Embedding a Logical Theory of Constructions in Agda

Ana Bove Peter Dybjer

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
{bove,peterd}@chalmers.se

Andrés Sicard-Ramírez [*]

Department of Fundamental Sciences
EAFIT University
Medellín, Colombia
asicard@eafit.edu.co

Abstract

We propose a new way to reason about general recursive functional programs in the dependently typed programming language Agda, which is based on Martin-Löf's intuitionistic type theory. We show how to embed an external programming logic, Aczel's Logical Theory of Constructions (LTC) inside Agda. To this end we postulate the existence of a domain of untyped functional programs and the conversion rules for these programs. Furthermore, we represent the inductive notions in LTC (intuitionistic predicate logic with equality, and totality predicates) as inductive notions in Agda. To illustrate our approach we specify an LTC-style logic for PCF, and show how to prove the termination and correctness of a general recursive algorithm for computing the greatest common divisor of two numbers.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs–Logics of programs; D.2.4 [*Software Engineering*]: Software/Program Verification–Correctness proofs

General Terms Languages, Theory, Verification

Keywords Logical theory of constructions, type theory, general recursion

1. Introduction

Assume that we want to use a proof assistant for verifying programs written in a standard functional language such as Haskell (Peyton Jones 2003), where functions can be defined by general recursion. What do we do?

The most obvious idea is to build a dedicated proof assistant tailored for this functional language. For example, the LCF system (Gordon et al. 1979) could be used in this way, since the terms in LCF come from a typed lambda calculus with recursive function definitions and recursive data types, that is, from a core functional programming language. Recent examples of dedicated proof assistants for functional languages include the Sparkle system for Clean (de Mol et al. 2002), and the Plover system for Haskell (Harrison and Kieburtz 2005). However, building such a system from scratch is a formidable task. Not only do we need to decide what logical rules the system has and build a theorem prover for them, but we also need to write libraries, user interfaces, provide automatic theorem proving support, etc.

Another possibility is to use an existing mature generic proof assistant such as Isabelle (Paulson 1994), and implement our logic of functional programs inside it, see for example Dybjer and Sander (1989). In this paper we shall propose a variation of this approach, where we implement a logic of general recursive functional programs inside the proof assistant Agda (Norell 2007; Agda wiki) for dependent type theory.

The question of encoding general recursive (and possibly partial) functions in a dependent type theory which allows only total functions has been recently considered by a number of authors. For example, Bove and Capretta (2005) turn such partial functions into total ones by adding an extra argument to the function, a termination witness. Another possibility is to encode functional programs as relations (Gonzalía 2006; Barthe et al. 2006).

However, both these methods have drawbacks and limitations. One is that you do not verify the general recursive program as it stands, but an Agda representation of it. Another is that they do not deal with the general case of verifying all possibly higher order functional programs.

Here we propose a method which has neither of these drawbacks. Although we still need to encode a general recursive program inside Agda, this encoding does not change the general recursive program much, it only uses the standard representation of an untyped lambda calculus in a logical framework using higher order abstract syntax.

We first present a rather standard "external" logic for functional programs, Aczel's Logical Theory of Constructions (LTC). (The term "external" program logics comes from Girard (1986) who contrasts it to the "integrated" logics arising from the identification of programs and proofs on which systems like Agda and Coq build.) We then show how to encode LTC in Agda.

The original purpose of LTC-style logics (there are several) was as targets for lambda calculus (or realisability) interpretations of Martin-Löf type theory, see Aczel (1977), Aczel et al. (1991) and Smith (1984). LTC-style logics are also closely related to Feferman's systems for *explicit mathematics* (Feferman and Jäger 1996).

LTC will not suffice for proving all interesting properties of functional programs. To reason about infinite streams and partiality we will have use for principles which are not available in LTC, such as coinduction and principles based on domain theory, for example Scott induction. However, as Dybjer (1985) emphasised, LTC will go a long way if we restrict ourselves to behaviours of programs on total elements of data structures. It is a system in which we have a

[*] This research was performed during a research visit to Chalmers University of Technology which was funded by the ALFA network LERnet and EAFIT University.

natural interpretation of Martin-Löf type theory, and in this sense LTC is at least as strong as Martin-Löf type theory. Moreover, it is a system where we can reason about general recursive functions (defined by fixed points) in a natural way, as our example will show.

The CoVer project. The ideas behind the present paper arose during the CoVer project (Combining Verification Methods in Software Development), a joint project involving the Programming Logic, Functional Programming, and Formal Methods research groups at Chalmers University of Technology. The goal of this project was to build a system for verifying Haskell programs using a combination of automatic and interactive theorem proving, and random testing. To our disposal we already had several separate tools: for example, an earlier version of the Agda system ("Agda 1") for interactive proof in dependent type theory, automatic theorem provers for classical first order predicate logic, and the random testing tool QuickCheck (Claessen and Hughes 2000). To reach our goal we needed to integrate these tools into a functioning system for verifying "real" Haskell programs.

