$

$$\frac{\mathrm{w} = \mathbf{nil}}{\mathsf{caseL}(\mathrm{w}, e_1, e_2) \longrightarrow e_1}\;\textsc{Op-CaseL1}$$

$$\frac{\mathrm{w} = \mathbf{cons}(\mathrm{w}_1, \mathrm{w}_2)}{\mathsf{caseL}(\mathrm{w}, e_1, e_2) \longrightarrow e_2\,[\mathrm{w}_1]\,[\mathrm{w}_2]}\;\textsc{Op-CaseL2}$$

Figure 8. Operational semantics

$$\frac{u : \tau \in \Gamma \qquad \emptyset \vdash \tau \text{ datatype}}{\Gamma \vdash u : \tau} \qquad \Gamma \vdash \mathbf{unit} : \mathbf{1} \qquad \Gamma \vdash \mathbf{zero} : \mathbf{nat}$$

$$\frac{\Gamma \vdash \mathrm{w} : \mathbf{nat}}{\Gamma \vdash \mathbf{succ}\,\mathrm{w} : \mathbf{nat}} \qquad \Gamma \vdash \mathbf{nil} : \mathbf{list}$$

$$\frac{\Gamma \vdash \mathrm{w}_1 : \mathbf{nat} \qquad \Gamma \vdash \mathrm{w}_2 : \mathbf{list}}{\Gamma \vdash \mathbf{cons}(\mathrm{w}_1, \mathrm{w}_2) : \mathbf{list}}$$

Figure 9. Typing of datatype language

$$\frac{U : \mathcal{K} \in \Phi}{\Gamma;\Phi \vdash U : \mathcal{K}} \qquad \frac{\Gamma;\Phi \vdash P_1 : \mathrm{prop} \qquad \Gamma;\Phi \vdash P_2 : \mathrm{prop}}{\Gamma;\Phi \vdash P_1 \Rightarrow P_2 : \mathrm{prop}}$$

$$\frac{\Gamma, u : \tau;\Phi \vdash P : \mathrm{prop}}{\Gamma;\Phi \vdash \forall u : \tau . P : \mathrm{prop}} \qquad \frac{\Gamma;\Phi, U : \mathcal{K} \vdash P : \mathrm{prop}}{\Gamma;\Phi \vdash \forall U : \mathcal{K} . P : \mathrm{prop}}$$

$$\frac{\Gamma, u : \tau;\Phi \vdash P : \mathcal{K}}{\Gamma;\Phi \vdash \lambda u : \tau . P : \tau \to \mathcal{K}} \qquad \frac{\Gamma;\Phi \vdash P : \tau \to \mathcal{K} \qquad \Gamma \vdash \mathrm{w} : \tau}{\Gamma;\Phi \vdash P\,\mathrm{w} : \mathcal{K}}$$

$$\frac{\Gamma;\Phi, U : \mathcal{K} \vdash P : \mathcal{K}'}{\Gamma;\Phi \vdash \lambda U : \mathcal{K} . P : \mathcal{K} \to \mathcal{K}'}$$

$$\frac{\Gamma;\Phi \vdash P_1 : \mathcal{K} \to \mathcal{K}' \qquad \Gamma;\Phi \vdash P_2 : \mathcal{K}}{\Gamma;\Phi \vdash P_1\, P_2 : \mathcal{K}'}$$

$$\frac{\forall i, \Gamma;\Phi, U : \mathcal{K} \vdash P_i : \mathrm{prop} \qquad \forall i, \mathrm{wfc}_U(P_i)}{\Gamma;\Phi \vdash \mathrm{IndRel}(U : \mathcal{K})\{\vec{P}\} : \mathcal{K}}$$

$$\frac{\Gamma \vdash \mathrm{w} : \mathbf{nat} \qquad \Gamma;\Phi \vdash P_1 : \mathcal{K} \qquad \Gamma;\Phi \vdash P_2 : \mathbf{nat} \to \mathcal{K} \to \mathcal{K}}{\Gamma;\Phi \vdash \mathrm{ElimN}\ \mathrm{w}\, P_1\, P_2 : \mathcal{K}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{w} : \mathbf{list} \\ \Gamma;\Phi \vdash P_1 : \mathcal{K} \qquad \Gamma;\Phi \vdash P_2 : \mathbf{nat} \to \mathbf{list} \to \mathcal{K} \to \mathcal{K}\end{array}}{\Gamma;\Phi \vdash \mathrm{ElimL}\ \mathrm{w}\, P_1\, P_2 : \mathcal{K}}$$

Figure 10. Kinding of propositions

Lemma 4.3 *If* $\Gamma, u_0 : \tau_0;\Phi;\Psi \vdash M : P$ *and* $\Gamma \vdash w_0 : \tau_0$, *then* $\Gamma;\Phi;\Psi[w_0/u_0] \vdash M[w_0/u_0] : P[w_0/u_0]$.

Lemma 4.4 *If* $\Gamma, u_0 : \tau_0;\Phi;\Psi;\Sigma \vdash e : \sigma$ *and* $\Gamma \vdash w_0 : \tau_0$, *then* $\Gamma;\Phi;\Psi[w_0/u_0];\Sigma[w_0/u_0] \vdash e[w_0/u_0] : \sigma[w_0/u_0]$.

Lemma 4.5 *If* $\Gamma;\Phi, U_0 : \mathcal{K}_0 \vdash P_1 : \mathcal{K}_1$ *and* $\Gamma;\Phi \vdash P_0 : \mathcal{K}_0$, *then* $\Gamma;\Phi \vdash P_1[P_0/U_0] : \mathcal{K}_1$.

Lemma 4.6 *If* $\Gamma;\Phi, U_0 : \mathcal{K}_0;\Psi \vdash M_1 : P_1$ *and* $\Gamma;\Phi \vdash P_0 : \mathcal{K}_0$, *then* $\Gamma;\Phi;\Psi[P_0/U_0] \vdash M_1[P_0/U_0] : P_1[P_0/U_0]$.

Lemma 4.7 *If* $\Gamma;\Phi, U_0 : \mathcal{K}_0;\Psi;\Sigma \vdash e : \sigma$ *and* $\Gamma;\Phi \vdash P_0 : \mathcal{K}_0$, *then* $\Gamma;\Phi;\Psi[P_0/U_0];\Sigma[P_0/U_0] \vdash e[P_0/U_0] : \sigma[P_0/U_0]$.

Lemma 4.8 *If* $\Gamma;\Phi;\Psi X_0 : P_0 \vdash M_1 : P_1$ *and* $\Gamma;\Phi;\Psi \vdash M_0 : P_0$, *then* $\Gamma;\Phi;\Psi \vdash M_1[M_0/X_0] : P_1$.

Lemma 4.9 *If* $\Gamma;\Phi;\Psi X_0 : P_0;\Sigma \vdash e : \sigma$ *and* $\Gamma;\Phi;\Psi \vdash M_0 : P_0$, *then* $\Gamma;\Phi;\Psi;\Sigma \vdash e[M_0/X_0] : \sigma$.

Lemma 4.10 *If* $\Gamma;\Phi;\Psi;\Sigma, x_0 : \sigma_0 \vdash e_1 : \sigma_1$ *and* $\Gamma;\Phi;\Psi;\Sigma \vdash e_0 : \sigma_0$, *then* $\Gamma;\Phi;\Psi;\Sigma \vdash e_1[e_0/x_0] : \sigma_1$.

We will also need the following simple lemmas.

Lemma 4.11 *If* $u \notin \mathrm{fv}_{\mathrm{dt}}(\sigma)$ *then* $\sigma[w/u] = \sigma$.

Lemma 4.12 *If* $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$ *then* $u \notin \mathrm{fv}_{\mathrm{dt}}(\sigma)$.

The type preservation theorem for our language follows.

$$\frac{X : P \in \Psi}{\Gamma;\Phi;\Psi \vdash X : P} \qquad \frac{\Gamma;\Phi;\Psi, X : P \vdash M : P'}{\Gamma;\Phi;\Psi \vdash \lambda X : P.M : P \Rightarrow P'}$$

$$\frac{\Gamma;\Phi;\Psi \vdash M : P \Rightarrow P' \qquad \Gamma;\Phi;\Psi \vdash M' : P}{\Gamma;\Phi;\Psi \vdash M\,M' : P'}$$

$$\frac{\Gamma, u : \tau;\Phi;\Psi \vdash M : P}{\Gamma;\Phi;\Psi \vdash \lambda u : \tau.M : \forall u : \tau.P}$$

$$\frac{\Gamma;\Phi;\Psi \vdash M : \forall u : \tau.P \qquad \Gamma \vdash \mathrm{w} : \tau}{\Gamma;\Phi;\Psi \vdash M\,\mathrm{w} : P[\mathrm{w}/u]}$$

$$\frac{\Gamma;\Phi, U : \mathcal{K};\Psi \vdash M : P}{\Gamma;\Phi;\Psi \vdash \lambda U : \mathcal{K}.M : \forall U : \mathcal{K}.P}$$

$$\frac{\Gamma;\Phi;\Psi \vdash M : \forall U : \mathcal{K}.P' \qquad \Gamma;\Phi \vdash P : \mathcal{K}}{\Gamma;\Phi;\Psi \vdash M\,P : P'[P/U]}$$

$$\frac{\begin{array}{c}P_{rec} \equiv \forall P : \mathbf{nat} \to \mathrm{prop}\,.P\,\mathbf{zero} \Rightarrow \\ (\forall n : \mathbf{nat}.P\,n \Rightarrow P\,(\mathbf{succ}\,n)) \Rightarrow (\forall n : \mathbf{nat}.P\,n)\end{array}}{\Gamma;\Phi;\Psi \vdash \mathrm{elimN} : P_{rec}}$$

$$\frac{\begin{array}{c}P_{rec} \equiv \forall P : \mathbf{list} \to \mathrm{prop}\,.P\,\mathbf{nil} \Rightarrow \\ (\forall n : \mathbf{nat}.\forall l : \mathbf{list}.P\,l \Rightarrow P\,(\mathbf{cons}(n,l))) \Rightarrow (\forall l : \mathbf{list}.P\,l)\end{array}}{\Gamma;\Phi;\Psi \vdash \mathrm{elimL} : P_{rec}}$$

$$\frac{\Gamma;\Phi \vdash P : \mathrm{prop} \qquad P \equiv \mathrm{IndRel}(U : \mathcal{K})\{\vec{P}\}}{\Gamma;\Phi;\Psi \vdash \mathrm{ctorRel}_P\ i : P_i}$$

$$\frac{\Gamma;\Phi \vdash P : \mathrm{prop} \qquad P \equiv \mathrm{IndRel}(U : \mathcal{K})\{\vec{P}\}}{\Gamma;\Phi;\Psi \vdash \mathrm{elimRel}_P : \mathrm{ElimPrinciple}(U, \vec{P})}$$

$$\frac{P =_{\beta\iota} P' \qquad \Gamma;\Phi;\Psi \vdash M : P}{\Gamma;\Phi;\Psi \vdash M : P'}$$

Figure 11. Typing of proof objects

Theorem 4.13 (Type preservation) *If $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$ and $e \longrightarrow e'$, then $\cdot;\cdot;\cdot;\cdot \vdash e' : \sigma$.*

Proof We perform structural induction on the derivation of the step relation $e \longrightarrow e'$.

Case Op-CLApp1. We have that $e \equiv e_1\,e_2$, $e_1 \longrightarrow e_1'$ and that $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$. By inversion of the typing derivation, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \sigma_1 \to \sigma$ and that $\cdot;\cdot;\cdot;\cdot \vdash e_2 : \sigma_1$. By the inductive hypothesis for $e_1 \longrightarrow e_1'$ we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1' : \sigma_1 \to \sigma$, and by applying the T-clApp typing rule, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1'\,e_2 : \sigma$, which proves our goal since $e' \equiv e_1'\,e_2$.

Case Op-CLApp2. From the step relation rule we have that $e_2 \longrightarrow e_2'$. We also have by inversion for typing that $\cdot;\cdot;\cdot;\cdot \vdash v_1 : \sigma_1 \to \sigma$ and that $\cdot;\cdot;\cdot;\cdot \vdash e_2 : \sigma_1$. By the inductive hypothesis we get that $\cdot;\cdot;\cdot;\cdot \vdash e_2' : \sigma_1$, and by applying the T-clApp rule we get that $\cdot;\cdot;\cdot;\cdot \vdash e' \equiv v_1\,e_2' : \sigma$.

Case Op-CLApp3. From the step relation rule we have that v_1 is of the form $\lambda x : \sigma_0.e_0$. By inversion of the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash v_1\,v_2 : \sigma$ we get that $\cdot;\cdot;\cdot;\cdot \vdash v_1 : \sigma_0 \to \sigma$ and that $\cdot;\cdot;\cdot;\cdot \vdash v_2 : \sigma_0$. By inversion of the typing derivation for v_1 we get that $\cdot;\cdot;\cdot;x : \sigma_0 \vdash e_0 : \sigma$. We now use the substitution

$$\frac{x : \sigma \in \Sigma}{\Gamma;\Phi;\Psi;\Sigma \vdash x : \sigma}\ \text{T-clVar}$$

$$\frac{\Gamma;\Phi;\Psi;\Sigma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma;\Phi;\Psi;\Sigma \vdash \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2}\ \text{T-clFun}$$

$$\frac{\Gamma;\Phi;\Psi;\Sigma \vdash e_1 : \sigma_1 \to \sigma_2 \qquad \Gamma;\Phi;\Psi;\Sigma \vdash e_2 : \sigma_1}{\Gamma;\Phi;\Psi;\Sigma \vdash e_1\,e_2 : \sigma_2}\ \text{T-clApp}$$

$$\frac{\Gamma;\Phi;\Psi;\Sigma, x : \sigma \vdash e : \sigma}{\Gamma;\Phi;\Psi;\Sigma \vdash \mathsf{fix}\,x : \sigma.e : \sigma}\ \text{T-clFix}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{w} : \tau \\ \Gamma;\Phi \vdash P : \tau \to \mathrm{prop} \qquad \Gamma;\Phi;\Psi \vdash M : P\,\mathrm{w}\end{array}}{\Gamma;\Phi;\Psi;\Sigma \vdash \langle \mathrm{w}, M\rangle : \mathsf{D}[\tau, P]}\ \text{T-clPack}$$

$$\frac{\begin{array}{c}\Gamma;\Phi;\Psi;\Sigma \vdash e_1 : \mathsf{D}[\tau, P] \\ \Gamma, u : \tau;\Phi;\Psi, X : P\,u;\Sigma \vdash e_2 : \sigma \qquad u \notin \mathsf{fv}_{\mathrm{dt}}(\sigma)\end{array}}{\Gamma;\Phi;\Psi;\Sigma \vdash \mathsf{let}\ (u,X) = e_1\ \mathsf{in}\ e_2 : \sigma}\ \text{T-clUnpack}$$

$$\frac{\Gamma, u : \tau;\Phi;\Psi;\Sigma \vdash e : \sigma}{\Gamma;\Phi;\Psi;\Sigma \vdash \lambda u : \tau.e : \forall u : \tau.\sigma}\ \text{T-clFunDt}$$

$$\frac{\Gamma;\Phi;\Psi;\Sigma \vdash e_1 : \forall u : \tau.\sigma \qquad \Gamma \vdash \mathrm{w}_2 : \tau}{\Gamma;\Phi;\Psi;\Sigma \vdash e_1\,[\mathrm{w}_2] : \sigma[\mathrm{w}_2/u]}\ \text{T-clAppDt}$$

$$\frac{\Gamma;\Phi, U : \mathcal{K};\Psi;\Sigma \vdash e : \sigma}{\Gamma;\Phi;\Psi;\Sigma \vdash \lambda U : \mathcal{K}.e : \forall U : \mathcal{K}.\sigma}\ \text{T-clFunPr}$$

$$\frac{\Gamma;\Phi;\Psi;\Sigma \vdash e_1 : \forall U : \mathcal{K}.\sigma \qquad \Gamma;\Phi \vdash P_2 : \mathcal{K}}{\Gamma;\Phi;\Psi;\Sigma \vdash e_1\,[P_2] : \sigma[P_2/U]}\ \text{T-clAppPr}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{w}_1 : \mathbf{nat} \qquad \Gamma;\Phi;\Psi;\Sigma \vdash e_2 : \sigma[\mathbf{zero}/u] \\ \Gamma;\Phi;\Psi;\Sigma \vdash e_3 : \forall u' : \mathbf{nat}.\sigma[\mathbf{succ}\,u'/u]\end{array}}{\Gamma;\Phi;\Psi;\Sigma \vdash \mathsf{caseN}(\mathrm{w}_1, e_2, e_3) : \sigma[\mathrm{w}_1/u]}\ \text{T-clCaseN}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{w}_1 : \mathbf{list} \qquad \Gamma;\Phi;\Psi;\Sigma \vdash e_2 : \sigma[\mathbf{nil}/u] \\ \Gamma;\Phi;\Psi;\Sigma \vdash e_3 : \forall u_1 : \mathbf{nat}.\forall u_2 : \mathbf{list}.\sigma[\mathbf{cons}(u_1,u_2)/u]\end{array}}{\Gamma;\Phi;\Psi;\Sigma \vdash \mathsf{caseL}(\mathrm{w}_1, e_2, e_3) : \sigma[\mathrm{w}_1/u]}\ \text{T-clCaseL}$$

Figure 12. Typing of computational language

$$(\lambda u : \tau.P)\,\mathrm{w} \to_\beta P[\mathrm{w}/u] \qquad (\lambda U : \mathcal{K}.P)\,P' \to_\beta P[P'/U]$$

$$\mathrm{ElimN}\ \mathbf{zero}\,P_0\,P_S \to_\iota P_0$$

$$\mathrm{ElimN}\,(\mathbf{succ}\,\mathrm{w})\,P_0\,P_S \to_\iota P_S\,\mathrm{w}\,(\mathrm{ElimN}\ \mathrm{w}\,P_0\,P_S)$$

$$\mathrm{ElimL}\ \mathbf{nil}\,P_n\,P_c \to_\iota P_n$$

$$\mathrm{ElimL}\,(\mathbf{cons}(\mathrm{w}_1, \mathrm{w}_2))\,P_n\,P_c \to_\iota P_c\,\mathrm{w}_1\,\mathrm{w}_2\,(\mathrm{ElimL}\ \mathrm{w}_2\,P_n\,P_c)$$

Figure 13. Beta- and iota-reduction for propositions

lemma 4.10 and get that $\cdot;\cdot;\cdot;\cdot \vdash e_0[v_2/x] : \sigma$ which proves our goal since $e' \equiv e_0[v_2/x]$.

