Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 249 (2009) 61–74
www.elsevier.com/locate/entcs

Another Look at Function Domains

Ana Bove

Department of Computer Science and Engineering,
Chalmers University of Technology, 412 96 Göteborg, Sweden
telephone: +46-31-7721020, fax: +46-31-7723663, Email: bove@chalmers.se

Abstract

Bove and Capretta have presented a method to deal with partial and general recursive functions in constructive type theory which relies on an inductive characterisation of the domains of the functions. The method separates the logical and the computational aspects of an algorithm, and facilitates the formal verification of the functions being defined. For nested recursive functions, the method uses Dybjer' schema for simultaneous inductive-recursive definitions. However, not all constructive type theories support this kind of definitions.
Here we present a new approach for dealing with partial and general recursive functions that preserves the advantages of the method by Bove and Capretta, but which does not rely on inductive-recursive definitions. In this approach, we start by inductively defining the graph of the function, from which we *first* define the domain and *afterwards* the type-theoretic version of the function. We show two ways of proving the formal specification of the functions defined with this new approach: by induction on the graph, or by using an induction principle in the style of the induction principle associated to the domain predicates of the Bove-Capretta method.

Keywords: General recursive functions, partial functions, nested functions, constructive type theory

1 Introduction

Bove and Capretta [6] have presented a method to deal with partial and general recursive functions in constructive type theory [13,10]. The idea is simple: given a function written in a Haskell-like style [17], we (automatically) extract the domain of the function and use that domain to define the function in type theory. The domain of the function is formally defined as an inductive predicate. The type-theoretic version of the function takes now an extra argument, a proof that the input belongs to the domain of the function, and is defined by structural recursion on this extra argument. In many cases, the type-theoretic version of the function is defined after its domain has been already defined. However, when the function is nested, the domain predicate and the type-theoretic version of the function need to be formalised following Dybjer' schema for simultaneous inductive-recursive definitions [11].

There are two clear advantages when using the Bove-Capretta method.

First, the method separates the logical and the computational aspects of an algorithm. This allows a function to be defined independently of its termination

doi:10.1016/j.entcs.2009.07.084

behaviour and hence, it is possible to also use the method to formalise partial functions in an easy way.

Second, if we work in constructive type theory, the inductive domain predicate has an associated induction principle which is very useful when proving the formal specification of the function.

In Section 2, we illustrate the Bove-Capretta method on a well-known nested recursive function, McCarthy's 91 function. There, besides presenting the formal definition of the function, we also show how to prove some of its properties. For further reading on the method, including its limitations, the reader is referred to [5,6,4] where the method is presented and exemplified for a constructive type theory in the spirit of Martin-Löf's type theory. For the applicability of the method to the Calculus of Inductive Constructions (CIC), see Chapter 15 of Bertot's and Castéran's book [3].

As already mentioned, for nested recursive functions the method uses Dybjer' schema for simultaneous inductive-recursive definitions. However, this schema is not supported in all constructive type theories and their subsequent interactive theorem provers; for example, such schema is not supported in the CIC nor in the proof assistant Coq [3], based on the CIC.

In this paper, we investigate a different approach to function domain which allows us to formally define a nested function without the need of a simultaneous inductive-recursive schema. As before, the type-theoretic version of the function takes as extra argument the proof that the input belongs to the domain of the function. However, in this new approach the function is always defined *after* the domain has been defined, avoiding then a simultaneous definition of the function and its domain even when the function is nested.

We follow a standard approach here: given the Haskell-like version of a function f, we inductively define a binary predicate $\_ \downarrow \_$ representing the graph of the function. An element n is in the domain of the f if there exists another element m such that $n \downarrow m$. If n is in the domain of the f, the result of applying the type-theoretic version of f to n (and to the proof that n actually belongs to the domain of f) is the element m such that $n \downarrow m$. In Section 3, we give the formal definition of the graph, the domain and the function itself for the example of McCarthy's 91 function.

