© 2021 World Scientific Publishing Company
https://doi.org/10.1142/9789811236488_0008

Chapter 8

The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq

Dominique Larchey-Wendling[*,‡] and Jean-François Monin[†,§]

[*] *Université de Lorraine, CNRS, Bâtiment LORIA, BP 239, 54 506 Vandoeuvre-lès-Nancy, France*

[†] *Université Grenoble Alpes, CNRS, Bâtiment IMAG, 700 av. centrale, 38 401 St Martin d'Hères, Grenoble INP, VERIMAG, France*

[‡] *dominique.larchey-wendling@loria.fr*

[§] *jean-francois.monin@univ-grenoble-alpes.fr*

We present the *Braga method* which we use to get verified OCaml programs by extraction from fully specified Coq terms. Unlike structural recursion which is accepted as is by Coq, the Braga method works systematically with more involved recursive schemes, including the non-terminating schemes of partial algorithms, nested or mutually recursive schemes, etc. The method is based on two main concepts linked together: an inductive description of the computational graph of an algorithm and an inductive characterization of its domain. The computational graph mimics the structure of recursive calls of the algorithm and serves both (a) as a guideline for the definition of a domain predicate of which the inductive structure is compatible with recursive calls; and (b) as a conformity predicate to ensure that the Coq algorithm logically reflects the original algorithm at a low-level. We illustrate the Braga method on various concrete recursive algorithms, including unbounded search, "fold-left" from the tail, non-terminating depth-first search, Paulson's normalization algorithm and first-order unification, the last two algorithms being examples of nested recursive schemes. The method allows us to easily show partial correctness and characterize termination in each case, and in addition, the intended OCaml algorithm is faithfully extracted from Coq code. All the results are implemented in Coq and freely accessible on GitHub.

1. Introduction

The ability to describe partial recursive functions which can have non-terminating computations, and to reason on them, is very useful because this is a natural room for many complex algorithms, and usual functional languages don't impose any restriction on termination. In complement, Coq is a proof-assistant celebrated for years for its success in different fields of mathematics and computer science. In particular, it is a tool of choice for the certification of algorithms written in functional programing languages such as OCaml or Haskell thanks to one of its a powerful features called *program extraction*, which can be summarized as follows. A faithful Coq version `prog` of the target program is written in the functional language embedded in Coq. Correctness properties of `prog` are then proved at will and, in the end of the process, an OCaml (say) version of `prog` is automatically extracted. As far as we are confident in this automated extraction, the resulting OCaml program satisfies the expected correctness properties. A well-known impressive example using this technique is the certified compiler for the C language developed in the CompCert project [1].

However, a challenging discrepancy is raised here because at a deep level of the logic implemented by Coq, only total functions encoded by terminating algorithms are allowed. It would be a strong impairment not to be able to encode as Coq functions a larger class of algorithms based on complex recursive schemes, including nested recursion or functions entailing computations that can terminate or loop forever, depending on the effective parameters given as input. In such situation, it is very important in practice to be able to reason (with formal support) on correctness properties *before* getting knowledge or even *in order to* get knowledge on termination issues.

We believe that thanks to the *Braga method*, named as a tribute to our initial summary presentation at TYPES 2018 [2], a large class of functional algorithms which were considered as out of reach before can now be certified. We support this claim here by a number of significant examples illustrating the range of possibilities offered by this approach. We even present a small example of a certified program implementing an algorithm that *cannot* be directly written in OCaml. In addition to this document, the Coq code corresponding to these examples is available at

https://github.com/DmxLarchey/The-Braga-Method.

The Braga method, to be explained in much further details in this chapter, digests and improves previous work developed in the last decades based on well-founded relations, inductive–recursive schemes, etc. However it can altogether be presented in a very short amount of space (see Section 3). In a nutshell, a relational version of the functional program f of interest is written under the form of an inductive relation $\mathbb{G}$ that mimics the structure of recursive calls; an inductive characterization of the domain of f is inferred from $\mathbb{G}$, either as a custom inductive predicate $\mathbb{D}$ or, equivalently, as a binary relation to be managed through the standard accessibility predicate of Coq. The subtle point is to ensure that recursive calls are safely expressed with a structurally smaller domain argument. This can be either automatically obtained using the `inversion` tactic of Coq or, if one prefers an explicit approach, using concise terms where the structural decrease shows up very clearly.

The chapter is organized as follows. For self-containedness, Section 2 presents the necessary background on Coq, including some fine points about structural recursion or the non-interference principle between the universes respectively devoted to observable data and functions on one side, and to their logical properties on the other. The reader in a hurry and already aware of these aspects can skip this section and directly start with Section 3 on page 325 where the basics of the Braga method are presented and illustrated on very simple algorithms which, at first glance, seem inexpressible in Coq because absolutely no clue is available on their convergence domain. Section 4 is devoted to additional tools that provide interesting variants of the Braga method. The first one is based on the constructive version of the generic accessibility predicate based on a binary relation given to it as a parameter, which is used in the Coq standard library to characterize well-foundedness, a standard tool for well-founded recursion. The second one is a technique for simulating induction–recursion in a type theory without this feature — this is currently the case in Coq. Then Sections 5–8 illustrate how the method and its variants can be applied on more complex situations involving: in Section 5, a non-standard approach to the well-known `fold_left` function on lists; in Section 6, depth-first search, another potentially non-terminating algorithm; and in Sections 7 (Paulson normalization algorithm of if-then-else expressions) and 8 (first-order unification), examples of nested recursion, with a presentation of the last ingredient of the Braga method. Finally, the relationship between previous work and our approach is given in Section 9.

2. Background Material

We provide here a light introduction to the main principles under the hood of Coq that should be sufficient for the non-specialist to grasp the main intuitions in the work presented here. This is by no means a somewhat complete presentation of Coq and the interested reader is referred to the abundant literature on the subject, for instance the book by Bertot and Castéran [3].

2.1. *Types, propositions and terms*

Coq is essentially a strongly typed functional programing language, with a very powerful type system called the calculus of (co)inductive constructions (CIC) with Universes. At the same time, Coq is a proof assistant implementing the so-called Curry–Howard–De Bruijn isomorphism, where theorems are types inhabited by their proofs, a central idea to be illustrated in more detail below.

As already suggested, the types in CIC are themselves organized along a hierarchy of universes generically denoted by `Type`, at the bottom of which a special type is of interest for us in this chapter: the sort `Prop` of propositions — we will often use the shorthand $\mathbb{P}$. For data types and functions on them we will use `Type`.

The two basic constructs for defining types are functional types, e.g. $A \to B$, which is the type of functions from A to B, and inductive types whose canonical inhabitants are exhaustively described with special injective[a] functions called constructors. When A and B are propositions, $A \rightarrow B$ is the type of functions returning a proof of B given a proof of A as input. In other words, the arrow $\rightarrow$ is interpreted as the logical implication between propositions.

Among common examples of inductive data types, we have `bool` for Booleans, whose constructors are `true` and `false`, and `nat` for Peano natural numbers, with two constructors noted $0 : \texttt{nat}$ and $\texttt{S} : \texttt{nat} \rightarrow \texttt{nat}$, where `S` represents the successor function. We commonly use digital notation, for example 2 for `S` (`S` 0). Note that an inductive type can be recursive, but it is not mandatory. For instance in `nat`, `S` has an argument of type `nat` but no recursivity is involved in `bool`.

Two special inductive propositions are of interest: `False` which has zero constructor, and then cannot be proved in the empty environment, and `True` which has exactly one constructor called $\texttt{I} : \texttt{True}$, i.e. the proposition `True` is trivially proved by `I`.

We will use a number of shorthands: $\mathbb{B}$ for `bool`, $\mathbb{N}$ for `nat`, $\bot$ for `False` and $\top$ for `True`. Additionally, in Sections 5, 6 and 8, we will use the inductive type of (polymorphic) lists over a given base type $X : \texttt{Type}$, denoted $\mathbb{L}\,X$, and defined as

$$l : \mathbb{L}\,X ::= \texttt{nil} \mid \texttt{cons}\ x\ l, \quad \text{where } x : X$$

in BNF notation. The symbol $[\,]$ is a short notation for the empty list `nil` and the infix notation $x :: l$ represents $(\texttt{cons}\,x\,l)$, i.e. the list l augmented with the value $x : X$ at its head. We assume some familiarity with lists and we will denote that type as $\mathbb{L}\,X$ in the rest of this chapter. Note that the Coq syntax corresponding to the above definition would be:

$$\texttt{Inductive}\ \mathbb{L}\ (X : \texttt{Type}) : \texttt{Type} := [\,] : \mathbb{L}\,X \mid (x : X) :: (l : \mathbb{L}\,X) : \mathbb{L}\,X.$$

Further lists operators and notations include the list $x_1 :: x_2 :: x_3 :: [\,]$ denoted as $[x_1; x_2; x_3]$, appending the lists $l, m : \mathbb{L}\,X$ denoted $l +\!\!+ m$ and satisfying the equations $[\,] +\!\!+ m = m$ and $(x :: l) +\!\!+ m = x :: (l +\!\!+ m)$. The list reversal function $\texttt{rev} : \mathbb{L}\,X \rightarrow \mathbb{L}\,X$ satisfying $\texttt{rev}\,[\,] = [\,]$ and $\texttt{rev}\,(x :: l) = (\texttt{rev}\,l) +\!\!+ [x]$ is also assumed. Finally, we describe a

[a]There are cases where constructors of dependent types are not provably injective but we can ignore these subtleties for the discussion here.

more visual way to introduce inductive definitions, with rules. For lists, this would look like

$$\texttt{Inductive}\ \mathbb{L}\ (X : \texttt{Type}) : \texttt{Type} := \qquad \frac{}{[\,] : \mathbb{L}\,X} \qquad \frac{x : X \quad l : \mathbb{L}\,X}{x :: l : \mathbb{L}\,X}$$

and we hope that the reader will be able to switch between BNF definitions (mostly for simple inductive types), rule-based definitions (mostly for inductive predicates, see later) and the regular Coq syntax when reading source code. As a final comment on lists for now, note that the type parameter X is declared *implicit* in most list operators including $[\,]$, $::$, $+\!\!+$ and `rev`. Hence it is not syntactically present in expressions and is recovered from the context most of the time.

Using function application and other constructs we can form typed terms; $t : T$ states that the term t has type T. For instance we have $0 : \mathbb{N}$ and $\texttt{S}\,0 : \mathbb{N}$. Abstraction, written $\lambda x : X, t$ (following the syntax suggested by Coq's standard library) denotes a function taking an argument x of type X in input, whose body is given by t — x is just a name, whereas t and X can be complex expressions. When the type is clear from the context, it can be omitted. We also use common shorthand notations, for example $\lambda x\,y, t$ for $\lambda x, (\lambda y, t)$ and $(f\ x\ y)$ for $\big((f\ x)\ y\big)$.

Common functions such as negation or conjunction on Boolean values in $\mathbb{B}$ are defined by *pattern matching* using the following syntax:

```
Definition neg (b : 𝔹) : 𝔹 :=
    match b with
        | true  ⇒ false
        | false ⇒ true
    end.
```

Common functions on the type of Peano natural numbers $\mathbb{N}$ such as addition are defined by pattern matching and *recursion,* with the keyword `Fixpoint` in place of the keyword `Definition`:

```
Fixpoint add (n m : ℕ) : ℕ :=
    match n with
        | 0   ⇒ m
        | S p ⇒ S (add p m)
    end.
```

Importantly, only total functions can be defined. In particular, looping computations are forbidden. This imposes an important restriction on recursion: recursive calls are allowed only on *structurally smaller* arguments. On the above example, n is `S` p in the second pattern, hence the recursive call is allowed because p is a strict subterm of `S` p. We go back to this in detail below since it is the central issue tackled in this chapter. Coq provides features for defining notations, for instance `add` x y is noted $x + y$ as usual.

Predicates are functions from a type (or several types) to $\mathbb{P}$. An important special case is equality, which happens to be yet another inductive type, with a single constructor corresponding to reflexivity (equality on X provides the smallest reflexive binary relation on X, and pattern-matching on a proof of equality happens to yield the Leibniz rule).[b]

Universal quantification also corresponds to a functional type. For instance, $\forall n : \mathbb{N}, n = n + 0$ is seen as the type of functions from natural numbers n to proofs of equalities between n and $n + 0$. This is a typical example of *dependent typing*, where the type of the result (the proposition $n = n + 0$) depends on the value n given in input. Indeed, this formula can be proved either by induction on n, or by directly programming a recursive function f on n that starts with a pattern-matching on n; when n is 0, the type of the result is $0 = 0+0$ which reduces to $0 = 0$ by computation of `add`, and then is trivially proved by reflexivity of the $=$ equality predicate. When n is $\mathtt{S}\,p$, the type of the result is $\mathtt{S}\,p = \mathtt{S}\,p + 0$ which reduces to $\mathtt{S}\,p = \mathtt{S}(p + 0)$ by computation, then solved using $p = p + 0$ obtained by a *recursive call* to f, namely $f\,p$. Such a function can be applied to any closed value, e.g. $\mathtt{S}\,(\mathtt{S}\,0)$, providing a proof of $2 = 2 + 0$. If desired, this proof can be then reduced by computation and after two steps, it boils down to a proof of $2 = 2$ by reflexivity. This illustrates that computations can be performed on proofs. In the present case, the result is very small (informally, just "by reflexivity") but in general the result can be a huge proof tree, where many lemmas and theories have been expanded. It is not really an issue, we will soon see why. To close this aspect, remark that the usual principle of induction on $\mathbb{N}$ is itself actually inhabited by a structural recursive function on $\mathbb{N}$.

[b]This approximation of the exact nature of $=$ in Coq is sufficient for our needs.

Another important dependent type is $\exists x : T, P\,x$, which is inhabited by pairs (x, ρ_x), where x, the witness, inhabits X and ρ_x is a proof of $P\,x$. More precisely, it is an inductive type having a single constructor of type $\forall x : X,\ P\,x \to \exists y,\ P\,y$ named `ex_intro`. For the sake of brevity we write (x, ρ_x) for $\texttt{ex\_intro}\,x\,\rho_x$.

Coq provides also Σ-types, denoted by $\{x : X \mid P\,x\}$, which are also inhabited by pairs (x, ρ_x) where $\rho_x : P\,x$. Only the label of the constructor changes, `exist` instead of `ex_intro`. Although the Σ-types $\exists x : T,\ P\,x$ and $\{x : X \mid P\,x\}$ look isomorphic, there is a big difference between them: $\exists x : T,\ P\,x$ is of sort $\mathbb{P}$, whereas $\{x : X \mid P\,x\}$ is of sort `Type`. Remember that while $\mathbb{P}$ is a type, it is also the lowest sort in the `Type` hierarchy of sorts, and these two existential quantifiers, $\exists x, \ldots$ and $\{x : X \mid \ldots\}$ outline an important distinction between sort $\mathbb{P}$ and sort `Type` to be discussed in the next section.

2.2. *Non-interference from `Prop` to `Type`*

We first state the non-interference property which plays a key role in the work presented here.

Pieces of information available in `Prop` cannot be exploited in `Type`.

This informal motto will be expressed with more technical words below. In order to explain its meaning, we consider two similar statements, one expressed with $\exists$ and the next one with a Σ-type.

Assume $x : \mathbb{N}$ and a hypothesis $H_x : \exists n, n + n = x$. Then by pattern-matching on H_x, we can get its two components, that is, $n_0 : \mathbb{N}$ and an equality $\rho : n_0 + n_0 = x$, allowing us to build a proof ρ' of $\texttt{S}\,n_0 + \texttt{S}\,n_0 = \texttt{S}\,(\texttt{S}\,x)$ and then a proof $(\texttt{S}\,n_0, \rho')$ of $\exists n, n + n = \texttt{S}\,(\texttt{S}\,x)$. This proof, reflecting an informal reasoning starting with *let* n_0 *be the number such that...* is implemented by a term $\texttt{match}\ H_x\ \texttt{with}\ (n_0, \rho) \Rightarrow\ \ldots\ (\texttt{S}\,n_0, \rho')\ \texttt{end}$. With an additional abstraction step, we get a function $\lambda H_x : (\exists n, n + n = x),\ \texttt{match} \ldots \texttt{end}$ of type $(\exists n, n + n = x) \to \big(\exists n, n + n = \texttt{S}\,(\texttt{S}\,x)\big)$.

Similarly, an inhabitant of $\big\{n \mid n{+}n = \texttt{S}\,(\texttt{S}x)\big\}$ can be constructed from an inhabitant in $\{n \mid n + n = x\}$, yielding after an abstraction step a function $\Phi_{\text{even}} : \{n \mid n + n = x\} \to \big\{n \mid n + n = \texttt{S}\,(\texttt{S}\,x)\big\}$.

Now, consider the application $\Phi_{\text{even}}\ (3, \rho_3)$ where ρ_3 is a proof of $3{+}3 = 6$. Its computation will return a pair $(4, \rho_4)$ with $\rho_4 : 4{+}4 = 8$.

In a more general situation, we have a function $\Psi : \{x : \mathbb{N} \mid P\,x\} \to \{y : \mathbb{N} \mid Q\,y\}$. The intuitive meaning of the input is a number x packed with a proof of a precondition $P\,x$, and the intuitive meaning of the output is a number y packed with a proof of the constraint $Q\,y$.

Another convenient way to type a function Ψ which takes an x such that $P\,x$ is satisfied and returns a constrained y is

$$\forall x : \mathbb{N},\, P\,x \to \{y : \mathbb{N} \mid Q\,y\}.$$

Here $\{x : \mathbb{N} \mid P\,x\}$ is unpacked, so that we get a function with *two* arguments, x then a proof of $P\,x$. An interesting advantage of this formulation is that Q is in the scope of x, we can then consider a postcondition relating y with x as in this common pattern:

$$\Psi :\ \forall x : X,\, P\,x \to \{y : Y \mid Q\,x\,y\}.$$

Note that in the Braga method, we will use extensively this pattern with a special conformity relation $\mathbb{G}$ for Q and its domain $\mathbb{D}$ for P. Using the infix notation $x \mapsto_{\mathbb{G}} y$ for $\mathbb{G}\,x\,y$ this will then be written as:

$$\Psi :\ \forall x : X,\, \mathbb{D}\,x \to \{y : Y \mid x \mapsto_{\mathbb{G}} y\}.$$

Now, consider a computation of $\Psi\,35\,\rho$ with ρ a proof of $P\,35$. It yields a pair (y, ρ') with ρ' a proof of $Q\,35\,y$. When the computation is completed, both y and ρ' are said to be in *normal form*. What does it mean? From y, a natural number, we get a normal value such as 3141. For ρ' we get a term corresponding to a *normal proof term* as illustrated above on $2 = 2 + 0$ in page 311.

However in practice, we have a different interest in the two parts of this result: we want to know the normal value of the result, for instance, the amount of the income tax to be payed at the end of the year, rather than a complicated expression yielding this value. On the other hand, the normal form of ρ' is of little interest to the end user, who basically wants to know that the result y (say 3141) satisfies the postcondition $Q\,x\,y$, provided the input x (say 35) satisfies the precondition $P\,x$. Potentially interesting aspects of the proof ρ' could be the kind of properties (algebraic, etc.) used in the reasoning, but this has nothing to do with the normal form of ρ'. Indeed, the computation of this normal form can be performed in theory, which is important for meta-theoretical considerations such as the justification of the logical rules used and the consistency of the underlying logical system.

However, in order to ensure that computing on the proof part is actually not necessary, an important principle must be respected: the computation of y from x does not depend on the proofs attached to them. This is the very meaning of the non-interference principle stated at the beginning of this section. This is often stated in the literature by qualifying terms in `Type` as *informative* and statements in $\mathbb{P}$ as *logical* or *non-informative*, though this terminology is somewhat misleading. Intuitively, logical statements behave like secret comments. As those comments live in the same logical framework, where proofs are seen as typed functions, computations *could* be performed on them as well. But we don't want those computations to have an impact on the data returned as outputs. To enforce this non-interference property, Coq applies a very simple rule:

Pattern-matching on a term of sort `Prop`
to construct a term of sort `Type` is forbidden.[c]

Assume for instance that our context provides a data $D_x : \{n \mid n+n = x\}$, expressing that we have a *public* n which is the half of x. Then by pattern-matching, H_x can be freely decomposed into some n and an associated proof, which can then be used to construct an inhabitant of $\{n \mid n + n = \mathtt{S}\,(\mathtt{S}\,x)\}$, witnessing that we can compute the half of $2 + x$. This is the job done by Φ_{even}.

On the other hand, assume that we only have an existential hypothesis $H_x : \exists n, n + n = x$. The point is that an inhabitant (n, ρ_n) of $\exists n, n + n = x$ contains a number n *intended to be hidden* — it is just a helper for expressing that x is even. Nevertheless, H_x can also be decomposed into a secret n and an associated proof, *provided we only try to construct a proof* of another proposition; for instance, saying that $2 + x$ is even as well — as an aside, the latter proof embeds a secret $\mathtt{S}\,n$.

However, $H_x : \exists n, n+n = x$ *cannot* be exploited by the same simple pattern-matching strategy to construct a *data* such as a Boolean value, a natural number, either alone or packed inside a Σ-type. In order to get the half of x and then compute the half of $2 + x$, more work is needed. Essentially, we first write a recursive program that computes the half of an even number, or more accurately, a number

[c]There is a very small number of harmless exceptions, to be discussed later.

```
Fixpoint half $(x : \mathbb{N}) : (\exists n, n+n=x) \to \{n \mid n+n=x\}$ :=
  let $\Phi_{\text{even}} :\ \{n \mid n+n=x\} \to \{n \mid n+n=\mathtt{S}\,(\mathtt{S}\,x)\} := \ldots$
  in match $x$ with
    | $0$          $\Rightarrow \lambda H_0, (0, E_0)$
    | $\mathtt{S}\,0$        $\Rightarrow \lambda H_1, \ldots$ *(absurd case)*
    | $\mathtt{S}\,(\mathtt{S}\,x')$ $\Rightarrow \lambda H_{\mathtt{SS}}, \Phi_{\text{even}}$ (half $x'$ ...)
  end.
```

Fig. 1. Fully specified function computing the half of an even number (sketch).

packed with a proof that it is even, that is a function

$$\mathtt{half} :\ \forall x,\ (\exists n, n+n=x) \to \{n \mid n+n=x\}$$

then we can decompose the result returned by $\mathtt{half}\,x\,H_x$ which inhabits the Σ-type $\{n \mid n+n=x\}$, in order to get the half of x and then compute the half of $2+x$. A sketch of the function `half` is given in Fig. 1. The function Φ_{even} was described at the beginning of Section 2.2. The recursive call needs an effective parameter of type $\exists n, n+n=x'$, to be provided from $H_{\mathtt{SS}} : \exists n, n+n = \mathtt{S}\,(\mathtt{S}\,x')$. When x is 1, we have an absurd case: from $H_1 : \exists n, n+n=1$ it is possible to derive $\bot$. Let us call φ the latter proof of $\bot$. As $\bot$ is an empty (or zero-case) type, a pattern matching on φ provides a fake inhabitant of $\{n \mid n+n=1\}$. This is one of the rare exceptions to the rule given above, since $\bot$ is in sort $\mathbb{P}$ whereas $\{n \mid n+n=1\}$ is in sort `Type`. We come back to this issue in more detail in Section 2.7, where more subtle ways of getting a fake inhabitant in a so-called informative type from a proof of an absurd proposition will be discussed.