Case OP-FIX. We have $e \equiv \mathsf{fix}\, x : \sigma.e_0$ and $e' \equiv e_0[\mathsf{fix}\, x : \sigma.e_0/x]$. By inversion of the typing derivation for e ($\cdot;\cdot;\cdot;\cdot \vdash \mathsf{fix}\, x : \sigma.e_0 : \sigma$) we get that $\cdot;\cdot;\cdot;x : \sigma \vdash e_0 : \sigma$. By applying the substitution lemma 4.10 for this derivation plus the typing derivation for e, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_0[\mathsf{fix}\, x : \sigma.e_0/x] : \sigma$, which proves our goal.

Case OP-DTAPP1. By inversion of the typing derivation for $e_1\ [\mathrm{w}_2]$, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \forall u : \tau.\sigma$ and that $\cdot \vdash \mathrm{w}_2 : \tau$. We also have $e_1 \longrightarrow e_1'$ from the step relation. From the inductive hypothesis we have that $\cdot;\cdot;\cdot;\cdot \vdash e_1' : \forall u : \tau.\sigma$ and by applying the typing rule T-CLAPPDT, we get that $\cdot;\cdot;\cdot;\cdot \vdash e' \equiv e_1'\ [\mathrm{w}_2] : \sigma$.

Case OP-DTAPP2. From the step relation we have that the value v_1 is of the form $\lambda u : \tau.e_0$. By inversion of the typing derivation for e we get $\cdot;\cdot;\cdot;\cdot \vdash e \equiv v_1\ [\mathrm{w}_2] : \sigma[\mathrm{w}_2/u]$, $\cdot;\cdot;\cdot;\cdot \vdash v_1 : \forall u : \tau.\sigma$ and $\cdot \vdash \mathrm{w}_2 : \tau$. Also, by inversion of the typing derivation for $v_1 = \lambda u : \tau.e_0$ we get that $u : \tau;\cdot;\cdot;\cdot \vdash e_0 : \sigma$. Now we apply the substitution lemma 4.4 for this derivation plus the derivation for w_2, and get $\cdot;\cdot;\cdot;\cdot \vdash e_0[\mathrm{w}_2/u] : \sigma[\mathrm{w}_2/u]$, which proves our goal since $e' \equiv e_0[\mathrm{w}_2/u]$.

Case OP-PRAPP1. From the step relation we have $e_1 \longrightarrow e_1'$. By inversion of the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash e \equiv e_1\ [P_2] : \sigma$, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \forall U : \mathcal{K}.\sigma$ and $\cdot;\cdot \vdash P_2 : \mathcal{K}$. By the inductive hypothesis for $e_1 \longrightarrow e_1'$ we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1' : \forall U : \mathcal{K}.\sigma$, and by applying the typing rule T-CLAPPPR we get that $\cdot;\cdot;\cdot;\cdot \vdash e' \equiv e_1'\ [P_2] : \sigma$.

Case OP-PRAPP2. From the step relation we have that the form of the value v_1 is $\lambda U : \mathcal{K}.e_0$. By inversion of the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash v_1\ [P_2] : \sigma$ we get $\sigma = \sigma_0[P_2/U]$, $\cdot;\cdot;\cdot;\cdot \vdash v_1 : \forall U : \mathcal{K}.\sigma_0$ and $\cdot;\cdot \vdash P_2 : \mathcal{K}$. By inversion of the typing derivation for v_1 (which will end with a T-CLFUNPR rule if we take into consideration the specific form of v_1), we get that $\cdot;U : \mathcal{K};\cdot;\cdot \vdash e_0 : \sigma_0$. By applying the substitution lemma 4.7 for this derivation plus the derivation for P_2, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_0[P_2/U] : \sigma_0[P_2/U]$, which proves our goal, since $e' \equiv e_0[P_2/U]$ and $\sigma = \sigma_0[P_2/U]$.

Case OP-LET1. From the step relation we have $e_1 \longrightarrow e_1'$. By inversion of the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash \mathsf{let}\,(u,X) = e_1 \mathsf{\ in\ } e_2 : \sigma$, we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \mathsf{D}[\tau,P]$, $u : \tau;\cdot;X : P u;\cdot \vdash e_2 : \sigma$ and $u \notin \mathrm{fv}_{\mathrm{dt}}(\sigma)$. From the inductive hypothesis for $e_1 \longrightarrow e_1'$ we get that $\cdot;\cdot;\cdot;\cdot \vdash e_1' : \mathsf{D}[\tau,P]$. We can now apply the same typing rule T-CLPACK, in order to get $\cdot;\cdot;\cdot;\cdot \vdash \mathsf{let}\,(u,X) = e_1' \mathsf{\ in\ } e_2 : \sigma$, which proves our goal.

Case OP-LET2. From the step relation we have that v_1 is a value of the form $\langle \mathrm{w},M\rangle$. By inversion of the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash \mathsf{let}\,(u,X) = v_1 \mathsf{\ in\ } e_2 : \sigma$, we get as above that $\cdot;\cdot;\cdot;\cdot \vdash v_1 : \mathsf{D}[\tau,P]$, $u : \tau;\cdot;X : P u;\cdot \vdash e_2 : \sigma$ and $u \notin \mathrm{fv}_{\mathrm{dt}}(\sigma)$. Since $v_1 = \langle \mathrm{w},M\rangle$ we can apply inversion for its typing derivation and get that $\cdot \vdash \mathrm{w} : \tau$, $\cdot;\cdot \vdash P : \tau \to \mathrm{prop}$, and $\cdot;\cdot;\cdot \vdash M : P\mathrm{w}$. First we apply the substitution lemma 4.4 for the typing derivations of e_2 and w. From this we get that $\cdot;\cdot;X : (P u)[\mathrm{w}/u];\cdot \vdash e_2[\mathrm{w}/u] : \sigma[\mathrm{w}/u]$. But since $u \notin \mathrm{fv}_{\mathrm{dt}}(\sigma)$, we have that $\sigma[\mathrm{w}/u] = \sigma$ by the lemma 4.11. Thus, by performing the substitutions, we get $\cdot;\cdot;X : P\mathrm{w};\cdot \vdash e_2[\mathrm{w}/u] : \sigma$. Now we can use the substitution lemma 4.9 for this derivation and the derivation of M to get $\cdot;\cdot;\cdot;\cdot \vdash e_2[\mathrm{w}/u][M/X] : \sigma$. This proves our goal, since $e' \equiv e_2[\mathrm{w}/u][M/X]$.

Case OP-CASEN1. We have $e = \mathsf{caseN}(v_1,e_2,e_3)$, $e' = e_2$ and that $\mathrm{w}_1 = \mathbf{zero}$. By typing inversion for $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$ we get that $\sigma = \sigma'[\mathrm{w}_1/u]$ and that $\cdot;\cdot;\cdot;\cdot \vdash e_2 : \sigma'[\mathbf{zero}/u]$. Using the equality $\mathrm{w}_1 = \mathbf{zero}$ we get that $\sigma'[\mathrm{w}_1/u] = \sigma'[\mathbf{zero}/u]$, so our goal is proved.

Case OP-CASEN2. We have by the step relation rule that $e = \mathsf{caseN}(\mathrm{w}_1,e_2,e_3)$, $\mathrm{w}_1 = \mathbf{succ}\,\mathrm{w}$ and $e' = e_3\,[\mathrm{w}]$. By inversion of the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash \mathsf{caseN}(v_1,e_2,e_3) : \sigma$ we get that $\sigma = \sigma'[\mathrm{w}_1/u]$, $\cdot \vdash \mathrm{w}_1 : \mathbf{nat}$, $\cdot;\cdot;\cdot;\cdot \vdash e_2 : \sigma'[\mathbf{zero}/u]$ and $\cdot;\cdot;\cdot;\cdot \vdash e_3 : \forall u' : \mathbf{nat}.\sigma'[\mathbf{succ}\,u'/u]$. By inversion of the dataterm typing derivation for $\mathrm{w}_1 = \mathbf{succ}\,\mathrm{w}$ we get that $\cdot \vdash \mathrm{w} : \mathbf{nat}$. Now we apply the typing rule T-CLAPPDT for e_3 and w and get $\cdot;\cdot;\cdot;\cdot \vdash e_3\,[\mathrm{w}] : \sigma'[\mathbf{succ}\,u'/u][\mathrm{w}/u']$. But from the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash e_2 : \sigma'[\mathbf{zero}/u]$ we can deduce that $u' \notin \mathrm{fv}_{\mathrm{dt}}(\sigma'[\mathbf{zero}/u])$, using the lemma 4.12. From this we also get that $u' \notin \mathrm{fv}_{\mathrm{dt}}(\sigma')$. Thus by the lemma 4.11 we get that $\sigma'[\mathrm{w}/u'] = \sigma'$. Now we have that $\sigma'[\mathbf{succ}\,u'/u][\mathrm{w}/u'] = \sigma'[\mathrm{w}/u'][\mathbf{succ}\,\mathrm{w}/u] = \sigma'[\mathbf{succ}\,\mathrm{w}/u]$ which proves our goal.

Case OP-CASEL1.

Case OP-CASEL2.

Case OP-CASEL3. Similar to the above three cases.

For the progress theorem we will need the following lemma which gives us the form of an expression when we know that it is a value and that it has a certain type.

Lemma 4.14 (Canonical forms) *If e is a value and $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$ then,*

1. *if $\sigma = \mathsf{D}[\tau,P]$, then $e = \langle w,M\rangle$.*
2. *if $\sigma = \sigma_1 \to \sigma_2$, then $e = \lambda x : \sigma_1.e_0$.*
3. *if $\sigma = \forall u : \tau.\sigma_0$, then $e = \lambda u : \tau.e_0$.*
4. *if $\sigma = \forall U : \mathcal{K}.\sigma_0$, then $e = \lambda U : \mathcal{K}.e_0$.*

We now formulate and prove the progress theorem for our language.

Theorem 4.15 (Progress) *If $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$ then either e is a value or there exists e' such that $e \longrightarrow e'$.*

Proof We will prove the theorem by performing structural induction on the typing derivation $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$.

Case T-CLVAR. Impossible because the context Σ is empty.

Case T-CLFUN. By the typing rule we have that $e = \lambda x : \sigma_1.e$, thus e is a value.

Case T-CLAPP. By the typing rule we have $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \sigma_1 \to \sigma$ and $\cdot;\cdot;\cdot;\cdot \vdash e_2 : \sigma_1$. By the inductive hypothesis for e_1 we either have that e_1 is a value or that there is an e_1' such that $e_1 \longrightarrow e_1'$. In the latter case, we apply the step relation rule OP-CLAPP1 and get an $e' \equiv e_1'\, e_2$ such that $e \longrightarrow e'$. In the former case, we apply the inductive hypothesis for e_2. If e_2 is not a value there exists e_2' such that $e_2 \longrightarrow e_2'$; we can apply the rule OP-CLAPP2 to get an $e' \equiv e_1\, e_2'$ (since e_1 is a value), such that $e \longrightarrow e'$. If e_2 is a value then we proceed as follows. We apply the canonical forms lemma for e_1 which is a value to get that e_1 is of the form $\lambda x : \sigma.e_0$. We can now apply rule OP-CLAPP3 to get an $e' \equiv e_0[e_2/x]$ such that $e \longrightarrow e'$.

Case T-CLFIX. We can trivially apply rule OP-FIX to get an e' such that $e \longrightarrow e'$.

Case T-CLPACK. In this case $e = \langle \mathrm{w},M\rangle$ so e is a value.

Case T-CLUNPACK. By the typing rule we have $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \langle \tau,P\rangle$. By applying the inductive hypothesis on this derivation, we have that either e_1 is a value, or that there exists e_1' such that $e_1 \longrightarrow e_1'$. In the latter case, we apply rule OP-LET1 and get $e' \equiv \mathsf{let}\,(u,X) = e_1' \mathsf{\ in\ } e_2$ such that $e \longrightarrow e'$. In the former case, we apply the canonical forms lemma and get that e_1 is of the form $\langle \mathrm{w}, M\rangle$. Now we can apply rule OP-LET2 in order to get an $e' \equiv e_2[\mathrm{w}/u][M/X]$ such that $e \longrightarrow e'$.

Case T-CLFUNDT. In this case $e = \lambda u : \tau.e_0$, thus e is a value.

Case T-CLAPPDT. From the typing rule we have that $\cdot;\cdot;\cdot;\cdot \vdash e_1 : \lambda u : \tau.\sigma_0$, so we can apply the inductive hypothesis. If e_1 is not a value then there exists e_1' such that $e_1 \longrightarrow e_1'$, in which case we apply rule OP-DTAPP1 to get an e' such that $e \longrightarrow e'$. If e_1 is a value then by the canonical forms lemma we get that e_1 is of the form $\lambda u : \tau.e_0$, and we can thus apply the rule OP-DTAPP2 to get an e' such that $e \longrightarrow e'$.

Case T-CLFUNPR.

Case T-CLAPPPR. Similar to the above two cases.

Case T-CLCASEN. From the typing rule we have that $\cdot;\cdot;\cdot;\cdot \vdash \mathrm{w}_1 : \mathbf{nat}$. By inversion of the dataterm typing rules, we have either that $\mathrm{w}_1 = \mathbf{zero}$ or that $\mathrm{w}_1 = \mathbf{succ}\,\mathrm{w}'$. In the former case, we can apply rule OP-CASEN1 to get an e' such that $e \longrightarrow e'$; while in the latter case we can apply rule OP-CASEN2.

Case T-CLCASEL. Similar to the above case.

By combining the preservation and progress theorems we prove the type-safety theorem for our language.

Theorem 4.16 (Type safety) *If $\cdot;\cdot;\cdot;\cdot \vdash e : \sigma$ then either e is a value or there exists e' such that $\cdot;\cdot;\cdot;\cdot \vdash e' : \sigma$ and $e \longrightarrow e'$.*

This leads to the following interesting corollary:

Corollary 4.17 *If $\cdot;\cdot;\cdot;\cdot \vdash e : \mathsf{D}[\tau, P]$ and $e \longrightarrow^* e'$ and e' is a value, then there exists a proof object M such that $e' = \langle w, M\rangle$ and $\cdot;\cdot;\cdot \vdash M : P\,w$.*

This means that if a term of the impure computational language has a proof-package type, then we can extract a valid proof object out of it if its evaluation terminates.

4.2 Consistency of logic

To show that the logical framework of our language design is reasonable, we need to show that it is consistent, that is, that logical paradoxes are not provable in it. This is equivalent to not being able to prove the false proposition, which, in higher-order logic is encoded as $\forall U : \mathrm{prop}.U$. Thus we need to show that this proposition is not provable in our system. Normally we would do this by proving subject reduction, strong normalization and the confluence property for the language of propositions. From these we would be able to argue that a proof object with type $\forall U : \mathrm{prop}.U$ is not derivable. Still, we have based our logical framework on the Calculus of inductive Constructions, for which we know that this proposition is not inhabited. The dataterm and Prop universe that we have defined are a strict subset of the CiC language. From these it follows directly that there exists no proof term in our language that inhabits the proposition $\forall U : \mathrm{prop}.U$, therefore our logic is consistent.

4.3 Proof-erasure semantics

What is the run-time overhead of having to create and manipulate proof terms? To answer this question, we define proof-erasure semantics for our language, where the proof terms are completely removed from the runtime representation of our programs. The proof-erased version of an expression e is the result of the contextual closure over expressions of the following function:

$$\begin{aligned} [\![\langle \mathrm{w}, M\rangle]\!]_{pe} &= \langle \mathrm{w}, \cdot\rangle \\ [\![\mathsf{let}\,(u, X) = e_1 \;\mathsf{in}\; e_2]\!]_{pe} &= \mathsf{let}\,(u, \cdot) = [\![e_1]\!]_{pe} \;\mathsf{in}\; [\![e_2]\!]_{pe} \end{aligned}$$

Also, we can define a new set of values based on this conversion, which essentially just drops the proof object out of the dataterm-proof packages.