In type theory, associated to the inductively defined graph of the function we have an induction principle, which we shall call *graph induction* here, that can be used to prove properties of the function. In Section 4, we revisit the proofs presented in Section 2 and we prove them using graph induction.

Alternatively, we can use graph induction to prove an induction principle, which we shall call *domain induction principle* here, in the style of the induction principle associated to the domain predicate (for the corresponding function) on the Bove-Capretta method. In Section 5, we show how to prove the domain induction principle associated to our example, and we revisit once more the proofs presented in Section 2 and prove them using the domain induction principle.

We conclude in Section 6, where we also discuss some related work.

Some of the proofs we refer to in this text have been placed in Appendix A. This will hopefully make the reading of this paper easier for those not interesting in the details of the actual code of the proofs.

All the definitions and proofs that we present here have been performed in the proof assistant Agda [1], which is based on Martin-Löf's type theory and developed at Chalmers University of Technology by Ulf Norell. Agda syntax is very similar to the one for Haskell and we will not say much about it here, unless we think an explanation is necessary. Function definition is done à la Haskell, that is, with a set of equations and by pattern matching on one or more of the arguments of the function. The difference is that, since type theory is a theory of total functions, the pattern matching must be exhaustive and the recursive calls must be on structurally smaller arguments for the function to be accepted by both the type checker and the termination checker. For documentation, examples and tutorials on the Agda system please consult Agda's wiki page [1]. Ulf Norell's Ph.D. thesis [16] is also a good source of information about Agda and its implementation.

2 The Bove-Capretta Method

Let us start by presenting a Haskell version of McCarthy's 91 function:

```
f n | 100 < n = n - 10
    | n <= 100 = f (f (n + 11))
```

As we have already said, the type-theoretic version of the function following the Bove-Capretta method (which we call `f91` below) takes as an extra argument the proof that the input belongs to the domain of the function (which we call `dom91` below). We now analyse the above definition of `f` in order to identify its domain. It is clear that the function terminates on any element n such that $100 < n$. On the other hand, if n <= 100 then the function terminates on n if it terminates on the arguments $n + 11$ and `f`$(n + 11)$. From this analysis, it is easy to see the simultaneous dependency between the function and its domain in the presence of nested recursive calls.

Assume the standard definitions of the functions $+$ and $-$, and of the relations $<$ and <= over Natural number [1]. Following the Bove-Capretta method, the definitions of the domain and the type-theoretic version of McCarthy's 91 function are given below:

```
mutual
  data dom91 : Nat -> Set where
    dom100< : (n : Nat) -> 100 < n -> dom91 n
    dom<=100 : (n : Nat) -> n <= 100 -> (p : dom91 (n + 11)) ->
               dom91 (f91 (n + 11) p) -> dom91 n
```

[1] By abuse of notation, we denote in the same way the Boolean inequality functions < and <=, and the relations which hold when such inequalities are true. We believe the user can distinguish, depending on the particular context, whether the functions or the relations are being used.

```
f91 : (n : Nat) -> dom91 n -> Nat
f91 ._ (dom100< n _) = n - 10
f91 ._ (dom<=100 n _ p1 p2) = f91 (f91 (n + 11) p1) p2
```

The inductive domain predicate consists of a constructor for each of the cases we identified in the analysis. As it can be seen, the information in each constructor is exactly the one we described before.

The function is defined by recursion on the proof of (`dom91` n). We proceed by pattern matching on this proof and obtain two cases, one for each of the constructors of `dom91`. For each recursive call in the second equation we have to provide a proof that the argument of the recursive call belongs to the domain of the function: these proofs are exactly the two last arguments of the second constructor of `dom91`.

Some remarks about Agda are in order here. Agda can interpret decimal representation of natural numbers. As in Haskell, Agda uses the underscore character "_" to denote an argument which is not needed in the right hand side. The "." notation is not relevant in this work and can be ignored. It is related to the accessibility of patterns and it has been shown here simply to be faithful to the actual Agda code. The interested user can refer to Chapter 2 of Norell's thesis [16] for an explanation about the use of "." in the left hand side of a definition.