2.3. *Harmless eliminations from `Prop` to `Type`*

The rule stated above which strictly forbids eliminations from sort `Prop` to `Type`, or so-called *large eliminations*. It has been relaxed to allow for exceptional and harmless large eliminations. The part of the Coq community which is concerned by these harmless large eliminations from sort $\mathbb{P}$ to sort `Type` usually calls them *singleton eliminations*; see [4] for an up-to-date and comprehensive discussion. However we find this “singleton” denomination a bit misleading and call them “harmless” instead. What qualifies as harmless has a precise meaning, but we here give the intuition of why such large eliminations have been considered acceptable.

Indeed, provided no information of propositional nature can leak into a computation — more precisely propositional information that would allow to choose between diverging computational paths, — then matching on a proof of a proposition in $\mathbb{P}$ to build in term in a `Type` is allowed. This happens when the constructor of the inductive proposition contains only parameters of sort $\mathbb{P}$. Hence typically when there are no constructors at all like for the $\bot$ empty proposition. This also holds for the logical conjunction $A \wedge B$ of which the sole constructor is `conj` $A\,B : A \rightarrow B \rightarrow A \wedge B$, hence `conj` $A\,B$ has two parameters, one is a proof of A, and the other a proof of B, both A and B being of sort $\mathbb{P}$.

However, the case of the logical disjunction $A \vee B$ with two constructors is very illuminating. These constructors are `or_introl` $A\,B : A \rightarrow A \vee B$ and `or_intror` $A\,B : B \rightarrow A \vee B$. Taken separately, both of these constructors could be considered harmless but a pattern matching on a proof of $A \vee B$ would reveal a Boolean information, i.e. which constructor of either `or_introl` and `or_intror` was used in the proof, or else which of A or B has a proof, hence a leak of logical information.

So, if there are two or more constructors for an inductive proposition, an information is indeed hidden in the choice of the constructor, and this information cannot be allowed to leak. Not having more than one constructor could then explain the origin of the singleton elimination terminology. However, note that the proposition $\exists x : X,\ P\,x$ has only one constructor, `ex_intro` $X\,P : \forall x : X,\ P\,x \rightarrow \exists x : X,\ P\,x$, but this constructor has two parameters of which the first, i.e. $x : X$ is of sort `Type`, and not $\mathbb{P}$. It cannot thus be eliminated to build a term in `Type`. This is why we find that the "singleton" qualifier does not properly cover the range of those allowed eliminations from $\mathbb{P}$ to `Type`, and instead, we call them "harmless large" eliminations, or simply "harmless" eliminations.

2.4. *Program extraction*

At this stage, we get functions working on data (in `Type`) packed with correctness proofs (in $\mathbb{P}$), with the additional knowledge which is that computing on proofs is not needed to get the data part of the result.

We can then use an important feature of Coq, allowing us to *extract* from such functions the part which is dedicated to data. To this effect, Coq just erases the code dedicated to proofs. For example, the type of `half` after $\mathbb{P}$-erasure would be $\mathbb{N} \to \mathbb{N}$: the second input for the precondition is erased, as well as the second component of the result (a proof of the postcondition). More generally, the type of a function $\Phi : \forall x : X,\ P\,x \to \{y : Y \mid Q\,x\,y\}$, after $\mathbb{P}$-erasure, becomes $X \to Y$.

However, the term obtained after raw $\mathbb{P}$-erasure is in general not acceptable as a Coq term, because Φ would no longer be a *total* function (over the whole type X). This phenomenon is witnessed on the above version of `half`, which is not defined on odd inputs. Code extraction actually targets mainstream functional languages such as OCaml or Haskell, where partial functions are allowed. For instance, the OCaml code obtained after extraction of `half` is a minor variant of

```
let rec half x =
    let phi_even n = S n
    in match x with
        | 0        → 0
        | S _      → assert false (* absurd case *)
        | S (S x') → phi_even (half x')
```

Note that the extraction process adds an element to be considered to the *Trusted Code Base* (TCB), i.e. the set of programs on which the confidence of a system claimed to be correct relies, on top of the kernel of Coq and the OCaml compiler. Program extraction was introduced in Coq more than 30 years ago by C. Paulin-Mohring [5]. The interested reader may consult a more recent overview by P. Letouzey [6]. Here, we rely on the correctness of the (currently implemented) Coq type-checker (kernel) and extraction mechanism, and consider their own verification/certification to be orthogonal to our work. To lower the TCB, we mention the lively MetaCoq project that deals with those issues [7].

2.5. *Loose additional remarks on Coq*

There is much more to say on Coq. On its theoretical background, the reader has surely noticed the constructive aspects of the logic behind Coq. It is clear that there is no room for a general principle of excluded middle (XM), as far as we work in the realm of data formalized by the universes beyond $\mathbb{P}$. Still, for extraction purposes, XM can be safely used at the level of $\mathbb{P}$, since justifications at this level are carefully erased in extracted programs. Note, however, that corrupting Coq with a contradictory set of axioms, even just in the $\mathbb{P}$ sort, allows for the construction of non-terminating programs in Coq, see Section 2.7 for additional details.

We close this section with a practical remark on the development of functions or proofs in Coq. Coq provides an interactive mode allowing the user to construct a term step by step by the means of *tactics*. Elementary tactics correspond to basic constructs such as λ-abstraction or pattern-matching. On top of them a large number of high level-tactics are available, allowing the user to automate tedious parts or goals solvable by semi-decision procedures. On the opposite side we have a powerful tactic called `refine`, allowing to provide an incomplete proof term where some subterms, to be filled later, are represented by a "_" joker. We often use this style in the work presented here, in order to clearly present the function to be extracted.

2.6. *Structural recursion*

Structural recursion is the very foundation of induction (or recursion) in the inductive type theory of Coq [3]. Except for co-recursion which is somehow dual, every other form of recursion described below ultimately derives from structural recursion. However, at first glance, it looks like it imposes a strong restriction on acceptable fixpoints.

A famous example of structural recursion is the *reverse and append* of lists of type $\mathbb{L}\,X$, a function characterized by the two recursive equations:

$$\texttt{rev\_app}\ l\ [\,] = l \quad \text{and} \quad \texttt{rev\_app}\ l\ (x :: m) = \texttt{rev\_app}\ (x :: l)\ m.$$

It is straightforward to encode those two equations this way in Coq:

```
Fixpoint rev_app {X : Type} (l m : 𝕃 X) {struct m} :=
    match m with
        | []      ⇒ l
        | x :: m' ⇒ rev_app (x :: l) m'
    end.
```

Intentionally, the above code is very verbosely presented to help for the comments below. The function `rev_app` is polymorphic in its X : `Type` parameter which is declared *implicit* by putting braces $\{\ldots\}$ around it instead of optional parentheses $(\ldots)$. It is a simple exercise to show that the identity `rev_app` $l\ m = $ `rev` $m \mathbin{+\!\!+} l$ holds for any values of $l, m : \mathbb{L}\,X$. However, we are not interested in the semantics of the function here but how it illustrates structural recursion.

Let us explain what makes the above `Fixpoint` definition structurally acceptable. The rule which Coq enforces is that one of the two parameters — here the second one m, — must always be *structurally smaller* on any recursive subcall. In general, Coq is able to detect which parameter may structurally decrease although it does not always find the right one. Here, we forced its hand with the optional $\{$`struct` $m\}$ declaration. Note that the rule says that the `struct` parameter must decrease structurally but it says nothing about the other parameters. Also, beware that on every subcall of a given `Fixpoint` definition, it is the same parameter that must decrease structurally.

But what does structural decrease mean? Well, this has a precise definition embedded in the *guard condition* that Coq enforces on `Fixpoint`s. We are not going to describe it in full details but just give the basic intuitions which are sufficient here:

- the `struct` parameter must be typed in an inductive type;
- in any recursive subcall of the body of the `Fixpoint`, the value of the `struct` parameter must be a *subterm* of the input value, according the inductive structure of the type.[d]

Hence typically, the first parameter l in `Fixpoint` `rev_app` does not decrease because there is a subcall where its value is $\_ :: l$. More generally, consider a recursive function *fct* having $n \geq 1$ parameters $x_1, \ldots x_n$ where x_i is expected to be structurally decreasing. For the following definition to be accepted:

`Fixpoint` $\mathit{fct}\ x_1 \ldots (x_i : T) \ldots x_n$ $\{$`struct` $x_i\}$:=
    $\ldots\ (\mathit{fct}\ e_1 \ldots e_n) \ldots$

the expression e_i has to reduce at type checking time to a subterm of x_i. To this effect, e_i may be syntactically smaller (e.g. p if x is $\mathtt{S}\,p$).

[d]Note that subterms are recognized up to the convertibility equivalence relation.

But subterm recognition also traverses `match` constructs, hence a term e_i of the form `match` e'_i `return` T `with` *patterns* `end` where, again, all cases considered in *patterns* reduce themselves to a subterm of x_i, is also recognized as a subterm of x_i. The structural decrease requirement in the guard condition ensures that there is a terminating strategy for the reduction of `Fixpoint`s. This cannot be proved within Coq but has been verified on paper for various versions of the calculus of (Inductive) constructions [8]. Intuitively, terms of inductive types can be seen as well-founded trees and the guard condition ensures that recursive subcalls always get you closer to the leaves of those trees, leaves after which no recursive subcall can occur anymore.

The guard condition is safe for termination, but it also imposes very strong restrictions on the kind of `Fixpoint`s that can be type-checked by Coq. For instance, consider the following equations for the factorial function on $\mathbb{N}_b$, i.e. positive integers in binary representation:

$$\texttt{fact}_b\,0_b = 1_b \quad\text{and}\quad \texttt{fact}_b\,n = n\cdot\texttt{fact}_b\,(n-1) \quad \text{when } n \neq 0_b.$$

Then $n-1$ (the result of a computation of the minus binary function) cannot be recognized as a subterm of n, even though it is provably smaller for the strict order over $\mathbb{N}_b$ (when $n \neq 0_b$). Hence directly encoding this definition as a `Fixpoint` would not be accepted by the Coq type-checker.

However, it is possible to write a Coq function $\texttt{fact}_b$ satisfying the same fixpoint equations, and critically, such that the OCaml program automatically extracted from $\texttt{fact}_b$ Coq term is:

$$\texttt{let rec fact}_b\ n = \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n\cdot\texttt{fact}_b\ (n-1).$$

In this example, it is not too complicated because we could use measure-based or well-founded recursion as explained in Section 4.1, but it can become really tricky when extracting algorithms which are inherently partial algorithms.

Regarding structurally decreasing fixpoints, we will now assume them, i.e. we won't necessarily write the Coq `Fixpoint` definition corresponding to structurally decreasing equations and leave this task to the reader. We just make the critical remark that the structurally decreasing parameter $x_i : T$, although it must belong to an inductive type T, does not need to belong to an *informative* type, i.e. its type T can be of sort $\mathbb{P}$. In that case, extraction magically removes this parameter: termination is statically ensured at type-checking time of the Coq version, provided that inputs satisfy the expected preconditions, then run-time checks are erased in the extracted version.

2.7. *Eliminating (proofs of) the empty proposition (or type)*

We discuss the role played by the empty proposition $\bot$ and the empty type `Empty_set`, both defined as inductive but with no way to construct a *closed* term:

```
Inductive ⊥ : ℙ := .   Inductive Empty_set : Type := .
```

in the common/shared `Init` part of the Coq standard library. Indeed, these predicates have zero/no rule to build a (proof) term for them. Corresponding to this above inductive definition of $\bot$, Coq automatically builds the (non-dependent) eliminator

```
Definition False_rect (T : Type) (f : ⊥) : T :=
    match f : ⊥ return T with end.
```

which allows, from a proof $f : \bot$, to build a term in any given type T : `Type`. The optional `return` T clause can be omitted when Coq is able to infer the type of the result (T in this case). Note that the `match` $f : \bot$ `with end` construct, which is a pattern matching with *zero* patterns, types correctly against any given type.

Moreover, this construct has an additional property of outmost importance for us: it is considered as *structurally smaller than any term of type T* (when T is an inductive type). This is just a special case of the rule given above in Section 2.6 for `match` e'_i `return` T `with` *patterns* `end`: here e'_i is f of type $\bot$, and as $\bot$ has zero constructor, the *patterns* part boils down to nothing.

Note, however, that when T is of sort `Type`, the construct `match` _ : $\bot$ `return` T `with end`, and hence `False_rect`, both contain an elimination from sort $\mathbb{P}$ to sort `Type`, a scheme which is permitted only for harmless eliminations, see Section 2.3.

On the other hand, the construct `match` _ : `Empty_set with end` which also types against any given type, is a regular elimination (not a harmless one), because it proceeds from sort `Type` to sort `Type`. Also, when considering

$$\texttt{Definition False\_ind } (P : \mathbb{P})\ (f : \bot) : P := \\ \quad \texttt{match } f : \bot \texttt{ return } P \texttt{ with end}.$$

which is a restriction of `False_rect` to sort $\mathbb{P}$ sharing the very same code, the elimination is a regular one from sort $\mathbb{P}$ to sort $\mathbb{P}$.

When considering extraction, for all these constructs that match on a term of an empty inductive type, i.e. `match` $\_ : E$ `with end` where E is either $\bot$, `Empty_set` or any other inductive type with no constructor, the extracted code proceeds with raising an exception like in, e.g.

```
let false_rect _ = assert false (* absurd case *)
```

witnessing a situation that is not supposed to occur at runtime.

We now switch to another way to interpret the elimination of empty inductive types computationally: by looping forever — at least, by pretending to do so. We define $\texttt{False\_loop}_\top$, an alternate elimination scheme of $\bot$ to T : `Type`, this time not involving any harmless large elimination:

$$\texttt{Definition False\_loop}_\top\ (T : \texttt{Type})\ (f : \bot) : T := \\ \big(\texttt{fix loop } (x : \top)\ \{\texttt{struct } x\} := \texttt{loop } (\texttt{match } f \texttt{ return } \top \texttt{ with end})\big)\ \texttt{I}.$$

Recall that $\top$ is a simple inductive proposition with one constructor called `I`. The pattern matching on x occurs when building an alternate proof of $\top$, a regular elimination from sort $\mathbb{P}$ to sort $\mathbb{P}$. Typing succeeds because `match` f `with end` types against any type, including $\top$. The satisfaction of structural decrease comes from the rule given above. Indeed, note that using

$$\texttt{fix loop } x\ \{\texttt{struct } x\} := \texttt{loop } x$$

as a replacement for `loop` above would have failed because x is not a (strict) subterm of itself. But in the definition of $\texttt{False\_loop}_\top$, the construct `match` f `return` $\top$ `with end` is recognized both as having type $\top$ and as being structurally smaller than x.

In the above definition of $\texttt{False\_loop}_\top$, $\top$ can be replaced by any inhabited inductive type. An interesting variant is to take ... $\bot$

itself, since a proof a $\bot$ is available, namely f. The definition can then be presented in a slightly simplified way as follows.

```
Definition False_loop⊥ (T : Type) : ⊥ → T :=
  fix loop f {struct f} := loop (match f return ⊥ with end).
```

On the extraction side, f of sort $\mathbb{P}$ will be removed. As functions in OCaml have at least one argument, we explicitly provide an additional one of type `unit`, the inductive type with one element called `tt`.

```
Definition False_loop (T : Type) : ⊥ → T :=
(fix loop t f {struct f} := loop tt (match f return ⊥ with end)) tt.
```

The code extracted from `False_loop` is now very different from that of `False_rect`. We get a forever loop

```
let  false_loop _ = let rec loop _ = loop () in loop ()
```

when applied to any argument of any type. Hence, after extraction, we get another possible computational interpretation of the empty type: *looping forever* instead of abruptly interrupting on an *error*. These correspond to two usual interpretations of partiality.

The above example of `False_loop` invites a side discussion about a misleading extrapolation of the normalization property of Coq.[e] Indeed, we make the following important observation:

The fact that (axiom free) Coq terms are normalizing does not imply *that the corresponding extracted OCaml terms terminate.*

Obviously, the `False_loop` term above and its extraction directly justify this statement as a would be counter-example. It would be incorrect to believe in an implication between Coq term normalization and OCaml normalization because this would forget that while erasing logical contents, the extraction process maps Coq terms to partial OCaml functions in which the logical domain arguments disappear. This could lead to errors — including non-termination — if one applies an extracted function to an argument not satisfying its precondition. This is precisely what could happen with the `loop`

[e]or even strong normalization on important fragments of Coq.

above that has any empty domain. Moreover, as we will discover, the Braga method actually relies on this ability to extract partial algorithms, for which partial correctness properties can then be established.

Extracted programs should normally not hit an absurdity, except of course when called on arguments which do not fit their (Coq) precondition, in which case they might return anything, interrupt or loop forever.[f] From a strict programmer's point of view, exceptions are much better behaved than fake results or loops because you get some control on what went wrong at runtime. However, logically, `False_rect` T _ or a direct `match` _ : $\bot$ `return` T `with end` both contain a harmless elimination (when T : `Type`), which could be viewed as an issue in some contexts [4]. Can we satisfy both a high programming standard (avoiding loops as much as possible) and a high logical standard (avoiding harmless eliminations)? The answer is yes, using `Empty_set` as an intermediate step:

```
Definition False_exc (T : Type) (f : ⊥) : T :=
    match False_loop Empty_set f return T with end.
```

In this case, we first eliminate $\bot$ into `Empty_set` using `False_loop`, so without using harmless elimination, and then `Empty_set` into T using a `match` _ : `Empty_set return` T `with end` construct, again without using harmless elimination because it proceeds from `Type` to `Type`. Extraction wise, we obtain the best of both worlds, i.e.

```
let false_exc _ = assert false (* absurd case *)
```

because the infinite loop, recognized as dead code by the extraction process, is just erased.

This discussion can be seen as a bit technical and peculiar to the typing rules of Coq and the required structural decrease, but we will use these features extensively to produce inversion (or projection) lemmas that satisfy the structural decrease constraint.

The section closes on the following *take-home lesson*: when one needs to eliminate a proof of $\bot$ against a `Type`, one can avoid harmless elimination using `False_loop`, or better `False_exc`. However,

[f]This situation might be avoidable, when it makes sense to extract the application of a function to specific closed arguments, instead of extracting the function itself.

when eliminating $\bot$ against say D : `Prop`, typically when establishing a domain property, then we advise for `False_ind` or a direct `match` _ : $\bot$ `return` D `with end`, especially since these constructs produce terms that are moreover accepted as structurally smaller.

3. The Braga Method

In type-theoretic frameworks such as Coq, where all functions are total, it is still possible to manage partial functions by considering an additional argument in $\mathbb{P}$ containing a proof that the previous arguments are in the expected domain [9, 10]. A first example was provided with the `half` function in Section 2.2, which was intended to be defined only on even numbers. In that case, another option was to relax the requirements and to return, for instance the euclidian quotient of the input by 2, or even an arbitrary value on odd inputs, e.g. 10 for 1, 11 for 3, etc. Such (somewhat cheating) options are not always available. For instance, we define here a predicate `is_cons` on lists and use it to build a function which returns the first element of a non-empty list.

```
Implicit Type l : 𝕃 X.
Definition is_cons l : ℙ := match l with _ :: _ ⇒ ⊤ | _ ⇒ ⊥ end.
Definition head l : is_cons l → X :=
    match l with
        | x :: t ⇒ λG, x
        | _      ⇒ λG, match G with end
    end.
```

In this common pattern, it is important to see that the second argument of head, acting as a precondition (or a guard) is pushed in the result returned by the match construct, which is typical of *dependent pattern matching* where not only the output value depends on the pattern, but also the output type. Each branch is then a function taking a guard as an argument, whose type is made specific according to the case considered. In the first case ($x :: t$), the specialized type of G is $\top$ and is not used. In all remaining cases (denoted by the _ wildcard or *joker*), the type of G is $\bot$, an empty type, allowing us to use `match` G `with end` as a fake inhabitant of X. Avoiding the (sometimes reluctantly accepted) elimination from $\mathbb{P}$ to `Type` here, one could alternatively get a fake inhabitant of X as `False_exc` $X\,G$ from Section 2.7. In both cases, the term G acts like a *Trojan horse* silently carrying an information about the original contents of l, to be revealed and used when needed. We will see many other uses of this idea.

Coming back to recursive functions, we can say that the domain of a partial recursive function corresponds to input values such that the computation actually returns an output, without getting lost in an infinite loop for instance.

The first central idea of the Braga method *is to define this domain* (*denoted* $\mathbb{D}$ *with subscripts*) *using an inductive predicate that mimics the structure of recursive calls.*

We will call these *custom inductive domain predicates* and they make it possible to define and reason on the desired function *before* getting additional knowledge on its actual domain. Even for total functions, proving totality may require preliminary technical partial correctness lemmas, so a usable formal definition is needed first. Such examples will be presented in Sections 7 and 8.

3.1. *Custom inductive domain predicates*

We first illustrate the Braga method on a very simple case where the domain depends on a higher-order argument in a completely uncontrollable way.

Given an arbitrary type X, a function $g : X \to X$, a halting test function $b : X \to \mathbb{B}$, and an initial value $x : X$, we would try to count the minimum number n of iterations of g over x needed to get a point where the test holds, that is $b\,(g^n\ x) = \texttt{true}$; but of course, with arbitrary g, b and x, we don't even know if such an n exists at all. Two algorithms easily come to mind, with or without accumulator, in OCaml syntax:

$$\texttt{let rec ns}\ x = \texttt{if}\ b\ x\ \texttt{then}\ 0\ \texttt{else}\ 1 + \texttt{ns}\,(g\ x),$$
$$\texttt{let rec nsa}\ x\ n = \texttt{if}\ b\ x\ \texttt{then}\ n\ \texttt{else nsa}\ (g\ x)\ (1 + n).$$

A simple question is: does the tail-recursive call `nsa` x 0 always return the same value as `ns` x?