If we inspect our original operational semantics in figure 8, we will see that they do not depend at all at the proof objects. Thus, if we define semantics for the proof-erased language they will look exactly the same as the semantics for the language that we already have. The only difference will be that any resulting value that is a dataterm-proof package, will not carry the proof term. Therefore, the original and the proof-erased languages are semantically equivalent, since a trivial bisimulation exists between them: by directly mapping operational semantics rules from the original language to the proof-erased language we have that $e \longrightarrow e' \Leftrightarrow [\![e]\!]_{pe} \longrightarrow_{pe} [\![e']\!]_{pe}$. This means that we can choose to have proof terms erased and not present in the runtime representation of our computational terms, while preserving the meanings of our programs. Therefore the runtime overhead of keeping and manipulating proof terms is extinguished, as the validity of proofs is guaranteed by static semantics together with termination of the evaluation of the relevant expression. Of course, depending on the level of reliability that we want, we can still choose to keep proof terms at runtime, so that their validity can be verified externally if need be.

The proof-erasure semantics raise another point: we can have a very efficient runtime representation for our dataterm-proof packages; specifically we can represent such packages just as dataterms. We have argued previously that an efficient, native representation can be used for the dataterms. The proof-erasure semantics tell us that the same representation can be used for such packages too. For example, we can represent a $\mathsf{D}[\mathbf{nat}, P]$ package as an ML BigNum type – an arbitrary precision natural numbers type that uses an array of word-size integers, and performs arithmetic operations by leveraging machine-level operations between integers. Furthermore, if we trust the implementation of the arithmetic operations, we can add trusted primitives to our language that perform these operations and provide proof that they are correct. Since the runtime representation of proof terms under proof-erasure semantics is nil, the native implementations of these operations can be used without worrying about building up proof terms.

If we inspect the operational semantics rule more closely, we can see that proof-erasure semantics could also remove abstraction over propositions and application of propositions. Thus we can add the following two rules to our proof-erasure function.

$$\begin{aligned} [\![\lambda U : \mathcal{K}.e]\!]_{pe} &= e \\ [\![e_1\,[P_2]]\!]_{pe} &= e_1 \end{aligned}$$

Though now the mapping of the operational semantics is not one-to-one, since in the proof-erased version there will be no equivalent to the OP-PRAPP1 rule, our argument about the two languages being semantically equivalent would still hold. This is actually the proof-erasure version that we will use later on in this paper, since it leaves us with simple computational terms.

Note that in a later version of our language design we will most probably add some kind of proposition reflection operation. This means that when given a proposition P (for example, as an argument to a function that quantifies over all propositions), we will be able to inspect the actual form of the proposition and take different actions depending on its form. In that case, proof erasure would not preserve the semantics of the language, since all information about proofs is lost at runtime. Still, a similar result can be proven, by defining a proof-reduced version of the language, where only the proposition that a proof term proves is kept as part of dataterm-proof packages. In other words, instead of preserving the proof term itself, only its type is kept at runtime, something that reduces the associated overhead. Propositional abstraction and application would of course need to be maintained in this case.

In both cases, we can say that our language design is characterized proof-irrelevance, because we never look into the actual proof term of a proposition. This is true even in the case where propositional reflection is added. In general, our view is that the actual structure of a proof term is uninteresting from a computational perspective, so we don't find a notion of going into its structure as something that will become part of our computational language.

5. Encoding data-term level functions

As we have mentioned in chapter 2, our logical framework is very close to CiC, its largest difference being that our dataterm universe (the equivalent to the Set universe) does not employ any notion of computation. This means that we cannot directly define functions like plus in our logical framework which can be evaluated to their result when given two arguments. This also has the implication that we cannot directly prove a proposition involving a term in non-normal form by providing a proof of the same proposition for the term in normal form. For example, we cannot directly prove the proposition $P(1+1)$ from a proof of $P(2)$.

In this chapter we will see how to recover this lost expressivity. In order to do this, we first try to approximate CiC by adding back computations to our data-term universe. Then, we see how to regain the extra power from this change in our original language.

First, let's add terms for lambda-abstraction, function application and inductive elimination to our dataterm universe. We extend our existing syntax with the following forms:

$$\begin{array}{rl} (\textit{Datatypes})\ \tau & ::= \cdots \mid \tau_1 \to \tau_2 \\ (\textit{Dataterms})\ \mathrm{w} & ::= \cdots \mid \lambda u:\tau.\mathrm{w} \mid \mathrm{w}_1\,\mathrm{w}_2 \mid \mathbf{elimnat}(\mathrm{w},\mathrm{w}_1,\mathrm{w}_2) \\ & \quad \mid \mathbf{elimlist}(\mathrm{w},\mathrm{w}_1,\mathrm{w}_2) \end{array}$$

We then define beta- and iota-reduction for these terms. We denote these reductions as β' and ι' in order to distinguish them from beta- and iota-reductions between propositions (that our original language already has). We define them as follows:

$$\begin{array}{rl} (\lambda u:\tau.\mathrm{w})\,\mathrm{w}' & \longrightarrow_{\beta'} \mathrm{w}[\mathrm{w}'/u] \\ \mathbf{elimnat}(\mathbf{zero},\mathrm{w}_1,\mathrm{w}_2) & \longrightarrow_{\iota'} \mathrm{w}_1 \\ \mathbf{elimnat}(\mathbf{succ}\,\mathrm{w}',\mathrm{w}_1,\mathrm{w}_2) & \longrightarrow_{\iota'} \mathrm{w}_2\,\mathrm{w}'\,\mathbf{elimnat}(\mathrm{w}',\mathrm{w}_1,\mathrm{w}_2) \\ \mathbf{elimlist}(\mathbf{nil},\mathrm{w}_1,\mathrm{w}_2) & \longrightarrow_{\iota'} \mathrm{w}_1 \\ \mathbf{elimlist}(\mathbf{cons}(\mathrm{w}',\mathrm{w}''),\mathrm{w}_1,\mathrm{w}_2) & \longrightarrow_{\iota'} \mathrm{w}_2\,\mathrm{w}'\,\mathrm{w}''\,\mathbf{elimlist}(\mathrm{w}'',\mathrm{w}_1,\mathrm{w}_2) \end{array}$$

We denote the reflexive, symmetric and transitive closure of the union of these two relations as $=_{\beta'\iota'}$.

With these additions, we can now define the plus function between natural numbers as follows:

$$x+y \equiv \lambda x.\lambda y.\mathbf{elimnat}(x,y,\lambda x'.\lambda r'.\mathbf{succ}\,r')$$

From the definitions for beta- and iota-reduction, it follows directly that:

$$1+1 =_{\beta'\iota'} 2$$

The typing rules for the extended syntax are obvious so we do not provide them here. Now, we can extend our typing rules for our propositional language, in order to make sure that propositions about beta-iota-equivalent dataterms are identified. This is done so that we are able to establish a large number of proofs that are otherwise not provable. We add the following conversion rule to the typing of proof terms:

$$\frac{P =_{\beta'\iota'} P' \qquad \Gamma;\Phi;\Psi \vdash M : P}{\Gamma;\Phi;\Psi \vdash M : P'}\ \textsc{B'I'conv}$$

Here we have overloaded the $=_{\beta'\iota'}$ relation to also denote the contextual closure over propositions of the relation we defined above.

What does this typing rule mean? We are basically saying that if we have a proof term for a proposition P, that same proof term is also a valid proof for any proposition that results by replacing any dataterm in proposition P by a dataterm that is beta-iota-equivalent to it. Let's see a small example of how this is used (and how propositions which were not provable otherwise are easy to establish). Assume that we can create a proof of $\mathrm{w} = \mathrm{w}$ for any dataterm w, and that this is the only proof regarding equality that we can fundamentally create – an assumption which is actually true for the default encoding of equality in our framework. How do I construct a proof of $1+1=2$? If it wasn't for the B'I'CONV rule, I would only be able to prove either $1+1=1+1$ or $2=2$. But adding this rule, and since $1+1=_{\beta'\iota'}2$, the same proof term M that proves any of these two propositions can serve as the proof of the proposition $1+1=2$.

We are ready now to consider how to recover this functionality in our original language. First, in our original dataterm universe, it is clear that functions like plus cannot be defined. But how can we reason about such functions in our framework? The answer is that we can encode them as predicates in our propositional language. Instead of defining plus as a function, we can define it as a three-place predicate $Q(x,y,z)$ which encodes the fact that z is the result of the addition of x and y. A possible way to define such a predicate in our language would be to use an inductive definition. that encodes the two axioms of addition: the fact that $0+y=y$ and the fact that $x+y=z$ implies $(x+1)+y=(z+1)$. We can therefore define Q as follows:

$$\begin{array}{l} Q(x,y,z) = \mathrm{IndRel}(\mathrm{plus} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \to \mathrm{prop})\{ \\ \quad \forall y.\,\mathrm{plus}\,0\,y\,y \mid \\ \quad \forall x,y,z.\,\mathrm{plus}\,x\,y\,z \Rightarrow \mathrm{plus}\,(\mathbf{succ}\,x)\,y(\mathbf{succ}\,z)\} \end{array}$$

Now, we can use this predicate to talk about the result of the addition of two numbers x and y in our propositions: it will be the term z for which $Q(x,y,z)$ is provable. Thus, we can rewrite propositions involving applications of this function so that they only use the predicate Q. For example, we can rewrite the proposition $1+1>0$ as $\forall r.Q(1,1,r) \Rightarrow r>0$.

Let us now consider the general case. Suppose that we have an expression e of natural number type in non-normal form. We can create an encoding of e in our propositional language as a predicate which is only inhabited by its normal form. Let's denote this predicate as $[\![e]\!]$ – how exactly we arrive to that predicate is the subject of the next two subsections. We can now rewrite any proposition $P(e)$ that involves this expression as $\forall r.\,[\![e]\!]\ r \Rightarrow P(r)$. By repeating this process for all non-normal dataterms in the proposition P, we can arrive at a proposition that is directly expressible in our language, since it only involves dataterms in normal form. This is true even though the original $P(e)$ proposition is not directly expressible (since our language does not permit dataterms in non-normal form).

We have now regained the expressibility of the extended language in our original language. But we have not yet established the fact that the propositions that the B'I'CONV rule makes provable are still provable in our restricted setting. Let's consider this rule again. What it essentially gives us is an automatic way to use a proof of $P(e)$ as a proof of $P(e')$ when $e =_{\beta'\iota'} e'$. This is entirely equivalent to having a way to use a proof of $P(e)$ as a proof of $P(\mathrm{w})$ and vice-versa, where w is the normal form of e. In our original language, we would encode $P(e)$ as $\forall r.\,[\![e]\!]\ r \Rightarrow P(r)$, and $P(\mathrm{w})$ would be left as is. Thus, the equivalent of the B'I'CONV rule would be to have an automatic way to go from a proof of $\forall r.\,[\![e]\!]\ r \Rightarrow P(r)$ to a proof of $P(\mathrm{w})$ and vice-versa. In that case, we would be able to prove exactly the same propositions as in the extended language, because we will have recovered the missing conversion rule.

How can we support this automatic conversion in our language? We can define two functions in our computational language that perform this conversion – from a proof about the normal form to a proof of the not-reduced term, and vice versa. These functions should have the types:

$$f_1 :: \forall P : \textbf{nat} \to \text{prop}.\forall u : \textbf{nat}.\mathsf{D}[\mathbf{1}, P(u)] \to \mathsf{D}[\mathbf{1}, \forall r. [\![e]\!]\ r \Rightarrow P(r)]$$
$$f_2 :: \forall P : \textbf{nat} \to \text{prop}.\forall u : \textbf{nat}.\mathsf{D}[\mathbf{1}, \forall r. [\![e]\!]\ r \Rightarrow P(r)] \to \mathsf{D}[\mathbf{1}, P(u)]$$

Let's see how this functions should work. The first should calculate the normal form of the expression e into a variable r', and should then compare r' with the dataterm u. If they're equal, we will have a proof of $u = r'$, otherwise the conversion function will fail with an error. Now if we had a proof M that the only term that inhabits the $[\![e]\!]$ predicate is in fact u, our job would be complete: we would prove the proposition $\forall r. [\![e]\!]\ r \Rightarrow P(r)$ by proving that every such r would be equal to u and therefore the proof $P(u)$ that we already have suffices. Therefore we need this proof M of the proposition $\forall r. [\![e]\!]\ r \Rightarrow u = r$ (which is equivalent to saying that only u inhabits the $[\![e]\!]$ predicate). Since $u = r'$, it would be enough to have a proof of $\forall r. [\![e]\!]\ r \Rightarrow r' = r$.

For the second function, we have a proof of the proposition $\forall r. [\![e]\!]\ r \Rightarrow P(r)$ and need to prove $P(u)$. Therefore, it would be enough to have a proof of the fact that u inhabits the $[\![e]\!]$ predicate, that is, a proof of the proposition $[\![e]\!]\ u$. Let's assume that we calculate again the normal form of the expression e into a variable r', and also that we have a proof M that $[\![e]\!]\ r'$. Then, we could compare u with r', and if they're equal, we' re done: we rewrite the proof M as $[\![e]\!]\ u$ since $u = r'$, apply that to the input proof, and get the desired proof we were aiming at. If they're not equal, our conversion function should just fail.

From this discussion it is apparent that we need to have a way in our computational language to calculate the normal form r of any expression e, and also to construct proofs of the two propositions: $\forall r'. [\![e]\!]\ r' \Rightarrow r = r'$ and $[\![e]\!]\ r$. Equivalently, we need a way to construct a dataterm-proof package in our computational language, with the following type:

$$\mathsf{D}[\textbf{nat}, \lambda r. [\![e]\!]\ r \wedge \forall r'. [\![e]\!]\ r' \Rightarrow r = r']$$

Having such a term for each expression e would be enough to define the above two functions f_1 and f_2, essentially regaining the conversion rule B'I'CONV. The way to do this will be described in the following two sections. Furthermore, from the discussion about how f_1 and f_2 should work, it should be apparent that what we are essentially doing is implementing this rule in our computational language, instead of having it be a part of our type-checker.

In the next two sections, we will see two different approaches at encoding the expressions e of the extended language as predicates of our propositional language, and also the accompanying computational-level terms that provide the dataterm-proof packages that we described above. These encodings are defined at the meta-level (outside our framework); a future research goal will be to extend our computational language so that they can directly be expressed within our framework.

5.1 First translation

The first translation of expressions of the extended language into our propositional language that we defined is based on the following two ideas: first, we define normal forms for expressions of any type and provide a predicate for each expression that is inhabited only by its single normal form. Second, we try to avoid depending crucially on iota-reduction between propositions.

The source language for which we provide translations is presented in figure 14; compared to the extended language we defined in the previous section, we have limited ourselves to natural numbers, and have defined slightly different syntax for the inductive elimination principle. The constraint to natural numbers is just to simplify the presentation; the ideas we present could be extended to other datatypes too. We have separated the terms of the language in two syntactic categories: values and expressions. Values are terms that are not reduced further; they consist of constructors of natural numbers, variables, and lambda-abstractions. Expressions are terms that might be reduced using beta- or iota-reduction, thus consist of all values plus function application and the elimination form for natural numbers.

$$\begin{array}{rl} (\textit{Types})\ \tau & ::= \textit{nat} \mid \tau_1 \to \tau_2 \\ (\textit{Values})\ v & ::= x \mid \textbf{zero} \mid \textbf{succ}\,x \mid \lambda x : \tau.e \\ (\textit{Expressions})\ e & ::= v \mid e_1\,e_2 \mid \textbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\ v \end{array}$$

Figure 14. Dataterms plus functions language syntax

$$\frac{e_1 \longrightarrow e_1'}{e_1\,e_2 \longrightarrow e_1'\,e_2}\ \text{FSAPP1} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\,e_2 \longrightarrow v_1\,e_2'}\ \text{FSAPP2}$$

$$\frac{v_1 = \lambda x : \tau.e}{v_1\,v_2 \longrightarrow e[v_2/x]}\ \text{FSBETA}$$

$$\frac{v = \textbf{zero}}{\textbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\ v \longrightarrow e_0}\ \text{FSIOTA1}$$

$$\frac{v = \textbf{succ}\,v'}{\textbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\ v \longrightarrow e_s[v'/x'][\textbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\ v'/z']}\ \text{FSIOTA2}$$

Figure 15. Operational semantics of dataterms plus functions language

The operational semantics for this language are shown in figure 15. The core of these semantics are the rules FSBETA and FSIOTA* which correspond to beta- and iota-reduction. As is well known, this language is terminating, so every expression will produce a value after a finite sequence of steps. We write this fact as $\forall e.\exists v.e \longrightarrow^* v$, and refer to the value v that e evaluates to as its canonical form. Note here that this language is powerful enough to write not only all primitive recursive functions, but also non-primitive recursive total functions like the Ackermann function.

Let's go into the specifics of our translation. As we already mentioned, to regain the expressiveness of this language into our framework we define encodings of these terms in two levels of our framework, namely the logical level and the computational language. The rough idea behind the translation at the logical level is that we want to have a one-place predicate P for each expression e, such that a proof of $P\,r$ exists for some r and guarantees that r is equal to the canonical form of the expression e (namely, if $e \longrightarrow^* v$, then $P\,v$ is provable, and $P\,r$ implies $r = v$).