As we have already said, the induction principle derived from the inductive predicate `dom91` is very useful when proving properties of the function. Its type is as follows:

```
(P : (n : Nat) -> dom91 n -> Set) ->
((n : Nat) -> (h : 100 < n) -> P n (dom100< n h)) ->
((n : Nat) -> (h : n <= 100) ->
  (p1 : dom91 (n + 11)) -> (p2 : dom91 (f91 (n + 11) p1)) ->
  P (n + 11) p1 -> P (f91 (n + 11) p1) p2 ->
  P n (dom<=100 n h p1 p2)) ->
(n : Nat) -> (p : dom91 n) -> P n p
```

(If we have a proof that `dom91` is satisfied by every natural number, we could eliminate the dependency of the above induction principle on the domain predicate and obtain a simpler induction principle where the predicate has just type `Nat -> Set`.)

In a system like Agda, however, we can express structural induction directly by using pattern matching. According to the Curry-Howard isomorphism, a proof by structural induction corresponds to a definition of a function by structural recursion: recursive calls correspond to the use of induction hypotheses. This way of writing proofs is actually very convenient since the use of pattern matching facilitates the reading and the understanding of the proof being defined.

We demonstrate this with the proofs of two properties about the behaviour of our function. In the first property, we show that n is smaller than the result of the function on n, plus 11. In the second property, we prove that the result of applying the function to n is 91 if $n <= 101$, and $n - 10$ otherwise. In both cases, in order to state the property, we must require a proof that the argument n is in the domain of the function. In both proofs, we use induction on the domain predicate.

(Observe that the first property can be proved without the need of induction but simply by using the second property and some arithmetic properties; however we are interested in inductive proofs here hence the proof we present.)

Before we move into the actual proofs regarding the behaviour of the function, we present the type of some auxiliary lemmas we use; we believe are trivial to understand from their types. Below, `_||_` is the logic disjunction of sets with constructors `inl` and `inr`, and elimination `||-elim`, `_==_` is the propositional equality, `False` is the empty set and `falseElim` its elimination. Curly brackets are used to introduce implicit arguments, that is, those arguments which the system should be able to figure out by itself once the lemma is being applied.

```
co : (n m : Nat) -> n < m || m <= n
<vs<= : {n m : Nat} -> n < m -> m <= n -> False
trans< : {n m l : Nat} -> n < m -> m < l -> n < l
<-10+11 : (n : Nat) -> n < (n - 10) + 11
<+to< : {n m : Nat} -> (l : Nat) -> n + l < m + l -> n < m
<to+1<= : {n m : Nat} -> n < m -> n + 1 <= m
+11-10==+1 : (n : Nat) -> (n + 11) - 10 == n + 1
 91<=100 : 91 <= 100
```

We now show the Agda code of the proof of our first property. We choose to use a let-expression here to make the inductive hypotheses explicit (we shall not do so in subsequent sections).

```
<result : {n : Nat} -> (p : dom91 n) -> n < (f91 n p) + 11
<result (dom100< n _) = <-10+11 n
<result (dom<=100 n _ p1 p2) =
     let ih1 : n + 11 < (f91 (n + 11) p1) + 11
         ih1 = <result p1
         ih2 : f91 (n + 11) p1 < (f91 (f91 (n + 11) p1) p2) + 11
         ih2 = <result p2
     in trans< (<+to< 11 ih1) ih2
```

When $100 < n$, the function returns $n - 10$ and it is easy to show the desired property with the help of some trivial arithmetic properties. When $n <= 100$, the two inductive hypotheses, `ih1` and `ih2`, are proofs of the statement for the two recursive calls. Given these results, it is then easy to show by transitivity of $<$ and simple arithmetic results that the inequality holds even for the argument n.