Due to the structural decrease requirement, there is no straightforward way to write down `ns` and `nsa` in Coq, then to state the expected theorem, not to mention proving it. However, it is clear that `ns` and `nsa` have the same domain $\mathbb{D}_{\texttt{ns}}$, which can be inductively expressed because, looking at the definitions, if $b\,x$ is `true` then x is in $\mathbb{D}_{\texttt{ns}}$ and, if $b\,x$ is `false` and $g\;x$ is in $\mathbb{D}_{\texttt{ns}}$, then x is in $\mathbb{D}_{\texttt{ns}}$ as well

`Inductive` $\mathbb{D}_{\texttt{ns}} : X \to \mathbb{P} :=$

$$\frac{b\,x = \texttt{true}}{\mathbb{D}_{\texttt{ns}}\;x}\;[\mathbb{D}^{\texttt{tt}}_{\texttt{ns}}\;x] \qquad \frac{b\,x = \texttt{false} \quad \mathbb{D}_{\texttt{ns}}\,(g\,x)}{\mathbb{D}_{\texttt{ns}}\;x}\;[\mathbb{D}^{\texttt{ff}}_{\texttt{ns}}\;x]\,.$$

We then look at Coq terms with the following shape:

```
Fixpoint  fct x (D : 𝔻_ns x) {struct D} : ℕ :=
    match b x with
        | true  ⇒ . . .
        | false ⇒ . . .  fct (g x) (proj D)  . . .
    end.
```

The point is to find a suitable expression for *proj* D, which is expected to be a proof of $\mathbb{D}_{\texttt{ns}}\;(g\;x)$ structurally smaller than D. We have to be very accurate here. This projection only makes sense for the second inductive rule called $\mathbb{D}^{\texttt{ff}}_{\texttt{ns}}$ and, in this case, D is $\mathbb{D}^{\texttt{ff}}_{\texttt{ns}} x E D_{gx}$, where E is a proof of $b\,x = \texttt{false}$ and D_{gx} a proof of $\mathbb{D}_{\texttt{ns}}\;(gx)$; *proj* D must then be D_{gx} itself.[g] However, as for `head` above, an additional guard argument is needed in order to have a properly defined function even in the irrelevant cases. Looking at the rules for $\mathbb{D}_{\texttt{ns}}$, we can take $b\,x = \texttt{false}$ for the guard and, in the rest of this chapter, *proj* D will be written $\pi_{\mathbb{D}_{\texttt{ns}}}\;D\;G$. The *guarded projection* $\pi_{\mathbb{D}_{\texttt{ns}}}$ is defined as follows, with the help of a basic lemma stating that a Boolean cannot be simultaneously equal to `true` and to `false`.

```
Fact  true_false {x : 𝔹} :    x = true  →  x = false  →  ⊥.
Definition  π_𝔻ns {x} (D : 𝔻_ns x) : b x = false → 𝔻_ns (g x) :=
    match D with
        | 𝔻^tt_ns x E        ⇒ λG, match true_false E G with end
        | 𝔻^ff_ns x E D_gx   ⇒ λG, D_gx
    end.
```

[g] A term isomorphic to D_{gx} would not be enough, Coq is quite fussy about structural ordering. For instance in $\mathbb{N}$, $y := \texttt{S}x$ is a subterm of $t := \texttt{S}y$ as expected, but $\texttt{S}\,x$ is *not* a subterm of $t := \texttt{S}\,(\texttt{S}\,x)$, because here $\texttt{S}\,x$ is reconstructed from x.

The Trojan horse used here is different from the former one used for `head`, that was a term whose type reduced to $\bot$ in the branch, whereas the Trojan horse used for $\pi_{\mathbb{D}_{\texttt{ns}}}$ reduces to a proof G of $b\ x = \texttt{false}$, where x is actually the first component of D when D is $\mathbb{D}^{\texttt{tt}}_{\texttt{ns}}\ x\ E$. Here, G happens to allow us to derive again a proof of $\bot$ but, in general, the purpose of a Trojan horse is to prove specific propositions other than $\bot$. Also, the reader might here recognize the pattern `match` $\ldots : \bot$ `return` P `with end` discussed in Section 2.7 that is both of arbitrary type, here $\mathbb{D}_{\texttt{ns}}\,(g\ x) : \mathbb{P}$, and *structurally smaller than any term which inhabits an inductive type*. Here P is $\mathbb{D}_{\texttt{ns}}\,(g\,x)$, whereas P was, e.g. $\top$ in $\texttt{False\_loop}_\top$ in Section 2.7.

Now, we can write recursive calls by feeding an additional argument containing a proof of $b\,x = \texttt{false}$. To this effect, we use again a Trojan horse which is here a proof of $b\,x = b_x$, where b_x is going to be the constructor (`true` or `false`) corresponding to each case, as specified in the first line of the `match` construct.[h] We then write `ns` and `nsa` as in Figure 2.

That is it! We can then prove the expected lemma as a corollary of a statement generalized on all n.

Lemma	$\texttt{ns\_nsa\_n\_direct} : \forall\, x\, n\, D,\ \texttt{nsa}\ x\ n\ D = \texttt{ns}\ x\ D + n.$
Corollary	$\texttt{ns\_nsa\_direct} : \forall\, x\, D,\ \texttt{nsa}\ x\ 0\ D = \texttt{ns}\ x\ D.$

Proof. The main lemma `ns_nsa_n_direct` is proved by dependent induction on D, implemented as a `Fixpoint`. The proof is very

```
Fixpoint ns $x$ $(D : \mathbb{D}_{\texttt{ns}}\,x)$ {struct $D$} : $\mathbb{N}$ :=
    match $b\,x$ as $b_x$ return $b\,x = b_x \to \_$ with
        | true  $\Rightarrow \lambda\_,\ 0$
        | false $\Rightarrow \lambda G, \texttt{S}\,(\texttt{ns}\ (g\,x)\ (\pi_{\mathbb{D}_{\texttt{ns}}}\ D\ G))$
    end eq_refl.

Fixpoint nsa $x$ $(n : \mathbb{N})$ $(D : \mathbb{D}_{\texttt{ns}}\,x)$ {struct $D$} : $\mathbb{N}$ :=
    match $b\,x$ as $b_x$ return $b\,x = b_x \to \_$ with
        | true  $\Rightarrow \lambda\_,\ n$
        | false $\Rightarrow \lambda G, \texttt{nsa}\ (g\,x)\ (\texttt{S}\,n)\ (\pi_{\mathbb{D}_{\texttt{ns}}}\ D\ G)$
    end eq_refl.
```

Fig. 2. Coq terms for `ns` and `nsa`, by structural recursion on $D : \mathbb{D}_{\texttt{ns}}\,x$.

[h] This corresponds to the trick used for a long time in Coq for the implementation of the tactic `case_eq`.

short because the above definitions of ns/nsa provide the following equalities (even conversions, actually) for free:

$$\begin{array}{rr} \mathtt{ns}\ 0\ \mathbb{D}^{\mathtt{tt}}_{\mathtt{ns}} = 0 & \mathtt{ns}\ x\ (\mathbb{D}^{\mathtt{ff}}_{\mathtt{ns}}\ y\ D) = \mathtt{S}\,(\mathtt{ns}\ (g\,x)\ D), \\ \mathtt{nsa}\ 0\ n\ \mathbb{D}^{\mathtt{tt}}_{\mathtt{ns}} = n & \mathtt{nsa}\ x\ n\ (\mathbb{D}^{\mathtt{ff}}_{\mathtt{ns}}\ y\ D) = \mathtt{ns}\ (g\,x)\ (\mathtt{S}\ n)\ D. \end{array} \tag{1}$$

□

The guarded projection $\pi_{\mathbb{D}_{\mathtt{ns}}}$ can also be obtained in a cheap but (possibly) mysterious way, using the `inversion` tactic of Coq. The reader is invited to display the rather heavy term produced by `inversion` and to guess why the result is structurally smaller as desired (even though Coq says it is so). The explicit yet small version shown above is yet another variation on small inversions [11]. As the needed guarded projection is a special case of inversion, we use indifferently for it the name guarded projection (in general, omitting "guarded" for brevity) or inversion in the rest of this chapter. The algorithm considered here in LISP style, with a recursive call inside an `else` branch. However, in most situations, recursive calls are inside a branch of a more general pattern matching. A more appropriate technique for writing suitable projections will be presented in Section 5.

3.2. *Inductive definition of the graph of a recursive function*

Note that the argument D for the domain is involved in a deep way in the above formalization, which makes it very easy to get lost in a dead end. For instance, the value returned by ns x D seem to depend on the particular proof D given in input. Though it cannot be the case, because informative values do not depend on proofs in the $\mathbb{P}$ universe, this meta-theoretical knowledge cannot be directly exploited and for more complex functions, the presence of D becomes very troublesome. In general, there is no convenient way to derive recursive equations such as the ones given in (1), which provide crucial inference steps.

For this reason, and another related to nested recursion to be developed later, we introduce an additional inductive definition (denoted here by $\mathbb{G}$ with subscripts).

Now the second central idea of the Braga method*: as for its domain $\mathbb{D}$, the inductive relation $\mathbb{G}$ mimics the structure of recursive calls, but in contrast with $\mathbb{D}$, the relation $\mathbb{G}$ takes the output as* *well into account, providing a description of the input-output relation between arguments and result.*

We call this relation the *computational graph* of the function.

3.2.1. *The algorithm without an accumulator*

For instance in the case of `ns`, we have the following inductive rules, with the infix notation $x \mapsto_{\texttt{ns}} y$ for $\mathbb{G}_{\texttt{ns}}\, x\, y$ used as

$$\texttt{Inductive}\ \ \mathbb{G}_{\texttt{ns}} : X \to \mathbb{N} \to \mathbb{P} := \quad \frac{b\,x = \texttt{true}}{x \mapsto_{\texttt{ns}} 0} \qquad \frac{b\,x = \texttt{false} \quad g\,x \mapsto_{\texttt{ns}} o}{x \mapsto_{\texttt{ns}} \texttt{S}\, o}\ .$$

Observe that $\mathbb{G}_{\texttt{ns}}$ is nothing but a relational and agnostic presentation of `ns`, without any claim about termination and partial correctness properties. On the other hand, $\mathbb{D}_{\texttt{ns}}$ is obtained from $\mathbb{G}_{\texttt{ns}}$ just by removing the output. Indeed, favoring the prefix notation $\mathbb{G}_{\texttt{ns}}\ x\ o$ over the infix $x \mapsto_{\texttt{ns}} o$, as side by side comparison gives

$$\begin{array}{ccc}
\dfrac{b\,x = \texttt{true}}{\mathbb{G}_{\texttt{ns}}\ x\ 0} & \rightsquigarrow & \dfrac{b\,x = \texttt{true}}{\mathbb{D}_{\texttt{ns}}\, x} \\[2ex]
\dfrac{b\,x = \texttt{false} \quad \mathbb{G}_{\texttt{ns}}\ (g\,x)\ o}{\mathbb{G}_{\texttt{ns}}\ x\ (\texttt{S}\, o)} & \rightsquigarrow & \dfrac{b\,x = \texttt{false} \quad \mathbb{D}_{\texttt{ns}}\, (g\,x)}{\mathbb{D}_{\texttt{ns}}\, x}\ .
\end{array}$$

The prefix notation makes it particularly straightforward to infer the custom domain predicate $\mathbb{D}_{\texttt{ns}}$ from computational graph $\mathbb{G}_{\texttt{ns}}$: for each rule of $\mathbb{G}_{\texttt{ns}}$, map it to a rule of $\mathbb{D}_{\texttt{ns}}$ by erasing the output/right argument of the $\mathbb{G}_{\texttt{ns}}$ predicate.[i]

Then a property which is both very useful and easy to show is that the computational graph is the graph of a (partial) function, i.e. a deterministic relation.

$$\texttt{Fact}\ \ \mathbb{G}_{\texttt{ns}}\texttt{\_fun}\ x\ o_1\ o_2 : \quad x \mapsto_{\texttt{ns}} o_1 \ \to\ x \mapsto_{\texttt{ns}} o_2 \ \to\ o_1 = o_2.$$

[i]Note that this simple idea of erasing fails with nested recursive algorithms but can be nonetheless circumvented using the graph to recover lost outputs, see Section 7.

Proof. Rewrite it as $\forall x\, o_1,\ x \mapsto_{\mathtt{ns}} o_1 \to \forall o_2,\ x \mapsto_{\mathtt{ns}} o_2 \to o_1 = o_2$ and proceed by induction on $x \mapsto_{\mathtt{ns}} o_1$ and inversion of $x \mapsto_{\mathtt{ns}} o_2$. □

In most practical situations, one first defines the computational graph, then derives the inductive domain from it. The point of defining $\mathbb{G}_{\mathtt{ns}}$ is to enable us to state the type of a slightly enriched version of `ns`, where the type of the result embeds a postcondition expressing that inputs and outputs are related according to $\mathbb{G}_{\mathtt{ns}}$:

$$\forall x,\ \mathbb{D}_{\mathtt{ns}}\ x \to \{o : \mathbb{N} \mid x \mapsto_{\mathtt{ns}} o\}. \qquad (2)$$

A function having this type, called `ns_pwc` (for *packed with conformity* to the computational graph), can then be defined as in Fig. 3. The heart of this code is inside the `refine` tactic, where we can recognize the contents of the expected function and additional stuff related to the structural decrease of D on the one hand, outputting a Σ-type instead of a natural number on the other hand. The positions marked by $\mathcal{O}^?_1$ and $\mathcal{O}^?_2$ denote terms for postconditions to be filled later, using very basic tactics in this case:

- for $\mathcal{O}^?_1$: constructing a proof of $x \mapsto_{\mathtt{ns}} 0$ from a proof of the guard G of type $b\,x = \mathtt{true}$;
- and for $\mathcal{O}^?_2$: constructing a proof of $g\,x \mapsto_{\mathtt{ns}} \mathsf{S}\,o$ from a proof G of $b\,x = \mathtt{false}$ and a proof C_o of $x \mapsto_{\mathtt{ns}} o$.

Note that in the actual Coq code, these marks $\mathcal{O}^?_1/\mathcal{O}^?_2$ are replaced with the _ joker that the `refine` tactic interprets as a hole to be filled later on. Finally, we point out that the proof ends with the keyword `Qed` — as opposed to the keyword `Defined` — registering `ns_pwc` as a term opaque to evaluation. Because `ns_pwc` outputs a result and a

```
Fixpoint ns_pwc $x$ $(D : \mathbb{D}_{\mathtt{ns}}\ x) : \{o \mid x \mapsto_{\mathtt{ns}} o\}$.
Proof. refine(
    match $b\,x$ as $b_x$ return $b\,x = b_x \to$ _ with
        | true  $\Rightarrow \lambda G$, exist _ 0 $\mathcal{O}^?_1$
        | false $\Rightarrow \lambda G$,
                  let $(o, C_o)$ := ns_pwc $(g\,x)$ $(\pi_{\mathbb{D}_{\mathtt{ns}}}\ D\,G)$
                  in  exist _ $(\mathsf{S}\,o)$ $\mathcal{O}^?_2$
    end eq_refl).
    [$\mathcal{O}^?_1$] : now constructor 1.
    [$\mathcal{O}^?_2$] : now constructor 2.
Qed.
```

Fig. 3. Coq proof term `ns_pwc` of the conform-by-construction `ns` algorithm.

proof of its conformity, there is no need to be able to compute with this term: conformity to $\mathbb{G}_{\texttt{ns}}$ is enough to completely characterize the output value w.r.t. the input value.

As for the above mentioned direct definition of `ns`, the domain argument in the recursive call is $\pi_{\mathbb{D}_{\texttt{ns}}}\,D\,G$, we already know that it is structurally smaller than D. This termination certificate can also be delayed with a _ joker if needed.[j]

Using the projections π_1 and π_2 of the standard library available on Σ-types, we derive

$$\texttt{Definition ns}\ x\ (D : \mathbb{D}_{\texttt{ns}}\,x) := \pi_1(\texttt{ns\_pwc}\ x\ D).$$
$$\texttt{Fact ns\_spec}\ x\ (D : \mathbb{D}_{\texttt{ns}}\,x) : \quad x \mapsto_{\texttt{ns}} \texttt{ns}\ x\ D.$$

where $\pi_2(\texttt{ns\_pwc}\ x\ D)$ is used as witness of conformity of the output value. The OCaml code automatically extracted from `ns` is as expected.[k]

$$\texttt{let rec ns}\ x = \texttt{match}\ b\,x\ \texttt{with true} \to 0 \mid \texttt{false} \to \texttt{S}\,\big(\texttt{ns}\,(g\,x)\big).$$

3.2.2. *The algorithm using an accumulator*

Next we proceed in the same way with the second function. Its recursive equations are encoded in the computational graph:

$$\texttt{Inductive}\ \mathbb{G}_{\texttt{nsa}} : X \to \mathbb{N} \to \mathbb{N} \to \mathbb{P} :=$$
$$\frac{b\,x = \texttt{true}}{x; n \mapsto_{\texttt{nsa}} n} \qquad \frac{b\,x = \texttt{false} \quad g\,x; \texttt{S}\,n \mapsto_{\texttt{nsa}} o}{x; n \mapsto_{\texttt{nsa}} o}\ .$$

Again, we use the mixfix notation $x; n \mapsto_{\texttt{nsa}} o$ to denote the predicate $\mathbb{G}_{\texttt{nsa}}\,x\,n\,o$ and we show that $\mathbb{G}_{\texttt{ns}}$ and $\mathbb{G}_{\texttt{nsa}}$ are related as follows:

$$x \mapsto_{\texttt{ns}} o \ \to\ x; 0 \mapsto_{\texttt{nsa}} o. \tag{3}$$

This is a special case of $x \mapsto_{\texttt{ns}} o \ \to\ \forall n,\ x; n \mapsto_{\texttt{nsa}} o + n$, which we prove by induction on $x \mapsto_{\texttt{ns}} o$.

[j]See, e.g. the example of depth-first search in Fig. 17 on page 354.

[k]Non-essential remark: this is the case if g and b are declared with the keyword `Parameter`, making them constants to be realized at extraction time. Otherwise, parameters g and b are added to `ns` according to a scoping feature of Coq called `Section` and then appear in the actual extracted code.

```
Fixpoint nsa_pwc $x$ $n$ $(D : \mathbb{D}_{\texttt{ns}}\, x) : \{o \mid x; n \mapsto_{\texttt{nsa}} o\}$.
Proof. refine(
    match $b\,x$ as $b_x$ return $b\,x = b_x \to$ _ with
        | true  $\Rightarrow \lambda G,$ exist _ $n$ $\mathcal{O}_1^?$
        | false $\Rightarrow \lambda G,$
                    let $(o, C_o)$ := ns_pwc $(g\,x)$ $(\mathsf{S}\,n)$ $(\pi_{\mathbb{D}_{\texttt{ns}}}\, D\, G)$
                    in  exist _ $o$ $\mathcal{O}_2^?$
    end eq_refl).
    [$\mathcal{O}_1^?$] : now constructor 1.
    [$\mathcal{O}_2^?$] : now constructor 2.
Qed.
```

Fig. 4. Coq proof term `nsa_pwc` of the conform-by-construction `nsa` algorithm.

The domain of `nsa` does not depend on n, so we still use $\mathbb{D}_{\texttt{ns}}$ to define a function `nsa_pwc` : $\forall x\, n,\ \mathbb{D}_{\texttt{ns}}\, x \to \{o \mid x; n \mapsto_{\texttt{nsa}} o\}$, fully displayed in Fig. 4, and along the same lines as for `ns_pwc`. Then we get `nsa` : $\forall x\, n, \mathbb{D}_{\texttt{ns}}\, x \to \mathbb{N}$ which satisfies $\forall x\, n\, D,\ x; n \mapsto_{\texttt{nsa}} \texttt{nsa}\ x\ n\ D$ by projecting the output Σ-type.

Finally we can reason on $\mathbb{G}_{\texttt{nsa}}$ to prove properties on `nsa`. A first useful property of $\mathbb{G}_{\texttt{nsa}}$ is its determinism, i.e.

$$\texttt{Fact}\ \ \mathbb{G}_{\texttt{nsa}}\texttt{\_fun}\ x\ n\ o_1\ o_2 :\quad x; n \mapsto_{\texttt{nsa}} o_1 \ \to\ x; n \mapsto_{\texttt{nsa}} o_2 \ \to\ o_1 = o_2.$$

Proof. We proceed by induction on $x; n \mapsto_{\texttt{nsa}} o_1$ and then by inversion of $x; n \mapsto_{\texttt{nsa}} o_2$. □

In addition to the conformity of `nsa` w.r.t. $\mathbb{G}_{\texttt{nsa}}$, we also need its completeness, that is, $x; n \mapsto_{\texttt{nsa}} o \to \forall D,\ o = \texttt{nsa}\ x\ n\ D$. This is an easy consequence of the determinism of $\mathbb{G}_{\texttt{nsa}}$ and of the conformity of `nsa` w.r.t. $\mathbb{G}_{\texttt{nsa}}$. The desired theorem $\forall x D, \texttt{nsa}\ x\ 0\ D = \texttt{ns}\ x\ D$ follows by combining the conformity of `ns`, property (3) and the completeness of `nsa`.

3.3. *Low-level and high-level properties*

We can now prove the low-level termination property of `ns`: the domain $\mathbb{D}_{\texttt{ns}}$ is as large as possible, encompassing exactly the input values x for which an output value o such that $x \mapsto_{\texttt{ns}} o$ exists, i.e. the projection of the computational graph $\mathbb{G}_{\texttt{ns}}$.

$$\texttt{Fact}\ \ \mathbb{D}_{\texttt{ns}}\texttt{\_p}\mathbb{G}_{\texttt{ns}} :\quad \forall x : X,\ \mathbb{D}_{\texttt{ns}}\, x \leftrightarrow \exists o : Y,\ x \mapsto_{\texttt{ns}} o.$$

Proof. For the *only if* direction, the required value is obviously $\mathtt{ns}\,x\,D$ where $D : \mathbb{D}_{\mathtt{ns}}\,x$, i.e. because of `ns_spec`, a value o s.t. $x \mapsto_{\mathtt{ns}} o$ is precisely what `ns` outputs on its domain. For the *if* direction, it is enough to show $\forall x\, o,\ x \mapsto_{\mathtt{ns}} o \to \mathbb{D}_{\mathtt{ns}}\,x$ and we proceed by induction on the proof of the graph predicate $x \mapsto_{\mathtt{ns}} o$. □

The process we followed so far is somehow automatic, meaning that we only use the syntactic information available for the algorithm `ns`. As a consequence, manipulating $\mathbb{D}_{\mathtt{ns}}$ either directly through its constructors or as the projection of $\mathbb{G}_{\mathtt{ns}}$ are not high-level ways to manipulate the domain.