One proposal for using Agda for verifying Haskell programs was the monadic embedding of Haskell into Agda by Abel, Benke, Bove, Hughes, and Norell (Abel et al. 2005a). Different monads can be chosen for different Haskell programs. If a direct translation into Agda is possible, then the identity monad is chosen. If the Haskell program terminates on a decidable subset of the input type, then the Maybe monad can be chosen. Potentially, other monads can be used for dealing with general recursion where the termination predicate is a priori undecidable, although this possibility was not explored in the paper.

The question of how to deal with the whole of Haskell, including the "awkward squad" of features, was a lively topic of discussion in the CoVer project. The adopted approach was to use GHC's (Glasgow Haskell compiler) reduction of full Haskell to an "external core language", and then to provide several translators from the core language for different verification purposes (into first order logic for the purpose of automatic verification, and into Agda 1 for interactive verification). We will not address the "whole of Haskell" issue in this paper, although future work will benefit from the experience which was accumulated during the CoVer project.

Although the long term goal is to verify "real" functional programs (e g written in Haskell), we explain our approach in a simple setting. For the time being, we limit our discussion to the core functional programming language PCF (Plotkin 1997), present the rules of an LTC-style logic for PCF, and show how to verify a general recursive PCF-program for computing the greatest common divisor of two numbers. It is straightforward to extend the discussion to versions of PCF with lists and other recursive data types.

Paper overview. Sections 2 and 3 present the version of PCF that we use in this paper and the version of LTC which is restricted to PCF, respectively. In Section 4 we describe how to embed LTC in the proof assistant Agda. Section 5 introduces the example we use throughout this paper, a recursive PCF-program that computes the greatest common divisor (`gcd`) of two numbers. In Section 6 we give a termination proof for `gcd`. In Section 7 we prove that our program is correct, in other words, that `gcd` indeed returns the greatest common divisor of its inputs. We conclude in Section 8 with some important directions for further developments of this approach.

Agda code. We show only excerpts of the Agda code (available at `www1.eafit.edu.co/asicard/code/plpv-code-09.tgz`) accompanying this paper. Note that some simple arithmetic properties used in this proof have been left as postulates. We make heavy use of Agda's infix and mix-fix operators, and of Agda's implicit arguments to facilitate readability. In an infix or mix-fix operator, the places for the arguments are marked with "_". Implicit arguments are delimited with curly brackets instead of parentheses.

2. PCF-terms

For simplicity, we restrict our discussion to a version of the functional programming language PCF, where we have added a constant error for explicit error handling. The abstract syntax is:

$$\begin{aligned} t ::= {} & x \mid t\ t \mid \lambda x.t \mid \mathsf{fix}\ x.t \mid 0 \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \mid \mathsf{iszero}\ t \\ & \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{error} \end{aligned}$$

To illustrate our method, we shall verify the following PCF-program which computes the greatest common divisor of two naturals numbers using Euclid's algorithm:

$$\begin{aligned} & \mathsf{fix}\ g.\lambda m.\lambda n.\mathsf{if}\ (\mathsf{iszero}\ n) \\ & \qquad \mathsf{then\ if}\ (\mathsf{iszero}\ m)\ \mathsf{then\ error\ else}\ m \\ & \qquad \mathsf{else\ if}\ (\mathsf{iszero}\ m)\ \mathsf{then}\ n \\ & \qquad\qquad \mathsf{else\ if}\ m \succ n\ \mathsf{then}\ g\ (m-n)\ n \\ & \qquad\qquad\qquad \mathsf{else}\ g\ m\ (n-m) \end{aligned}$$

We omit the PCF definitions of the functions $-$ (minus) and $\succ$ (greater-than) on natural numbers.

3. LTC for PCF

We use LTC as a generic name for a family of related logical systems which have been used by Aczel (1977, 1980, 1989) and Smith (1984) for the purpose of interpreting Martin-Löf type theory in type-free logical systems.

Some of these systems are based on combinators and stay strictly within the realm of first order logic (Aczel 1977), whereas others are based on the lambda calculus and do not. Since they play a role for constructive foundations, these systems are usually intuitionistic but we may also consider classical versions. As we will explain later, we are planning to use automatic theorem provers for classical first order logic and for that purpose, it will be important to work in a classical combinator based system. However, for the purpose of this presentation, we will work in a system based on the lambda calculus and will only use intuitionistic logic.

The idea to use LTC for verifying functional programs is discussed by Dybjer (1985, 1990), and we shall now present a mechanisation of this approach. It suffices here to use a version of LTC which is adapted to PCF, where we only have booleans and natural numbers as basic data. In Aczel's and Smith's work, other kinds of data were considered, most notably the *internal propositions* which are used for interpreting universes in Martin-Löf type theory. This interesting part of LTC is however not needed here and is not part of our LTC system for PCF.