To show the second property we actually prove three auxiliary lemmas. The types of the first two lemmas are given below; induction is not needed in their proofs.

```
res-100< : {n : Nat} -> (p : dom91 n) -> 100 < n ->
           f91 n p == n - 10

res-==100 : {n : Nat} -> (p : dom91 n) -> n == 100 ->
            f91 n p == 91
```

The proof of the first lemma is trivial. The second lemma uses the first one on the result of both recursive calls. Transitivity and substitutivity of equality (see Section A for the type of these properties), and a few simple arithmetic results are also needed in the proof.

The type of the last auxiliary lemma is presented next, see Section A.1 for the Agda code of its proof:

```
res-<=100 : {n : Nat} -> (p : dom91 n) -> n <= 100 ->
            f91 n p == 91
```

Again we proceed by induction on the proof that (`dom91` n). We get two cases.

The first constructor of `dom91` corresponds to the case where $100 < n$, which contradicts the hypothesis that $n <= 100$.

In the second constructor of `dom91`, we pattern match on the proof that $n <= 100$ and we obtain two cases: either $n < 100$ or $n == 100$ (second and third equation in the code in Section A.1 respectively). For this very last case, we simply call the lemma `res-==100` to obtain the desired proof.

In the remaining case, that is, when $n < 100$, we analyse an intermediate result: the fact that either $100 < n+11$ or $n+11 <= 100$. If $100 < n+11$, we call `res-100<` on the argument of the first recursive call and we do induction on the argument of the second recursive call. If $n + 11 <= 100$, we do induction on the arguments of both recursive calls.

3 The Graph, the Domain and the Function

Following a very standard approach, we first define an inductive relation, which we shall call $\_ \downarrow \_$, representing the graph of the function.

```
data _↓_ : Nat -> Nat -> Set where
  100< : (n : Nat) -> 100 < n -> n ↓ n - 10
  <=100 : (n x y : Nat) -> n <= 100 ->
          n + 11 ↓ x -> x ↓ y -> n ↓ y
```

Given the graph, we now define the domain and the type-theoretic version of the function, which we shall call `Dom91` and `F91` respectively, also in a standard way.

```
Dom91 : Nat -> Set
Dom91 n = Exists Nat (\m -> n ↓ m)

F91 : (n : Nat) -> Dom91 n -> Nat
F91 _ (exists m _) = m
```

Observe that these definitions are no longer mutually dependent: given the graph, first the domain is defined as an existential set, then the function is defined provided its argument satisfies the domain predicate.

We can easily relate the result of the function to the element that witnesses that the input belongs to the domain of the function, that is, the element m for which $n \downarrow m$, and also show that the input is related to both those elements in the graph.

```
result : (n : Nat) -> (p : Dom91 n) -> F91 n p == witness p

im-↓ : (n : Nat) -> (p : Dom91 n) -> n ↓ F91 n p

res-↓ : (n : Nat) -> (p : Dom91 n) -> n ↓ witness p
```

For our example, it is also straightforward to prove that the graph indeed represents a function, and that the result of the function is independent of the actual proof that the input argument belongs to the domain of the function.

```
unique-res : (n r l : Nat) -> n ↓ r -> n ↓ l -> r == l

dom-prf-ind : (n : Nat) -> (p : Dom91 n) -> (q : Dom91 n) ->
               F91 n p == F91 n q
```

If wanted, the recursive equations for the function can be easily derived.

```
eq-100< : (n : Nat) -> (p : Dom91 n) -> 100 < n ->
           F91 n p == n - 10

eq-<=100 : (n : Nat) -> (p : Dom91 n) -> n <= 100 ->
         Exists (Dom91 (n + 11))
                (\p1 -> Exists (Dom91 (F91 (n + 11) p1))
                               (\p2 -> F91 n p ==
                                       F91 (F91 (n + 11) p1) p2))
```

Finally, the non-recursive lemmas `res-100<` and `res-==100` can also be proved with the same types as before (except that we now use `Dom91` and `F91` in place of `dom91` and `f91`, respectively) and very similar proofs.