Of course, one needs human intervention to design interesting/useful alternative characterizations. In the case of $\mathbb{D}_{\mathtt{ns}}$, we can for instance show:

$$\texttt{Fact}\ \mathbb{D}_{\mathtt{ns}}\texttt{\_high\_level}\ (x : X) : \quad \mathbb{D}_{\mathtt{ns}}\,x \leftrightarrow \exists n : \mathbb{N},\ b\,(g^n\,x) = \mathtt{true}$$

since a call to g on x generates a sequence of subcalls $g^0(x)$, $g^1(x), g^2(x), \ldots$ until the first of those input values gives b the value `true`. Note that the above result could be strengthened further because `ns` actually computes the first possible match for $b\,(g^n\,x) = \mathtt{true}$, if there is one at all; see `ns_partially_correct` in the Coq code.

4. Accessibility, Well-foundedness and Induction–Recursion

The main tool for ensuring termination in the Braga method is the inductive definition of a suitable domain $\mathbb{D}$ derived from the code of a functional algorithm under study f, together with associated structurally decreasing projection functions $\pi_{\mathbb{D}}$ as illustrated in the previous sections. However, a traditional approach to recursion is to guess a well-founded relation R which is expected to support the termination of f in all cases. These two views can be reconciled to some extent by focussing on the constructive definition of a generic accessibility predicate `Acc` parameterized by R, which is the main ingredient in Coq for defining well-founded relations. The usual approach to defining well-founded recursive functions in Coq consists in providing a suitable R as an eureka, then to prove that R is well-founded and finally to feed a standard high-level feature of Coq (e.g. `Program Fixpoint` or `Equations`) with R.

Instead of directly writing the domain $\mathbb{D}$ as a custom inductive predicate, an alternate approach is possible, by defining first a binary relation $\prec_f^{\mathrm{sc}}$ along similar lines, again by looking at the shape of the recursive calls in f. When $\prec_f^{\mathrm{sc}}$ happens to be well-founded, tools inspired by the traditional approach can be used as well.

Once again, a strong point of the Braga method is that it works even when $\prec_f^{\mathrm{sc}}$ *is* not *well-founded.*

This distinguishes the Braga method from the above mentioned approaches because it allows to postpone the study of termination, as long as needed.[1]

In this variant of the Braga method, `Acc` is seen as a generic $\mathbb{D}$ predicate parameterized by $\prec_f^{\mathrm{sc}}$. An interesting benefit of this variant is that the key projection function, to be used in recursive calls for building a structurally smaller domain argument, is defined once for all: it is just `Acc_inv` of the standard Coq library. In the opposite direction, one can also consider `Acc` as a special inductive relation and `Acc_inv` as a particular (though important) case of a projection function $\pi_{\mathbb{D}}$. Things are partly simplified because `Acc` has a single constructor. However, a light contribution of the second author to the Coq standard library (in `Logic/ConstructiveEpsilon.v`) shows that a dedicated domain predicate sometimes provides code which can compete with `Acc`.

This section ends with an introduction to induction–recursion, which can be used in association with the Braga method to write fixpoint equations of the recursive function under study.

4.1. *Well-founded recursion*

Well-founded recursion is a principle that allows to justify termination of recursive calls based on a well-founded order (or relation). Considering a relation $R : X \to X \to \mathbb{P}$, it is well-founded if no infinite descending chain of the form $\ldots\ R\ x_n\ R\ \ldots\ R\ x_1\ R\ x_0$ exists

[1]This does not make, e.g. `Equations` incompatible with the Braga method at all. In fact, `Equations` can perfectly be used in conjunction with it.

in the type X. This can be refined by defining the *well-founded part* of the relation R as the x_0 which are not the starting points of infinite descending chains, and then simply characterizing well-founded relations as those where the well-founded part is the whole type X.

The classical characterization of the well-founded part of R is given an inductive counterpart in Coq using the *accessibility predicate*:

$$\texttt{Inductive Acc}\ \{X : \texttt{Type}\}\ (R : X \to X \to \mathbb{P})\ (x : X) : \mathbb{P} :=$$

$$\frac{\forall y : X,\ R\,y\,x \to \texttt{Acc}\,R\,y}{\texttt{Acc}\,R\,x}\ [\texttt{Acc\_intro}]$$

and one can indeed show that $\texttt{Acc}\,R\,x_0$ entails no infinite descending chain starts at x_0. However, the converse only holds under some classical assumptions, typically excluded-middle and dependent choice. Hence the `Acc` predicate is usually considered the proper way to characterize well-foundedness in inductive type theory.

$$\texttt{Definition well\_founded}\ \{X\}\ (R : X \to X \to \mathbb{P}) := \forall x : X,\ \texttt{Acc}\,R\,x.$$

Defined this way, `well_founded` satisfies most of the closure properties of (the classical characterization of) well-foundness including the (transfinite) recursion principle:

$$\begin{aligned}&\texttt{Theorem well\_founded\_induction\_type}\ \{X\ R\}\ (\_ : \texttt{well\_founded}\ R) :\\ &\quad \forall P : X \to \texttt{Type},\ \big(\forall x : X, (\forall y : X,\ R\,y\,x \to P\,y) \to P\,x\big)\\ &\qquad \to \forall x : X,\qquad\qquad\qquad\qquad P\,x.\end{aligned}$$

A way to read this statement is the following: each time one needs to show $\forall x, P\,x$, i.e. provide a dependent function mapping $x : X$ to a value in type $P\,x$, one can further assume the induction hypothesis $IH_x : \forall y,\ R\,y\,x \to P\,y$ at x, which provides $P\,y$ for all the values $y : X$ that are R-smaller than x.

In many cases, the programmers seek a simple relation R of the form $R := \lambda x\,y : X, \lfloor x \rfloor < \lfloor y \rfloor$ where $\lfloor\cdot\rfloor : X \to \mathbb{N}$ is a $\mathbb{N}$-based measure and $< : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ is the strict natural order. For instance, $\texttt{fact}_b$ algorithm of Section 2.6 or breadth-first search algorithms can be implemented using measure-based induction [12].

Note that although it is a very common strategy, it is not always applicable, e.g. the decreasing measure might simply not be total computable, as in the case of the *Tortoise and the Hare* algorithm [13]. In such case, one could of course use Hilbert's description operator as is done in HOL4 for instance [14], but at the cost of adding a non-logical axiom to Coq that is highly incompatible with the constructive world view, and potentially inconsistent with other logical axioms.[m]

Although well-founded recursion via `well_founded_induction_type` is more general than measure-based recursion to define non-structurally recursive functions in Coq, it has a major drawback: one needs to devise the well-founded relation R *before* actually defining the recursive function.

First of all, it might be the case that no such well-founded relation exists, typically for partial algorithms. But even for totally defined functions, complications might become unbearable when writing nested recursive functions that call themselves on their own output values like, e.g. McCarthy's F91 function [15].

4.2. *Accessibility-based recursion*

Coming to theoretical foundations of the herein called Braga method, we revert back to the definition of the `Acc` predicate. It allows to implement and extract not only total functions but also *partial functions* via its fully-dependent recursor:

$$\begin{aligned}&\texttt{Theorem}\ \ \texttt{Acc\_rect}'\ X\,R\ (P : \forall x,\ \texttt{Acc}\,R\,x \to \texttt{Type}) :\\ &\quad\Big(\forall x\ A_x, \big(\forall y\ (H_{yx} : R\,y\,x),\ P\ y\ (\texttt{Acc\_inv}\,A_x\,y\,H_{yx})\big) \to P\,x\,A_x\Big)\\ &\quad \to \forall x\ A_x, \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad P\,x\,A_x\end{aligned}$$

which reads quite differently than `well_founded_induction_type` above. Indeed, the well-foundedness of R has disappeared and instead we witness the accessibility $A_x : \texttt{Acc}\,R\,x$ of x as an extra argument.

But before describing further the interpretation of the type of `Acc_rect′`, let us recall `Acc_inv`, the inversion/projection lemma for

[m]That is, such an addition could silently corrupt Coq to the point where $\bot$ becomes provable.

the `Acc` predicate implemented with a trivial pattern matching:

$$\texttt{Definition Acc\_inv}\ \{X\,R\}\ x\ (A_x : \texttt{Acc}\,R\,x) : \forall y,\ R\,y\,x \to \texttt{Acc}\,R\,y :=$$
$$\texttt{match}\ A_x\ \texttt{with Acc\_intro}\ \_\ H \Rightarrow H\ \texttt{end}.$$

This definition ensures that whenever one applies $\texttt{Acc\_inv}\,A_x$ to any y such that $R\,y\,x$ one can get a proof of $\texttt{Acc}\,R\,y$ which is also structurally smaller than $A_x : \texttt{Acc}\,R\,x$.

Now we give a possible interpretation of $\texttt{Acc\_rect}'$ as an induction principle for defining a partial function f. Let us assume that we can somehow ensure the identity $\mathbb{D}_f = \texttt{Acc}\,R$ between the intended domain $\mathbb{D}_f$ of f and the accessibility predicate $\texttt{Acc}\,R$. We then write $D_x : \mathbb{D}_f\,x$ instead of $A_x : \texttt{Acc}\,R\,x$ and we are in position to define a partial, dependent function

$$f :\ \forall x\ (D_x : \mathbb{D}_f\,x),\ P\ x\ D_x.$$

In this case, applying $\texttt{Acc\_rect}'$ reads as following: provided x and a proof $D_x : \mathbb{D}_f\,x$, while building a value in $P\,x\,D_x$ we can further assume the induction hypothesis at x

$$I\!H_x :\ \forall (y : X)\ (H_{yx} : R\,y\,x),\ P\ y\ (\texttt{Acc\_inv}\,D_x\,y\,H_{yx}).$$

That is, we can assume a value in $P\,y\,D_y$ for every y that is R-below x, where $D_y := \texttt{Acc\_inv}\,D_x\,y\,H_{yx}$ is a particular proof for $\mathbb{D}_f\,y$ build from D_x and $H_{yx} : R\,y\,x$. Further note that the type family P may depend not only on x but also on the proof D_x of $\mathbb{D}_f\,x$.

We follow up with a detailed review of the code of $\texttt{Acc\_rect}'$ because it contains important ideas that the Braga method also makes use of. For $P : \forall x,\ \texttt{Acc}\,R\,x \to \texttt{Type}$ satisfying the assumption

$$H_P : \forall x\ A_x,\ \big(\forall y\ (H_{yx} : R\,y\,x),\ P\ y\ (\texttt{Acc\_inv}\,A_x\,y\,H_{yx})\big) \to P\,x\,A_x$$

we may define $\texttt{Acc\_rect}'$ as the following fixpoint:

$$\texttt{Fixpoint Acc\_rect}'\ x\ (A_x : \texttt{Acc}\ R\ x)\ \{\texttt{struct}\ A_x\} : P\ x\ A_x :=$$
$$H_P\ x\ A_x\ \big(\lambda y\ H_{yx},\ \texttt{Acc\_rect}'\ y\ (\texttt{Acc\_inv}\ A_x\ y\ H_{yx})\big).$$

This code is a slight variant of the one occurring in Coq's standard library module `Wf` under the name `Fix_F`. It shows precisely how *structural recursion* is used to achieve `Acc`-based recursion and *a fortiori* well-founded recursion. The structurally decreasing argument in the definition of $\texttt{Acc\_rect}'$ is the proof $A_x : \texttt{Acc}\,R\,x$ and the guardedness condition is ensured by the pattern-matching on A_x performed inside the `Acc_inv` term: `Acc_inv` $A_x\ y\ H_{yx}$ is recognized as a subterm of A_x. For Coq specialists, we also point out that `Acc_rect`$'$ *does not* perform harmless (large) elimination: there is no elimination from $\mathbb{P}$ to `Type` because `Acc_inv` is applied only when building the `struct` argument of sort $\mathbb{P}$, i.e. this is just a regular elimination from $\mathbb{P}$ to $\mathbb{P}$.

But, these theoretical considerations put aside, aren't we back to square one? We still need to find R such that $\mathbb{D}_f$ and `Acc` R match, or at least that `Acc` R covers the domain $\mathbb{D}_f$.

Fortunately, concrete algorithms like those defined by recursive equations always contain a *canonical relation* that can be used for R. This is the *recursive subcall/call* relation below denoted by the $\preccurlyeq^{\mathrm{sc}}$ infix symbol. To understand this characterization of the domain $\mathbb{D}_f =$ `Acc` $\preccurlyeq^{\mathrm{sc}}_f$ of f, one could think in classical terms where `Acc` $\preccurlyeq^{\mathrm{sc}}_f x$ holds for the values x such that no infinite $\preccurlyeq^{\mathrm{sc}}_f$-decreasing sequence exists. As $(\cdot) \preccurlyeq^{\mathrm{sc}}_f x$ captures precisely the direct recursive subcalls that can be triggered by a call at x, `Acc` $\preccurlyeq^{\mathrm{sc}}_f x$ means termination of any sequence of recursive subcalls starting from x, hence the termination of the computation at x.

4.3. *The domain as subcall/call accessibility*

We illustrate this characterization of the domain $\mathbb{D}_f =$ `Acc` $\preccurlyeq^{\mathrm{sc}}_f$ on the previous example of `ns` of Section 3.1, and we later show why this example challenges well-founded recursion. Consider the following algorithm described by the OCaml program:

$$\texttt{let rec ns}\ x = \texttt{if}\ b\,x\ \texttt{then}\ 0\ \texttt{else}\ 1 + \texttt{ns}\,(g\,x),$$

where $b : X \to \mathbb{B}$, $g : X \to X$ are already defined total functions. If one picks R a relation for which $R\,(g\,x)\,x$ holds for any $x : X$, then using via $IH_x\ (g\,x)$, one can access the value `ns` $(g\,x)$ while defining `ns` x.

Of course, one cannot simply choose any such relation R because it may well be that $R\,x\,x$ holds for any x and thus `Acc` R would give an empty domain.[n] To avoid such a situation, we pick the smallest

[n]Think, e.g. $g\,x = x$.

possible relation $\preccurlyeq^{\mathrm{sc}}_{\mathtt{ns}} : X \to X \to \mathbb{P}$ linking calls with subcalls that actually occur, here simply defined by the single inductive rule

$$\texttt{Inductive}\ \preccurlyeq^{\mathrm{sc}}_{\mathtt{ns}} : X \to X \to \mathbb{P} := \quad \frac{b\,x = \mathtt{false}}{g\,x \preccurlyeq^{\mathrm{sc}}_{\mathtt{ns}} x}\ .$$

Note the $b\,x = \mathtt{false}$ premise which restricts the rule on the actual recursive calls, i.e. the subcall $\mathtt{ns}\,(g\,x)$ does not occur when $b\,x = \mathtt{true}$.

Given this definition of $\mathbb{D}'_{\mathtt{ns}}$ as $\mathtt{Acc} \preccurlyeq^{\mathrm{sc}}_{\mathtt{ns}}$, we can use $\mathtt{Acc\_rect}'$ to give a first implementation in *proof style*:

```
Definition ns_Acc : ∀x, D'_ns x → ℕ.
Proof.
    induction 1 as [x _ IH_D] using Acc_rect'.
    case_eq (b x); intros G.
    + exact 0.
    + apply S, (IH_D (g x)).
      now constructor.
Defined.
```

However, this definition makes it really hard to prove some critical properties of the resulting term $\mathtt{ns}_{\mathtt{Acc}}$. For instance, we would like to be able to show the equation $\mathtt{ns}_{\mathtt{Acc}}\,x\,D = 0$ whenever $b\,x = \mathtt{true}$ holds, and the fixpoint equation $\mathtt{ns}_{\mathtt{Acc}}\,x\,D = \mathtt{S}\big(\mathtt{ns}_{\mathtt{Acc}}\,(g\,x\,D')\big)$ for some $D' : \mathbb{D}'_{\mathtt{ns}}\,x$ when $b\,x = \mathtt{false}$. But this can be very difficult because opaque proof terms often stand in the way of the evaluation that would normally give them to us for free, as reflexive identity. To make those proof terms transparent might involve opening a large amount of proof terms of lemmas of the standard library (due to dependencies), and such proofs might involve very large terms saturating the type-checker, which is precisely the reason why they were made opaque in the first place.

Another critique is that the above term $\mathtt{ns}_{\mathtt{Acc}}$ somehow hides the fixpoint computation behind $\mathtt{Acc\_rect}'$ of which, unless inlined, the code is not visible. To solve both of these problems, we use the computational graph $\mathbb{G}_{\mathtt{ns}} : X \to \mathbb{N} \to \mathbb{P}$ as defined in Section 3.2 encoding the relation $x \mapsto_{\mathtt{ns}} o$ to be read as $\mathtt{ns}$ terminates on input value x and outputs the value o, or $\mathtt{ns}\,x = o$ for short. Instead of just outputting a value of type $\mathbb{N}$, we write the fully specified $\mathtt{ns\_pwc}_{\mathtt{Acc}}$ version of `ns`, packed with correctness as

$$\texttt{ns\_pwc}_{\texttt{Acc}} : \forall x : X,\ \mathbb{D}'_{\texttt{ns}}\, x \to \{o : \mathbb{N} \mid x \mapsto_{\texttt{ns}} o\}.$$

We furthermore inline `Acc_rect'` inside the definition of $\texttt{ns\_pwc}_{\texttt{Acc}}$ to fully display the computational content of the term in Fig. 5. We can then project the output Σ-type to get

$$\texttt{Definition}\ \ \texttt{ns}\ x\ (D : \mathbb{D}'_{\texttt{ns}}\ x) := \pi_1(\texttt{ns\_pwc}_{\texttt{Acc}}\ x\ D).$$

and its specification

$$\texttt{Fact}\ \ \texttt{ns\_spec}\ x\ (D : \mathbb{D}'_{\texttt{ns}}\ x) : \quad x \mapsto_{\texttt{ns}} \texttt{ns}\ x\ D.$$

with $\pi_2(\texttt{ns\_pwc}_{\texttt{Acc}}\ x\ D)$ containing the conformity proof of the output value.

We can also recover the "natural" constructors mimicking those of the custom domain predicate $\mathbb{D}_{\texttt{ns}}$ as two constructors $\mathbb{D}^{1'}_{\texttt{ns}}$ and $\mathbb{D}^{2'}_{\texttt{ns}}$ below which serve as an alternative to the `Acc_intro` constructor implied by the definition $\mathbb{D}'_{\texttt{ns}} := \texttt{Acc} \preccurlyeq^{\text{sc}}_{\texttt{ns}}$:

$$\mathbb{D}^{1'}_{\texttt{ns}} : \forall x,\ b\,x = \texttt{true} \to \mathbb{D}'_{\texttt{ns}}\,x \quad \mathbb{D}^{2'}_{\texttt{ns}} : \forall x,\ b\,x = \texttt{false} \to \mathbb{D}'_{\texttt{ns}}(g\,x) \to \mathbb{D}'_{\texttt{ns}}\,x$$

The fixpoint equations can easily be deduced by combining `ns_spec` and the functionality of $\mathbb{G}_{\texttt{ns\_fun}}$. As $\texttt{ns\_pwc}_{\texttt{Acc}}$ is packed with its conformity with $\mathbb{G}_{\texttt{ns}}$, there is no need to unfold or evaluate its expression to get these next two equations

$$\texttt{ns}\ x\ (\mathbb{D}^{1'}_{\texttt{ns}}\ x\ E) = 0 \quad \text{and} \quad \texttt{ns}\ x\ (\mathbb{D}^{2'}_{\texttt{ns}}\ x\ E\ D) = \texttt{S}\,\big(\texttt{ns}\ (g\,x)\ D\big)$$

as witnessed by the Coq `Qed` directive ending the proof term of Fig. 5, intended to be *opaque* to evaluation.

```
Fixpoint ns_pwc_Acc x (D : D'_ns x) : {o | x ↦_ns o}.
Proof. refine(
    match b x as b_x return b x = b_x → _ with
        | true  ⇒ λG, exist _ 0 O?_1
        | false ⇒ λG,
                    let (o, C_o) := ns_pwc_Acc (g x) T?_1
                    in  exist _ (S o) O?_2
    end eq_refl).
    1, 2 : cycle 1.  (* reordering of proof obligations *)
    [T?_1] : apply Acc_inv with (1 := D); now constructor.
    [O?_1] : now constructor 1.
    [O?_2] : now constructor 2.
Qed.
```

Fig. 5. Coq fixpoint for $\texttt{ns\_pwc}_{\texttt{Acc}}$ with $\mathbb{D}'_{\texttt{ns}} := \texttt{Acc} \preccurlyeq^{\text{sc}}_{\texttt{ns}}$.

This construction with $\mathbb{D}'_{\texttt{ns}}$ defined as $\texttt{Acc}\preccurlyeq^{\text{sc}}_{\texttt{ns}}$ provides exactly the same tools as the construction with custom domain predicates. We could now proceed with the study of the high-level properties of `ns` in a similar way.

4.4. *A failure of well-founded recursion*

In the section, we discuss how this particular algorithm scheme of `ns` challenges well-founded recursion, contrary to `Acc`-based recursion. Let us consider $b : \mathbb{N} \to \mathbb{B}$ to be the identity test with 1, i.e. $b\,x := x = 1$ and $g : \mathbb{N} \to \mathbb{N}$ to be defined such that

$$g\,n := \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd.} \end{cases}$$

Then the computation of `ns` generates the Syracuse sequence $g^0\,x$, $g^1\,x$, $g^2\,x$, ... until it eventually reaches the value 1. It is easy to show that the domain of `ns` in this case is exactly the values x for which the Syracuse sequence from x ends up in the cycle $1, 4, 2, 1$. This follows directly from $\mathbb{D}_{\texttt{ns}}$`_high_level` (see page 334).

Hence $\mathbb{D}_{\texttt{ns}}/\mathbb{D}'_{\texttt{ns}}$ is a predicate of which the totality problem is still unresolved at the present time and b.t.w., despite its very simple statement, a highly difficult mathematical problem [16]. *A fortiori,* there is no known measure nor well-founded order that could be used to justify the eventual termination of the Syracuse sequence into the length 3 cycle.