Our LTC system for PCF is an intuitionistic predicate logic with equality, where the terms are PCF-terms (as defined above). There are two predicate symbols: B for *total boolean*, and N for *total natural number*. (Note that LTC does not contain predicate symbols for representing the usual PCF-types, which also contain *partial* values). We have the following axioms and axiom schemata:

- *Conversion rules* for the PCF-terms:

$$\begin{aligned}\forall t\,t'.\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t\ \mathsf{else}\ t' &= t\\ \forall t\,t'.\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t\ \mathsf{else}\ t' &= t'\\ \mathsf{pred}\ 0 &= 0\\ \forall t.\mathsf{pred}\ (\mathsf{succ}\ t) &= t\\ \mathsf{iszero}\ 0 &= \mathsf{true}\\ \forall t.\mathsf{iszero}\ (\mathsf{succ}\ t) &= \mathsf{false}\\ \forall t\,t'.(\lambda x.t)\ t' &= t[x := t']\\ \forall t.\ \mathsf{fix}\ x.\ t &= t[x := \mathsf{fix}\ x.\ t]\end{aligned}$$

  where $t[x := t']$ is the capture-free substitution of x for t' in t,

- *Discrimination rules* for constructors:

$$\neg(\mathsf{true} = \mathsf{false})$$
$$\forall t.\ \neg(0 = \mathsf{succ}\,t)$$

- *Introduction rules* for B and N:

$$\begin{array}{ll}\mathsf{B}\ \mathsf{true} & \mathsf{N}\ 0\\ \mathsf{B}\ \mathsf{false} & \forall t.\mathsf{N}\ t \to \mathsf{N}\ (\mathsf{succ}\ t),\end{array}$$

- *Elimination rules* for B and N expressing proof by case analysis on boolean values, and proof by mathematical induction, respectively:

$$P\ \mathsf{true} \to P\ \mathsf{false} \to \forall t.(\mathsf{B}\ t \to P\ t)$$
$$P\ 0 \to \forall t.(\mathsf{N}\ t \to P\ t \to P\ (\mathsf{succ}\ t)) \to \forall t.(\mathsf{N}\ t \to P\ t).$$

In LTC we use the totality predicates for expressing termination. If a function f maps a total natural number to a total natural number

$$\forall t.\mathsf{N}\ t \to \mathsf{N}\ (f\ t),$$

then f terminates and outputs a (total) natural number whenever the input is a (total) natural number. This is because a term which is equal to (convertible to) a canonical form, also reduces to that canonical form. See Dybjer (1985) for a detailed discussion of this point.

Using the totality predicate N we can thus express that our `gcd` program always terminates, unless both arguments are 0:

$$\begin{aligned}&\forall m\ n.\mathsf{N}\ m \to \mathsf{N}\ n \to \neg(m = 0 \wedge n = 0) \to\\ &\quad \mathsf{N}(\mathsf{gcd}\ m\ n)\end{aligned}$$

The correctness of the algorithm, that is, the fact that it outputs the greatest common divisor of the two inputs, unless they are both 0, is expressed as follows:

$$\begin{aligned}&\forall m\ n.\mathsf{N}\ m \to \mathsf{N}\ n \to \neg(m = 0 \wedge n = 0) \to\\ &\quad \mathsf{CD}\ m\ n\ (\mathsf{gcd}\ m\ n) \wedge\\ &\quad \forall d.(\mathsf{CD}\ m\ n\ d \to d \leqslant (\mathsf{gcd}\ m\ n))\end{aligned}$$

where $(\mathsf{CD}\ m\ n\ d)$ stands for $(d \mid m \wedge d \mid n)$ when $\_ \mid \_$ is the divisibility predicate. Note that in LTC we distinguish between the less-than-or-equal *predicate* $\leqslant$ and the less-than-or-equal *function* $\preccurlyeq$. We have

$$m \leqslant n \leftrightarrow m \preccurlyeq n = \mathsf{true}$$

All these notions are readily definable in LTC.

4. Representing LTC in Agda

Agda is a proof assistant based on Martin-Löf type theory. It can also be used as a programming language with dependent types. Like in other modern functional programming languages, we can define data types and functions by pattern matching over elements that belong to such data types. The difference is that, if we intend to use the system as a theorem prover, we must ensure its logical consistency and hence, that all the defined functions are total. For this purpose, Agda performs a *coverage check* for pattern matching definitions and a *termination check* for recursive calls; both checks are purely syntactical. There is also a *positivity check* for data types. The reader who is interested in learning more about the Agda language and how to use it for programming and proving can look at (Bove and Dybjer 2008; Agda wiki) for a gentle introduction. In what follows, we will assume that the reader is familiar with dependently typed programming.

So, how can we represent LTC in Agda? The first thought is to use Agda as a *logical framework* along the lines of the Edinburgh Logical Framework (ELF) (Harper et al. 1993). Like Agda, ELF is a lambda calculus with dependent types and a few universes. A logical theory (like LTC) is encoded in ELF by adding constants of the appropriate types. Some constants encode the syntactic constructs (term formers, predicate symbols), others encode proof objects for inference rules. In Agda we can add such constants by declaring them as *postulates*. In this way we can use Agda as a logical framework in essentially the same way as ELF.

4.1 Logical Framework-style Encoding of LTC

PCF-terms. We employ the usual method for encoding the untyped lambda calculus.