We define this translation in two steps. First, we show how to translate each value v into a data-term or predicate of our language. This is done using the function $[\![\cdot]\!]_v$ shown in figure 16. Second, we show how to translate each expression e into a one-place predicate P such that if $e \longrightarrow^* v$, the only r for which we can construct a proof of the proposition $P\,r$ is $r = [\![v]\!]_v$. This translation is denoted using the function $[\![\cdot]\!]_p$ and is shown in figure 17. The two translations are mutually recursive, so we present the reasoning behind them in tandem.

The translation $[\![\cdot]\!]_v$ is trivial for natural numbers, i.e. $[\![\textbf{succ}\,\textbf{zero}]\!]_v = \textbf{succ}\,\textbf{zero}$. A lambda-abstraction needs to be translated as a relation, since we cannot directly define it as a function. For example a lambda-abstraction $\lambda x.e$ that expects a **nat** and returns a **nat** should

Types returned

$$[\![\mathbf{nat}]\!]_v = \mathbf{nat}$$

$$[\![\tau_1 \to \tau_2]\!]_v = [\![\tau_1]\!]_v \to [\![\tau_2]\!]_v \to \text{prop}$$

Actual translation

$$[\![x]\!]_v = x$$

$$[\![\mathbf{zero}]\!]_v = \mathbf{zero}$$

$$[\![\mathbf{succ}\, v']\!]_v = \mathbf{succ}\ [\![v']\!]_v$$

$$[\![\lambda x : \tau . e]\!]_v = \lambda x : [\![\tau]\!]_v . [\![e]\!]_p$$

Figure 16. Converting values into dataterms or predicates

be translated as a two-place relation R that is inhabited by the pairs (x', v') for all natural numbers x', where v' is the canonical form of the expression $e[x'/x]$. The way to encode a relation like that in our framework is to define a two-place predicate P of the type $\mathbf{nat} \to \mathbf{nat} \to \text{prop}$, such that we can construct proof objects proving the proposition $P\,x'\,v'$. There is a simple way to achieve this: we define P as $\lambda x. [\![e]\!]_p$. For each x', $P\,x'$ will be the one place predicate $[\![e[x'/x]]\!]_p$, which by construction of $[\![\cdot]\!]_p$, will be only inhabited by the value v'. That means, we will be able to construct a proof of $P\,x'\,v'$. When we define the translation we need to keep in mind that function types get translated into predicate types; to help us with that we have overloaded the translation $[\![\cdot]\!]_v$ to operate on types too.

Let's move on to define the translation $[\![\cdot]\!]_p$. As we said, this translation should yield a predicate P for each expression e such that if $e \longrightarrow^* v$ then we can construct a proof of $P\,v$, and of the fact that for any r for which we have a proof of $P\,r$, that r is equal to v (this is what our computational-level functions will do). For values, defining such a proposition is a trivial thing: since $v \longrightarrow^* v$, P is just the equality to $[\![v]\!]_v$ relation, namely $P \equiv (\text{eq}\,[\![v]\!]_v)$. Here we have supposed that we have defined an equality predicate for each type of dataterm or proposition, as follows:

$$\begin{array}{lcllcl} \text{eq}_\tau & : & \tau \to \tau \to \text{prop} & \equiv & \lambda x : \tau . \,\text{IndRel}(U : \tau \to \text{prop})\{U\,x\} \\ \text{refl}_\tau & : & \forall x : \tau . x =_\tau x & \equiv & \lambda x : \tau . \,\text{ctorRel}_{=_\tau x}\, 0 \end{array}$$

We drop the subscript τ since it can easily be inferred from the context, and use the syntax $x = y$ for $\text{eq}\, x\, y$.

For beta-redexes, we need to calculate the canonical forms r_1 and r_2 of e_1 and e_2 respectively. Since e_1 is of arrow type, r_1 will be a two place predicate (as is evident from the type returned by $[\![\tau \to \tau']\!]_v$) representing the function that e_1 evaluates to. Furthermore, for each r_2, there will be a single r that satisfies the proposition $r_1\,r_2\,r$. This r is exactly the single term that should inhabit the predicate $[\![e_1\,e_2]\!]_p$. We thus construct this predicate in a way that reflects this fact.

For iota-redexes, we create a two-place inductive predicate, which is inhabited for each natural number n by the canonical form of the expression $\mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\,n$. This inductive predicate has two constructors, which essentially reflect the FSIOTA1 and FSIOTA2 rules of the operational semantics. Then, we apply the translation of the value we're actually eliminating on to this predicate, getting a one-place predicate that is inhabited by the canonical form of the original expression $\mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\,v$.

Types returned

$$[\![\tau]\!]_p = [\![\tau]\!]_v \to \text{prop}$$

Actual translation

$$[\![v]\!]_p = (\text{eq}\,[\![v]\!]_v)$$

$$\begin{array}{c} [\![e_1\,e_2 : \tau']\!]_p = \lambda r : [\![\tau']\!]_v . \\ \forall r_1 : [\![\tau \to \tau']\!]_v . [\![e_1]\!]_p\, r_1 \Rightarrow \\ \forall r_2 : [\![\tau]\!]_v . [\![e_2]\!]_p\, r_2 \Rightarrow \\ r_1\, r_2\, r \end{array}$$

$$\begin{array}{c} [\![\mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\,v : \tau]\!]_p = \\ \text{IndRel}(\text{ENPred} : \mathbf{nat} \to [\![\tau]\!]_v \to \text{prop})\{ \\ \forall r : [\![\tau]\!]_v . [\![e_0]\!]_p\, r \Rightarrow \text{ENPred}\, 0\, r \\ |\, \forall x' : \mathbf{nat}. \forall z' : [\![\tau]\!]_v . U\, x'\, z' \Rightarrow \\ \forall r : [\![\tau]\!]_v . [\![e_s]\!]_p\, r \Rightarrow \text{ENPred}\, (\mathbf{succ}\, x')\, r\} \\ [\![v]\!]_v \end{array}$$

the above inductive predicate has the following elimination principle:

$$\begin{array}{c} \text{useENPred} = \forall P : \mathbf{nat} \to \tau \to \text{prop} . (\forall r. [\![e_0]\!]_p\, r \Rightarrow P\, 0\, r) \Rightarrow \\ (\forall x'. \forall z'. \text{ENPred}\, x'\, z' \Rightarrow P\, x'\, z' \Rightarrow \forall r. [\![e_s]\!]_p\, r \Rightarrow P\,(\mathbf{succ}\, x')\, r) \Rightarrow \\ \forall x : \mathbf{nat}. \forall y : \tau . \text{ENPred}\, x\, y \Rightarrow P\, x\, y \end{array}$$

Figure 17. Converting expressions into predicates

As an example, let's consider the translation of the addition function between natural numbers, namely the expression:

$$\lambda x. \lambda y. \mathbf{elimnat}\{y \,|\, x'.z'.\mathbf{succ}\, z'\}\, x$$

By following the rules of the translation we end up with the following one-place predicate:

$$\begin{array}{l} \text{eq}(\lambda x. \\ \quad \text{eq}(\lambda y. \\ \quad\quad \text{IndRel}(\text{Add} : \mathbf{nat} \to \mathbf{nat} \to \text{prop})\{ \\ \quad\quad\quad \forall r : \mathbf{nat}. x = r \Rightarrow \text{Add}\, 0\, r \\ \quad\quad\quad |\, \forall x' : \mathbf{nat}. \forall z' : \mathbf{nat}. \text{Add}\, x'\, z' \Rightarrow \\ \quad\quad\quad\ \ \forall r : \mathbf{nat}. (\mathbf{succ}\, z') = r \Rightarrow \text{Add}\, (\mathbf{succ}\, x')\, r \\ \quad\quad\quad \} \\ \quad\quad) \\ \quad) \end{array}$$

Let's now move on to the translation functions into the computational language. The rough idea behind this translation is to have a way to compute the canonical form v of any expression e, plus to prove the fact that v *is* the *single* canonical form of the expression e. This is done by returning a proof object for the fact that $[\![e]\!]_p$ is inhabited only by v. Thus, for any expression e of type τ we want to construct a computational-level term with the type:

$$\exists r : [\![\tau]\!]_v . [[\![e]\!]_p\, r \wedge \forall r'. [\![e]\!]_p\, r' \Rightarrow r = r']$$

By construction of $[\![\cdot]\!]_p$, having a term of such a type means that we have a way to find the encoding of the unique value v for which $e \longrightarrow^* v$ holds.

That is not the whole story, though. Consider the case where the type τ of the expression e is a function type like $\mathbf{nat} \to \mathbf{nat}$. Then the single r that the above term will give us will be a relation of type $\mathbf{nat} \to \mathbf{nat} \to \text{prop}$. In that case, when we apply e to v we want to be able to find the canonical form that the expression $e\,v$ will evaluate to. To do that, we also need a computational level function that given a canonical form as an argument returns us the canonical form of the result; plus the proof that $e\,v$ actually evaluates to that form. This means that we need a computational term of the type $\forall y : nat.\exists z : nat.[r\,y\,z \wedge \forall z'.r\,y\,z' \Rightarrow z = z']$. Another way to see this is that we need some kind of guarantee of the $(\forall, \exists!)$ mode of the relation, and the extra computational term that we need is exactly this guarantee. We denote the type of this term as $\mathsf{cltct}_\tau(r:\tau)$ where r is the relation for which we want to construct this function and τ is the type of the relation; the term itself is denoted as $\mathsf{cltct}(e)$.

We are still not fully done though. Consider the case where e is of type $(\mathbf{nat} \to \mathbf{nat}) \to \mathbf{nat}$. According to the above, we would need to have a function of the type $\forall r' : \mathbf{nat} \to \mathbf{nat} \to \mathrm{prop}.\exists z : nat.[r\,r'\,z \wedge \forall z'.r\,r'\,z' \Rightarrow z = z']$. That function doesn't have any information about the mode of the r' relation – equivalently there is no way to actually get the result of applying r' to an argument. Thus we also require a similar function for r', leading the overall type $\mathsf{cltct}_\tau(r:\tau)$ to be equal to

$$\begin{array}{l}\forall r' : \mathbf{nat} \to \mathbf{nat} \to \mathrm{prop}.\\ \quad (\forall y : \mathbf{nat}.\exists z : \mathbf{nat}.[r'\,y\,z \wedge \forall z'.r'\,y\,z' \Rightarrow z = z']) \to\\ \quad \exists z' : \mathbf{nat}.[r\,r'\,z \wedge \forall z'.r\,r'\,z' \Rightarrow z = z']\end{array}$$

The original type of the computational-level term that we construct for an expression e has to take these things into account. We denote the computational level term for an expression $e:\tau$ as $\mathsf{clpkg}(e)$; the type of this term is denoted as $\mathsf{clpkg}_\tau(r:\tau)$. In 18 and 19 we present these translation functions plus the logical-level terms that these translations depend on. Note that we have slightly deviated from the notation used in the previous section for the language, for example by using let-notation to open up existential packages and products. Also note that we have assumed that the language has general existential quantification over dataterms and props (instead of existential packages), and that it includes a product type. Even though these are not presented in the previous section, they are straightforward to add. Especially in the case of existential quantification, the key ideas behind it are already present in our language in the form of the dataterm-proof object existential package – the typing behind the existential types and the proof of type safety would be entirely similar to the ones we already have for the existential package. Last, let us note that we have used some extra type annotations and the redundant annotation *proof of* $\cdot$ *is* $\cdots$ which gives the proposition that a proof term that follows proves, in order to make the translation somewhat easier to follow and type-check.

How do we argue that this translation is actually valid? First, we can be convinced that the translation into the propositional language accurately encodes the semantics of the source language, due to it being relatively small and straightforward. Then, we need to show that the translation into the computational language is also valid, e.g. that all the terms are well-typed under our typing rules, and that all terms terminate in a finite number of steps. This is what the following two lemmas prove:

Lemma 5.1 *The terms* $\mathsf{cltct}(v:\tau)$ *and* $\mathsf{clpkg}(e:\tau)$ *are well-typed for all expressions* $e:\tau$*, and have types* $\mathsf{cltct}_\tau(v:\tau)$ *and* $\mathsf{clpkg}_\tau(e:\tau)$ *respectively.*

This follows by inspecting the translation itself. In most cases, it is relatively easy to see why terms are well-typed. The most difficult case is the second branch of the caseN construct when translating the inductive elimination form, where we need to prove that $\forall r'.\mathrm{ENPred}\,(\mathbf{succ}\,x')\,r' \Rightarrow r = r'$. This proceeds by performing inductive elimination on the ENPred term, and essentially requiring a proof of:

$$\forall x''.\forall z''.\mathrm{ENPred}\,x''\,z'' \Rightarrow \forall r.\,[\![e_s]\!]_p\; r \Rightarrow x' = x'' \Rightarrow r = r'$$

The problem here is that even if we have a proof about $[\![e_s]\!]_p$ that would enable us to remove the $\forall r.\,[\![e_s]\!]_p\; r$ part, this proof talks about $[\![e_s]\!]_p$ using x' and z' instead of x'' and z''. Thus, we build proofs of $x' = x''$ and $z' = z''$, so we can convert our proof regarding the mode of $[\![e_s]\!]_p$ to a proof involving x'' and z'' that permits us to prove that $r = r'$ from the $\forall r.\,[\![e_s]\!]_p\; r$ part.

Lemma 5.2 *The evaluation of the terms* $\mathsf{cltct}(e:\tau)$ *and* $\mathsf{clpkg}(e:\tau)$ *terminates.*

This is easy to prove, considering that the only recursive call that could result in non-termination is the use of the fix-point of f in the translation of the inductive elimination form. We see that this recursive call uses f on a smaller argument than its original argument, so after a finite number of steps, the evaluation of f into any natural number argument will terminate. Essentially, this function is defined by structural recursion, and no other terms can cause non-termination, therefore all the defined terms terminate.

5.2 Simpler translation

The notion of canonical forms of terms of functional types complicates the above translation very much. Also, the proof procedures we end up with in the computational language aren't very easy to use, something that is apparent if we consider the number of steps (and different computational level functions) that a simple proof like $1+1=2$ would require. We have designed another translation that leverages the $\beta\iota-$reduction of the propositional level in order to provide easier to work with encodings of the dataterm plus functions language.

The basic idea behind this translation is that natural numbers are encoded as one-place predicates inhabited by their normal form – just like in the previous translation; but functions are encoded as functions at the propositional level. The departure from the previous translation is that $\iota-$reduction is critically leveraged, by having parts of the propositions that involve large elimination on natural numbers. For example the "is-0" function $\lambda x.\mathbf{elimnat}(x, 1, \lambda x'.\lambda z'.0)$ can be encoded as a propositional level function of type $(\mathbf{nat} \to \mathrm{prop}) \to (\mathbf{nat} \to \mathrm{prop})$ (keeping in mind that natural numbers are one-place predicates). This function gets a $\mathbf{nat} \to \mathrm{prop}$ encoding of a natural number (which we call X), and returns an encoding of the result as a one-place predicate $\mathbf{nat} \to \mathrm{prop}$ inhabited only by the canonical form of the result. Suppose X is only inhabited by x. Then, in the case where $x = \mathbf{zero}$, this one-place predicate should just be the encoding of 1; in the case where $x = \mathbf{succ}\,x'$, it should be the encoding of 0. This can be written in our propositional language as: $\lambda X : \mathbf{nat} \to \mathrm{prop}.(\lambda r : \mathbf{nat}.\forall x : \mathbf{nat}.X\,x \Rightarrow (\mathrm{ElimN}\,x\,(\lambda r'.r' = 1)(\lambda x' : \mathbf{nat}.\lambda z' : \mathbf{nat} \to \mathrm{prop}.\lambda r'.r' = 0))\,r)$. This is a propositional-level function; thus, when we apply the encoding of a natural number to it, e.g. $\lambda r.r = \mathbf{zero}$, it is beta-reduced to: $(\lambda r : \mathbf{nat}.\forall x : \mathbf{nat}.x = \mathbf{zero} \Rightarrow (\mathrm{ElimN}\,x\,(\lambda r'.r' = 1)(\lambda x' : \mathbf{nat}.\lambda z' : \mathbf{nat} \to \mathrm{prop}.\lambda r'.r' = 0))\,r)$ Obviously $x = \mathbf{zero}$ is the only x that satisfies this predicate. After proving this fact we can show that this proposition is equivalent to the simpler proposition $\lambda r.\mathrm{ElimN}\,0\,(\lambda r'.r' = 1)(\lambda x' : \mathbf{nat}.\lambda z' : \mathbf{nat} \to \mathrm{prop}.\lambda r'.r' = 0))\,r$, which, after $\beta\iota-$reduction is equal to $\lambda r.r = 1$. This proposition is obviously only inhabited by the canonical of the original expression. So we see that $\beta\iota-$reduction helps us along the way to simplify the propositions that our translation forms into more meaningful propositions.