Given that well-founded recursion assumes the domain to be total, there would be no way to define this instance of `ns` unless at some point, someone comes up with a totality proof for $\mathbb{D}'_{\texttt{ns}}$ moreover based on a well-founded relation. On the contrary, $\texttt{Acc}\preccurlyeq^{\text{sc}}_{\texttt{ns}}$-based recursion (or custom domain predicates) are perfectly at ease with partial functions and the implementation of the Syracuse sequence can trivially be extracted as the above instance of `ns`.

4.5. *Inductive–Recursive schemes*

Induction–recursion consists in the simultaneous definition of a predicate and a fixpoint such that the predicate might make reference to the fixpoint values. The concept was formally introduced by Dybjer [17] and used widely for the representation of partial recursion in type theory, e.g. in the seminal work of Bove and Capretta [10].

A bit at odds with the Coq understanding of accessibility characterized by the specific but parametric `Acc` predicate, the domain predicates used for inductive–recursive scheme by Bove and Capretta [10] are also called accessibility predicates. To us, they look much more like our custom inductive domain predicates, however with the main difference that their accessibility predicates must belong to sort `Type` because the fixpoints to which they are attached proceed by pattern matching and recursion on them.

Anyway, Coq does not currently implement inductive–recursive schemes. Also, in the peculiar distinction between "non-informative" propositions in $\mathbb{P}$ and "informative" `Type`s that is crucial for extraction in Coq, pattern matching based on domain constructors in $\mathbb{P}$ would not be accepted: it is already forbidden for regular fixpoints definitions.[o]

Following Bove and Capretta [10] and the fully predicative world view of Agda [18], one could of course consider `Type`-based domain predicates in which case pattern matching on them would be allowed. However, this approach would lead to terms with an entirely different computational contents: computation would proceed by matching on domain predicates instead of matching on input parameters. This would of course reflect into the extracted terms which would contain those informative domain arguments. But erasing the non-informative domain argument is precisely the feature we are using to get clean extracted terms [19].

Nonetheless, our approach is compatible with induction–recursion in the sense that we can simulate those schemes in Coq. In fact, they form a quite convenient approach at proving partial correctness properties as an alternative to induction on the computational graph predicate. In practice, they allow to work with partial functions instead of relational reasoning.

Simulating induction–recursion consists in the implementation of a (proof irrelevant) eliminator (i.e. induction principle) for the domain predicate and of fixpoint equations for the function.

[o]With the exception of the *singleton elimination rule*, see Section 2.3.

This approach is favored in Sections 6 and 8 while inductive–recursive schemes and computational graph-based induction are compared in Section 7. In this work, we do not provide a systematic description of induction–recursion but instead favor examples to hint at how it behaves in practice.

5. Odd Functions on Lists

Objectives and disclaimer. In most cases, recursive calls are inside branches of a pattern-matching construct, rather than in a simple `if-then-else` construct. The components of the constructor currently analyzed can then be directly exploited in the projections $\pi_{\mathbb{D}}$ introduced with the first central idea of the Braga method, see Section 3. To illustrate this, we consider here basic functions on lists, that are neither complicated nor efficient in any way. But they happen to provide an unusual and in some sense natural reference for well-known functions, especially OCaml `fold_left` which seems never to be formally specified. We even consider a version which is *not* even directly programmable in OCaml. This becomes the case after a simple transformation but anyway, the reference program obtained in this way, though simple, does not fit the simple scheme by structural recursion. Thanks to the Braga method we can reason on these functions (and even their ideal non-programmable version) and show that they are related as expected with the standard efficient versions.

5.1. *On the correctness of* `fold_left`

Let us start with a well-known example, reverting a list, which is traditionally presented in two ways: a simple version `naive_rev` which recursively uses an auxiliary function `consr`, such that `consr` $u\ y$, also denoted by $u +\!: y$, is the list u postfixed by the single element y and a more sophisticated version `eff_rev` using an accumulator. It is well known that `eff_rev` is better behaved: it is linear-time in the length of the input, whereas `naive_rev` is quadratic-time. However, `naive_rev` is simpler and better at proving algebraic properties. So it is common to consider it as a specification of revert, and to prove that `eff_rev` returns the same result as `naive_rev`. In this approach the associativity of the append function $+\!\!+$ plays a crucial role, given the fact $u +\!: y = u +\!\!+ [y]$. On the other hand, `eff_rev` is a special case of the `fold_left` function. But what should be the specification of `fold_left`? Things become clearer if we (attempt to) write the recursive equations of `naive_rev` in the converse way.

$$\begin{aligned}
&\texttt{naive\_rev}\,[\,] &&= [\,]\\
&\texttt{naive\_rev}\,(y :: u) &&= \texttt{naive\_rev}\,u +\!: y\\[1ex]
&\texttt{naive\_rev\_conv}\,[\,] &&= [\,]\\
&\texttt{naive\_rev\_conv}\,(u +\!: y) &&= y ::\ \texttt{naive\_rev\_conv}\,u.
\end{aligned}$$

Similarly, a reference version of `foldl_ref` $f\ b_0$ would be:

$$\begin{aligned}
&\texttt{foldl\_ref}\ f\ b_0\ [\,] &&= b_0\\
&\texttt{foldl\_ref}\ f\ b_0\ (u +\!: z) &&= f\ (\texttt{foldl\_ref}\ f\ b_0\ u)\ z.
\end{aligned}$$

These equations, which formalize common informal explanatory drawings, correspond to nothing but the mirror version of `fold_right`. Note that, in these equations, f and b_0 are constants. In particular, b_0 is *not* an accumulator. Therefore, in the rest of this chapter, we consider that f and b_0 are given once for all and we simplify the previous equations as follows.

$$\texttt{foldl\_ref}\ [\,] = b_0 \quad \text{and} \quad \texttt{foldl\_ref}\ (u +\!: z) = f\ (\texttt{foldl\_ref}\ u)\ z.$$

Figure 6 contains a program in OCaml syntax which reflects those equations, but this is not a regular program because the second pattern is written with a function call instead of constructors. From an algebraic perspective, the pair $([\,], +\!:)$ shares the same desired properties (injectivity, discrimination and covering) as $([\,], ::)$ for decomposing a list. But beyond algebraic meaningfulness, an explicit way to get the components of each "constructor" is needed.

```
let rec foldl_ref l = match l with  (* fake *)
    | []      → b0
    | u +: z → f (foldl_ref u) z
```

Fig. 6. A fake ideal reference program for `fold_left`.

Nonetheless it is possible to recover a regular functional program after a small additional work. Let us introduce an auxiliary *non-recursive* type `lr` defined in OCaml syntax as follows.

$$\texttt{type}\ \alpha\ \texttt{lr} = \texttt{Nilr} \mid \texttt{Consr of}\ \alpha\ \texttt{list} * \alpha.$$

The first argument of `Consr` is purposely a `list`, and *not* a `lr`. We then consider the regular reference OCaml program `foldl_ref` (without parameters f and b_0) given in Fig. 7. In this program, `l2r` is the obvious bijective function from α `list` to α `lr`, whose inverse is the even more obvious function `r2l` which interprets `Nilr` by the $[\,]$ constant and `Consr` by the $+:$ operator.

In other words, the constructor `Consr` is a concrete reflection of the $+:$ function. The regular pattern matching on the left-hand side of Fig. 8 can be seen as the actual meaning of the fake scheme on the right-hand side which is suggested by the above recursive equations.

Note that `naive_rev_conv` can be implemented using the same pattern, yielding a program having the same complexity as `naive_rev`.

On the same model, `foldl_ref` of Fig. 7 can serve as an inefficient, but clear reference program for the usual `fold_left`. In order to provide a formal Coq proof of the equivalence between them, a suitable definition of `foldl_ref` in Coq is required, as well as tools for reasoning about it. The above recursive function does not fit into the usual scheme of definitions by structural recursion, but we can use the Braga method.

First, we introduce in Fig. 9 a relational presentation $\mathbb{G}_{\texttt{foldl}}$ for the graph of `foldl_ref`. We consider $\mathbb{G}_{\texttt{foldl}}$ as a binary relation $\mapsto_{\texttt{fl}}$ between an input in $\mathbb{L}\,A$ and an output in B, with additional

```
let rec foldl_ref l = match l2r l with
    | Nilr          → b₀
    | Consr (u, z)  → f (foldl_ref u) z
```

Fig. 7. A regular reference program for `fold_left`.

```
match l2r l with               |  match l with    (* fake *)
   | Nilr          → ...       |     | []       → ...
   | Consr (u, z)  → ...       |     | u +: z   → ...
```

Fig. 8. Implementation of a fake match.

$$\texttt{Inductive}\ \ \mathbb{G}_{\texttt{foldl}} : \mathbb{L}\,A \to B \to \mathbb{P}\ \ \texttt{and}\ \ \mathbb{G}_{\texttt{flr}} : \texttt{lr}\,A \to B \to \mathbb{P}\ \ :=$$
$$\frac{}{\texttt{Nilr} \mapsto_{\texttt{flr}} b_0} \qquad \frac{u \mapsto_{\texttt{fl}} b}{\texttt{Consr}\ u\ z \mapsto_{\texttt{flr}} f\ b\ z} \qquad \frac{\texttt{l2r}\ l \mapsto_{\texttt{flr}} b}{l \mapsto_{\texttt{fl}} b}$$

Fig. 9. Basic relational presentation of `fold_left`.

$$\texttt{Inductive}\ \ \mathbb{G}_{\texttt{foldl}} : \mathbb{L}\,A \to B \to \mathbb{P} := \qquad \frac{}{[\,] \mapsto_{\texttt{fl}} b_0} \qquad \frac{u \mapsto_{\texttt{fl}} b}{u +: z \mapsto_{\texttt{fl}} f\ b\ z}$$

Fig. 10. High-level relational presentation of `fold_left`.

$$\texttt{Inductive}\ \ \mathbb{D}_{\texttt{foldl}} : \mathbb{L}\,A \to \mathbb{P} := \qquad \frac{}{\mathbb{D}_{\texttt{foldl}}\ [\,]} \qquad \frac{\mathbb{D}_{\texttt{foldl}}\ u}{\mathbb{D}_{\texttt{foldl}}\ (u +: z)}$$

Fig. 11. Inductive definition of the domain of `fold_left`, based on Fig. 10.

constant parameters f and b_0. In situations where more details are needed we will use the heavier notation $\mathbb{G}^{f,b_0}_{\texttt{foldl}}$. In order to define $\mathbb{G}_{\texttt{foldl}}$, a Coq version of `lr` and `l2r` is needed first. This is an easy exercise, as well as the definition of `r2l` and the proofs that `l2r` and `r2l` are inverse of each other.

This presentation is a straightforward translation of the program given in Fig. 7. However in the present case, it is more naturally described in Fig. 10, with $+:$ instead of `Consr`, pretending that we are going to directly implement the fake `match` of Fig. 6 without the artificial intermediary of `lr`.

The next step is to write the inductive definition of the domain $\mathbb{D}_{\texttt{foldl}}$ of $\mathbb{G}_{\texttt{foldl}}$. We just ignore its last (output) argument. The constant parameters f and b_0 are irrelevant here since they are only used for computing the output. A first definition of $\mathbb{D}_{\texttt{foldl}}$ is given in Fig. 11. Actually, an equivalent predicate $\mathbb{D}_{\texttt{lz}}$ is used in order to fulfill an objective of this section. Note that these predicates are suitable to all functions which visit lists from right to left. A projection $\pi_{\mathbb{D}_{\texttt{lz}}} : \mathbb{D}_{\texttt{lz}}\ (u +: z) \to \mathbb{D}_{\texttt{lz}}\ u$ returning a structurally smaller term can then be blindly defined using the `inversion` tactic of Coq, however an explicit definition will be given in Section 5.2.

A conform-by-construction `fold_left` can then be defined as in Figure 12. As for `ns`, the heart of this code is inside the `refine` tactic, with a crucial use of $\pi_{\mathbb{D}_{\texttt{lz}}}$ in the recursive call and two proof obligations for the postcondition. A technical difference is that here

```
Let Fixpoint foldl_pwc $l$ $(D : \mathbb{D}_{\texttt{lz}}\ l) : \{b \mid l \mapsto_{\texttt{fl}} b\}$.
Proof.
    gen_help $l$ $\mathbb{G}_{\texttt{foldl}}$; apply up_llP in $D$; revert $D$.
    refine ( match l2r $l$ with
        | Nilr       $\Rightarrow \lambda D\ T,$ exist _ $b_0$ $\mathcal{O}^?_1$
        | Consr $u$ $z$ $\Rightarrow \lambda D\ T,$
                    let $(b, C_b) :=$ foldl_pwc $u$ $(\pi_{\mathbb{D}_{\texttt{lz}}}\ D)$
                    in       exist _ $(f\ b\ z)$ $\mathcal{O}^?_2$
    end).
    $[\mathcal{O}^?_1]$ : apply $T$; constructor 1.
    $[\mathcal{O}^?_2]$ : apply $T$; constructor 2; exact $C_b$.
Qed.
```

Fig. 12. Coq proof term `foldl_pwc` of the conform-by-construction `foldl` algorithm.

we have two Trojan horses. The first one is D whose type $\mathbb{D}_{\texttt{lz}}\ l$ has been replaced by $\mathbb{D}_{\texttt{lz}}\,(\texttt{r2l}\,(\texttt{l2r}\ l))$ using `up_llP`, and the second one is $T : \forall y, \texttt{r2l}\,(\texttt{l2r}\ l) \mapsto_{\texttt{fl}} y \ \rightarrow\ l \mapsto_{\texttt{fl}} y$, introduced by `gen_help`. Lemmas `up_llP` and `gen_help` are justified by a simple rewriting step. In this way, the pattern-matching of $\texttt{l2r}\ l$ changes expressions $\texttt{r2l}\,(\texttt{l2r}\ l)$ respectively by `r2l Nilr` and $\texttt{r2l}\,(\texttt{Consr}\ u\ z)$ in the two branches. In the first, we get $D : \mathbb{D}_{\texttt{lz}}\,[]$ and $T : \forall y, [\,] \mapsto_{\texttt{fl}} y \rightarrow l \mapsto_{\texttt{fl}} y$. In the second we get $D : \mathbb{D}_{\texttt{lz}}\ (u +\!\!: z)$ and $T : \forall y, u +\!\!: z \mapsto_{\texttt{fl}} y \ \rightarrow\ l \mapsto_{\texttt{fl}} y$, so everything is in place for feeding $\pi_{\mathbb{D}_{\texttt{lz}}}$ and proving the postconditions.

As for `ns`, we easily get a Coq version of `foldl_ref` and a proof that it satisfies $\mathbb{G}_{\texttt{foldl}}$ using the standard projections on Σ-types π_1 and π_2. The extraction of `foldl_ref` yields exactly the expected OCaml code.

In this case study, we are interested in proving that the usual (linear-time) implementation of `fold_left` returns the same result as `foldl_ref`. To this effect we first define this function (where f is a hidden parameter) by easy structural recursion in the list in input, and we prove that it is *complete* w.r.t. $\mathbb{G}_{\texttt{foldl}}$.

```
Fixpoint foldl $b$ $l$ : $B$ :=
    match $l$ with  $[\,] \Rightarrow b$ | $x :: l \Rightarrow$ foldl $(f\ b\ x)$ $l$  end.
Theorem foldl_compl $b$ $l$ :   $l \mapsto_{\texttt{fl}} b \ \rightarrow\ b =$ foldl $b_0$ $l$.
```

The proof is by trivial induction on $l \mapsto_{\texttt{fl}} b$, using a simple lemma saying that $\texttt{foldl}\ f\ b\ (u +\!\!: z)$ is always equal to $f\ (\texttt{foldl}\ f\ b\ u)\ z$.

Finally, we get the expected corollary, expressed with an explicit f.

$$\texttt{Theorem}\ \texttt{foldl\_equiv\_partial}\ f\ b\ l\ (D : \mathbb{D}_{\texttt{lz}}\, l) :$$
$$\texttt{foldl}\ f\ b\ l = \texttt{foldl\_ref}\ f\ b\ l\ D.$$

Actual termination is obtained separately and total correctness of `fold_left` is just a special case of `fold_equiv_partial`. As expected for such a very simple case study, the proofs are very light, between one and three lines of elementary explicit scripts without automation or heavy machinery.

Back to the revert function, we can prove, along the same approach, that `eff_rev` returns the same result as `naive_rev_converse`, without referring to an alien function ($+\!\!+$) and its algebraic properties. In particular the graph is nicely symetric. Its domain is $\mathbb{D}_{\texttt{lz}}$, the same as for `foldl_ref`.

5.2. *Projections*

We define here the projection used in order to have a clearly structurally smaller domain argument in the recursive call of `foldl_pwc`. Though $\mathbb{D}_{\texttt{foldl}}$ can indeed be used, we replace it with the equivalent definition given in Fig. 13, which is based on the graph of Fig. 9. The main reason is that the auxiliary $\mathbb{D}_{\texttt{lr}}$ illustrates a situation which is close to most common examples, where the pattern-matching is expressed against the main argument of the function (l here). The projection is then easier to define, without interference with additional equality proofs. We first focus on this part by defining $\pi_{\mathbb{D}\texttt{lr}} : \mathbb{D}_{\texttt{lr}}\,(\texttt{Consr}\, u\, z) \to \mathbb{D}_{\texttt{lz}}\, u$ as in Fig. 14. The term returned in the interesting case is D_{u0} which is clearly the intended subterm of D. Note the use of a Trojan horse $G : \texttt{shape}\ r$, where $\texttt{shape}\ r$ plays the same role as `is_cons` at the beginning of Section 3. When D is $\mathbb{D}^{\texttt{N}}_{\texttt{lr}}$, then its type is $\mathbb{D}_{\texttt{lr}}\ r$ with $r = \texttt{Nilr}$, so that $\texttt{shape}\, r$, the type of G, reduces to $\bot$.

$$\texttt{Inductive}\ \mathbb{D}_{\texttt{lz}} : \mathbb{L}\,A \to \mathbb{P}\quad \texttt{and}\quad \mathbb{D}_{\texttt{lr}} : \texttt{lr}\,A \to \mathbb{P}\quad :=$$

$$\frac{}{\mathbb{D}_{\texttt{lr}}\ \texttt{Nilr}}\ [\mathbb{D}^{\texttt{N}}_{\texttt{lr}}]\qquad \frac{\mathbb{D}_{\texttt{lz}}\ u}{\mathbb{D}_{\texttt{lr}}\ (\texttt{Consr}\ u\ z)}\ [\mathbb{D}^{\texttt{C}}_{\texttt{lr}}\ u\ z]\qquad \frac{\mathbb{D}_{\texttt{lr}}\ (\texttt{l2r}\ l)}{\mathbb{D}_{\texttt{lz}}\ l}\ [\mathbb{D}^{\texttt{l}}_{\texttt{lz}}]$$

Fig. 13. Inductive definition of the domain of `fold_left`, based on Fig. 9.

```
Definition shape (r : lr A) : ℙ :=
    match r with Consr u z ⇒ ⊤ | _ ⇒ ⊥ end.
Definition π𝔻lr {u z} (D : 𝔻lr (Consr u z)) : 𝔻lz u :=
    match D in 𝔻lr r return
            let u₀ := match r with Consr u₀ z₀ ⇒ u₀ | _ ⇒ u end
            in  shape r → 𝔻lz u₀ with
       | 𝔻ᶜlr u₀ z₀ D_u₀ ⇒ λG, D_u₀
       | 𝔻ᴺlr            ⇒ λG, match G with end
    end I.
```

Fig. 14. Projection function for $\mathbb{D}_{\texttt{lr}}$.

```
Definition lrleft r : shape r → 𝕃 A :=
    match r with Consr u z ⇒ λ_, u | _ ⇒ λG, False_elim _ G end.
Definition π𝔻lr {u z} (D : 𝔻lr (Consr u z)) : 𝔻lz u :=
    match D in 𝔻lr r return ∀G, 𝔻lz (lrleft r G) with
       | 𝔻ᶜlr u₀ z₀ D_u₀ ⇒ λG, D_u₀
       | 𝔻ᴺlr            ⇒ λG, match G with end
    end I.
```

Fig. 15. Projection function for $\mathbb{D}_{\texttt{lr}}$ with an auxiliary function.

There is a subtle point about the u component of $\texttt{Consr}\,u\,z$. In the course of the pattern matching of D, the type of D is originally considered as being $\mathbb{D}_{\texttt{lr}}\,r$ and the identity $r = \texttt{Consr}\,u\,z$ is lost: r becomes either `Nilr` (the fake case handled by the Trojan horse G), or $\texttt{Consr}\,u_0\,z_0$, so we need to reconnect u_0 with u. This is performed by stating that the type of the result in the `return` clause is $\mathbb{D}_{\texttt{lz}}\,u_0$, where u_0 is the first component of r when r is $\texttt{Consr}\,u_0\,z_0$. However, u_0 has to be defined in all cases for r, so a default value has to be provided. In the case of the type `lr` we could take the ad-hoc `Nilr`. For the sake of generality it is much better to make no assumption on the type of u_0, but we just remark, as in [20] that a suitable candidate is necessarily available at this stage : u itself.

Another option for $\pi_{\mathbb{D}\texttt{lr}}$ is to first define an auxiliary function `lrleft` along the same lines as for `head` at the beginning of Section 3, as illustrated in Fig. 15. In addition to r, this function takes a guard argument G of type $\texttt{shape}\,r$. In the absurd case where r is `Nilr`, we don't mind to find a value, using for `False_elim` one of the functions detailed in Section 2.7. This option is especially valuable if a safe version like `False_loop` or `False_exc` is chosen, avoiding harmless `Prop` to `Type` eliminations issue.[p]

However, as for `ns` in Section 3.1, in the target algorithm, the pattern-matching is expressed not against the argument of the function (l here), but on a function of l, which is here `l2r`. A similar work is done with an auxiliary equality proof. The expression `same` $G\,D_r$ just says that in the type of D_r, `l2r` l can be rewritten as `consr` $u\,z$ in the presence of $G:\ l = u +: z$.

```
Definition $\pi_{\mathbb{D}_{\mathtt{lz}}}\ \{u\ z\}\ \big(D : \mathbb{D}_{\mathtt{lz}}\,(u +: z)\big) : \mathbb{D}_{\mathtt{lz}}\,u :=$
    match $D$ in $\mathbb{D}_{\mathtt{lz}}\,l$ return $l = u +: z \rightarrow$ _ with
        | $\mathbb{D}^1_{\mathtt{lz}}\ l\ D_r \Rightarrow \lambda G, \pi_{\mathbb{D}_{\mathtt{lr}}}\,(\mathtt{same}\,G\,D_r)$
    end eq_refl.
```

6. Potentially Non-terminating Depth-First Search

Depth-first search is an algorithm for traversing or searching tree-based or graph-based data-structures [21]. The standard *traversing* `dfs` algorithm is generally presented using the recursive equations of Fig. 16 leading to potential non-termination on some inputs; see the discussion ending the section for a non-terminating example on an infinite graph. The structure of `dfs` is similar to that of our initial example `ns` introduced in Section 3.1 but it has two input parameters instead of only one.