First we postulate a domain of PCF-terms:

```
postulate D : Set
```

Then we postulate the term constructors for PCF, using *higher order abstract syntax* for representing the variable binding operations λ and `fix` as higher order functions.

```
postulate
  λ   : (D -> D) -> D    -- abstraction and app.
  _‘_ : D -> D -> D

  fix : (D -> D) -> D    -- fixed point operator

  zero   : D             -- partial nat. numbers
  succ   : D -> D
  pred   : D -> D
  iszero : D -> D

  true           : D     -- partial booleans
  false          : D
  if_then_else_ : D -> D -> D -> D

  error : D              -- error
```

Intuitionistic predicate logic with equality. It is also well-known how to encode intuitionistic predicate logic with equality. The set `D` will also be the domain of our predicate logic. As usual, we implement propositions in predicate logic as types in Agda. Using postulates, we can introduce each logical constant as a set former, and each inference rule (introduction and elimination) as constants of the corresponding types. We show here only the rules for the equality predicate; the other logical constants are represented in a similar way. Note that we use `==` for *propositional* equality, while $=$ stands for *definitional* equality in Agda.

```
postulate
  _==_     : D -> D -> Set
  ==-refl  : (d : D) -> d == d
  ==-subst : (P : D -> Set) {d₁ d₂ : D} ->
             d₁ == d₂ -> P d₁ -> P d₂
```

The equality predicate over `D` is represented by a function that takes two elements in `D` and returns a set (that is, an element in `Set`): the set of proofs that the two elements of `D` are equal. Using `==-refl` we can only construct proofs that an element is equal to itself. The substitutivity property `==-subst` states that, if two elements of `D` are equal, and if we are able to prove an arbitrary predicate `P` over `D` for the first of the two elements, then we can produce a proof that the second element also satisfies the predicate.

Conversion and discrimination rules. The next step is to represent the conversion and discrimination rules.

We only show a few conversion rules here: those for predecessor, the beta-rule, and the conversion rule for fixed points:

```
postulate
  CP₁ : pred zero                == zero
  CP₂ : (n : D) -> pred (succ n) == n

  beta : (f : D -> D) -> (a : D) ->
         (λ f) ‘ a == f a

  Cfix : (f : D -> D) -> fix f == f (fix f)
```

The two rules that discriminate between constructors are represented as follows:

```
postulate
  true≠false : ¬ (true == false)
  0≠S        : {n : D} -> ¬ (zero == succ n)
```

As usual in intuitionistic logic, negation is here defined in terms of implication and absurdity.

Rules for the totality predicates. We show only the rules for the totality predicate for natural numbers:

```
postulate
  N : D -> Set
  N-z : N zero
  N-s : {n : D} -> N n -> N (succ n)

  N-ind : (P : D -> Set) -> P zero ->
          ({n : D} -> N n -> P n -> P (succ n)) ->
          {n : D} -> N n  -> P n
```

where we assume that `zero` and `succ` are the Agda representation of the PCF-terms 0 and succ from Section 2, respectively.

The first rule (the formation rule) declares a unary predicate symbol. The next two are the introduction rules. The last one is the induction rule.

Consistency. The consistency of our LTC for PCF is unproblematic. We can for example interpret `D` as a domain in the sense of Scott (1976) and interpret PCF-terms in `D` as in denotational semantics. Then we can interpret == as equality of denotations and verify the conversion rules. See Aczel (1980) for an account of *Frege structures* which model the full LTC-language.

4.2 Making use of Agda's Inductive Notions

The main disadvantage of encoding the logical framework as presented above is that it makes limited use of the support for writing proofs that Agda can provide. One of the main strengths of Agda is its support for inductive data types, including inductive families, and function definitions using pattern matching on such inductive data types. To benefit from this feature, it would be better if we encode the inductive notions used in our program logic as data types in Agda, rather than as postulates. First, the logical constants can be represented as set formers defined by their introduction rules in the usual way. Moreover, the totality predicates for boolean values and natural numbers can be implemented as inductively defined predicates over `D`.

Inductively defined sets and families are introduced in Agda with the `data` construct. The elements of the set are introduced by giving the name of the constructors and their types. We can then define the elimination rules (for the logical constants and the totality predicates) by pattern matching, as usual. If this is the only use we make of pattern matching we get an implementation of LTC in Agda, which is equivalent to the logical framework implementation.

However, to make full use of Agda's support for proof by pattern matching, we will not restrict ourselves to using the elimination rules above. Instead we will allow proof by pattern matching in general, as long as it is accepted by Agda's coverage and termination checker. This means that we actually work in an extension of LTC since, by working in this way, new, more general, induction principles become available. Of course, it is expected that this extension is conservative, although a rigorous proof of this would be quite hard, and is outside the scope of this paper.

What about the set `D`? Can it also be inductively defined in Agda? Yes, this is possible, we can for example build a Scott domain for PCF in Agda, see for example Hedberg (1996) for an implementation of constructive domain theory in ALF, a precursor of Agda. However, we will then not be able to use Agda's intensional equality type for equality of domain elements; equality of domain elements will be represented by a non-trivial equivalence relation. As a consequence we would need to work with the *setoid* `(D,==)` and this would be inconvenient. Since we will never prove properties by induction on `D`, it is preferable to postulate the existence of `D` and the conversion rules for terms on D.