4 Graph Induction

As we have already said, associated to the inductive definition of the graph we have an induction principle which, combined with the pattern matching facilities of Agda, allows us to prove properties of the defined function in a very elegant way. A revisited proof of `<result` can then be as follows:

```
<result' : {n z : Nat}-> (q : n ↓ z) -> n < z + 11
<result' (100< n h) = <-10+11 n
<result' (<=100 n x y _ p1 p2) =
         trans< (<+to< 11 (<result' p1)) (<result' p2)

<result : {n : Nat} -> (p : Dom91 n) -> n < (F91 n p) + 11
<result (exists _ q) = <result' q
```

Here we use an auxiliary function which is proved by induction on its graph argument. When pattern matching on this argument we obtain two cases, corresponding to the two constructors of a graph. The proof of this auxiliary function has a similar structure than the one presented in the previous section (except that we chose here

not to use a let-expression).

Alternatively, we could prove the main statement directly by first pattern matching on the proof that (`Dom91` n) to obtain (`exists` m q), and then by pattern matching on q (which is a proof that $n \downarrow m$) to obtain the two cases corresponding to the two constructors of $\_ \downarrow \_$ as it is shown below:

```
<result : {n : Nat} -> (p : Dom91 n) -> n < (F91 n p) + 11
<result (exists .(n - 10) (100< n h)) = <-10+11 n
<result (exists .y (<=100 n x y _ p1 p2)) =
   trans< (<+to< 11 (<result (exists x p1))) (<result (exists y p2))
```

The new proof of `res-<=100` is also very similar to that presented in the previous section. Here again, we can choose between having an auxiliary function where we pattern match directly on the graph argument, or having only one function where we first pattern match on the domain argument and then pattern match on its graph component. In Section A.2 we show only this last alternative.

5 Domain Induction Principle

An alternative way to prove properties about our function is to use graph induction to prove an induction principle in the style of the induction principle associated to the predicate `dom91` (see page 4 for its type), and then use this new induction principle to prove the desired properties.

The new domain induction principle, which we here call `domain-ind` is easy to prove by graph induction, see Section A.3.

Using this induction principle to prove our first property results in a compact code.

```
<result : (n : Nat) -> (p : Dom91 n) -> n < (F91 n p) + 11
<result = domain-ind (\n p -> n < (F91 n p) + 11)
                     (\n _ -> <-10+11 n)
                     (\_ _ _ _ q1 q2 -> trans< (<+to< 11 q1) q2)
```

Unfortunately, the new proof of the auxiliary lemma for our second property is not as nice as the previous proofs of it, see Section A.3 Agda code of the lemma.

We could add here that if we use the induction principle associated to the predicate `dom91` to prove this property when referring to the function `f91`, we obtain almost the same code as in Section A.3, the only difference being that in that case we would need a proof that n satisfies `dom91` (instead of `Dom91`) to be passed as an argument to `res-==100`. What this observation tells us is that the difference in the structure and readability of the proof does not depend on the new approach, but on the use of the induction principle to obtain the proof in place of the pattern matching and the structural recursion on an inductively defined argument, for which the Agda system is designed.

6 Conclusions

We presented an alternative method to define partial and general recursive functions in constructive type theory. Here, instead of defining an inductive predicate characterising the domain of the function, we inductively define the graph of the function. From the graph, the domain of the function is defined and thereafter the type-theoretic version of the function, which takes an extra argument with a proof that the input satisfies the domain.

This way of defining functions is as powerful as the one presented by Bove-Capretta with inductive domain predicates in the sense that the induction principle associated to the domain predicate of a particular function in the Bove-Capretta method can be obtained by induction on the graph of that function.

An advantage of this new approach with respect to the Bove-Capretta one is that nested recursive functions can be formalised in the theory without the need of a simultaneous inductive-recursive schema; such a schema is not supported by all constructive type theories.