Before we see the translation itself, let us note that the source language that we use is changed slightly: the **elimnat** construct is written with syntax $\mathbf{elimnat}(e, e_1, e_2)$, where e is the natural number expression we' re eliminating on, and e_1 and e_2 are the cases for **zero** and **succ** respectively. Also, we have limited the type that this expression might return to **nat** and $\mathbf{nat} \to \mathbf{nat}$. This

Types returned

$$\mathsf{clpkg}_\tau(e:\tau) = \mathsf{clpkg}_\tau(\llbracket e \rrbracket_p : \llbracket \tau \rrbracket_p)$$
$$\mathsf{clpkg}_\tau(R:\llbracket \tau \rrbracket_p) = \exists r : \llbracket \tau \rrbracket_v . [R\,r \wedge \forall r'.R\,r' \Rightarrow r = r'] * \mathsf{cltct}_\tau(r : \llbracket \tau \rrbracket_v)$$
$$\mathsf{cltct}_\tau(v:\tau) = \mathsf{cltct}_\tau(\llbracket v \rrbracket_v : \llbracket \tau \rrbracket_v)$$
$$\mathsf{cltct}_\tau(r:\llbracket \tau_1 \to \tau_2 \rrbracket_v) = \forall y : \llbracket \tau_1 \rrbracket_v . \mathsf{cltct}_\tau(y : \llbracket \tau_1 \rrbracket_v) \to \mathsf{clpkg}_\tau(r\,y : \llbracket \tau_2 \rrbracket_p)$$
$$\mathsf{cltct}_\tau(r : \llbracket \mathbf{nat} \rrbracket_v) = \top$$

Logical-level terms

$\top$	:	prop	=	$\mathrm{IndRel}(U : \mathrm{prop})\{U\}$
True	:	$\top$	=	$\mathrm{ctorRel}_\top\ 0$
false	:	prop	=	$\mathrm{IndRel}(U : \mathrm{prop})\{U\}$
useFalse	:	$\mathrm{false} \to \forall P : \mathrm{prop}\,.P$	=	$\lambda f.\lambda P.\,\mathrm{elimRel}_{\mathrm{false}}\ P\,f$
eq	:	$\tau \to \tau \to \mathrm{prop}$	=	$\lambda x : \tau.\,\mathrm{IndRel}(U : \tau \to \mathrm{prop})\{U\,x\}$
		where τ can be either a dataterm type or a prop-kind		
useEq	:	$\forall x : \tau.\forall P : \tau \to \mathrm{prop}\,.P\,x \Rightarrow \forall y : \tau.x = y \Rightarrow P\,y$	=	$\lambda x.\,\mathrm{elimRel}_{\mathrm{eq}\,x}$
reflexivity x	:	$x = x$	=	$\mathrm{ctorRel}_{\mathrm{eq}\,x}\ 0$
symmEq	:	$x = y \Rightarrow y = x$	=	$\mathrm{useEq}\ x\,(\lambda r.r = x)\,(\mathrm{reflexivity}\ x)$
succEq	:	$x = y \Rightarrow \mathbf{succ}\,x = \mathbf{succ}\,y$	=	$\mathrm{useEq}\ x\,(\mathrm{eq}\,(\mathbf{succ}\,x))\,(\mathrm{reflexivity}\,(\mathbf{succ}\,x))\,y$
predEq	:	$\mathbf{succ}\,x = \mathbf{succ}\,y \Rightarrow x = y$	=	$\mathrm{useEq}\,(\mathbf{succ}\,x)\,(\lambda n.\,\mathrm{ElimN}\ n\,\top\,(\lambda x'.\lambda\_.x = x'))\,(\mathrm{reflexivity}\ x)\,y$
zeroNotSucc	:	$\forall x.\mathbf{succ}\,x = 0 \Rightarrow \mathrm{false}$	=	$\lambda x.\,\mathrm{useEq}\,(\mathbf{succ}\,x)\,(\lambda n.\,\mathrm{ElimN}\ n\ \mathrm{false}\,(\lambda\_.\lambda\_.\top))\ \mathrm{True}\ \mathbf{zero}$
$P_1 \wedge P_2$	:	prop	=	$\mathrm{IndRel}(U : \mathrm{prop})\{P_1 \Rightarrow P_2 \Rightarrow U\}$
useAnd	:	$\forall P : \mathrm{prop}\,.(P_1 \Rightarrow P_2 \Rightarrow P) \Rightarrow P_1 \wedge P_2 \Rightarrow P$	=	$\mathrm{elimRel}_{P_1 \wedge P_2}$
mkAnd	:	$P_1 \Rightarrow P_2 \Rightarrow P_1 \wedge P_2$	=	$\mathrm{ctorRel}_{P_1 \wedge P_2}\ 0$
left	:	$P_1 \wedge P_2 \Rightarrow P_1$	=	$\mathrm{useAnd}\ P_1\,(\lambda X_1.\lambda X_2.X_1)$
right	:	$P_1 \wedge P_2 \Rightarrow P_2$	=	$\mathrm{useAnd}\ P_2\,(\lambda X_1.\lambda X_2.X_2)$

Figure 18. Converting an expression to its canonical form

limits the generality of our translation, but was only done so that the presentation is more readable and easier to explain; extending the idea to any type is fairly straighforward. This limitation doesn't limit our ability to express functions beyond primitive recursive functions, like the Ackermann function.

In figure 20 we see the translation of terms of the source language into the propositional level of our framework. We have assumed that an equality relation is defined between natural numbers as follows:

$$\mathrm{eq} : \mathbf{nat} \to \mathbf{nat} \to \mathrm{prop} = \lambda x : \mathbf{nat}.\,\mathrm{IndRel}(U : \mathbf{nat} \to \mathrm{prop})\{U\,x\}$$

We write $(\mathrm{eq}\,x)\,y$ as $x = y$, or simply as $(\mathrm{eq}\,x)$ when only one argument is applied, which then denotes the "equal-to-x" one-place predicate.

One other thing that we need to comment on is the definition of the translation for the **elimnat** construct. The first part of it is straightforward – we require that the argument we' re eliminating on is evaluated to a normal form denoted as n. In the second part, we want to return a predicate of type $\llbracket \tau \rrbracket$ depending on the actual value of n. This is done by using large-elimination on the natural number n. When $n = \mathbf{zero}$, we want to return the predicate that corresponds to the encoding of the base case of the elimination, namely e_1. When $n = \mathbf{succ}\,n'$ we want to return the encoding of the inductive step of the elimination, namely e_2, applied to the predecessor and the result at the predecessor; iota-elimination at the propositional level will take care of that.

Note though that the original large-elimination construct ElimN creates a proposition of the expected kind $\mathcal{K}$, when given a natural number n, a proposition of kind $\mathcal{K}$ (which corresponds to the case where $n = 0$), and a proposition of kind $\mathbf{nat} \to \mathcal{K} \to \mathcal{K}$ (which corresponds to the case where $n = \mathbf{succ}\,n'$). In our case, we want to return a proposition of the kind $\llbracket \tau \rrbracket$. The third argument (the inductive step case) should correspond to the encoding of e_2, but this would be of kind $\llbracket \mathbf{nat} \to \tau \to \tau \rrbracket = (\mathbf{nat} \to \mathrm{prop}) \to \llbracket \tau \rrbracket \to \llbracket \tau \rrbracket$. Thus we make a slightly modified ElimN$'$ construct which expects this kind of propositions and is easily translatable to the original ElimN construct.

Let's see how the addition function between natural numbers would be translated using this approach. As before, we define this function to be equal to:

$$\lambda x.\lambda y.\mathbf{elimnat}(x, y, \lambda x'.\lambda z'.\mathbf{succ}\,z')$$

By following the translation, we end up with the following predicate encoding this function:

$$\begin{array}{l} \lambda X : \mathbf{nat} \to \mathrm{prop}\,.\lambda Y : \mathbf{nat} \to \mathrm{prop}\,.\lambda r : \mathbf{nat}. \\ \quad \forall r_x : \mathbf{nat}.X\,r_x \Rightarrow \\ \quad (\mathrm{ElimN}'\ r_x\,Y\,(\lambda X'.\lambda Z.\lambda r'.\forall r_z.Z\,r_z \Rightarrow \mathbf{succ}\,r_z = r'))\,r \end{array}$$

Now we are ready to move on to see how we can create terms at the computational level that correspond to this translation. Ideally we want a computational term corresponding to a natural-number expression to give us the dataterm that is the canonical form of the expression, plus a proof that this dataterm is actually the normal form of the expression; and that it is the only such dataterm. Thus for an expression e of type **nat** we would want a computational-level term of the type:

$$\mathsf{D}[\mathbf{nat}, \lambda r : \mathbf{nat}.\,\llbracket e \rrbracket\ r \wedge \forall r' : \mathbf{nat}.\,\llbracket e \rrbracket\ r' \Rightarrow r = r']$$

For an expression e of functional type $\mathbf{nat} \to \mathbf{nat}$ we would like to generate a computational-level function that takes the encoding of the **nat** argument, the package of its canonical form with the associated proofs and returns a package of the canonical form of the result of the function for the given argument, plus proofs of canonicity and uniqueness. The type of the term we would want to

Actual translation

$$\begin{array}{rcl}
\mathsf{cltct}(\lambda x.e : \tau_1 \to \tau_2) & = & \lambda x : \llbracket \tau_1 \rrbracket_v . \lambda x_{tct} : \mathsf{cltct}_\tau(x : \llbracket \tau_1 \rrbracket_v).\mathsf{clpkg}(e : \tau_2) \\
\mathsf{cltct}(x : \tau_1 \to \tau_2) & = & x_{tct} \\
\mathsf{cltct}(v : \mathbf{nat}) & = & [\mathrm{True}] \\
\mathsf{clpkg}(v : \tau) & = & \quad \mathsf{pack}\ \llbracket v \rrbracket_v\ \mathsf{as}\ r\ \mathsf{in} \\
& & \quad [\mathrm{mkAnd}\,(\textit{proof of}\ \ \llbracket v \rrbracket_p\, \llbracket v \rrbracket_v \equiv (\mathrm{eq}\ \llbracket v \rrbracket_v)\ \llbracket v \rrbracket_v\ \textit{is}\ \mathrm{reflexivity}\ \llbracket v \rrbracket_v) \\
& & \qquad\qquad (\textit{proof of}\ \forall r'.\,\llbracket v \rrbracket_p\, r' \Rightarrow r = r' \equiv \forall r'.r = r' \Rightarrow r = r'\ \textit{is}\ \lambda r'.\lambda p.p)] * \mathsf{cltct}(v : \tau) \\
\mathsf{clpkg}(e_1\, e_2 : \tau) & = & \quad \mathsf{let}\,((r_1, X_{r_1} :: [\llbracket e_1 \rrbracket_p\, r_1 \wedge \forall r'_1.\,\llbracket e_1 \rrbracket_p\, r'_1 \Rightarrow r_1 = r'_1]), \\
& & \qquad f_{r_1} :: \forall y : \llbracket \tau' \rrbracket_v .\mathsf{cltct}_\tau(y : \llbracket \tau' \rrbracket_v) \to \exists r : \llbracket \tau \rrbracket_v .[r_1\, y\, r \wedge \forall r'.r_1\, y\, r' \Rightarrow r = r'] * \mathsf{cltct}_\tau(r : \llbracket \tau \rrbracket_v)) = \\
& & \qquad \mathsf{clpkg}(e_1)\ \mathsf{in} \\
& & \quad \mathsf{let}\,((r_2, X_{r_2} :: [\llbracket e_2 \rrbracket_p\, r_2 \wedge \forall r'_2.\,\llbracket e_2 \rrbracket_p\, r'_2 \Rightarrow r_2 = r'_2]), \\
& & \qquad f_{r_2} :: \mathsf{cltct}_\tau(r_2 : \llbracket \tau' \rrbracket_v)) = \mathsf{clpkg}(e_2)\ \mathsf{in} \\
& & \quad \mathsf{let}\,((r, X_r :: [r_1\, r_2\, r \wedge \forall r'.r_1\, r_2\, r' \Rightarrow r = r']), f_r :: \mathsf{cltct}_\tau(r : \llbracket \tau \rrbracket_v)) = f_{r_1}\, r_2\, f_{r_2}\ \mathsf{in} \\
& & \quad \mathsf{pack}\ r\ \mathsf{as}\ r\ \mathsf{in} \\
& & \quad [\mathrm{mkAnd}\,(\textit{proof of}\ \ \llbracket e_1\, e_2 \rrbracket_p\, r \equiv \forall r_1 : \llbracket \tau \to \tau' \rrbracket_v .\,\llbracket e_1 \rrbracket_p\, r_1 \Rightarrow \forall r_2 : \llbracket \tau \rrbracket_v .\,\llbracket e_2 \rrbracket_p\, r_2 \Rightarrow r_1\, r_2\, r\ \textit{is} \\
& & \qquad\qquad \lambda r'_1.\lambda p_1.\lambda r'_2.\lambda p_2.\textit{proof of}\ r'_1\, r'_2\, r\ \textit{is}\ \mathrm{useEq}\ r_1\ (\lambda r''_1.r''_1\, r'_2\, r) \\
& & \qquad\qquad\quad (\textit{proof of}\ r_1\, r'_2\, r\ \textit{is}\ \mathrm{useEq}\ r_2\ (\lambda r''_2.r_1\, r''_2\, r)\ (\mathrm{left}\ X_r)\ r'_2\ (\mathrm{right}\ X_{r_2}\, r'_2\, p_2)) \\
& & \qquad\qquad r'_1\ (\mathrm{right}\ X_{r_1}\, r'_1\, p_1)) \\
& & \qquad\qquad (\textit{proof of}\ \forall r'.(\forall r_1 : \llbracket \tau \to \tau' \rrbracket_v .\,\llbracket e_1 \rrbracket_p\, r_1 \Rightarrow \forall r_2 : \llbracket \tau \rrbracket_v .\,\llbracket e_2 \rrbracket_p\, r_2 \Rightarrow r_1\, r_2\, r') \Rightarrow r = r'\ \textit{is} \\
& & \qquad\qquad \lambda r'.\lambda p.\,\mathrm{right}\ X_r\, r'\ (\textit{proof of}\ r_1\, r_2\, r'\ \textit{is}\ p\, r_1\, X_{r_1}\, r_2\, X_{r_2}))] * f_r \\
\mathsf{clpkg}(\mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\, v : \tau) & = & \quad (\mathsf{fix}\, f :\ (\forall y : \mathbf{nat}.\exists r.[\llbracket \mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\, y \rrbracket_p\, r \wedge \forall r'.\,\llbracket \mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\, y \rrbracket_p\, r' \Rightarrow r = r'] \\
& & \qquad\quad * \mathsf{cltct}_\tau(r : \llbracket \tau \rrbracket_v)). \\
& & \quad \lambda y : \mathbf{nat}.\mathsf{caseN}(y, \\
& & \qquad\quad \mathsf{let}\ ((r, X_r :: [\llbracket e_0 \rrbracket_p\, r \wedge \forall r'.\,\llbracket e_0 \rrbracket_p\, r' \Rightarrow r = r']), f_r)\ =\ \mathsf{clpkg}(e_0)\ \mathsf{in}\, \mathsf{pack}\ r\ \mathsf{as}\ r\ \mathsf{in} \\
& & \qquad\qquad [\mathrm{mkAnd}\,(\textit{proof of}\ \ \llbracket \mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\, 0 \rrbracket_p\, r \equiv \mathrm{ENPred}\, 0\, r\ \textit{is}\ \ \mathrm{ctorRel}_{\mathrm{ENPred}}\ 0\, r\, (\mathrm{left}\ X_r)) \\
& & \qquad\qquad\qquad (\textit{proof of}\ \forall r'.\,\llbracket \mathbf{elimnat}\{e_0 \,|\, x'.z'.e_s\}\, 0 \rrbracket_p\, r' \Rightarrow r = r'\ \textit{is}\ \lambda r'.\lambda p. \\
& & \qquad\qquad\qquad\quad \mathrm{useENPred}\,(\lambda n.\lambda r'.\,\mathrm{ElimN}\ n\, (r = r')\, (\lambda x.\lambda p.\top)) \\
& & \qquad\qquad\qquad\quad (\textit{proof of}\ \forall r_0.\,\llbracket e_0 \rrbracket_p\, r_0 \Rightarrow r = r_0\ \textit{is}\ \lambda r_0.\lambda p_0.\,\mathrm{right}\ X_r\, r_0\, p_0) \\
& & \qquad\qquad\qquad\quad (\lambda x'.\lambda z'.\lambda p_1.\lambda r'.\lambda p_2.\,\mathrm{True})\, 0\, r'\, p)] * f_r, \\
& & \qquad\quad \lambda x'.\mathsf{let}\ ((z', X_{z'} :: [\mathrm{ENPred}\, x'\, z' \wedge \forall r'.\,\mathrm{ENPred}\, x'\, r' \Rightarrow z' = r']), z'_{tct})\ =\ f\, x'\ \mathsf{in} \\
& & \qquad\quad \mathsf{let}\ ((r, X_r :: [\llbracket e_s \rrbracket_p\, r \wedge \forall r'.\,\llbracket e_s \rrbracket_p\, r' \Rightarrow r = r']), f_r)\ =\ \mathsf{clpkg}(e_s)\ \mathsf{in}\, \mathsf{pack}\ r\ \mathsf{as}\ r\ \mathsf{in} \\
& & \qquad\qquad [\mathrm{mkAnd}\,(\textit{proof of}\ \ \mathrm{ENPred}\,(\mathbf{succ}\, x')\, r\ \textit{is}\ \ \mathrm{ctorRel}_{\mathrm{ENPred}}\ 1\, x'\, z'\, (\mathrm{left}\ X_{z'})\, r\, (\mathrm{left} X_r)) \\
& & \qquad\qquad\qquad (\textit{proof of}\ \forall r'.\,\mathrm{ENPred}\,(\mathbf{succ}\, x')\, r' \Rightarrow r = r'\ \textit{is}\ \lambda r'.\lambda p. \\
& & \qquad\qquad\qquad\quad \mathrm{useENPred}\,(\lambda n.\lambda r'.\,\mathrm{ElimN}\ n\, \top\, (\lambda x''.\lambda _.x' = x'' \Rightarrow r = r'))\ \mathrm{True} \\
& & \qquad\qquad\qquad\quad (\lambda x''.\lambda z''.\lambda p_0 : \mathrm{ENPred}\, x''\, z''.\lambda _.\lambda r''.\lambda p_1 : \llbracket e_s \rrbracket_p\, [x'', z''/x', z']\, r''.\lambda p_2 : x' = x''. \\
& & \qquad\qquad\qquad\quad \mathsf{let}\, p_3\ ::\ z' = z''\ = \mathrm{right}\ X_{z'}\, z''\, (\mathrm{useEq}\ x''\, (\lambda x.\,\mathrm{ENPred}\, x\, z'')\, p_0\, x'\, (\mathrm{symmEq}\ x''))\ \mathsf{in} \\
& & \qquad\qquad\qquad\quad \mathsf{let}\, p_4\ ::\ \llbracket e_s \rrbracket_p\, [x''/x']\, r''\ = \mathrm{useEq}\ z''\, (\lambda z'.\,\llbracket e_s \rrbracket_p\, r'')\, p_1\, z'\, (\mathrm{symmEq}\ p_3)\ \mathsf{in} \\
& & \qquad\qquad\qquad\quad \mathsf{let}\, p_5\ ::\ \llbracket e_s \rrbracket_p\, r''\ = \mathrm{useEq}\ x''\, (\lambda x'.\,\llbracket e_s \rrbracket_p\, r'')\, p_4\, x'\, (\mathrm{symmEq}\ p_2)\ \mathsf{in} \\
& & \qquad\qquad\qquad\quad \mathrm{right}\ X_r\, r''\, p_4) \\
& & \qquad\qquad\qquad\quad (\mathbf{succ}\, x')\, r'\, p\, (\mathrm{reflexivity}\ x'))] * f_r))\ \llbracket v \rrbracket_v
\end{array}$$