Despite its apparent simplicity and its lack of nested calls, we consider `dfs` to be a particularly interesting algorithm to implement as an illustration of the Braga method because of this potential

$$\begin{array}{lll}
\mathtt{dfs}\ v\ [\,] & = v & \\
\mathtt{dfs}\ v\ (x :: l) & = \mathtt{dfs}\ v\ l & \text{if } x \in v \\
\mathtt{dfs}\ v\ (x :: l) & = \mathtt{dfs}\ (x :: v)\ (\mathtt{succs}\,x \mathbin{+\!\!+} l) & \text{if } x \notin v
\end{array}$$

Fig. 16. Equations describing the `dfs` algorithm.

[p]This issue is not raised in the first version of $\pi_{\mathbb{D}_{\mathtt{lr}}}$ presented in Fig. 14 since there is no need to eliminate G to describe the type returned by the `match` G construct.

non-termination, leading to a quite non-trivial characterization of its (termination) domain, based on invariants to be discussed later on. The ability to manipulate the partial algorithm and derive partial correctness properties will be critical to the characterization of its termination domain.

6.1. *Preliminaries*

We consider a potentially infinite graph described by a type $\mathcal{V}$: `Type` of *vertices* and a function `succs` : $\mathcal{V} \rightarrow \mathbb{L}\,\mathcal{V}$ finitely enumerating the *successors* of a vertex. These assumptions restrict the study to finitely branching directed graphs but these are standard assumptions for depth-first search.

To convert equations of Fig. 16 into a definitive algorithm, we need to assume a membership test function over lists of vertices `mem` : $\mathcal{V} \rightarrow \mathbb{L}\,\mathcal{V} \rightarrow \mathbb{B}$ that we denote infix $x \in^? v := \mathtt{mem}\,x\,v$, and with the specification:

$$\texttt{Parameter mem\_true\_iff}: \quad \forall x\,v,\ x \in^? v = \texttt{true} \leftrightarrow x \in v.$$

Then we can show that

$$\texttt{Corollary mem\_iff}: \quad \forall x\,v,\ \wedge \begin{cases} x \in^? v = \texttt{true} \leftrightarrow x \in v \\ x \in^? v = \texttt{false} \leftrightarrow x \notin v. \end{cases}$$

Note that `mem` could be derived from an equality decider[q] over $\mathcal{V}$, but we refrain from specifying it more: the particular implementation might depend on the specific structure of vertices to be more efficient than a sequence of identity tests.

6.2. *The computational graph and the domain*

We define the computational graph $\mathbb{G}_{\mathtt{dfs}}$ of the `dfs` algorithm as a ternary relation $\mathbb{G}_{\mathtt{dfs}}\,v\,l\,o$ between the inputs $(v\ l : \mathbb{L}\,\mathcal{V})$ and the output $o : \mathbb{L}\,\mathcal{V}$, denoted with the mixfix notation $v \sqcup l \mapsto_{\mathtt{d}} o$, and to be

[q]usually implementable for data-types but, contrary to OCaml, not available in any type in Coq, e.g. typically not available over function types.

read as "`dfs` v l outputs o". It is composed of the three following inductive rules that mimic the equations of Fig. 16:

$$\texttt{Inductive}\ \mathbb{G}_{\texttt{dfs}} : \mathbb{L}\,\mathcal{V} \to \mathbb{L}\,\mathcal{V} \to \mathbb{L}\,\mathcal{V} \to \mathbb{P} :=$$

$$\frac{}{v \sqcup [\,] \mapsto_{\texttt{d}} v} \quad \frac{x \in v \quad v \sqcup l \mapsto_{\texttt{d}} o}{v \sqcup x :: l \mapsto_{\texttt{d}} o} \quad \frac{x \notin v \quad x :: v \sqcup \texttt{succs}\, x +\!\!+\, l \mapsto_{\texttt{d}} o}{v \sqcup x :: l \mapsto_{\texttt{d}} o}\ .$$

The graph $\mathbb{G}_{\texttt{dfs}}$ is a mostly straightforward formal encoding of the otherwise informal equations defining `dfs`. For simplicity, here we assume $\mathbb{G}_{\texttt{dfs}}$ to faithfully encode those equations in its three rules, but this will not matter at all for total correctness. It might only be of relevance when considering the operational semantics of the extracted code.

We show that the computational graph $\mathbb{G}_{\texttt{dfs}}$ of `dfs` is functional, i.e. it outputs at most one value on any given pair of inputs:

$$\texttt{Fact}\ \mathbb{G}_{\texttt{dfs}}\texttt{\_fun}\ v\ l\ o_1\ o_2 :\ \ v \sqcup l \mapsto_{\texttt{d}} o_1 \ \to\ v \sqcup l \mapsto_{\texttt{d}} o_2 \ \to\ o_1 = o_2.$$

Proof. By induction on the first predicate of type $v \sqcup l \mapsto_{\texttt{d}} o_1$ and inversion on the second predicate of type $v \sqcup l \mapsto_{\texttt{d}} o_2$. □

We characterize the domain $\mathbb{D}_{\texttt{dfs}}$ of `dfs` with a custom inductive predicate following the three rules of the graph $\mathbb{G}_{\texttt{dfs}}$ but ignoring/erasing the third (output) argument[r]:

$$\texttt{Inductive}\ \mathbb{D}_{\texttt{dfs}} : \mathbb{L}\,\mathcal{V} \to \mathbb{L}\,\mathcal{V} \to \mathbb{P} :=$$

$$\frac{}{\mathbb{D}_{\texttt{dfs}}\, v\ [\,]}\ [\mathbb{D}^1_{\texttt{dfs}}\, v] \qquad \frac{x \in v \quad \mathbb{D}_{\texttt{dfs}}\, v\ l}{\mathbb{D}_{\texttt{dfs}}\, v\ (x :: l)}\ [\mathbb{D}^2_{\texttt{dfs}}\, v\, x\, l]$$

$$\frac{x \notin v \quad \mathbb{D}_{\texttt{dfs}}\,(x :: v)\ (\texttt{succs}\, x +\!\!+\, l)}{\mathbb{D}_{\texttt{dfs}}\, v\ (x :: l)}\ [\mathbb{D}^3_{\texttt{dfs}}\, v\, x\, l]\ .$$

The correctness of this characterization of $\mathbb{D}_{\texttt{dfs}}$ w.r.t. the projection of $\mathbb{G}_{\texttt{dfs}}$ on its two inputs will be established later on.

[r]This works in the case of `dfs` because it is not a nested recursive algorithm, but it will fail and must be refined in the case of, e.g. Paulson's normalization algorithm of Section 7.

6.3. *A term for `dfs` that conforms to its computational graph*

We have enough structure to build the fully specified `dfs`, that is the algorithm *packed with conformity* to the computational graph $\mathbb{G}_{\texttt{dfs}}$ of type

$$\texttt{dfs\_pwc} : \forall v\, l,\ \mathbb{D}_{\texttt{dfs}}\, v\ l \to \{o \mid v \sqcup l \mapsto_{\texttt{d}} o\}$$

of which the exhaustive term reported in Fig. 17. It is implemented as a `Fixpoint` of which the `struct` argument is the non-informative domain predicate $D : \mathbb{D}_{\texttt{dfs}}\, v\ l$. Using the handy `refine` tactic, we mostly separate the *computational contents* presented in programming style, from the *logical contents* presented in proof style (i.e. as combinations of tactics).

The computational contents strictly follows the intended OCaml algorithm that we wish to extract. Some of the logical contents, essentially names for introduced hypotheses, must be reported in there but we try to keep it is as minimal as possible.

The logical contents — composed of *proof obligations* — splits into, on the one hand *termination certificates* such as $\mathcal{T}_1^?$, and on the

```
Let Fixpoint dfs_pwc $v$ $l$ $(D : \mathbb{D}_{\texttt{dfs}}\, v\ l)$ {struct $D$} : $\{o \mid v \sqcup l \mapsto_{\texttt{d}} o\}$.
Proof. refine(
  match $l$ with
      | []     $\Rightarrow \lambda D,$    exist _ $v$ $\mathcal{O}_1^?$
      | $x :: l \Rightarrow \lambda D,$
      match $x \in^? l$ as $b$ return $x \in^? l = b \to$ _ with
          | true   $\Rightarrow \lambda E,$
                 let $(o, G_o)$ := dfs_pwc $v$ $l$ $\mathcal{T}_1^?$
                 in  exist _ $o$ $\mathcal{O}_2^?$
          | false $\Rightarrow \lambda E,$
                 let $(o, G_o)$ := dfs_pwc $(x :: v)$ (succs $x$ ++ $l$) $\mathcal{T}_2^?$
                 in  exist _ $o$ $\mathcal{O}_3^?$
      end eq_refl
  end $D$).
  $1, 2, 4$ : cycle 1.  (* reordering of proof obligations *)
 [$\mathcal{T}_1^?$] : now apply $\pi_{\mathbb{D}_{\texttt{dfs}}}$_1 with $(1 := D)$.
 [$\mathcal{T}_2^?$] : now apply $\pi_{\mathbb{D}_{\texttt{dfs}}}$_2 with $(1 := D)$.
 [$\mathcal{O}_1^?$] : now constructor 1.
 [$\mathcal{O}_2^?$] : constructor 2; auto; apply mem_iff; auto.
 [$\mathcal{O}_3^?$] : constructor 3; auto; apply mem_iff; auto.
Qed.
```

Fig. 17. Coq proof term `dfs_pwc` of the fully specified `dfs` algorithm.

other hand *postconditions* such as $\mathcal{O}_1^?$. In real Coq code, these names all collapse to the wildcard _ (or joker) associated with the `refine` tactic but we distinguish them in here to better document them.

For instance, the termination certificate $\mathcal{T}_1^?$ corresponds to the subgoal:

$$[\mathcal{T}_1^?] : \ldots, x : \mathcal{V}, v\ l : \mathbb{L}\,\mathcal{V}, D : \mathbb{D}_{\texttt{dfs}}\, v\ (x :: l), E : x \in^? l = \texttt{true} \vdash \mathbb{D}_{\texttt{dfs}}\, v\ l.$$

We remark that the proof term for the inversion lemma below

$$\texttt{Lemma}\ \ \pi_{\mathbb{D}_{\texttt{dfs}}}\texttt{\_1}\ v\ x\ l : \quad \mathbb{D}_{\texttt{dfs}}\, v\ (x :: l) \to x \in^? v = \texttt{true} \to \mathbb{D}_{\texttt{dfs}}\, v\ l$$

must be *carefully crafted* because, used in the proof of the termination certificate $\mathcal{T}_1^?$, its output value of type $\mathbb{D}_{\texttt{dfs}}\, v\ l$ must type-check as a *subterm* of its first (unnamed) parameter of type $\mathbb{D}_{\texttt{dfs}}\, v\ (x :: l)$. In modern versions of Coq, one can safely rely on the `inversion` tactic to satisfy such a constraint. However, the obtained term might not be short and if a cleaner implementation of such an inversion lemma is required, one could for instance switch to small-inversions based on dependent pattern matching as discussed in Sections 3.1 and 5.2. We recall that it is standard to call such a result "inversion lemma" because it corresponds to the inversion of the second inductive rule defining $\mathbb{D}_{\texttt{dfs}}$, i.e. it implements pattern matching on a term with this (second) outer constructor. Here we also call these results projections because they recover the structural components of constructors.

The second projection lemma $\pi_{\mathbb{D}_{\texttt{dfs}}}\texttt{\_2}$ is used as termination certificate $\mathcal{T}_2^?$ and must thus satisfy the same structural decrease property.

$$\begin{array}{l}\texttt{Lemma}\ \ \pi_{\mathbb{D}_{\texttt{dfs}}}\texttt{\_2}\ v\ x\ l : \\ \quad \mathbb{D}_{\texttt{dfs}}\, v\ (x :: l) \to x \in^? v = \texttt{false} \to \mathbb{D}_{\texttt{dfs}}\, (x :: v)\ (\texttt{succs}\, x \mathbin{+\!\!+} l).\end{array}$$

Turning to postconditions like, e.g. $\mathcal{O}_2^?$

$$[\mathcal{O}_2^?] : \ \ldots, E : x \in^? l = \texttt{true}, G_o : v \sqcup l \mapsto_{\texttt{d}} o \vdash v \sqcup x :: l \mapsto_{\texttt{d}} o$$

these are much simpler to establish and their proofs consist mainly in the application of the corresponding rule/constructor of the graph $\mathbb{G}_{\texttt{dfs}}$.

Now we can define `dfs` by projecting on the first component of the Σ-type $\{o \mid v \sqcup l \mapsto_{\mathtt{d}} o\}$ that is the output of `dfs_pwc` and we get its specification with the second $\pi_2(\mathtt{dfs\_pwc}\ v\ l\ D)$.

$$\begin{array}{l}\mathtt{Definition}\ \mathtt{dfs}\ v\ l\ (D : \mathbb{D}_{\mathtt{dfs}}\, v\ l) := \pi_1(\mathtt{dfs\_pwc}\ v\ l\ D).\\ \mathtt{Fact}\ \mathtt{dfs\_spec}\ v\ l\ (D : \mathbb{D}_{\mathtt{dfs}}\, v\ l) : \quad v \sqcup l \mapsto_{\mathtt{d}} \mathtt{dfs}\ v\ l\ D.\end{array}$$

Since `dfs` is inherently a partial algorithm, let us pause a bit and consider again our definition of the domain predicate $\mathbb{D}_{\mathtt{dfs}}\, v\ l$ used to define `dfs`. Of course, one could naturally consider the projection of the graph $\mathbb{G}_{\mathtt{dfs}}$ on its inputs v and l as a definition of the domain, i.e. the pair of values v and l for which there is an output value o such that $v \sqcup l \mapsto_{\mathtt{d}} o$. It turns out that those two characterizations are equivalent:

$$\mathtt{Theorem}\ \mathbb{D}_{\mathtt{dfs}}\mathtt{\_eq\_}\mathbb{G}_{\mathtt{dfs}}\ v\ l : \quad \mathbb{D}_{\mathtt{dfs}}\, v\ l \ \leftrightarrow\ \exists o,\ v \sqcup l \mapsto_{\mathtt{d}} o.$$

Proof. The *only if* direction ($\rightarrow$) is trivial as an o satisfying $v \sqcup l \mapsto_{\mathtt{d}} o$ is precisely what $\mathtt{dfs}\, v\, l\, D$ outputs (according to `dfs_spec`). For the *if* direction ($\leftarrow$), we show by induction on the graph predicate $v \sqcup l \mapsto_{\mathtt{d}} o$ that $\mathbb{D}_{\mathtt{dfs}}\, v\ l$ holds. For this, we just use the constructors of $\mathbb{D}_{\mathtt{dfs}}$. □

6.4. *Reasoning about `dfs` and its domain*

We now complete our construction with a simulated induction–recursion scheme for `dfs` [10, 17] that will allow us to reason about $\mathbb{D}_{\mathtt{dfs}}$/`dfs`. First a proof-irrelevant recursor/eliminator for the domain $\mathbb{D}_{\mathtt{dfs}}$, leaving out guessable arguments[s] as a joker _ for concision:

$$\begin{array}{l}\mathtt{Theorem}\ \mathbb{D}_{\mathtt{dfs}}\mathtt{\_rect}\ (P : \forall v\, l,\ \mathbb{D}_{\mathtt{dfs}}\, v\ l \rightarrow \mathtt{Type}) :\\ \qquad \big(\forall v\, l\, D_1\, D_2,\ P\ v\ l\ D_1 \rightarrow P\ v\ l\ D_2\big)\\ \quad \rightarrow \big(\forall v,\ P\, \_\, \_\ (\mathbb{D}^1_{\mathtt{dfs}}\ v)\big)\\ \quad \rightarrow \big(\forall v\, x\, l\, H D,\ P\, \_\, \_\ D \rightarrow P\, \_\, \_\ (\mathbb{D}^2_{\mathtt{dfs}}\ v\ x\ l\ H\ D)\big)\\ \quad \rightarrow \big(\forall v\, x\, l\, H D,\ P\, \_\, \_\ D \rightarrow P\, \_\, \_\ (\mathbb{D}^3_{\mathtt{dfs}}\ v\ x\ l\ H\ D)\big)\\ \quad \rightarrow \big(\forall v\, l\, D, P\ v\ l\ D\big).\end{array}$$

[s]By guessable, we mean that they are recovered by Coq through unification.

Then the proof-irrelevance of `dfs`, and finally the fixpoint equations:

$$
\begin{array}{ll}
\texttt{Facts}: & \\
\quad \texttt{dfs\_pirr} : \forall v\, l\, D_1\, D_2, & \texttt{dfs}\ v\ l\ D_1 = \texttt{dfs}\ v\ l\ D_2. \\
\quad \texttt{dfs\_fix\_1} : \forall v, & \texttt{dfs}\ \_\ \_\ (\mathbb{D}^1_{\texttt{dfs}}\ v) = v. \\
\quad \texttt{dfs\_fix\_2} : \forall v\, x\, l\, H D, & \texttt{dfs}\ \_\ \_\ (\mathbb{D}^2_{\texttt{dfs}}\ v\ x\ l\ H\, D) = \texttt{dfs}\ \_\ \_\ D. \\
\quad \texttt{dfs\_fix\_3} : \forall v\, x\, l\, H D, & \texttt{dfs}\ \_\ \_\ (\mathbb{D}^3_{\texttt{dfs}}\ v\ x\ l\ H\, D) = \texttt{dfs}\ \_\ \_\ D.
\end{array}
$$

Proof. Direct consequences of `dfs_spec` and $\mathbb{G}_{\texttt{dfs}}$`_fun`. □

With the tools that simulate an inductive–recursive scheme, we can study `dfs` and give a more abstract characterization of its domain, and of what it computes using invariants.

6.5. *High-level correctness results and termination*

Even though this example is discussed in Krauss [14], we do not follow his outline. Indeed, his reasoning assumes finiteness of the type $\mathcal{V}$ of vertices. Here we manage `dfs` as a partial algorithm, hence assuming finiteness of $\mathcal{V}$ is unnecessary, and we get a high-level termination characterization independent of that assumption. Only in the end do we specialize `dfs` on a finite type of vertices, deriving totality nearly for free in that case.

We establish a first partial correctness result: a property of the output of `dfs` $v\ l$ under the hypothesis of its termination on that particular input $(v\ l : \mathbb{L}\,\mathcal{V})$. Here, we show that on its domain $\mathbb{D}_{\texttt{dfs}}$ of termination, `dfs` computes a least invariant as follows:

$$
\begin{array}{l}
\texttt{Definition dfs\_invariant}_t\ (v\ l : \mathbb{L}\,\mathcal{V})\ (i : \mathbb{L}\,\mathcal{V}) := \\
\qquad\qquad \wedge \begin{cases} v \mathbin{+\!\!+} l \subseteq i \\ \forall x,\ x \in i \rightarrow (x \in v \vee \texttt{succs}\, x \subseteq i). \end{cases} \\
\texttt{Theorem dfs\_invariant}\ v\ l\ (D : \mathbb{D}_{\texttt{dfs}}\, v\ l) : \\
\qquad\quad \wedge \begin{cases} \texttt{dfs\_invariant}_t\ v\ l\ (\texttt{dfs}\ v\ l\ D) \\ \forall i,\ \texttt{dfs\_invariant}_t\ v\ l\ i \rightarrow \texttt{dfs}\ v\ l\ D \subseteq i. \end{cases}
\end{array}
$$

Proof. By induction on D with $\mathbb{D}_{\texttt{dfs}}$`_rect`, and then rewriting using `dfs_pirr` and the fixpoint equations `dfs_fix_[123]`. □

Then we switch to the most difficult result to establish, i.e. the characterization of the domain $\mathbb{D}_{\texttt{dfs}}$ of termination of `dfs` using invariants:

$$\texttt{Theorem}\ \mathbb{D}_{\texttt{dfs}}\texttt{\_domain}\ v\ l: \quad \mathbb{D}_{\texttt{dfs}}\,v\ l \leftrightarrow \exists i, \texttt{dfs\_invariant}_t\ v\ l\ i.$$

Proof. According to the first conjunct of `dfs_invariant`, `dfs` outputs an invariant when called on its domain $\mathbb{D}_{\texttt{dfs}}$, thus the *only if* part is trivial. On the other hand, showing that the existence of an invariant implies the termination of `dfs` is much more complicated.

Assuming a fixed $i : \mathbb{L}\,\mathcal{V}$, we want to show

$$\forall v\, l,\ \texttt{dfs\_invariant}_t\ v\ l\ i \to \mathbb{D}_{\texttt{dfs}}\,v\ l.$$

We proceed by a nested induction:

(1) first on v using reverse strict list inclusion $\supsetneq$ as a well-founded relation;
(2) second by structural induction on l.

The relation $\supsetneq$ between the lists $(v\ w : \mathbb{L}\,\mathcal{V})$ is defined as

$$v \supsetneq w := w \subseteq v \wedge \exists x : \mathcal{V},\ x \in v \wedge x \notin w.$$

Of course this relation $\supsetneq$ is *not* well-founded in general, but it is when restricted to the sublists of some given fixed list, here the assumed global invariant i. We show that the binary relation $\lambda v\, w,\ v \supsetneq w \wedge v \subseteq i$ is indeed well-founded; this involves in particular the pigeon hole principle.

As a consequence, computing $\texttt{dfs}\ v\ (x :: l)$, the recursive subcalls to $\texttt{dfs}\ v\ l$ (when $x \in v$) and $\texttt{dfs}\ (x :: v)\ (\texttt{succs}\,x +\!\!+\, l)$ (when $x \notin v$) are both lesser in this nested scheme: in particular when $x \notin v$ holds, we have $v \subsetneq x :: v \subseteq i$.[t] Since the first parameter $(x :: v)$ is $\supsetneq$-smaller than v, the second parameter has no influence in the nested inductive scheme. □

Using the characterization by invariants, it is then almost straightforward to establish the monotonicity of $\mathbb{D}_{\texttt{dfs}}$:

$$\texttt{Fact}\ \mathbb{D}_{\texttt{dfs}}\texttt{\_mono}\ v\ v'\ l\ l' : v \subseteq v' \to l' \subseteq v' +\!\!+\, l \to \mathbb{D}_{\texttt{dfs}}\,v\ l \to \mathbb{D}_{\texttt{dfs}}\,v'\ l'$$

[t]As $x :: l \subseteq i$ is a property of the invariant i.

whereas, on the other hand, trying to show $\mathbb{D}_{\texttt{dfs}}\texttt{\_mono}$ by, e.g. direct induction on $\mathbb{D}_{\texttt{dfs}}\,v\;l$ is painful endeavor that is bound to end in misery.