We have shown our ideas in the formalisation of McCarthy's 91 function and in the proofs of two of its properties. These proofs have been done in two different ways: a) by induction on the graph; b) by first proving an induction principle in the style of the induction principle associated to the domain predicates of the Bove-Capretta method, and then using this principle to prove the properties. Which of these two ways of proving properties is the most adequate depends on the property in question and on the facilities present in the proof assistant for writing proofs. The Agda system is particularly suitable to work with definitions by pattern matching.

In our example, the original equations defining the function were mutually exclusive: either $100 < n$ or $n <= 100$, hence the graph obtained from the Haskell version of the function could be proved to indeed represent a function. This might not always be the case though, but we believe it is clear to the user how the code of the function could be modified so that patterns are mutually exclusive. Observe that exhaustiveness is not required and that it is inherent to the notion of partiality.

Mutual recursion presents no problem; the corresponding graphs will be mutually recursive, while the domains and the functions will be defined as in our example here.

Related Work

During the last decade there has been several ideas on how to deal with partial and general recursive functions in a total setting. We give a rather detailed survey of those ideas in the related work section of a previous paper [6]. Since the publication of that article, there have been a couple of papers that use co-inductive types to model partiality [9,7], and a suggestion to extend type theory with a type of partial recursive functions [8]. See the introduction section of this last article for a survey of other approaches to defining a type of partial functions in type theory.

The idea of using graph induction to prove properties of a function is certainly not new, and it will be impossible to give an accurate account of related work on

this topic. We then concentrate here on related work that use the notion of graph to define a partial function in a total setting, and to prove properties about it.

In [15], McKinna addresses the problem of how to prove properties of a function that has been already defined. In his work, which is done together with C. McBride, McKinna presents the idea of using graph induction for this purpose. While the presentation shows how one can use the Epigram system [14] (a system in the spirit of the Agda system we use here) to make the proofs, the issue he discuses is not really the one we address in here, in the sense that our starting point is how to define the function that we will later verify.

Barthe et al [2] present a tool to define and prove properties of recursive functions in Coq. From a pseudo-code of a function, the tool generates the graph of the function, basically with the same information as the graphs we use here. From this graph, the tool by Barthe et al can define the function in two different ways. In one of the approaches, the domain of the function and the function itself are defined basically as in the Bove-Capretta method. In the second approach, the user must provide a measure and proofs that each recursive call is performed on smaller arguments according to that measure. Given the graph and the measure, the proof obligations the user must prove are automatically generated by the tool. Using well-founded induction, the tool can now define the total function that is given by the graph. This second approach can then only be used for total functions. Among other things, an induction principle is generated for the function.

Using the tool developed by Barthe et al, partial functions can only be represented by using the first approach described above, that is, by defining the domain and the function as in the Bove-Capretta method. Coq type system does not support simultaneous inductive-recursive definitions, so no nested recursive function can be formalised with this tool. Since the second approach described above cannot be applied to partial functions there is no way to deal with partial and nested recursive functions using the tool described in [2].

Krauss also defines a function by first defining its graph, see [12]. As in our work, the resulting function is formalised independently of its termination behaviour and does not need a type system that supports inductive-recursive definitions. Krauss work is performed in the Isabelle/HOL setting and this is the reason behind many of the differences between his work and ours. The graphs defined by Krauss do not use a new variable to represent the result of each recursive call as in our graphs, but a single function variable which is constrained to the graph on all recursive calls. Using the THE operator provided by HOL (defined using Hilbert's indefinite description operator ϵ), the function can be defined from the graph. A domain is derived as the accessible part of the recursion relation in the graph; it contains basically the same information as the domains of Bove-Capretta but since the function has been defined independently of its domain, the domain of a nested function does not need to be simultaneously defined with the function. The domain is mainly used to restrict the applicability of the function. A partial induction rule in the style of the induction principle associated to the inductive domain predicates in the Bove-Capretta method is also derived.

References

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

[2] G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In M. Hagiya and P. Wadler, editors, *8th International Symposium on Functional and Logic Programming, FLOPS'06*, volume 3945 of *LNCS*, pages 114–129. Springer-Verlag, 2006.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Develpment. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[4] A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, volume 2646 of *LNCS*, pages 39–58, April 2002.