Figure 19. Converting an expression to its canonical form (continued)

generate will thus look like:

$$\forall y : \mathbf{nat} \to \mathrm{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r : \mathbf{nat}.\,\llbracket y \rrbracket\, r \wedge \forall r' : \mathbf{nat}.\,\llbracket y \rrbracket\, r' \Rightarrow r = r'] \to$$
$$\mathsf{D}[\mathbf{nat}, \lambda r : \mathbf{nat}.\,\llbracket e\, y \rrbracket\, r \wedge \forall r' : \mathbf{nat}.\,\llbracket e\, y \rrbracket\, r' \Rightarrow r = r']$$

In the general case, we denote the type of the computational term corresponding to the expression $e : \tau$ as $\mathsf{cltype}(e : \tau)$. The translation into actual computational terms is denoted as $\mathsf{clterm}(e)$. Both are defined in the figure 21. This is a complicated translation because of the fact that we need to prove uniqueness of canonical forms for arbitrary expressions; a number of proofs in our logical framework will be used in the process. The terms in our logical framework this translation depends upon are gathered in the same figure. Note that this translation only specifies how to translate the **elimnat** construct when it returns a **nat** type – but the given translation is extended to other return types in a straightforward manner.

We use various forms of syntactic sugar to make the presentation easier to understand. For example, we use the redundant annotation *proof of P is* $\cdots$ to specify the proposition that a proof term that follows proves. Also, we use reduntant type annotations at various places (denoted as $e :: \tau$) so it is easier to see that terms are well typed.

How do we prove that this translation is valid? Again we will have to show the following lemma:

Lemma 5.3 *The* $\mathsf{clterm}(e : \tau)$ *terms are well-typed for any expression e, and have the type* $\mathsf{cltype}(e : \tau)$.

The lemma follows by inspection of the translation, taking the typing rules into account.

Now we can prove a stronger guarantee for this relation, by leveraging the proof-erased version of our language that we have previously seen. We will show that the value that the terms will compute will essentially be the canonical form of the expression in the source language. This is what the following lemma states:

Types returned

$$[\![\mathbf{nat}]\!] = \mathbf{nat} \to \text{prop}$$

$$[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]$$

Actual translation

$$[\![x]\!] = x$$

$$[\![\mathbf{zero}]\!] = (\text{eq}\ \mathbf{zero})$$

$$[\![\mathbf{succ}\, e]\!] = \lambda r : \mathbf{nat}.\forall r' : \mathbf{nat}.\ [\![e]\!]\ r' \Rightarrow \mathbf{succ}\, r' = r$$

$$[\![\lambda x : \tau.e]\!] = \lambda x : [\![\tau]\!]\ .\ [\![e]\!]$$

$$[\![e_1\, e_2]\!] = [\![e_1]\!]\ [\![e_2]\!]$$

$$[\![\mathbf{elimnat}(e, e_1, e_2) : \mathbf{nat}]\!] = \lambda r : \mathbf{nat}.\forall n : \mathbf{nat}.\ [\![e]\!]\ n \Rightarrow \\ (\text{ElimN}'\ n\ [\![e_1]\!]\ [\![e_2]\!])\ r$$

$$[\![\mathbf{elimnat}(e, e_1, e_2) : \mathbf{nat} \to \mathbf{nat}]\!] = \\ \lambda x_1 : \mathbf{nat} \to \text{prop}\,.\lambda r : \mathbf{nat}.\forall n : \mathbf{nat}.\ [\![e]\!]\ n \Rightarrow \\ (\text{ElimN}'\ n\ [\![e_1]\!]\ [\![e_2]\!])\ x_1\ r$$

Syntactic sugar

$$\text{ElimN}'\ (\text{w} : \mathbf{nat})\ (P_0 : \mathcal{K})\ (P_S : (\mathbf{nat} \to \text{prop}) \to \mathcal{K} \to \mathcal{K}) = \\ \text{ElimN}\ \text{w}\, P_0\ (\lambda x : \mathbf{nat}.P_S\ (\text{eq}\ x))$$

Figure 20. Alternate translation of expressions into predicates

Lemma 5.4 *For every expression e : **nat** in the source language, where $e \longrightarrow^*_{\beta\iota} v$, we have that $\mathsf{clterm}(e : \mathbf{nat}) \longrightarrow^* v$ under the semantics of our computational language.*

This follows by inspection of the proof-erased version of the $\mathsf{clterm}(\cdot)$ translation, which is presented in figure 22, and observing that each step taken in the step relation of the source language is accurately reflected in one step of the step relation of our computational language.

5.3 A complete example

In this section we will see a completely worked out example of how to use the above translation to create proofs of interesting propositions. We will consider the primality testing problem. We want to create a computational-level function, that given any natural number n, checks whether it is prime or not, and returns a proof of this fact if it is prime.

To prove that n is prime, it is enough to show that all numbers greater than 1 and less than n are not divisors of n. It is equivalent to show that their greatest common divisor with n is 1. So our computational level function would go about constructing such a proof as follows: it would iterate over all natural numbers between 1 and n, and calculate the GCD of each number i with n. If the GCD is 1, then it proceeds with the proof, otherwise it fails. The calculation of the GCD should happen using the above translations, so that we get a proof term that witnesses that the function actually has this result.

Let's go into more detail as to how we can implement this. First, suppose that we have an implementation of the GCD function in the source language of our translations – we can certainly do this since GCD is a total recursive function. We denote this function as $\mathbf{gcd}\, x\, y$. We will now use the second translation that we defined above to get a one-place predicate which is inhabited by the normal form of the expression $\mathbf{gcd}\, x\, y$. We denote this predicate as $[\![\mathbf{gcd}\, x\, y]\!]$. The choice of the second translation is arbitrary – we could have used the first translation of section 5.1, but the second translation is somewhat easier to work with.

Using the same translation we can get a computational level function, which we call $\mathsf{gcdTactic}$, that given two arguments x and y gives us the normal form r of the expression $\mathbf{gcd}\, x\, y$ and also a proof that r is the only term that inhabits the predicate $[\![\mathbf{gcd}\, x\, y]\!]$. This term will have the following type:

$$\begin{aligned} \mathsf{gcdTactic} :: &\forall X : \mathbf{nat} \to \text{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r.X\, r \wedge \forall r'.X\, r' \Rightarrow r = r'] \\ &\forall Y : \mathbf{nat} \to \text{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r.Y\, r \wedge \forall r'.Y\, r' \Rightarrow r = r'] \\ &\mathsf{D}[\mathbf{nat}, \lambda r.\ [\![\mathbf{gcd}\, X\, Y]\!]\ r \wedge \forall r'.\ [\![\mathbf{gcd}\, X\, Y]\!]\ r' \Rightarrow r = r'] \end{aligned}$$

We can easily create a wrapper for this function that expects natural numbers for the arguments x and y and is therefore easier to work with. We call this term $\mathsf{gcdTactic}'$, and it will have the following type:

$$\mathsf{gcdTactic}' \quad :: \quad \forall x : \mathbf{nat}.\forall y : nat.\mathsf{D}[\mathbf{nat},\ \lambda r.\ [\![\mathbf{gcd}]\!]\ (\text{eq}\ x)\ (\text{eq}\ y)\ r \wedge \\ \forall r'.\ [\![\mathbf{gcd}]\!]\ (\text{eq}\ x)\ (\text{eq}\ y)\ r' \Rightarrow r = r']$$

Now let's consider how to encode the fact that a number n is prime in our propositional language. Essentially, we want to say that for all i that are greater than 1 and less than n, the GCD of i and n is 1. In CiC, we would write this as:

$$\text{prime}\ n \triangleq \forall i : \mathbf{nat}.i > 1 \Rightarrow i < n \Rightarrow \mathbf{gcd}\, i\, n = 1$$

In our framework, it is apparent from the previous section that the same proposition would be written as:

$$\text{prime}\ n \triangleq \forall i : \mathbf{nat}.i > 1 \Rightarrow i < n \Rightarrow \forall r : \mathbf{nat}.\ [\![\mathbf{gcd}]\!]\ (\text{eq}\ i)\ (\text{eq}\ n)\ r \Rightarrow 1 = r$$

What we ultimately want to arrive at is therefore a computational level function that creates a proof term for the proposition prime n for any natural number n. Of course this function might fail if n is not a prime number. This function will use the $\mathsf{gcdTactic}'$ function in the process of checking whether the GCD of two numbers is equal to 1.

We find that it is easier to define a special predicate to encode bounded quantification instead of using unbounded quantification and requiring proofs that the quantified variable lies between certain numbers. We call this predicate BoundedForall, and use the syntactic sugar $\forall i \in (1, n).P\, i$ to denote the term BoundedForall $P\, n$. We define this predicate as an inductive relation, with two axioms: first, that for any P, $\forall i \in (1, 2).P\, i$ holds, and second, that if $\forall i \in (1, n).P\, i$ holds and we have a proof of $P\, n$, then $\forall i \in (1, \mathbf{succ}\, n).P\, i$ also holds. This definition is written as follows in our language:

$$\begin{aligned} &\text{BoundedForall} : (\mathbf{nat} \to \text{prop}) \to \mathbf{nat} \to \text{prop} = \\ &\lambda P : \mathbf{nat} \to \text{prop}\,. \\ &\text{IndRel}(\text{BoundedForall} : \mathbf{nat} \to \text{prop})\{ \\ &\quad \text{bfBase} :: \text{BoundedForall}\ 2 \\ &\quad \text{bfStep} :: \forall n : \mathbf{nat}.\ \text{BoundedForall}\ n \Rightarrow P\, n \Rightarrow \text{BoundedForall}\, (\mathbf{succ}\, n) \\ &\} \end{aligned}$$

Now we can define the predicate about a number being prime using this bounded quantification:

$$\text{prime}\ n \triangleq \forall i \in (1, n).\forall r : \mathbf{nat}.\ [\![\mathbf{gcd}]\!]\ (\text{eq}\ i)\ (\text{eq}\ n)\ r \Rightarrow 1 = r$$

The computational level function that we ultimately want to create should therefore have the type:

$$\mathsf{primeTactic} :: \forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \\ \forall i \in (1, n).\forall r : \mathbf{nat}.\ [\![\mathbf{gcd}]\!]\ (\text{eq}\ i)\ (\text{eq}\ n)\ r \Rightarrow 1 = r]$$

How can we implement a function like this? We could iterate over all i in the $(1, n)$ range, and for each one we could use our $\mathsf{gcdTactic}'$ function to get the result of $\mathbf{gcd}\, i\, n$. We then need to

Types returned

$$\begin{array}{c}
\mathsf{cltype}(e:\mathbf{nat}) = \mathsf{D}[\mathbf{nat}, \lambda r:\mathbf{nat}.\, \llbracket e \rrbracket\; r \wedge \forall r':\mathbf{nat}.\, \llbracket e \rrbracket\; r' \Rightarrow r = r'] \\
\mathsf{cltype}(e:\tau_1 \to \tau_2) = \forall y:\tau_1 . \mathsf{cltype}(y:\tau_1) \to \mathsf{cltype}(e\,y:\tau_2)
\end{array}$$

Logical-level terms

$$\begin{array}{rcllcl}
\mathrm{eq} & : & \mathbf{nat} \to \mathbf{nat} \to \mathrm{prop} & & = & \lambda x:\mathbf{nat}.\, \mathrm{IndRel}(U:\mathbf{nat} \to \mathrm{prop})\{U\,x\} \\
\mathrm{useEq} & : & \forall x:\mathbf{nat}. \forall P:\mathbf{nat} \to \mathrm{prop}\,.P\,x \Rightarrow \forall y:\mathbf{nat}. x = y \Rightarrow P\,y & & = & \lambda x.\, \mathrm{elimRel}_{\mathrm{eq}\,x} \\
\mathrm{reflexivity}\,x & : & x = x & & = & \mathrm{ctorRel}_{\mathrm{eq}\,x}\, 0 \\
\mathrm{succEq} & : & x = y \Rightarrow \mathbf{succ}\,x = \mathbf{succ}\,y & & = & \mathrm{useEq}\,x\,(\mathrm{eq}\,(\mathbf{succ}\,x))\,(\mathrm{reflexivity}\,(\mathbf{succ}\,x))\,y \\
P_1 \wedge P_2 & : & \mathrm{prop} & & = & \mathrm{IndRel}(U:\mathrm{prop})\{P_1 \Rightarrow P_2 \Rightarrow U\} \\
\mathrm{useAnd} & : & \forall P:\mathrm{prop}\,.(P_1 \Rightarrow P_2 \Rightarrow P) \Rightarrow P_1 \wedge P_2 \Rightarrow P & & = & \mathrm{elimRel}_{P_1 \wedge P_2} \\
\mathrm{mkAnd} & : & P_1 \Rightarrow P_2 \Rightarrow P_1 \wedge P_2 & & = & \mathrm{ctorRel}_{P_1 \wedge P_2}\, 0 \\
\mathrm{left} & : & P_1 \wedge P_2 \Rightarrow P_1 & & = & \mathrm{useAnd}\, P_1\, (\lambda X_1 . \lambda X_2 . X_1) \\
\mathrm{right} & : & P_1 \wedge P_2 \Rightarrow P_2 & & = & \mathrm{useAnd}\, P_2\, (\lambda X_1 . \lambda X_2 . X_2)
\end{array}$$