We finish with the characterization of the domain of `dfs` $[\,]$, which is the standard way to call `dfs` on an empty list $v = [\,]$ of already visited vertices.

$$\begin{array}{l}
\textbf{Definition}\ \texttt{dfs\_nil\_invariant}_t\ v\ l\ i :=\\
\qquad\qquad l \subseteq i \wedge \forall x,\ x \in i \to \texttt{succs}\,x \subseteq i.\\
\textbf{Corollary}\ \texttt{dfs\_nil\_invariant}\ l\ (D : \mathbb{D}_{\texttt{dfs}}\,[\,]\ l) :\\
\qquad \wedge \begin{cases} \texttt{dfs\_nil\_invariant}_t\ l\ (\texttt{dfs}\ [\,]\ l\ D)\\ \forall i,\ \texttt{dfs\_nil\_invariant}_t\ l\ i \to \texttt{dfs}\ [\,]\ l\ D \subseteq i.\end{cases}\\
\textbf{Corollary}\ \mathbb{D}_{\texttt{dfs}}\texttt{\_nil\_domain}\ l :\\
\qquad\qquad \mathbb{D}_{\texttt{dfs}}\,[\,]\ l \leftrightarrow \exists i,\ \texttt{dfs\_nil\_invariant}_t\ l\ i.
\end{array}$$

Hence `dfs` $[\,]\ l$ terminates and computes the least list i containing l and invariant/stable under `succs`, precisely when such an invariant exists. We can further specialize the termination result $\mathbb{D}_{\texttt{dfs}}\texttt{\_domain}$ and prove totality for `dfs` in case the type $\mathcal{V}$ of vertices is finite, i.e. listable.

$$\textbf{Fact}\ \mathbb{D}_{\texttt{dfs}}\texttt{\_total} :\quad (\exists l_{\mathcal{V}} : \mathbb{L}\,\mathcal{V},\ \forall x : \mathcal{V},\ x \in l_{\mathcal{V}}) \to \forall v\,l,\ \mathbb{D}_{\texttt{dfs}}\,v\ l.$$

Proof. Use $\mathbb{D}_{\texttt{dfs}}\texttt{\_domain}$ and pick $i := l_{\mathcal{V}}$ as invariant. □

6.6. *Concluding remarks and extraction*

Note that in case $\mathcal{V}$ is not finite, e.g. $\mathcal{V} = \mathbb{N}$, then it is possible for $\mathbb{D}_{\texttt{dfs}}\,[\,]$ not to cover the whole input type $\mathbb{L}\,\mathcal{V}$. Indeed, with $\texttt{succs}\,n := [1 + n]$, then any invariant must be stable under successor, which means `dfs` $[\,]\,l$ terminates when and only when $l = [\,]$.

To finish, the extracted OCaml code confirms the operational behavior of `dfs` as we expected:

```
let rec dfs v l = match l with
  | []   -> v
  | x::l -> if   mem x v
            then dfs v l
            else dfs (x::v) (app (succs x) l)
```

Remember that the global parameters $\texttt{mem} : \alpha \to \alpha\ \texttt{list} \to \texttt{bool}$ and $\texttt{succs} : \alpha \to \alpha\ \texttt{list}$ are not extracted and have to be provided for this code to work.[u] An alternative approach would have been to make `mem` and `succs` parameters of `dfs` with the disadvantage of bloating the above code a bit without significantly improving the explanations of what is going on.

7. Paulson's `if-then-else` Normalization Algorithm

Paulson's normalization algorithm was the example which we chose to introduce the basics of the herein called Braga method at the TYPES 2018 conference [2]. It is described by the equations of Fig. 18. In this section, we both enter more in the details of the implementation of `nm` while we also develop four possible variants of the Braga method, characterized by

- defining the domain of `nm` either as a *custom inductive predicate*, or as the *accessibility predicate* of the subcall/call relation of `nm`;
- proving partial correctness either with the simulated proof-irrelevant *inductive–recursive scheme* of `nm`, or proceeding by induction on the *computational graph predicate* of `nm`.

These two binary choices give rise to four possible variants of the method and we discuss/compare all of them in this section.

7.1. *The computational graph and the inductive domain*

First, we define the inductive type of `if _ then _ else _` expressions

$$a, b, c : \Omega ::= \alpha \mid \omega\, a\, b\, c,$$

where α represents atomic expressions and $\omega\, a\, b\, c$ is a short notation for `if` a `then` b `else` c where a, b and c are expressions themselves.

$$\begin{aligned}
\mathtt{nm}\ \alpha &= \alpha\\
\mathtt{nm}\ (\omega\,\alpha\,y\,z) &= \omega\,\alpha\,(\mathtt{nm}\ y)\,(\mathtt{nm}\ z)\\
\mathtt{nm}\ (\omega\,(\omega\,a\,b\,c)\,y\,z) &= \mathtt{nm}\big(\omega\,a\,(\mathtt{nm}\ (\omega\,b\,y\,z))\,(\mathtt{nm}\ (\omega\,c\,y\,z))\big)
\end{aligned}$$

Fig. 18. Equations describing `nm`, Paulson's normalization algorithm.

[u]However, `app`/$+\!\!+$ is extracted but not displayed here.

This type is idealized for the purpose of simplifying the explanations here: there is only one atomic expression. Of course, a more realistic implementation would involve a type parameter for atomic expressions but this would not fundamentally change the discussion which follows in the section.

We define the computational graph reflecting the equations of Fig. 18 into a binary relation $e \mapsto_{\mathtt{n}} n$ which reads as "`nm` e terminates and outputs n". The choice of the letter n is to remind that the output is intended to be a normal form (of the input e).

$$\texttt{Inductive}\ \ \mathbb{G}_{\mathtt{nm}} : \Omega \to \Omega \to \mathbb{P} :=$$

$$\frac{}{\alpha \mapsto_{\mathtt{n}} \alpha} \qquad \frac{y \mapsto_{\mathtt{n}} n_y \qquad z \mapsto_{\mathtt{n}} n_z}{\omega\,\alpha\,y\,z \mapsto_{\mathtt{n}} \omega\,\alpha\,n_y\,n_z}$$

$$\frac{\omega\,b\,y\,z \mapsto_{\mathtt{n}} n_b \qquad \omega\,c\,y\,z \mapsto_{\mathtt{n}} n_c \qquad \omega\,a\,n_b\,n_c \mapsto_{\mathtt{n}} n_a}{\omega\,(\omega\,a\,b\,c)\,y\,z \mapsto_{\mathtt{n}} n_a}\,.$$

In line with the previous sections, $e \mapsto_{\mathtt{n}} n$ is just a convenient infix notation for the prefix $\mathbb{G}_{\mathtt{nm}}\ e\ n$ notation. We show that the graph $\mathbb{G}_{\mathtt{nm}}$ is a functional relation.

$$\texttt{Fact}\ \ \mathbb{G}_{\mathtt{nm}}\texttt{\_fun}\ e\ n_1\ n_2 : \quad e \mapsto_{\mathtt{n}} n_1 \ \to\ e \mapsto_{\mathtt{n}} n_2 \ \to\ n_1 = n_2.$$

Proof. By induction on $e \mapsto_{\mathtt{n}} n_1$ then inversion on $e \mapsto_{\mathtt{n}} n_2$. □

We give a first possible characterization of the domain $\mathbb{D}_{\mathtt{nm}}$ of `nm` by a custom domain predicate:

$$\texttt{Inductive}\ \ \mathbb{D}_{\mathtt{nm}} : \Omega \to \mathbb{P} :=$$

$$\frac{}{\mathbb{D}_{\mathtt{nm}}\ \alpha} \qquad \frac{\mathbb{D}_{\mathtt{nm}}\ y \quad \mathbb{D}_{\mathtt{nm}}\ z}{\mathbb{D}_{\mathtt{nm}}\,(\omega\,\alpha\,y\,z)}$$

$$\frac{\begin{array}{c}\mathbb{D}_{\mathtt{nm}}\,(\omega\,b\,y\,z) \qquad\qquad \mathbb{D}_{\mathtt{nm}}\,(\omega\,c\,y\,z)\\ \forall n_b\,n_c,\ \omega\,b\,y\,z \mapsto_{\mathtt{n}} n_b \ \to\ \omega\,c\,y\,z \mapsto_{\mathtt{n}} n_c \ \to\ \mathbb{D}_{\mathtt{nm}}\,(\omega\,a\,n_b\,n_c)\end{array}}{\mathbb{D}_{\mathtt{nm}}\,(\omega\,(\omega\,a\,b\,c)\,y\,z)}\,.$$

The intuition behind the construction of $\mathbb{D}_{\mathtt{nm}}$ is to simply erase the right-hand side part (i.e. output part) of $\mathbb{G}_{\mathtt{nm}}$: when we have $e \mapsto_{\mathtt{n}} n$, we only keep what is on the left of the $\mapsto_{\mathtt{n}}$ symbol and we get $\mathbb{D}_{\mathtt{nm}}\ e$. This is what we already did in the cases of the `ns` searching algorithm in Section 3.1, or of the depth first search algorithm `dfs` of Section 6. However neither `ns` nor `dfs` have nested calls while `nm` has two.

We now explain how to cope with nested calls when designing custom domain predicates. When there is a nested call, then its output is transferred on the left-hand side (i.e. the input part) of another premise and we simply cannot leave a dangling variable not referring to anything that way. So, we characterize/recover the erased output by using the computational graph $\mathbb{G}_{\mathtt{nm}}$ combined with universal quantification. This is what happens in the lower premise of the third rule.

The third central idea of the Braga method*: when dealing with nested or mutually recursive algorithms, one can use the computational graph predicate to characterize the output values of nested calls than come as input for the domain predicate.*

As hinted in the introduction of this section, we now discuss a second and alternate construction of the domain, denoted $\mathbb{D}'_{\mathtt{nm}}$, and based on a different intuition. First, we link calls to `nm` with the direct recursive subcalls they trigger in the $\preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$ binary subcall/call relation:

$$\texttt{Inductive}\ \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}} : \Omega \to \Omega \to \mathbb{P} :=$$

$$\frac{}{y \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}} \omega\,\alpha\,y\,z} \quad \frac{}{\omega\,b\,y\,z \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}} \omega\,(\omega\,a\,b\,c)\,y\,z} \quad \frac{\omega\,b\,y\,z \mapsto_{\mathtt{n}} n_b \qquad \omega\,c\,y\,z \mapsto_{\mathtt{n}} n_c}{\omega\,a\,n_b\,n_c \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}} \omega\,(\omega\,a\,b\,c)\,y\,z}\,.$$

$$\frac{}{z \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}} \omega\,\alpha\,y\,z} \quad \frac{}{\omega\,c\,y\,z \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}} \omega\,(\omega\,a\,b\,c)\,y\,z}$$

The relation $\preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$ is defined with inductive rules but if you look closely, $\preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$ never appears on any premise of any rule, hence induction is just a presentation/programming convenience here, not a requirement. Note, however, that $\mathbb{G}_{\mathtt{nm}}$ is used in the two premises of the rightmost rule, to characterize nested calls similarly to the case of the custom domain predicate $\mathbb{D}_{\mathtt{nm}}$.

Having linked recursive subcalls with $\preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$, following the general description of Section 4.3, we state that the domain is composed of the input values from which no infinite descending $\preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$-chain start, conventionally called the well-founded part of the $\preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$ relation, and inductively characterized by the accessibility predicate $\mathtt{Acc}\ \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}$.

$$\texttt{Definition}\ \mathbb{D}'_{\mathtt{nm}}\ (e : \Omega) := \mathtt{Acc}\ \preccurlyeq^{\mathrm{sc}}_{\mathtt{nm}}\ e.$$

Below we simply denote $\mathbb{D}_{\mathtt{nm}}$ for the domain predicate but note that the discussion would be mostly same were we to use the alternate definition $\mathbb{D}'_{\mathtt{nm}}$ instead. Only some technical details differ slightly but not the main results we present in here. We will however discuss some of these differences.

7.2. *The Coq term packed with a conformity certificate*

So, with either definition of the domain, be it $\mathbb{D}_{\mathtt{nm}}$ or $\mathbb{D}'_{\mathtt{nm}}$, we now implement the `nm` algorithm *packed with a conformity* certificate, as a term of type

$$\mathtt{nm\_pwc} : \forall e : \Omega,\ \mathbb{D}_{\mathtt{nm}}\ e \to \{n \mid e \mapsto_{\mathtt{n}} n\}.$$

Its computational contents is displayed in Fig. 19 but the contents of *proof obligations* is not displayed for concision. Theses are divided into three post conditions $\mathcal{O}^?_1$–$\mathcal{O}^?_3$ and five termination certificates $\mathcal{T}^?_1$–$\mathcal{T}^?_5$:

- the post conditions $\mathcal{O}^?_1$–$\mathcal{O}^?_3$ are proved very directly by applying the corresponding constructor/rule of the inductive definition of $\mathbb{G}_{\mathtt{nm}}$;
- the termination certificates $\mathcal{T}^?_1$–$\mathcal{T}^?_5$ have more complex proofs, in particular if the domain is defined as the custom predicate $\mathbb{D}_{\mathtt{nm}}$. In that case, one should be careful with the guardedness condition,

```
Let Fixpoint nm_pwc $e$ $(D : \mathbb{D}_{\mathtt{nm}}\ e)$ {struct $D$} : $\{n \mid e \mapsto_{\mathtt{n}} n\}$.
Proof. refine(
    match $e$ with
        | $\alpha$              $\Rightarrow \lambda D$, exist _ $\alpha$ $\mathcal{O}^?_1$
        | $\omega\,\alpha\,y\,z$        $\Rightarrow \lambda D$,
            let $(n_y, C_y)$ := nm_pwc $y$ $\mathcal{T}^?_1$ in
            let $(n_z, C_z)$ := nm_pwc $z$ $\mathcal{T}^?_2$ in
                          exist _ $(\omega\,\alpha\,n_y\,n_z)$ $\mathcal{O}^?_2$
        | $\omega\,(\omega\,a\,b\,c)\,y\,z \Rightarrow \lambda D$,
            let $(n_b, C_b)$ := nm_pwc $(\omega\,b\,y\,z)$ $\mathcal{T}^?_3$ in
            let $(n_c, C_c)$ := nm_pwc $(\omega\,c\,y\,z)$ $\mathcal{T}^?_4$ in
            let $(n_a, C_a)$ := nm_pwc $(\omega\,a\,n_b\,n_c)$ $\mathcal{T}^?_5$ in
                          exist _ $n_a$ $\mathcal{O}^?_3$
    end $D$).
    (* POs: termination certs $\mathcal{T}^?_{1\text{–}5}$; postconditions $\mathcal{O}^?_{1\text{–}3}$ *)
Qed.
```

Fig. 19. Coq proof term `nm_pwc` of the `nm` algorithm packed with correctness.

e.g. the proof term of $\mathcal{T}_1^?$

$$[\mathcal{T}_1^?] : \ldots, y : \Omega, z : \Omega, D : \mathbb{D}_{\mathtt{nm}}\,(\omega\,\alpha\,y\,z) \vdash \mathbb{D}_{\mathtt{nm}}\ y$$

should be built as a subterm of D. Because $\mathbb{D}_{\mathtt{nm}}$ has several constructors, this requires dependent pattern matching which is properly implemented by the `inversion` tactic and explicit projections by "small inversions", as explained in the previous sections.
In the case of the alternate definition $\mathbb{D}'_{\mathtt{nm}} := \mathtt{Acc}\ \prec^{\mathrm{sc}}_{\mathtt{nm}}$, a simple pattern matching on $D : \mathbb{D}'_{\mathtt{nm}}\ \_$ (as implemented in the `Acc_inv` lemma) is sufficient for ensuring structural decrease.

Now we can define `nm` by projecting on the first component of the Σ-type $\{n \mid e \mapsto_{\mathtt{n}} n\}$ containing the output value

$$\begin{array}{l}\mathtt{Definition}\ \ \mathtt{nm}\ e\ (D : \mathbb{D}_{\mathtt{nm}}\ e) := \pi_1(\mathtt{nm\_pwc}\ e\ D).\\ \mathtt{Fact}\ \ \mathtt{nm\_spec}\ e\ (D : \mathbb{D}_{\mathtt{nm}}\ e) :\ \ e \mapsto_{\mathtt{n}} \mathtt{nm}\ e\ D\end{array}$$

and with the second component $\pi_2(\mathtt{nm\_pwc}\ e\ D)$, we get its specification `nm_spec` expressing the conformity proof of the output value.

7.3. *The inductive–recursive scheme*

We build tailored inductive–recursive constructors for the domain. As `nm` is a nested recursive algorithm, the constructors refer to the function itself, more precisely, on the values it outputs in nested calls.

$$\begin{array}{lll}\mathtt{Facts}: & & \\ \quad \mathbb{D}^1_{\mathtt{nm}} : & & \mathbb{D}_{\mathtt{nm}}\ \alpha. \\ \quad \mathbb{D}^2_{\mathtt{nm}} : \forall y\,z, & & \mathbb{D}_{\mathtt{nm}}\ y \to \mathbb{D}_{\mathtt{nm}}\ z \to \mathbb{D}_{\mathtt{nm}}(\omega\,\alpha\,y\,z). \\ \quad \mathbb{D}^3_{\mathtt{nm}} : \forall a\,b\,c\,y\,z\,D_b\,D_c, & & \mathbb{D}_{\mathtt{nm}}\big(\omega\,a\,(\mathtt{nm}\ (\omega\,b\,y\,z)\ D_b)\,(\mathtt{nm}\ (\omega\,c\,y\,z)\ D_c)\big) \\ & \to & \mathbb{D}_{\mathtt{nm}}\big(\omega\,(\omega\,a\,b\,c)\,y\,z\big).\end{array}$$

Proof. Depending whether one chooses $\mathbb{D}_{\mathtt{nm}}$ or $\mathbb{D}'_{\mathtt{nm}}$, the proofs somewhat differ in here but they are always straightforward. □

We follow up on the inductive–recursive scheme for `nm` with a proof-irrelevant eliminator/induction principle for $\mathbb{D}_{\mathtt{nm}}$ (or else $\mathbb{D}'_{\mathtt{nm}}$).

It states that a predicate $P : \forall e, \mathbb{D}_{\mathtt{nm}}\ e \to \mathtt{Type}$ which is both proof-irrelevant and closed under the three constructors $\mathbb{D}^1_{\mathtt{nm}}$–$\mathbb{D}^3_{\mathtt{nm}}$ holds over the whole domain:

$$\begin{aligned}
&\mathtt{Theorem}\ \mathbb{D}_{\mathtt{nm}}\mathtt{\_rect}\ (P : \forall e, \mathbb{D}_{\mathtt{nm}}\ e \to \mathtt{Type}) :\\
&\quad \big(\forall e\, D_1\, D_2,\ P\ e\ D_1 \to P\ e\ D_2\big)\\
&\to \big(P\ \_\ \mathbb{D}^1_{\mathtt{nm}}\big)\\
&\to \big(\forall y\, z\, D_y\, D_z,\ P\ y\ D_y \to P\ z\ D_z \to P\ \_\ (\mathbb{D}^2_{\mathtt{nm}}\ y\ z\ D_y\ D_z)\big)\\
&\to \big(\forall a\, b\, c\, y\, z\, D_b\, D_c\, D_a,\ P\ \_\ D_b \to P\ \_\ D_c \to P\ \_\ D_a\\
&\qquad\qquad \to P\ \_\ (\mathbb{D}^3_{\mathtt{nm}}\ a\ b\ c\ y\ z\ D_b\ D_c\ D_a)\big)\\
&\to \big(\forall e\, D,\ P\ e\ D\big).
\end{aligned}$$

Proof. The technical details of the proof here depends on the choice of $\mathbb{D}_{\mathtt{nm}}$ or the alternate $\mathbb{D}'_{\mathtt{nm}}$, but in either case, it proceeds by `Fixpoint`s with $D : \mathbb{D}_{\mathtt{nm}}\ e$ as `struct` parameter. Then the pattern matching is on e —not D!— but we later implement careful inversion/projections of D to ensure decrease of the recursive sub-calls. It is very similar to the term build for `nm_pwc` in Fig. 19 except that here we do not need to control the computational contents so tightly because $\mathbb{D}_{\mathtt{nm}}$`_rect` is not intended to be extracted. □

We finish the construction of the inductive–recursive scheme for `nm` with the proof irrelevance of `nm` and fixpoint equations.

$$\begin{aligned}
&\mathtt{Facts} :\\
&\quad \mathtt{nm\_pirr} : \forall e\, D_1\, D_2, && \mathtt{nm}\ e\ D_1 = \mathtt{nm}\ e\ D_2.\\
&\quad \mathtt{nm\_fix\_1} : && \mathtt{nm}\ \alpha\ \mathbb{D}^1_{\mathtt{nm}} = \alpha.\\
&\quad \mathtt{nm\_fix\_2} : \forall y\, z\, D_y\, D_z, && \mathtt{nm}\ (\omega\,\alpha\,y\,z)\ (\mathbb{D}^2_{\mathtt{nm}}\ y\ z\ D_y\ D_z)\\
&&& = \omega\,\alpha\,(\mathtt{nm}\ y\ D_y)\,(\mathtt{nm}\ z\ D_z).\\
&\quad \mathtt{nm\_fix\_3} : \forall a\, b\, c\, y\, z\, D_b\, D_c\, D_a, && \mathtt{nm}\ \big(\omega\,(\omega\,a\,b\,c)\,y\,z\big)\ (\mathbb{D}^3_{\mathtt{nm}}\ \_\ \_\ \_\ \_\ \_\ D_b\ D_c\ D_a)\\
&&& = \mathtt{nm}\ \big(\omega\,a\,(\mathtt{nm}\ (\omega\,b\,y\,z)\ D_b)\,(\mathtt{nm}\ (\omega\,c\,y\,z)\ D_c)\big)\ D_a.
\end{aligned}$$

Proof. The proofs are very short and based on the functionality $\mathbb{G}_{\mathtt{nm}}$`_fun` of $\mathbb{G}_{\mathtt{nm}}$ and `nm_spec`. They are the same whether for $\mathbb{D}_{\mathtt{nm}}$ or $\mathbb{D}'_{\mathtt{nm}}$. □

7.4. *High-level partial correctness results*

Now that we have built the inductive–recursive scheme for nm, we can prove partial correctness properties of nm following the outline of Giesl [22]. Here, we present three of those partial correctness results, the first one being proved using the full inductive–recursive scheme and the two other results, by graph induction instead. These two approaches are in fact interchangeable in the case of nm.