[5] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *LNCS*, pages 121–135, September 2001.

[6] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708, February 2005. Cambridge University Press.

[7] A. Bove and V. Capretta. Computation by prophecy. In *Typed Lambda Calculi and Applications TLCA'07*, volume 4583 of *LNCS*, pages 70–83, June 2007.

[8] A. Bove and V. Capretta. A type of partial recursive functions. In O. Mohamed, C. Mu noz, and A. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 102–117. Springer, August 2008.

[9] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[10] T. Coquand and G. Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[11] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[12] A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning, 3rd International Joint Conference, IJCAR'06*, volume 4130 of *LNCS*, pages 589–603. Springer-Verlag, August 2006.

[13] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[14] C. McBride. Epigram. `www.e-pig.org`, 2007.

[15] J. McKinna. McCarthy-Painter induction. Available at `http://www.cs.ru.nl/~james/`, July 2003. Talk given at the Scottish Theorem Provers meeting.

[16] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[17] S. Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, April 2003.

A Missing Agda Codes

In the proofs below, we also use `trans`, which is a proof that the equality is transitive, and `subst` and `subst′`, which are proofs of the substitutivity properties of equality, left to right and right to left respectively. The type of these properties are as follows:

```
trans : {A : Set} -> {x y z : A} -> x == y -> y == z -> x == z

subst : {A : Set} -> (P : A -> Set) -> {x y : A} ->
        x == y -> P x -> P y
```

```
subst' : {A : Set} -> (P : A -> Set) -> {x y : A} ->
         x == y -> P y -> P x
```

In order to understand part of the proofs below, we need to look closer at the definition of the relation $<=$:

```
_<=_ : Nat -> Nat -> Set
n <= m = (n < m) || (n == m)
```

A.1 Proofs with the Bove-Capretta Method

In Agda, we can use the `with` constructor to analyse an intermediate result on the left hand side of a definition. Below, we analyse the result of `co` 100 $(n+11)$ which gives a proof (`inl` h') that $100 < n+11$ or a proof (`inr` h') that $n+11 <= 100$.

```
res-<=100 : {n : Nat} -> (p : dom91 n) -> n <= 100 ->
            f91 n p == 91
res-<=100 (dom100< n h1) h2 = falseElim (<vs<= h1 h2)
res-<=100 (dom<=100 n (inl h) p1 p2) _ with co 100 (n + 11)
... | inl h' =
        let res1 : f91 (n + 11) p1 == n + 11 - 10
            res1 = res-100< p1 h'
            res2 : f91 (n + 11) p1 == n + 1
            res2 = trans res1 (+11-10==+1 n)
        in res-<=100 p2 (subst' (\m -> m <= 100) res2 (<to+1<= h))
... | inr h' =
        let ih1 : f91 (n + 11) p1 == 91
            ih1 = res-<=100 p1 h'
        in res-<=100 p2 (subst' (\m -> m <= 100) ih1 91<=100)
res-<=100 (dom<=100 n (inr h) p1 p2) _ =
          res-==100 (dom<=100 n (inr h) p1 p2) h
```

A.2 Proofs using Graph Induction

As it can be seen, the structure of this proof is very similar to the one above. The differences are due to the type of the domain and the fact that we now do induction on the graph, which is one of the components of the new domain of a function.

```
res-<=100 : {n : Nat} -> (p : Dom91 n) -> n <= 100 ->
            F91 n p == 91
res-<=100 (exists .(n - 10) (100< n h1)) h2 =
          falseElim (<vs<= h1 h2)
res-<=100 (exists .y (<=100 n x y (inl h) p1 p2)) _
          with co 100 (n + 11)
... | inl h' =
        let res1 : F91 (n + 11) (exists x p1) == n + 11 - 10
            res1 = res-100< (exists x p1) h'
```

```
            res2 : F91 (n + 11) (exists x p1) == n + 1
            res2 = trans res1 (+11-10==+1 n)
        in res-<=100 (exists y p2)
                     (subst' (\m -> m <= 100) res2 (<to+1<= h))
... | inr h' = let ih1 : F91 (n + 11) (exists x p1) == 91
                   ih1 = res-<=100 (exists x p1) h'
               in res-<=100 (exists y p2)
                            (subst' (\m -> m <= 100) ih1 91<=100)
res-<=100 (exists .y (<=100 n x y (inr h) p1 p2)) _ =
          res-==100 (exists y (<=100 n x y (inr h) p1 p2)) h
```