Actual translation

$$\begin{array}{rcl}
\mathsf{clterm}(\mathbf{zero}) & = & \langle \mathbf{zero}, \mathrm{mkAnd}\,(\textit{proof of}\;\; \llbracket \mathbf{zero} \rrbracket\; \mathbf{zero} =_\beta (\mathrm{eq}\,\mathbf{zero})\, \mathbf{zero}\; \textit{is}\;\; \mathrm{reflexivity}\,x) \\
 & & \qquad\qquad\qquad (\textit{proof of}\; \forall r':\mathbf{nat}.\mathbf{zero} = r' \Rightarrow \mathbf{zero} = r'\; \textit{is}\; \lambda r'.\lambda p.p)\rangle \\
 & & \\
\mathsf{clterm}(\mathbf{succ}\,e) & = & \mathsf{let}\;(r, p_r :: \llbracket e \rrbracket\; r \wedge \forall r':\mathbf{nat}.\, \llbracket e \rrbracket\; r' \Rightarrow r = r') \;=\; \mathsf{clterm}(e)\;\mathsf{in} \\
 & & \langle \mathbf{succ}\,r, \mathrm{mkAnd}\,(\textit{proof of}\;\; \llbracket \mathbf{succ}\,e \rrbracket\; (\mathbf{succ}\,r) =_\beta \forall r':\mathbf{nat}.\, \llbracket e \rrbracket\; r' \Rightarrow \mathbf{succ}\,r = \mathbf{succ}\,r'\; \textit{is} \\
 & & \qquad\qquad\qquad\qquad \lambda r'.\lambda p_{r'} .\, \mathrm{succEq}(\textit{proof of}\;\; r = r'\; \textit{is}\;\; \mathrm{right}\; p_r\, r'\, p_{r'})) \\
 & & \qquad\qquad (\textit{proof of}\; \forall r'':\mathbf{nat}.(\forall r':\mathbf{nat}.\, \llbracket e \rrbracket\; r' \Rightarrow r'' = \mathbf{succ}\,r') \Rightarrow r'' = \mathbf{succ}\,r\; \textit{is} \\
 & & \qquad\qquad\qquad\qquad \lambda r''.\lambda p:\forall r':\mathbf{nat}.\, \llbracket e \rrbracket\; r' \Rightarrow r'' = \mathbf{succ}\,r' . p\, r\, (\mathrm{left}\; p_r))\rangle
\end{array}$$

$$\begin{array}{rcl}
\mathsf{clterm}(\lambda x:\tau.e) & = & \lambda x: \llbracket \tau \rrbracket\, . \lambda x_{tct} : \mathsf{cltype}(x:\tau).\mathsf{clterm}(e) \\
\mathsf{clterm}(x) & = & (x_{tct} :: \mathsf{cltype}(x:\tau)) \\
\mathsf{clterm}(e_1\, e_2) & = & \mathsf{clterm}(e_1)\; \llbracket e_2 \rrbracket\; \mathsf{clterm}(e_2) \\
\mathsf{clterm}(\mathbf{elimnat}(e, e_1, e_2) : \mathit{nat}) & = & \mathsf{let}\;(r_n, p_{r_n} :: \llbracket e \rrbracket\; r_n \wedge \forall r':\mathbf{nat}.\, \llbracket e \rrbracket\; r' \Rightarrow r_n = r') \;=\; \mathsf{clterm}(e)\;\mathsf{in} \\
 & & \mathsf{let}\; f \;=\; \mathsf{fix}\, f : \forall y:\mathbf{nat}.\mathsf{D}[\mathbf{nat}, \lambda r.(\mathrm{ElimN}'\; y\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r \wedge \forall r'.(\mathrm{ElimN}'\; y\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r' \Rightarrow r = r']. \\
 & & \qquad\qquad \lambda y:\mathbf{nat}.\mathsf{caseN}(y, \\
 & & \\
 & & \qquad\qquad\quad \mathsf{clterm}(e_1) \\
 & & \qquad\qquad\qquad :: (\mathsf{D}[\mathbf{nat}, \lambda r.\, \llbracket e_1 \rrbracket\; r \wedge \forall r'.\, \llbracket e_1 \rrbracket\; r' \Rightarrow r = r'] =_{\beta\iota} \\
 & & \qquad\qquad\qquad\quad \mathsf{D}[\mathbf{nat}, \lambda r.(\mathrm{ElimN}'\; 0\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r \wedge \forall r'.(\mathrm{ElimN}'\; 0\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r' \Rightarrow r = r']), \\
 & & \\
 & & \qquad\qquad\quad \lambda y':\mathbf{nat}. \\
 & & \qquad\qquad\quad ((\mathsf{clterm}(e_2) \\
 & & \qquad\qquad\qquad :: \forall x:\mathbf{nat} \to \mathrm{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r.x\, r \wedge \forall r':\mathbf{nat}.\, \llbracket x \rrbracket\; r' \Rightarrow r = r'] \to \\
 & & \qquad\qquad\qquad\quad \forall y:\mathbf{nat} \to \mathrm{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r.y\, r \wedge \forall r':\mathbf{nat}.\, \llbracket y \rrbracket\; r' \Rightarrow r = r'] \to \\
 & & \qquad\qquad\qquad\quad \mathsf{D}[\mathbf{nat}, \lambda r.\, \llbracket e_2 \rrbracket\; x\, y\, r \wedge \forall r':\mathbf{nat}.\, \llbracket e_2\, x\, y \rrbracket\; r' \Rightarrow r = r']) \\
 & & \qquad\qquad\quad (\mathrm{eq}\; y')\, \langle y', \mathrm{mkAnd}\,(\mathrm{reflexivity}\, y')\, (\lambda r'.\lambda p : (\mathrm{eq}\; y')\, r'.p)\rangle \\
 & & \qquad\qquad\quad (\mathrm{ElimN}'\; y'\; \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, (f\, y')) \\
 & & \qquad\qquad\qquad :: \mathsf{D}[\mathbf{nat}, \lambda r.\, \llbracket e_2 \rrbracket\; (\mathrm{eq}\; y')\, (\mathrm{ElimN}'\; y'\; \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r \wedge \cdots] \\
 & & \qquad\qquad\qquad \mathrm{but}\;\; \llbracket e_2 \rrbracket\; (\mathrm{eq}\; y')\, (\mathrm{ElimN}'\; y'\; \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket) =_\beta \\
 & & \qquad\qquad\qquad\quad (\lambda x'.\, \llbracket e_2 \rrbracket\; (\mathrm{eq}\; x'))\, y'\, (\mathrm{ElimN}'\; y'\; \llbracket e_2 \rrbracket\; \llbracket e_2 \rrbracket)) =_\iota \\
 & & \qquad\qquad\qquad\quad \mathrm{ElimN}\,(\mathbf{succ}\, y')\; \llbracket e_1 \rrbracket\; (\lambda x'.\, \llbracket e_2 \rrbracket\; (\mathrm{eq}\; x')) = \\
 & & \qquad\qquad\qquad\quad \mathrm{ElimN}'\,(\mathbf{succ}\, y')\; \llbracket e_2 \rrbracket\; \llbracket e_2 \rrbracket)\;\mathsf{in} \\
 & & \\
 & & \mathsf{let}\;(r^*, p_{r^*} :: (\mathrm{ElimN}'\; r_n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket) r^* \wedge \forall r':\mathbf{nat}.(\mathrm{ElimN}'\; r_n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r' \Rightarrow r^* = r') \;=\; f\, r_n\;\mathsf{in} \\
 & & \langle r^*, \mathrm{mkAnd} \\
 & & \qquad\quad (\textit{proof of}\;\; \llbracket \mathbf{elimnat}(e, e_1, e_2) \rrbracket\; r^* =_\beta \forall n:\mathbf{nat}.\, \llbracket e \rrbracket\; n \Rightarrow (\mathrm{ElimN}'\; n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r^*\; \textit{is} \\
 & & \qquad\quad \lambda n.\lambda p_n : \llbracket e \rrbracket\; n.\, \mathrm{useEq}\; r_n\, (\lambda z.(\mathrm{ElimN}'\; z\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r^*) \\
 & & \qquad\qquad\qquad\quad (\textit{proof of}\;\; (\mathrm{ElimN}'\; r_n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r^*\; \textit{is}\; (\mathrm{left}\; p_{r^*})) \\
 & & \qquad\qquad\qquad\quad (\textit{proof of}\;\; r_n = n\; \textit{is}\; (\mathrm{right}\; p_{r_n}\, n\, p_n))) \\
 & & \\
 & & \qquad\quad (\textit{proof of}\; \forall r^\bullet:\mathbf{nat}.\, \llbracket \mathbf{elimnat}(e, e_1, e_2) \rrbracket\; r^\bullet \Rightarrow r^* = r^\bullet\; \textit{is} \\
 & & \qquad\quad \textit{proof of}\; \forall r^\bullet:\mathbf{nat}.(\forall n:\mathbf{nat}.\, \llbracket e \rrbracket\; n \Rightarrow (\mathrm{ElimN}'\; n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r^\bullet) \Rightarrow r^* = r^\bullet\; \textit{is} \\
 & & \qquad\quad \lambda r^\bullet.\lambda p : (\forall n:\mathbf{nat}.\, \llbracket e \rrbracket\; n \Rightarrow (\mathrm{ElimN}'\; n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r^\bullet). \\
 & & \qquad\qquad\quad \mathrm{right}\; p_{r^*}\, r^\bullet\, (\textit{proof of}\;\; (\mathrm{ElimN}'\; r_n\, \llbracket e_1 \rrbracket\; \llbracket e_2 \rrbracket)\, r^\bullet\; \textit{is} \\
 & & \qquad\qquad\qquad\qquad\quad p\, r_n\, (\mathrm{left}\; p_{r_n})) \\
 & & \qquad\quad))\rangle
\end{array}$$

Figure 21. Alternate translation of expressions into computational terms

$$
\begin{array}{rcl}
\mathsf{clterm}(\mathbf{zero}) & = & \langle \mathbf{zero}, \cdot \rangle \\[1em]
\mathsf{clterm}(\mathbf{succ}\, e) & = & \ \mathsf{let}\ (r, \cdot) = \mathsf{clterm}(e)\ \mathsf{in} \\
& & \ \langle \mathbf{succ}\, r, \cdot \rangle \\[1em]
\mathsf{clterm}(\lambda x : \tau . e) & = & \lambda x_{tct} : \mathsf{cltype}(x : \tau).\mathsf{clterm}(e) \\
\mathsf{clterm}(x) & = & x_{tct} \\
\mathsf{clterm}(e_1\, e_2) & = & \mathsf{clterm}(e_1)\, \mathsf{clterm}(e_2) \\
\mathsf{clterm}(\mathbf{elimnat}(e, e_1, e_2) : nat) & = & \ \mathsf{let}\ (r_n, \cdot) = \mathsf{clterm}(e)\ \mathsf{in} \\
& & \ \mathsf{let}\ f = \mathsf{fix}\, f : \forall y : \mathbf{nat}.\mathsf{D}[\mathbf{nat}, \cdot]. \\
& & \qquad\qquad \lambda y : \mathbf{nat}.\mathsf{caseN}(y, \\
& & \qquad\qquad\quad \mathsf{clterm}(e_1), \\
& & \qquad\qquad\quad \lambda y' : \mathbf{nat}. \\
& & \qquad\qquad\quad \mathsf{clterm}(e_2)\, \langle y', \cdot \rangle\, (f\, y'))\ \mathsf{in} \\[0.5em]
& & \ \mathsf{let}\ (r^*, \cdot) = f\, r_n\ \mathsf{in} \\
& & \ \langle r^*, \cdot \rangle
\end{array}
$$

Figure 22. Proof-erased version of the $\mathsf{clterm}(\cdot)$ translation

compare this result with 1, and in the case where they are equal, we can obviously prove the $\forall r : \mathbf{nat}.\, [\![\mathbf{gcd}]\!]\ (\mathrm{eq}\ i)\, (\mathrm{eq}\ n)\, r \Rightarrow 1 = r$ proposition – since the right part of the proof term that the gcdTactic′ function gives us is exactly that proposition. If they are not equal, our overall primality test just fails (since we cannot construct a proof that the given number n is prime when in fact, it is not a prime). We can split this process in two computational level functions: a generic function to prove bounded-quantification propositions that goes over all numbers in a range, and a function that checks the result of the $\mathbf{gcd}\, x\, y$ function to see whether it is one, and returns a proof of $\forall r : \mathbf{nat}.\, [\![\mathbf{gcd}]\!]\ (\mathrm{eq}\ x)\, (\mathrm{eq}\ y)\, r \Rightarrow 1 = r$ if it is. These functions would therefore have the following types:

$$
\begin{array}{rcl}
\mathsf{bforallTactic} & :: & \forall P : \mathbf{nat} \to \mathrm{prop}\,.(\forall i : \mathbf{nat}.\mathsf{D}[\mathbf{1}, P\, i]) \to \\
& & \quad \forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \forall i \in (1, n).P\, i] \\
\mathsf{gcdIsOneCheck} & :: & \forall x : \mathbf{nat}.\forall y : \mathbf{nat}.\mathsf{D}[\mathbf{1},\, \forall r : \mathbf{nat}. \\
& & \quad [\![\mathbf{gcd}]\!]\ (\mathrm{eq}\ x)\, (\mathrm{eq}\ y)\, r \Rightarrow 1 = r]
\end{array}
$$

If we have these functions, the implementation of primeTactic is entirely straightforward:

$$
\begin{array}{l}
\mathsf{primeTactic} := \lambda n.\mathsf{bforallTactic} \\
\quad (\lambda i.\forall r : \mathbf{nat}.\, [\![\mathbf{gcd}]\!]\ (\mathrm{eq}\ i)\, (\mathrm{eq}\ n)\, r \Rightarrow 1 = r) \\
\quad (\lambda i.\mathsf{gcdIsOneCheck}\, i\, n)\, n
\end{array}
$$

Thus our work is complete. In figure 23 we present the implementations of the functions that we have mentioned. Note that we have made the assumption that we have a special construct called fail in our computational language that throws an error with type $\perp$. This construct is certainly easy to add to our computational language, even though we did not explicitly present it in the previous chapter.

Let us now consider what this example shows. We have shown that we can construct a function in our computational language that creates a proof of the primality of any prime number. We have done this even though initially our framework did not have a notion of total functions between dataterms. Also, considering the disscussion about proof-erasure semantics in the previous chapter, we can see that this implementation of primality testing can be made efficient, even though it provides an actual proof of its correct behavior. In order to do this, we would need to use proof-erasure semantics and redefine the gcdTactic function so that it uses the trusted primitive operations that we would add to our language to compute the GCD of two natural numbers. In that case, all computation involving proofs would not happen at runtime, and the actual runtime behavior of our primeTactic function would be identical to the same naïve primality testing algorithm implemented in a language like ML.

To better demonstrate this, we have provided the proof-erased version of the example in figure 24. In this, we assume that the gcdTactic function has been defined using trusted primitive operations. From this code we can see that the actual computation that is going on is what we would expect, without extra overhead: we loop from n-1 down to 2, checking whether the GCD of the index and the given number is 1; if this is true throughout the loop, then the primeTactic function succeeds, otherwise an error is produced (meaning that the given number was not prime).

How does this code compare to code that we would get through Coq proof extraction? In that case, natural numbers would be defined in ML as a normal datatype, using their unary representation, so all operations involving them would actually be very expensive. This would greatly hinder the runtime performance of the program, even though the rest of the code would look similar to what we have here.

6. Comparison

In this section we compare our language design with other relevant frameworks. We consider both proof assistants and dependently-typed languages, since our framework lies somewhere between these two systems.