Remark. One might object that we should not work with postulates which cannot be instantiated in Martin-Löf type theory. We reject this objections on the grounds that we are not here attempting to contribute to constructive foundations but to outline a genuinely practical approach to verification of functional programs. To this end, we do not mind using classical logic (for reasons given above) and axioms which are proved consistent using classical techniques.

To sum up, some of the axioms of LTC will still be postulated, others will be consequences of inductive definitions. We now present how this is done.

PCF-terms. This is just like in the logical frameworks encoding presented before. We postulate a domain of PCF-terms

```
postulate D : Set
```

and the existence of constants for representing the term constructors.

Intuitionistic predicate logic with equality. The logical constants are now represented as inductively defined set formers, as usual in Martin-Löf type theory. For example, the Agda code for existential quantification and for the equality predicate are as follows:

```
data ∃ (P : D -> Set) : Set where
  ∃-i : (witness : D)  -> P witness -> ∃ P

∃-fst : {P : D -> Set} -> ∃ P -> D
∃-fst (∃-i x px) = x

∃-snd : {P : D -> Set} -> (x-px : ∃ P) ->
        P (∃-fst x-px)
∃-snd (∃-i x px) = px

data _==_ (x : D) : D -> Set where
  ==-refl : x == x

==-subst : (P : D -> Set){x y : D} -> x == y ->
           P x -> P y
==-subst P ==-refl px = px
```

As mentioned above, we do not need to restrict ourselves to using the elimination rules for the logical constants, but can employ the more general pattern matching provided by Agda.

Conversion and discrimination rules. Just as in the logical framework representation, the conversion rules are postulated.

Rules for the totality predicates. The totality predicates are now represented as inductive families generated by the introduction rules. We only show the totality predicate for natural numbers.

```
data N : D -> Set where
  N-z : N zero
  N-s : {n : D} -> N n -> N (succ n)
```

We can now define the elimination rule for `N` by pattern matching:

```
N-ind : (P : D -> Set) -> P zero ->
        ({n : D} -> N n -> P n -> P (succ n)) ->
        {n : D} -> N n  -> P n
N-ind P p0 h N-z      = p0
N-ind P p0 h (N-s Nn) = h Nn (N-ind P p0 h Nn)
```

However, as already remarked, we will not restrict ourselves to using this elimination rule, but will use Agda's pattern matching on `B` and `N`.

Consistency. Establishing the consistency of the latter encoding of LTC in Agda is less straightforward than for the logical framework encoding. We need to establish the consistency of Agda extended with the postulates for `D` and the conversion rules. We start with the usual set-theoretic model of Martin-Löf type theory with inductive definitions following Dybjer (1991). In this model Agda's function spaces are interpreted as full set-theoretic function spaces and inductive notions are interpreted as least fixed points of monotone operators.

We then extend this model in two ways. First we model the additional postulates for `D` by constructing a Scott domain for `D`. Moreover, the induction principles accepted by Agda (with coverage checking and termination checking) are modelled by total functions in the set-theoretic model. To work out the details of this model would require substantial work and is outside the scope of this paper.

Note that building a set-theoretic model of Agda, even without the extra postulates for `D`, but with pattern matching, coverage checking, and termination checking would already require substantial work. It would entail the consistency of Agda, viewed as a logical system.

5. Example: Greatest Common Divisor

In Section 2 we showed a program `gcd` for computing the greatest common divisor of two numbers written in PCF. We can now use the representation of PCF-terms as elements of the set `D` in Agda, which we presented in the previous section, to define the `gcd` algorithm:

```
gcdh : D -> D
gcdh = \g -> λ (\m -> λ (\n ->
              if (iszero n)
              then (if (iszero m)
                    then error
                    else m)
              else (if (iszero m)
                    then n
                    else (if (m ≻ n)
                          then g ‘ (m - n) ‘ n
                          else g ‘ m ‘ (n - m)))))

gcd : D -> D -> D
gcd m n = fix gcdh ‘ m ‘ n
```

where both $-$ and $\succ$ are functions of type `D -> D -> D`.

From this definition of `gcd` we prove the following five *lemmas*, which will be useful when proving properties about the algorithm:

```
gcd-00 : gcd zero zero == error

gcd-S0 : {m : D} -> N m ->
         gcd (succ m) zero == succ m

gcd-0S : {n : D} -> N n ->
         gcd zero (succ n) == succ n

gcd-S>S : {m n : D} -> N m -> N n ->
          (succ m > succ n) ->
          gcd (succ m) (succ n) ==
                    gcd (succ m - succ n) (succ n)

gcd-S≤S : {m n : D} -> N m -> N n ->
          succ m ≤ succ n ->
          gcd (succ m) (succ n)  ==
                    gcd (succ m) (succ n - succ m)
```

Although they follow rather straightforwardly from the definition of `gcd` and the conversion rules for PCF, proving them is a surprisingly time consuming task in Agda. The problem is that Agda does not know how to normalise a PCF-programs, so each step in the proof has to be performed manually. The conversions rules for PCF-programs are given by postulates and hence do not contribute to the usual normalisation provided by Agda's type checker. As we describe in Section 8.1, we plan to connect a first order theorem prover to Agda, which we expect will automatise much of the reasoning needed to prove equations of this kind.