A.3 Proofs using the Domain Induction Principle

The new domain induction principle has the following type and proof:

```
domain-ind : (P : (n : Nat) -> Dom91 n -> Set) ->
  ((n : Nat) -> (h : 100 < n) -> P n (exists (n - 10) (100< n h))) ->
  ((n : Nat) -> (h : n <= 100) ->
     (p1 : Dom91 (n + 11)) -> (p2 : Dom91 (F91 (n + 11) p1)) ->
     P (n + 11) p1 -> P (F91 (n + 11) p1) p2 ->
     P n (exists (F91 (F91 (n + 11) p1) p2)
                 (<=100 n (F91 (n + 11) p1) (F91 (F91 (n + 11) p1) p2)
                        h (im-↓ _ p1) (im-↓ _ p2)))) ->
  (n : Nat) -> (p : Dom91 n) -> P n p
domain-ind P ih1 ih2 .n (exists .(n - 10) (100< n h)) = ih1 n h
domain-ind P ih1 ih2 .n (exists .y (<=100 n x y h p1 p2)) =
           ih2 n h (exists x p1) (exists y p2)
               (domain-ind P ih1 ih2 (n + 11) (exists x p1))
               (domain-ind P ih1 ih2 x (exists y p2))
```

As we have already mentioned, the proof of the auxiliary lemma using the domain induction principle is more difficult to follow.

```
res-<=100 : (n : Nat) -> (p : Dom91 n) -> n <= 100 ->
            F91 n p == 91
res-<=100 =
  domain-ind (\n p -> (n <= 100 -> F91 n p == 91))
    (\n h1 h2 -> falseElim (<vs<= h1 h2))
    (\n h1 p1 p2 q1 q2 _ ->
       ||-elim (\_ -> F91 (F91 (n + 11) p1) p2 == 91)
          (\h -> ||-elim (\_ ->  F91 (F91 (n + 11) p1) p2 == 91)
                    (\h' -> q2 (subst' (\m -> m <= 100)
                                       (trans (res-100< p1 h')
                                              (+11-10==+1 n))
                                       (<to+1<= h)))
                    (\h' -> q2 (subst' (\m -> m <= 100)
                                       (q1 h')
```

```
                                  91<=100))
            (co 100 (n + 11)))
(\h -> res-==100 (exists (F91 (F91 (n + 11) p1) p2)
                    (<=100 n (F91 (n + 11) p1)
                           (F91 (F91 (n + 11) p1) p2) h1
                           (im-↓ (n + 11) p1)
                           (im-↓ (F91 (n + 11) p1) p2)))
                 h)
h1)
```

There are a few reasons behind this less readable code, some of them related to the fact that Agda has mainly been design to work with definitions by pattern matching instead of with induction principles and recursive combinators. First, Agda does not provide a way to perform case analysis on the right hand side of a definition, and even though the result of analysing cases on the expression (`co 100 (n + 11)`) is basically the same as the result of performing an `||`-elimination on the same expression, the code is more readable when using the former way. In addition, performing an `||`-elimination instead of pattern matching on a proof that $n <= 100$ makes the code longer and more difficult to read. Finally, in the case where $n == 100$ (last case), we need to call the lemma `res-==100` with a proof that n satisfies `Dom91`; this proof is a bit cumbersome to obtain from the information available from the induction principle.