Coq proof assistant Our framework is very much based upon the Coq proof assistant [8], as is evident from its presentation thus far. Still, the Coq proof assistant differs from our framework in several key points. First, its Set universe that corresponds to our dataterm language includes a total computational language, that facilitates encoding and proving predicates involving functions between dataterms. We have seen how to recover this functionality without large extra overhead. Second, a proof of a certain specification can be used to extract a program that performs the computational part of the proof. For example, having a proof that for any number n there exists either a proof of $\mathrm{isPrime}(n)$ or of $\mathrm{not}\,(\mathrm{isPrime}(n))$, we could extract a program that checks whether a number is prime or not. Even though this is an interesting feature, this program will be written in a total functional subset of ML, and will not be as efficient as a normal implementation. A basic source of inefficiency will be the use of inefficient representations for datatypes like natural numbers, which we have shown can be handled efficiently in our system. Also, we ultimately want to use our language to develop proofs about low-level systems code, and this focus makes clear the fact that

```
gcdTactic     ::  $\forall X : \mathbf{nat} \to \text{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r.X\,r \wedge \forall r'.X\,r' \Rightarrow r = r'] \to$
                  $\forall Y : \mathbf{nat} \to \text{prop}\,.\mathsf{D}[\mathbf{nat}, \lambda r.Y\,r \wedge \forall r'.Y\,r' \Rightarrow r = r'] \to$
                  $\mathsf{D}[\mathbf{nat}, \lambda r.\, [\![\mathbf{gcd}\,X\,Y]\!]\; r \wedge \forall r'.\, [\![\mathbf{gcd}\,X\,Y]\!]\; r' \Rightarrow r = r']$
              :=  $\mathsf{clterm}(\mathbf{gcd} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat})$

gcdTactic'    ::  $\forall x : \mathbf{nat}.\forall y : nat.\mathsf{D}[\mathbf{nat}, \lambda r.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r \wedge \forall r'.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r' \Rightarrow r = r']$
              :=  $\lambda x : \mathbf{nat}.\lambda y : \mathbf{nat}.$
                  $\mathsf{gcdTactic}\,(\text{eq}\,x)\,\langle x,\ \text{mkAnd}(\textit{proof of}\ \ (\text{eq}\,x)\,x\ \textit{is}\ \ \text{reflexivity}\,x)$
                                                  $(\textit{proof of}\ \forall r'.(\text{eq}\,x)\,r' \Rightarrow (\text{eq}\,x)\,r'\ \textit{is}\ \lambda r'.\lambda p.p)\rangle$
                        $(\text{eq}\,y)\,\langle y,\ \text{mkAnd}(\text{reflexivity}\,y)(\lambda r'.\lambda p.p)\rangle$

bforallTactic ::  $\forall P : \mathbf{nat} \to \text{prop}\,.(\forall i : \mathbf{nat}.\mathsf{D}[\mathbf{1}, P\,i]) \to \forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \forall i \in (1,n).P\,i]$
              :=  $\lambda P : \mathbf{nat} \to \text{prop}\,.\lambda g : (\forall i : \mathbf{nat}.\mathsf{D}[\mathbf{1}, P\,i]).\mathsf{fix}\,f : \forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \forall i \in (1,n).P\,i].\lambda n : \mathbf{nat}.$
                  $\mathsf{caseN}(n,$
                    $\mathsf{fail}$ (* if n=0, then we fail as $\forall i \in (1,0).P\,i$ is not provable *),
                    $\lambda n'.\mathsf{caseN}(n',$
                      $\mathsf{fail}$ (* if n'=0 $\Rightarrow$ n=1, then we fail as $\forall i \in (1,1).P\,i$ is not provable *),
                      $\lambda n''.\mathsf{caseN}(n'',$
                        $\langle \mathbf{unit}, \textit{proof of}\ \ (\forall i \in (1,2).P\,i)\ \textit{is}\ \ \text{bfBase}\,P\rangle,$
                        $\lambda n'''.\mathsf{let}\ (\_, X_1 :: \forall i \in (1, S\,(S\,(n'''))).P\,i)\ =\ f\,(S\,(S\,n'''))\ \mathsf{in}$
                              $\mathsf{let}\ (\_, X_2 :: P\,(S\,(S\,n''')))\ =\ g\,(S\,(S\,n'''))\ \mathsf{in}$
                              $\langle \mathbf{unit}, \textit{proof of}\ \ (\forall i \in (1, S\,(S\,(S\,n'''))).P\,i)\ \textit{is}\ \ \text{bfStep}\,P\,(S\,(S\,n'''))\,X_1\,X_2\rangle))))$

gcdIsOneCheck ::  $\forall x : \mathbf{nat}.\forall y : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \forall r : \mathbf{nat}.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r \Rightarrow 1 = r]$
              :=  $\lambda x : \mathbf{nat}.\lambda y : \mathbf{nat}.$
                  $\mathsf{let}\ (r, X_1 :: [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r \wedge \forall r'.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r' \Rightarrow r = r')\ =\ \mathsf{gcdTactic'}\,x\,y\ \mathsf{in}$
                  $\mathsf{let}\ (\_, X_2 :: r = 1)\ =\ \mathsf{caseN}(r, \mathsf{fail}, \lambda r'.\mathsf{caseN}(r', \langle \mathbf{unit}, \text{reflexivity}\ 1\rangle, \mathsf{fail}))\ \mathsf{in}$
                  $\langle \mathbf{unit}, \textit{proof of}\ \forall r'.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r' \Rightarrow 1 = r'\ \textit{is}$
                        $\text{useEq}\ r\,(\lambda r.\forall r'.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r' \Rightarrow r = r')$
                        $(\textit{proof of}\ \forall r'.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,x)\,(\text{eq}\,y)\,r' \Rightarrow r = r'\ \textit{is}\ \text{right}\,X_1)$
                        $1\,(\textit{proof of}\ \ r = 1\ \textit{is}\ X_2)\rangle$

primeTactic   ::  $\forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \forall i \in (1,n).\forall r : \mathbf{nat}.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,i)\,(\text{eq}\,n)\,r \Rightarrow 1 = r]$
                  (* which is equivalent to $\forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \text{prime}\,n]$ *)
              :=  $\lambda n.\mathsf{bforallTactic}$
                        $(\lambda i.\forall r : \mathbf{nat}.\, [\![\mathbf{gcd}]\!]\; (\text{eq}\,i)\,(\text{eq}\,n)\,r \Rightarrow 1 = r)$
                        $(\lambda i.\mathsf{gcdIsOneCheck}\,i\,n)\,n$
```

Figure 23. Example: Using the defined translation to prove primality of a number

```
   $[\![\mathsf{gcdTactic'}]\!]_{pe}$    :=  $\lambda x : \mathbf{nat}.\lambda y : \mathbf{nat}.\mathsf{gcdTactic}\,\langle x, \cdot\rangle\,\langle y, \cdot\rangle$

 $[\![\mathsf{bforallTactic}]\!]_{pe}$   :=  $\lambda g : (\forall i : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \cdot]).\mathsf{fix}\,f : \forall n : \mathbf{nat}.\mathsf{D}[\mathbf{1}, \cdot].\lambda n : \mathbf{nat}.$
                             $\mathsf{caseN}(n, \mathsf{fail}$
                               $\lambda n'.\mathsf{caseN}(n', \mathsf{fail},$
                                 $\lambda n''.\mathsf{caseN}(n'', \langle \mathbf{unit}, \cdot\rangle,$
                                   $\lambda n'''.\mathsf{let}\ (\_, \cdot)\ =\ f\,(S\,(S\,n'''))\ \mathsf{in}$
                                         $\mathsf{let}\ (\_, \cdot)\ =\ g\,(S\,(S\,n'''))\ \mathsf{in}$
                                         $\langle \mathbf{unit}, \cdot\rangle))))$

$[\![\mathsf{gcdIsOneCheck}]\!]_{pe}$   :=  $\lambda x : \mathbf{nat}.\lambda y : \mathbf{nat}.$
                             $\mathsf{let}\ (r, \cdot)\ =\ \mathsf{gcdTactic'}\,x\,y\ \mathsf{in}$
                             $\mathsf{let}\ (\_, \cdot)\ =\ \mathsf{caseN}(r, \mathsf{fail}, \lambda r'.\mathsf{caseN}(r', \langle \mathbf{unit}, \cdot\rangle, \mathsf{fail}))\ \mathsf{in}$
                             $\langle \mathbf{unit}, \cdot\rangle$

  $[\![\mathsf{primeTactic}]\!]_{pe}$   :=  $\lambda n.\mathsf{bforallTactic}\,(\lambda i.\mathsf{gcdIsOneCheck}\,i\,n)\,n$
```

Figure 24. Example: Proof-erased version of the primality testing example

the program-extraction feature is not very useful to us. Third, manipulation of proofs in Coq happens either using a fixed set of tactics, or by writing a new tactic in ML in an essentially untyped manner. A tactic may create a proof term that is actually not valid as a proof for the proposition it claims to prove; a proof-checker must ensure that this is not the case at runtime, after the tactic has been executed. Our language gives us the benefit of manipulating proof terms programmatically, while maintaining information about their type and guaranteeing their validity. This means that tactics that we write in our language are written in a type-safe manner and always construct valid proofs of the propositions they claim to prove. Fourth, the Calculus of Inductive Constructions has an extra universe hierarchy over the Prop and Set universes, the Type(n) hierarchy, which is inhabited by similar types and terms as the Set universe. In our setting, it is not clear whether this hierarchy is essentially needed. Last, the Coq proof assistant has a large library of tactics to aid users in the creation of proofs; we hope that with the proper extensions to our language we will eventually be able to support these tactics and more.

HOL The HOL system [6] is a general theorem proving environment based upon the LCF approach to theorem provers. HOL is based on a certain type of higher-order logic with a small set of axioms. These axioms are then implemented as constructors of an abstract datatype called *theorem* inside a general computational language (which, in the case of HOL, is Standard ML). This abstract datatype represents theorems that we have proven correct. Since the only way to construct terms of this datatype is through the use of the encodings of the axioms of the logic, a term that has this datatype has certainly been produced by combining these axioms – therefore, it corresponds to a valid proof derivation in the logic. HOL also has a set of tactics, which are functions in ML that help us derive proofs; these tactics are implemented again by combining the axiom functions so they always produce valid proofs. Proof development in HOL happens totally inside the computational language (ML), by combining axioms and tactics accordingly. In many ways, this is similar to how proof development happens in our framework: first, our computational language naturally takes precedence over our logical language (as the logical language is just a part of the computational language); second, we can create tactics in our computational language that are guaranteed to create valid proofs and combine them with other proofs from our logical language as we see fit. The big difference with HOL is that in HOL, the *theorem* type which corresponds to our proof-package type is essentially untyped, since it doesn't reflect the proposition that the *theorem* proves in its type. Essentially, we can see our framework as adding a kind of dependent-types especially for theorems, so that we can statically have guarantees about the propositions that they prove. We believe that this will make the development of tactics and large proofs more modular and reliable, just as a richly-typed programming language helps us develop large and reliable programs compared to an untyped language.

Isabelle Isabelle [12, 11] is a general framework that is used to define logics and develop proofs in them. It is based on a logical core that is very close to HOL, but which enables to define a large number of object logics on top of it. A large number of logics have been defined inside this core logic, like ZFC Set theory, first-order logic, and HOL, which is by far the logic that Isabelle is mostly used with. The implementation of the logical core follows the LCF approach, which we described above for the HOL theorem prover – so the same points that we raised there compared with our framework still hold for Isabelle. Also, the extra level of indirection that Isabelle add is not directly useful for our purposes. We have a fixed logic that we believe is enough to encode and prove the theorems that low-level system code verification entails, something evidenced by the large amount of this kind of work that has been done using the CiC logic of the Coq proof assistant (which is close to our own logic). Isabelle implements a large number of powerful tactics and decision procedures that facilitate the proof development process considerably. Some examples are a term rewriting engine and a tableaux prover – there has even been effort to embed a full SMT solver [4]. These decision procedures are implemented inside ML, in an untyped manner as we have seen so far, while proof development happens in a style similar to Coq, by writing proof scripts at a higher-level language. Our language permits the decision procedures and the proofs to be developed inside the same computational language, limiting the impedance that writing a new decision procedure currently has (since one has to switch to ML and learn about the internals of the proof assistant). At any case, our aim for future work is to implement similar decision procedures as the ones that Isabelle has in a type-safe manner inside our computational language.

LF The Logical Framework LF [7, 13] is a general framework for defining logics, based upon the idea that syntax, inference rules and proofs can uniformly be represented using higher-order abstract syntax, as terms of a dependently-typed lambda-calculus [7]. Using LF we can represent a large number of logics and programming languages, and coupled with a meta-logical tool like Twelf [14] we can prove meta-theory results about them. No notion of computation whatsoever is part of the LF framework – all terms of the framework are just representation forms. This was a clear influence for us in the initial steps of our language design, something which is reflected in our choice of having the dataterm universe be composed just of representations. As we have said, we want to explore in the future the possibility of extending this universe in order to support higher-order abstract syntax in the style of LF. This extension would be useful if we would want to encode languages with binding in our dataterm universe, in order to be able to reason about them. The problem with LF and Twelf is that proof terms have to be written out in full – there is no way to automate the construction of trivial or non-trivial parts of the proofs. We address this problem in the same way that tactic-based proof assistants like Coq and Isabelle address it – by having computational-level functions that automatically create parts of the proof terms in order to facilitate their overall construction.

Type System for Certified Binaries In [16], Shao *et al.*describe a general type-system to be used by typed intermediate and assembly languages that is able to represent complex propositions and proofs. This type-system heavily draws upon CiC as we do in the present work, and we can essentially think of it as CiC with a collapsed Set and Prop universe. As an example, the paper presents a typed intermediate language that admits types from this type system. This language is in fact a general computation language, that uses singleton types in order to reflect values of its terms into its types; in this way, we are able to specify interesting propositions about the results of functions inside the type system, and also have ways to prove them. Examples are shown that demonstrate the fact that the type system is powerful enough to create programs in the computational language that provide a full proof of their partial correctness. Though the use case of this work compared to our language design is very different, the two systems are very close. Still, our system differs in various important ways: first, our computational language doesn't need singleton types to reflect the value of terms inside the type system, since these terms will be normal dataterms that the type system can use without losing decidability. Second, we maintain separation between the Prop and Set universes, and by removing total computations from the Set universe, we are able to represent dataterms efficiently during runtime. The most important difference is that in Shao *et al.*'s work the logical framework exists so that terms of the computational language can be certified with regard to certain properties, in our case we do not care about certifying the behavior of our computational-level functions; instead, we use these functions in order to create proof terms. That is, our focus is on the construction of the proof terms instead of the certification of the behavior of our computational-level terms. Thus, our different focus might prompt us to add features to our computational language, like proposition reflection operations, that would not make sense in Shao *et al.*'s original setting.

Dependently-typed programming languages A large number of dependently-typed languages have been proposed in the programming languages literature, and there are many cases where the type systems of such languages are expressive enough so that they are comparable to a logical framework like CiC. Such examples are ATS [19] and Hoare Type Theory [10]. Our system is similar to such languages in many ways, as it combines manipulation of proofs with a general computational language. Still, our last point in the above comparison remains: our focus is on constructing proofs and not on certifying the behavior of programs in our computational language. This inevitably leads us to current and future design choices that would not make sense otherwise. For example, our language does not support any form of dependently-typed data structures at the computational level, like lists dependent on their length, and we do not believe that we will need to add something like this in the future. This is because this sort of dependency would be something needed to certify certain properties about terms of the computational language, something that we have no interest in.

7. Future directions

In this chapter we present a number of research directions that we would like to explore in the future in order to extend our framework.

– First, we would like to extend our dataterm universe so that it can encode a larger class of datatypes. At the very least, we need to support general inductive types instead of natural numbers and lists – an extension to this class of types seems straightforward. An interesting research objective would be to try to see whether this universe could be extended with arrow types that correspond to the parametric function space, essentially adding higher-order abstract syntax to our dataterm universe. This would permit us to encode a large number of languages with binding directly inside this universe in a very natural way. What is not clear is how exactly these terms would be reasoned about in our logical language and used in our computational language. For example, should we define an elimination principle for them in the propositional level, as we currently do for natural numbers, and if yes, what should that principle look like? Also, how would we programmatically construct and destruct such terms inside our computational languages? There has been significant amount of work in answering similar questions (see for example [3, 15]), but it is not clear how this would be used in our particular language design.

– Second, we would like to add some support in our computational language for branching operations depending on the specific forms of propositions we are trying to prove. We can think of this as adding some kind of proposition-reflection constructs in our language. This would mean that we could create generic functions that take an arbitrary proposition as an argument, decompose it as they see fit, and try to create proofs for some part of them, simplifying the proposition that is left to be proved. This is essentially what a tactic or a decision procedure does, so it seems that such a construct will be necessary in our language. The problems to be addressed would be to investigate what these proposition-reflection constructs should actually look like, and how they would influence the design of our type system.

– A third problem to address in the future is the fact that the translation functions that we have defined from terms in the CiC-style Set universe to terms in our language are defined at the meta-level and not inside our language. We believe that by adding proposition reflection constructs as we mentioned above, these translation functions would be closer to being implementable in our language. What is not clear is whether our type system can be extended in a way such that these functions are directly implementable, while not sacrificing decidability of type-checking.

– Last, we would like to investigate how to implement useful tactics and decision procedures similar to the ones present in existing systems inside our language. This effort would of course persume extending our computational language with many features that we would expect a general computational language to have, like references, algebraic datatypes, etc. It is not clear yet whether the programming effort required to implement such tactics inside our language would be less than what existing systems would require. Trying to implement such tactics is also very important because it would give us a sense as to what extensions our language needs and whether our type system can be extended so that meaningful types are given to these tactics.

8. Conclusion

We have presented a new language design that aims to be a general proof development framework. Our design places significant importance in combining proof construction with a general computational language. Terms in this computational language that might involve side-effects and non-termination can construct proofs that are guaranteed to be valid, which can later be used as part of more complex proofs. We have shown how we arrive at this language design starting from existing and widely used similar frameworks and aiming to simplify them. We show that despite any simplifications, our language retains the expressivity and power of the original frameworks. The novelty in our approach lies at the fact that our proof language and computational language manipulate essentially the same kind of terms. Furthermore, these terms can be represented efficiently at runtime through native representations, without sacrificing the validity of the proofs we are trying to arrive at.

References

[1] H. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(3):181–215, 1997.

[2] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)(Apr. 2008)*.

[3] J. Despeyroux, F. Pfenning, and C. Schurmann. Primitive recursion for higher-order abstract syntax. *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA97)*, pages 147–163, 1997.

[4] P. Fontaine, J. Marion, S. Merz, L. Nieto, and A. Tiu. Expressiveness+ automation+ soundness: Towards combining SMT solvers and interactive proof assistants. *TACAS*, 3920:167–181, 2006.

[5] G. Gonthier. The four color theorem in Coq. *Talk given at the TYPES 2004 conference, December*, 2004.

[6] M. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press New York, NY, USA, 1993.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[8] G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

[9] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices*, 41(1):42–54, 2006.

[10] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. *ACM SIGPLAN Notices*, 41(9):62–73, 2006.

[11] T. Nipkow, L. Paulson, and M. Wenzel. Isabelle/HOL – A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS, 2002.

[12] L. Paulson. Isabelle: The Next 700 Theorem Provers. *Arxiv preprint cs.LO/9301106*, 2000.

[13] F. Pfenning. Logic programming in the LF logical framework. *Logical Frameworks*, pages 149–181, 1991.

[14] F. Pfenning and C. Schurmann. System description: Twelf – a meta-logical framework for deductive systems. *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, 1999.

[15] A. Poswolsky and C. Schurmann. Extended Report on Delphin: A Functional Programming Language with Higher-Order Encodings and Dependent Types. Technical report, Technical Report YALEU/DCS/TR-1375, Yale University, 2007.

[16] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 37(1):217–232, 2002.

[17] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

[18] F. Wiedijk. Comparing mathematical provers. *Springer LNCS*, 2594:188–202, 2003.

[19] H. Xi. Applied Type System. *Proceedings of TYPES 2003*, 3085:394–408, 2004.