Note that $>$ and $\leq$ above are relations, that is, operators that return a set, and not functions returning a truth value:

```
_>_ : D -> D -> Set
m > n = m ≻ n == true

_≤_ : D -> D -> Set
m ≤ n = m ≻ n == false
```

Similar definitions are given for the relations < and $\geq$.

In the next two sections we show that this algorithm is correct and terminates with the greatest common divisor of its two inputs.

6. Termination of `gcd`

The termination theorem for `gcd` states that if `m` and `n` are total natural numbers, then `gcd m n` is also a total natural number, provided that at least one of the numbers `m` or `n` is non-zero:

```
gcd-N : {m n : D} -> N m -> N n ->
        ¬ ((m == zero) ∧ (n == zero)) ->
        N (gcd m n)
```

We first prove some (easy) lemmas using the substitutivity property of equality:

```
gcd-S0-N : {m : D} -> N m -> N (gcd (succ m) zero)

gcd-0S-N : {n : D} -> N n -> N (gcd zero (succ n))

gcd-S>S-N : {m n : D} -> N m -> N n ->
            N (gcd (succ m - succ n) (succ n)) ->
            succ m > succ n ->
            N (gcd (succ m) (succ n))
```

```
gcd-S≤S-N : {m n : D} -> N m -> N n ->
            N (gcd (succ m) (succ n - succ m)) ->
            succ m ≤ succ n ->
            N (gcd (succ m) (succ n))
```

These four lemmas show that the left hand sides of the last four equations presented in Section 5 terminate. Note that the last two lemmas have an extra hypothesis stating that the result of the recursive call corresponding to the case we are considering (that is, the right hand side of the corresponding equation in Section 5) also terminates.

Now, given two total natural numbers `m` and `n`, we prove that either `m` < `n` or `m` ≥ `n` by pattern matching on the proofs that the numbers are total.

```
x>y∨x≤y : {m n : D} -> N m -> N n ->
          (m > n) ∨ (m ≤ n)
```

Next we prove two auxiliary lemmas. The first lemma concerns the case where the total numbers `m` and `n` are such that `m` > `n`

```
gcd-x>y-N :  {m n : D} -> N m -> N n ->
             (h : {m' n' : D} -> N m' -> N n' ->
                  (m' , n') <₂ (m , n) ->
                  ¬ ((m' == zero) ∧ (n' == zero))
                  -> N (gcd m' n')) ->
             m > n ->
             ¬ ((m == zero) ∧ (n == zero)) ->
             N (gcd m n)
```

The other auxiliary lemma is similar but concerns the case where `m` ≤ `n`. Both lemmas are proved by pattern matching on the proofs that the numbers are total and they call (some of) the previous four lemmas we presented at the beginning of this section.

Finally, we use course-of-value induction on the lexicographic order $<_2$ on pairs of (total) natural numbers

```
N-wf₂ : (P : D -> D -> Set) ->
        ({m n : D} -> N m -> N n ->
          ({m' n' : D} -> N m' -> N n' ->
          (m' , n') <₂ (m , n) -> P m' n') ->
          P m n ) ->
        {m n : D} -> N m -> N n -> P m n
```

to prove that `gcd` terminates. (`N-wf₂` can be proved by course-of-value induction on the usual order on (total) natural numbers which, in turn, can be proved by pattern matching on the proof that the numbers are total.)

Note that if in `N-wf₂` we take `P` such that

```
P : D -> D -> Set
P m n = ¬ ((m == zero) ∧ (n == zero)) ->
        N (gcd m n)) ->
```

then, the type that results from `N-wf₂`, when applied to `m` and `n` and proofs that they are total, is the same as the type that results from the auxiliary lemma `gcd-x>y-N`. Moreover, the premise what we called `h` in `gcd-x>y-N` corresponds to the premise of the induction step in the induction principle `N-wf₂`.

7. Correctness of `gcd`

We show here that the `gcd` algorithm presented in Section 5 returns the greatest common divisor of two numbers, provided that at least one of those two numbers is non-zero. The main proofs have the same structure as the termination proof above. Hence, we only show the main steps.

Let us first define the divisibility relation:

```
_|_ : D -> D -> Set
m | n = ¬ (m == zero) ∧ ∃ (\k -> n == k * m)
```

That two elements have a common divisor is expressed by

```
CD : D -> D -> D -> Set
CD m n d = (d | m) ∧ (d | n)
```

That an element is greater than any common divisor is expressed by

```
GRT : D -> D -> D -> Set
GRT m n g = (d : D) -> N d -> CD m n d -> d ≤ g
```

That an element is the greatest common divisor is expressed by

```
GCD : D -> D -> D -> Set
GCD m n g = CD m n g ∧ GRT m n g
```

Our correctness theorem is thus

```
gcd-GCD : {m n : D} -> N m -> N n ->
          ¬ ((m == zero) ∧ (n == zero)) ->
          GCD m n (gcd m n)
```

To prove this theorem we only need to combine a proof that the result of `(gcd m n)` is a common divisor of `m` and of `n` (called `gcd-CD` and presented in Section 7.1), and a proof that `(gcd m n)` is greater than any common divisors of `m` and `n` (called `gcd-GRT` and presented in Section 7.2). See Section 7.3 for the full proof of the correctness property.

7.1 Common Divisor

The proof of

```
gcd-CD : {m n : D} -> N m -> N n ->
         ¬ ((m == zero) ∧ (n == zero)) ->
         CD m n (gcd m n)
```

follows a similar structure to the proof of `gcd-N` presented in Section 6. The four lemmas we need to prove here are the following:

```
gcd-S0-CD : {m : D} -> N m ->
            CD (succ m) zero (gcd (succ m) zero)

gcd-0S-CD : {n : D} -> N n ->
            CD zero (succ n) (gcd zero (succ n))

gcd-S>S-CD :
  {m n : D} -> N m -> N n ->
  CD (succ m - succ n) (succ n)
     (gcd (succ m - succ n) (succ n)) ->
  succ m > succ n ->
  CD (succ m) (succ n) (gcd (succ m) (succ n))

gcd-S≤S-CD :
  {m n : D} -> N m -> N n ->
  CD (succ m) (succ n - succ m)
     (gcd (succ m) (succ n - succ m)) ->
  succ m ≤ succ n ->
  CD (succ m) (succ n) (gcd (succ m) (succ n))
```

Next, we need to prove the corresponding auxiliary lemmas. Then we use course-of-value induction on the lexicographic order on pair of (total) natural numbers to obtain the desired result, namely, that `(gcd m n)` is a common divisor of `m` and of `n`.

7.2 Greater than any Common Divisor

To prove that the result of the `gcd` algorithm is greater than any other common divisor of the two numbers, we use an auxiliary relation stating that any common divisor of `m` and `n` divides `g`:

```
DIVISIBLE : D -> D -> D -> Set
DIVISIBLE m n g = {d : D} -> N d ->
                  CD m n d -> d | g
```

We first prove that this relation holds for `m`, `n` and `(gcd m n)`:

```
gcd-DIVISIBLE : {m n : D} -> N m -> N n ->
                ¬ ((m == zero) ∧ (n == zero)) ->
                DIVISIBLE m n (gcd m n)
```

This proof follows the same structure as the other proofs. Namely, we first prove four lemmas showing that the relation hold for the result of the left hand side of the last four equations of `gcd` presented in Section 5. As usual, in the two cases where `gcd` performs a recursive call, we need to add an extra hypothesis stating that the property holds for the result of the recursive call. Then we prove the two corresponding auxiliary lemmas and finally, we wrap up all the intermediate results using course-of-value induction on the lexicographic order on pairs of (total) natural numbers.

Next, we show that for any natural number `g` that is a common divisor of two other elements and that is related to these elements by the `DIVISIBLE` relation, then `g` is also greater than any common divisor of the two other elements.

```
gcd-GRT : {m n g : D} -> N g -> CD m n g ->
          DIVISIBLE m n g -> GRT m n g
```

This property is proved by pattern matching on the proof that `g` is a total number, or in other words, that `(N g)` holds. In the first case `(N g)` is `N-z`, which means that `g` must be `zero`. We know here that `g | m` because `(CD m n g)` holds, but this is actually not possible because the definition of divisibility rules out the case `zero | _`. In the second case `(N g)` is `(N-s x)` and hence, `g` must be `(succ i)` for some element `i`. The `DIVISIBLE` relation says that any common divisor `d` of `m` and `n` must also divide `g`, from where we can derive that $d \leq g$ since `g` is non-zero.

In particular, `gcd-GRT` can be applied to `m`, `n` and `(gcd m n)` when at least one of `m` or `n` is non-zero, given that for this particular choice of inputs we can prove all the necessary hypothesis (as we have already showed in this paper), and thus obtain that `GRT m n (gcd m n)` holds.

7.3 Greatest Common Divisor

Putting all this together our correctness proof becomes:

```
gcd-GCD : {m n : D} -> N m -> N n ->
          ¬ ((m == zero) ∧ (n == zero)) ->
          GCD m n (gcd m n)
gcd-GCD {m} {n} Nm Nn mn≠0 =
  ∧-i gcd-cd
      (gcd-GRT (gcd-N Nm Nn mn≠0)
               gcd-cd
               (gcd-DIVISIBLE Nm Nn mn≠0))
  where gcd-cd : CD m n (gcd m n)
        gcd-cd = gcd-CD Nm Nn mn≠0
```

8. Future Work

8.1 Calling an Automatic Theorem Prover

Reasoning about a program such as `gcd` in our Agda implementation of LTC is very low level compared with ordinary reasoning about programs in Agda. For example, judgements of the form `n : Nat` (where `Nat` is the inductively defined set of natural numbers) are checked automatically by Agda, whereas LTC propositions of the form `(N n)` have to be proved manually by constructing proof objects `proof : N n`. Moreover, Agda automatically normalises terms by using its definitional equality rules, whereas simplification using the postulated conversion rules for elements in `D` has to be done manually.

However, much of this low-level reasoning could be done automatically, for example, by a theorem prover for first order predicate logic. Therefore we plan to connect an off-the-shelf theorem prover (like Vampire or Equinox) to Agda, and use it to automatise substantial parts of our LTC-proofs. Such a "plug-in" for first order logic was in fact already built for Agda 1 and Agda-Light (an experimental system), and we plan to do the analogous connection for the current version of Agda.

In order to make full use of a first order theorem prover, it becomes necessary to modify LTC above so that it is an official first order theory. We need to transform the above lambda calculus based version of LTC to a combinator based one. In practice, we use so called "super-combinators", that is, rather than translating everything into combinations of `S` and `K`, we introduce a new combinator for each lambda (so called *lambda lifting*). This leads to a very natural proof style where we would for example have a super-combinator `gcd` with its recursion equations as postulates.

During the CoVer project, there were several contributions in this direction which we expect to benefit from. First, Koen Claessen experimented with a method where Haskell programs were translated into first order logic, and theorems about them were proved by using off-the-shelf theorem provers. Gregoire Hamon wrote a program ("The CoVer translator") which automatically translated programs in the external core language of GHC into a language understandable by first order theorem provers. And as already mentioned above, Andreas Abel, Thierry Coquand, and Ulf Norell (Abel et al. 2005b) investigated the use of external first order theorem provers for proving theorems in Agda-Light.

8.2 Using Agda's Standard Library for Proofs in LTC

The `gcd` function calls auxiliary functions for subtraction and greater-than. Firstly, these functions are defined by primitive recursion and will pass Agda's termination checker. So there is no need to leave standard dependent type theory and move to LTC to prove properties about them. Moreover, they are basic functions, the properties of which can be expected to be found in a standard library.

So the question arises whether we can transfer such results about subtraction and greater-than to LTC. The answer is that we can apply the Aczel translation of Martin-Löf type theory into LTC. For example, the function

```
_+_ : Nat -> Nat -> Nat
```

would be translated into the LTC-function

```
_+'_ : D -> D -> D
```

with the property

```
(m n : D) -> N m -> N n -> N (m +' n)
```

And the commutativity theorem for +

```
(m n : Nat) -> m + n == n + m
```

will be translated into

```
(m n : D) -> N m -> N n -> m +' n == n +' m
```

References

Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying Haskell programs using constructive type theory. In *Proc. of the ACM SIGPLAN 2005 Haskell Workshop*, pages 62–73. ACM, 2005a.

Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. In B. Gramlich, editor, *Proc. of 5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *LNCS*, pages 285–301, 2005b.

Peter Aczel. What might the objects of the logical theory of constructions be? In P. Dybjer et al., editors, *Proc. of the Worshop on Programming Logic*, number 54 in Programming Methodology Group Reports, pages 122–139. Chalmers University of Technology, 1989.

Peter Aczel. Frege structures and the notion of proposition, truth and set. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 31–59. Amsterdan: North-Holland, 1980.

Peter Aczel. The strength of Martin-Löf's intuitionistic type theory with one universe. In S. Miettinen and J. Väänanen, editors, *Proc. of the Symposium on Mathematical Logic (Oulu, 1974)*, Report No. 2, Department of Philosopy, University of Helsinki, Helsinki, pages 1–32, 1977.

Peter Aczel, David P. Carlisle, and Nax Mendler. Two framework of theories and their implementation in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 3–39. Cambridge University Press, 1991.

Agda wiki. Available at `appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`, 2008.

Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2006. ISBN 3-540-33438-6.

Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708, February 2005. Cambridge University Press.

Ana Bove and Peter Dybjer. Dependent types at work, 2008. Lecture notes of a graduate course with the same name. Submitted for publication in the post-proceedings of the International Summer School on Language Engineering and Rigorous Software Development.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, volume 35.9 of *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000.

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers. Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages. 13th International Workshop, IFL 2001*, volume 2312 of *LNCS*, pages 55–71, 2002.

Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

Peter Dybjer. Program verification in a logical theory of constructions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 334–349, 1985. Appears in revised form as Programming Methodology Group Report 26, University of Gothenburg and Chalmers University of Technology, 1986.

Peter Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79, 1990.

Peter Dybjer and Herbert Sander. A functional programming approach to the specification and verification of concurrent systems. *Formal Aspects of Computing*, 1:303–319, 1989.

Solomon Feferman and Gerhard Jäger. Systems of explicit mathematics with non-constructive μ-operator, part II. *Ann. Pure Appl. Logic*, 79(1): 37–52, 1996.

Jean-Yves Girard. Linear logic and parallelism. In M. Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism*, volume 280 of *LNCS*, pages 166–182, 1986.

Carlos Gonzalía. *Relations in Dependent Type Theory*. PhD thesis, Chalmers University of Technology and University of Gothenburg, Department of Computer Science and Engineering, 2006.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.

William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, 2005.

Michael Hedberg. A type-theoretic interpretation of constructive domain theory. *J. Autom. Reasoning*, 16(3):369–425, 1996.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and University of Gothenburg, Department of Computer Science and Engineering, 2007.

Lawrence C. Paulson. *Isabelle. A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994. (With a contribution by T. Nipkow).

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, April 2003.

Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1997.

Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

Jan Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *The Journal of Symbolic Logic*, 49(3):730–753, 1984.