Let us start by showing that nm outputs expressions in normal form, i.e. when the Boolean condition b in `if` b `then _ else _` is always atomic. We characterized this notion inductively as

$$\texttt{Inductive normal} : \Omega \to \mathbb{P} := \quad \frac{}{\texttt{normal}\ \alpha} \quad \frac{\texttt{normal}\ y \quad \texttt{normal}\ z}{\texttt{normal}\,(\omega\,\alpha\,y\,z)}\ .$$

With this definition, we prove the following partial correctness result:

$$\texttt{Theorem nm\_normal}\ e\ (D : \mathbb{D}_{\texttt{nm}}\ e) : \quad \texttt{normal}\,(\texttt{nm}\ e\ D).$$

Proof. Here, we use the full inductive–recursive scheme of nm. The proof proceeds by induction on D using $\mathbb{D}_{\texttt{nm}}$`_rect`. There are four inductive cases to establish:

(1) the proof-irrelevance of $\lambda e\,D,\ \texttt{normal}\,(\texttt{nm}\ e\ D)$, follows trivially from that of nm proved as `nm_pirr`;
(2) for the second inductive case, we rewrite using `nm_fix_1` and get $\texttt{normal}\ \alpha$ which holds by the first rule of normal;
(3) for the third inductive case, we rewrite using `nm_fix_2` and we need to show $\texttt{normal}\,\big(\omega\,\alpha\,(\texttt{nm}\ y\ D_y)\,(\texttt{nm}\ z\ D_z)\big)$ while assuming $\texttt{normal}\,(\texttt{nm}\ y\ D_y)$ and $\texttt{normal}\,(\texttt{nm}\ z\ D_z)$ as induction hypotheses. Hence the second rule of normal does the job;
(4) for the fourth inductive case, after rewriting using `nm_fix_3`, we are invited to show

$$\texttt{normal}\,\Big(\texttt{nm}\ \big(\omega\,a\,(\texttt{nm}\ (\omega\,b\,y\,z)\ D_b)\,(\texttt{nm}\ (\omega\,c\,y\,z)\ D_c)\big)\ D_a\Big)$$

but this is precisely the statement of the third induction hypothesis.

This completes the four cases of the induction on $\mathbb{D}_{\texttt{nm}}\ e$. □

Let us now show that, while `nm` is normalizing, it also preserves the semantics of `if _ then _ else _` expressions. We could do this by explicitly defining a semantic interpretation of Ω but we proceed otherwise by defining an "equivalence" relation that would be satisfied by any reasonable semantic interpretation of Ω, i.e. any two equivalent expressions would necessarily have the same interpretation. We use the least congruence which allows for commutation in the composition of Boolean conditions, i.e. identifying `if` (`if` a `then` b `else` c) `then` y `else` z and `if` a `then` (`if` b `then` y `else` z) `else` (`if` c `then` y `else` z). This can be characterized inductively by the following rules:

$$\texttt{Inductive}\ \sim_\Omega : \Omega \to \Omega \to \mathbb{P} :=$$

$$\frac{}{\alpha \sim_\Omega \alpha} \qquad \frac{x \sim_\Omega y \quad y \sim_\Omega z}{x \sim_\Omega z} \qquad \frac{x \sim_\Omega x' \quad y \sim_\Omega y' \quad z \sim_\Omega z'}{\omega\, x\, y\, z \sim_\Omega \omega\, x'\, y'\, z'}$$

$$\frac{}{\omega\,(\omega\, a\, b\, c)\, y\, z \sim_\Omega \omega\, a\,(\omega\, b\, y\, z)\,(\omega\, c\, y\, z)}\,.$$

The reader might have noted that we left out the symmetry rule, hence $\sim_\Omega$ is only contained in the above mentioned congruence, even strictly b.t.w.[v] However, the symmetry rule is not needed and $\sim_\Omega$ is large enough to show the following partial correctness result:

$$\texttt{Theorem nm\_equiv}\ e\ (D : \mathbb{D}_{\texttt{nm}}\ e) : \quad e \sim_\Omega \texttt{nm}\ e\ D.$$

Proof. We could also proceed by induction on D using $\mathbb{D}_{\texttt{nm}}$`_rect` but here we want to illustrate the alternate method of graph induction. In that spirit, thanks to `nm_spec`, it is enough to show

$$\forall e\, n,\ e \mapsto_{\texttt{n}} n\ \to\ e \sim_\Omega n$$

and we establish this by induction on (the proof term of) $e \mapsto_{\texttt{n}} n$:

(1) for the first rule of $\mathbb{G}_{\texttt{nm}}$, we need to show $\alpha \sim_\Omega \alpha$ which is trivial using the first rule of $\sim_\Omega$;
(2) for the second rule of $\mathbb{G}_{\texttt{nm}}$, we need to show $\omega\, \alpha\, y\, z \sim_\Omega \omega\, \alpha\, n_y\, n_z$ while assuming $y \sim_\Omega n_y$ and $z \sim_\Omega n_z$ as induction hypotheses. We conclude with the third (or congruence) rule of $\sim_\Omega$;

[v] For example, one can prove that $\omega\, a\,(\omega\, b\, y\, z)\,(\omega\, c\, y\, z) \not\sim_\Omega \omega\,(\omega\, a\, b\, c)\, y\, z$, see `equiv_not_sym`.

(3) for the third rule of $\mathbb{G}_{\texttt{nm}}$, we need to show $\omega\,(\omega\,a\,b\,c)\,y\,z \sim_\Omega n_a$ while assuming $\omega\,b\,y\,z \sim_\Omega n_b$, $\omega\,c\,y\,z \sim_\Omega n_c$ and $\omega\,a\,n_b\,n_c \sim_\Omega n_a$ as induction hypotheses. We use the fourth rule of $\sim_\Omega$ combined with reflexivity, transitivity and congruence. Reflexivity (i.e. $\forall e,\ e \sim_\Omega e$) itself is proved separately by structural induction on e.

This concludes the three cases of $\mathbb{G}_{\texttt{nm}}$ graph induction. □

We remark that the graph induction method deployed in the previous proof (after having removed the reference to `nm` e D with `nm_spec`) does not involve any of the tools of its inductive–recursive scheme any more. In fact, it does not even involve `nm`, just its computational graph $\mathbb{G}_{\texttt{nm}}$.

Actually, graph induction can generally be used as an alternative way to capture extensional properties of `nm`, specifically because of `nm_spec`. However, to some users, directly manipulating the output values of `nm` through `nm` e D might be viewed favorably as opposed to using a relational description of it. It can also be more convenient when combining `nm` with other functions.

On the other hand, the graph induction method allows to avoid the construction of inductive–recursive scheme of `nm`, except for the domain constructors (see below `nm_term`), i.e. with graph induction, one does not need the proof-irrelevant eliminator $\mathbb{D}_{\texttt{nm}}$`_rect`, and neither proof-irrelevance of `nm` nor its fixpoint equations.

For us, we think both methods are fine and it is up to the user to decide which one he finds more convenient to a particular application.

Let us now prepare the termination proof of `nm`. For this we need a third partial correctness result stating that `nm` preserves a particular measure. We define the measure $\langle\!\langle\cdot\rangle\!\rangle : \Omega \to \mathbb{N}$ over Ω by structural induction:

$$\langle\!\langle\alpha\rangle\!\rangle := 1 \qquad \langle\!\langle\omega\,x\,y\,z\rangle\!\rangle := \langle\!\langle x\rangle\!\rangle\big(1 + \langle\!\langle y\rangle\!\rangle + \langle\!\langle z\rangle\!\rangle\big).$$

Observe that this definition ensures that $\langle\!\langle e\rangle\!\rangle$ is never 0,

Fact `ce_size_ge_1` e : $1 \leq \langle\!\langle e\rangle\!\rangle$.

Then we establish the following remarkable strict inequality [22]:

Fact `ce_size_special` $a\ b\ c\ y\ z$:

$$\langle\!\langle\omega\,a\,(\omega\,b\,y\,z)\,(\omega\,c\,y\,z)\rangle\!\rangle < \langle\!\langle\omega\,(\omega\,a\,b\,c)\,y\,z\rangle\!\rangle$$

by a mostly straightforward arithmetic computation. We show the following partial correctness result:

Theorem `nm_dec` e $(D : \mathbb{D}_{\texttt{nm}}\ e)$: $\quad \langle\!\langle \texttt{nm}\ e\ D \rangle\!\rangle \le \langle\!\langle e \rangle\!\rangle$.

Proof. Using `nm_spec`, it is enough to show

$$\forall e\, n,\ e \mapsto_{\texttt{n}} n \ \rightarrow\ \langle\!\langle n \rangle\!\rangle \le \langle\!\langle e \rangle\!\rangle$$

and we prove this by induction on the graph predicate $e \mapsto_{\texttt{n}} n$:

(1) for the first rule of $\mathbb{G}_{\texttt{nm}}$, we have to show $\langle\!\langle \alpha \rangle\!\rangle \le \langle\!\langle \alpha \rangle\!\rangle$ which is trivial;
(2) for the second rule of $\mathbb{G}_{\texttt{nm}}$, while assuming $\langle\!\langle n_y \rangle\!\rangle \le \langle\!\langle y \rangle\!\rangle$ and $\langle\!\langle n_z \rangle\!\rangle \le \langle\!\langle z \rangle\!\rangle$ as induction hypotheses, we have to show $\langle\!\langle \omega\, \alpha\, n_y\, n_z \rangle\!\rangle \le \langle\!\langle \omega\, \alpha\, y\, z \rangle\!\rangle$. This computes into $1 + \langle\!\langle n_y \rangle\!\rangle + \langle\!\langle n_z \rangle\!\rangle \le 1 + \langle\!\langle y \rangle\!\rangle + \langle\!\langle z \rangle\!\rangle$ easily solved by an arithmetic tactic;
(3) for the third rule of $\mathbb{G}_{\texttt{nm}}$, while assuming $\langle\!\langle n_b \rangle\!\rangle \le \langle\!\langle \omega\, b\, y\, z \rangle\!\rangle$, $\langle\!\langle n_c \rangle\!\rangle \le \langle\!\langle \omega\, c\, y\, z \rangle\!\rangle$ and $\langle\!\langle n_a \rangle\!\rangle \le \langle\!\langle \omega\, a\, n_b\, n_c \rangle\!\rangle$, we have to show $\langle\!\langle n_a \rangle\!\rangle \le \langle\!\langle \omega\, (\omega\, a\, b\, c)\, y\, z \rangle\!\rangle$. But by monotonicity we have

$$\langle\!\langle n_a \rangle\!\rangle \le \langle\!\langle \omega\, a\, n_b\, n_c \rangle\!\rangle \le \langle\!\langle \omega\, a\, (\omega\, b\, y\, z)\, (\omega\, c\, y\, z) \rangle\!\rangle$$

and we finish with the above remarkable inequality `ce_size_special`.

This concludes the three cases of $\mathbb{G}_{\texttt{nm}}$ graph induction. $\square$

7.5. *Termination and total correctness*

We conclude the theoretical study of `nm` with its termination proof, i.e. the domain $\mathbb{D}_{\texttt{nm}}$ holds over the whole input type:

Theorem $\mathbb{D}_{\texttt{nm}}$`_total` : $\quad \forall e : \Omega,\ \mathbb{D}_{\texttt{nm}}\ e$.

Proof. We proceed by strong induction on $\langle\!\langle e \rangle\!\rangle$ while using partial correctness `nm_dec`. Then we distinguish three cases: $e = \alpha$, $e = \omega\, \alpha\, y\, z$ or $e = \omega\, (\omega\, a\, b\, c)\, y\, z$ by pattern matching:

(1) of course $\mathbb{D}^1_{\texttt{nm}}$ establishes $\mathbb{D}_{\texttt{nm}}\ \alpha$;
(2) with $\mathbb{D}^2_{\texttt{nm}}$, proving $\mathbb{D}_{\texttt{nm}}\ (\omega\, \alpha\, y\, z)$ is reduced into proving both $\mathbb{D}_{\texttt{nm}}\ y$ and $\mathbb{D}_{\texttt{nm}}\ z$ which hold by induction. Indeed, it is easy to show $\langle\!\langle y \rangle\!\rangle < \langle\!\langle \omega\, \alpha\, y\, z \rangle\!\rangle$ and $\langle\!\langle z \rangle\!\rangle < \langle\!\langle \omega\, \alpha\, y\, z \rangle\!\rangle$;

(3) and finally, we use $\mathbb{D}^3_{\texttt{nm}}$ to establish $\mathbb{D}_{\texttt{nm}}\big(\omega\,(\omega\,a\,b\,c)\,y\,z\big)$. We are thus invited to prove $D_b : \mathbb{D}_{\texttt{nm}}(\omega\,b\,y\,z)$, $D_c : \mathbb{D}_{\texttt{nm}}(\omega\,c\,y\,z)$ and then $\mathbb{D}_{\texttt{nm}}\big(\omega\,a\,(\texttt{nm}\ \_\ D_b)\,(\texttt{nm}\ \_\ D_c)\big)$. By D_b and D_c are directly built using the induction hypothesis because $\langle\!\langle \omega\,u\,y\,z\rangle\!\rangle < \langle\!\langle \omega\,(\omega\,a\,b\,c)\,y\,z\rangle\!\rangle$ holds for $u \in \{b, c\}$. Then we use `ce_size_special` which allow to prove

$$\langle\!\langle \omega\,a\,(\texttt{nm}\ \_\ D_b)\,(\texttt{nm}\ \_\ D_c)\rangle\!\rangle \le \langle\!\langle \omega\,a\,(\omega\,b\,y\,z)\,(\omega\,c\,y\,z)\rangle\!\rangle < \langle\!\langle \omega\,(\omega\,a\,b\,c)\,y\,z\rangle\!\rangle.$$

Note that we use $\langle\!\langle \texttt{nm}\ (\omega\,u\,y\,z)\ D_u\rangle\!\rangle \le \langle\!\langle \omega\,u\,y\,z\rangle\!\rangle$ for $u \in \{b, c\}$ which comes from the partial correctness result `nm_dec`.

The three aforementioned cases covering the whole domain, the proof is completed. □

Considering this last proof, critically, a partial correctness result is used to establish termination: we need some properties of the output value to be able to establish termination. This is typical of nested recursive schemes and what makes them *a priori* hard/impossible to implement in the naive approach through structural induction. Even well-founded induction is difficult because the inductive structure of the domain depends on the output of the function itself.

We can conclude with the fully specified and terminating Paulson's normalization algorithm, i.e. total correctness of the `nm` algorithm:

```
Definition pnm $(e : \Omega)$ :    $\{n \mid$ normal $n \wedge e \sim_\Omega n\}$.
```

Extraction works flawlessly giving

```
type $\Omega = \alpha \mid \omega$ of $\Omega * \Omega * \Omega$
let rec pnm $e$ = match $e$ with
  | $\alpha$                $\to \alpha$
  | $\omega(\alpha, y, z)$          $\to \omega(\alpha,$ pnm $y,$ pnm $z)$
  | $\omega(\omega(a, b, c), y, z)$ $\to$ pnm$(\omega(a,$ pnm$(\omega(b, y, z)),$ pnm$(\omega(c, y, z))))$
```

8. First-Order Unification

Considering a type of terms, here binary trees denoted Λ, composed using the infix $\diamond$ operator and with leaves decorated either with variables like $\mu\,x$ or with constants like $\varphi\,c$, the unification of two given

$$\begin{array}{lllll}
\texttt{unif}\ (\mu\,x) & m & = \texttt{Some}\,[(x,m)] & \text{if } x \not\prec m \\
\texttt{unif}\ (\varphi\,c) & (\mu\,x) & = \texttt{Some}\,[(x,\varphi\,c)] & \\
\texttt{unif}\ (\varphi\,c) & (\varphi\,d) & = \texttt{Some}\,[\,] & \text{if } c = d \\
\texttt{unif}\ (m \diamond n) & (\mu\,x) & = \texttt{Some}\,[(x,m \diamond n)] & \text{if } x \not\prec m \diamond n \\
\texttt{unif}\ (m \diamond n) & (m' \diamond n') & = \texttt{Some}\,(\sigma \circ \nu) & \text{when } \begin{cases} \texttt{unif}\ m\ m' = \texttt{Some}\,\sigma \\ \texttt{unif}\ n\{\!\{\sigma\}\!\}\ n'\{\!\{\sigma\}\!\} = \texttt{Some}\,\nu \end{cases} \\
\texttt{unif}\ \_ & \_ & = \texttt{None} & \text{in all other cases}
\end{array}$$

Fig. 20. Equations describing the `unif` algorithm.

terms consists in finding a substitution of the variables so that under this substitution, the two terms become identical. Actually unification not only seeks a substitution, it seeks a most general one.

We study the same nested unification algorithm as Krauss [14] which was first informally described by Manna and Waldinger [23] and later verified both in classical and constructive settings; see Slind [24] and Monin [25] for more details. The unification algorithm `unif` (with occur-check) is conventionally presented using the equations of Fig. 20. There, the notation $x \not\prec m$ means that x does not occur check in m.[w] Note that contrary to the usual practice, we make the constructors μ and φ for atomic terms (respectively variables and constants) explicit herein — but with a compact notation — to avoid any formal ambiguity. The algorithm computes optional substitutions, i.e. either a substitution $\texttt{Some}\,\sigma$ or a void value `None`, and substitutions are represented as lists of variable/term pairs. Moreover $\sigma \circ \nu$ represents the composition of the two substitutions σ and ν.

All calls to `unif` are terminal[x] except for the case `unif` $(m \diamond n)\ (m' \diamond n')$. In that call, there are two subcalls: first on `unif` $m\ m'$ and then possibly on `unif` $n\{\!\{\sigma\}\!\}\ n'\{\!\{\sigma\}\!\}$ creating a nesting between these recursive subcalls. Decision for the occur check condition $x \prec^? m$ is also a recursive algorithm but it employs structural recursion over terms, hence is quite trivial to implement, verify, and extract.

A call to $\texttt{unif}\,m\,n$ produces either $\texttt{Some}\,\sigma$ where σ is then a most general unifier for m/n, or `None` in which case m and n cannot be unified. In this section, we formalize and mechanically establish exactly this functional specification along with the termination of the computation of $\texttt{unif}\,m\,n$ whatever the values of m and n.

[w]That is, x cannot occur in m unless $m = \mu\,x$.
[x]That is, they respond without invoking any further recursive subcall.

The `unif` algorithm, though idealized herein, is quite useful in practice, typically in first order theorem provers, but also Coq itself uses a refinement of (higher-order) unification. This combination of usefulness and tricky nesting in the recursive scheme makes `unif` a prime target for applying our method, and this example would have been put up-front were it not for the preliminary notions required to present it, and the number of matching subcases that have to be considered.

8.1. *Preliminaries*

Let us now completely formalize `unif` in inductive type theory. We assume two discrete types $\mathcal{V}$ (for variables) and $\mathcal{C}$ for (constants). By discrete, we mean that $\mathcal{V}$ and $\mathcal{C}$ are each provided with a Boolean equality decider:

$$\begin{array}{ll} =^?_{\mathcal{V}} : \mathcal{V} \to \mathcal{V} \to \mathbb{B} & \texttt{eqV\_spec} : \forall x\, y : \mathcal{V},\ x =^?_{\mathcal{V}} y = \texttt{true} \leftrightarrow x = y \\ =^?_{\mathcal{C}} : \mathcal{C} \to \mathcal{C} \to \mathbb{B} & \texttt{eqC\_spec} : \forall a\, b : \mathcal{C},\ a =^?_{\mathcal{C}} b = \texttt{true} \leftrightarrow a = b. \end{array}$$

Note that, from these, we also define dependent deciders

$$\begin{aligned} &\texttt{eqV\_dec} : \forall x\, y : \mathcal{V},\ \{x = y\} + \{x \neq y\}, \\ &\texttt{eqC\_dec} : \forall a\, b : \mathcal{C},\ \{a = b\} + \{a \neq b\}, \end{aligned}$$

that extract as their respective Boolean decider $=^?_{\mathcal{V}}$ and $=^?_{\mathcal{C}}$ but are more convenient to use when combining programming and proving.

Given the types for constants and variables, we build the type Λ of terms which are binary trees with leaves either in $\mathcal{V}$ or $\mathcal{C}$:

$$m, n : \Lambda ::= \mu\, x \mid \varphi\, c \mid m \diamond n \quad \text{with } x : \mathcal{V} \text{ and } c : \mathcal{C}.$$

It is trivial to extend equality deciders to Λ as

$$=^?_{\Lambda} : \Lambda \to \Lambda \to \mathbb{B} \quad \texttt{eqT\_spec} : \forall s\, t : \Lambda,\ s =^?_{\Lambda} t = \texttt{true} \leftrightarrow s = t.$$

We define recursively the size $[\![\cdot]\!] : \Lambda \to \mathbb{N}$ and the list of variables $\langle\!\langle \cdot \rangle\!\rangle : \Lambda \to \mathbb{L}\,\mathcal{V}$ of terms by the structurally recursive equations:

$$\begin{array}{lll} [\![\mu\, \_]\!] := 0 & [\![\varphi\, \_]\!] := 0 & [\![m \diamond n]\!] := 1 + [\![m]\!] + [\![n]\!] \\ \langle\!\langle \mu\, x \rangle\!\rangle := [x] & \langle\!\langle \varphi\, \_ \rangle\!\rangle := [\,] & \langle\!\langle m \diamond n \rangle\!\rangle := \langle\!\langle m \rangle\!\rangle +\!\!+ \langle\!\langle n \rangle\!\rangle. \end{array}$$

The occur check decision algorithm $\prec^? : \mathcal{V} \to \Lambda \to \mathbb{B}$ is also defined by structural recursion

$$x \prec^? \mu\,\_ := \texttt{false} \qquad\qquad x \prec^? \varphi\,\_ := \texttt{false}$$
$$x \prec^? m \diamond n := \mu\,x =^?_\Lambda m \,||\, \mu\,x =^?_\Lambda n \,||\, x \prec^? m \,||\, x \prec^? n$$

and specified by

Fact `trm_vars_occ_check` $x\ m$: $\quad x \prec m \leftrightarrow m \neq \mu\,x \wedge x \in \langle\!\langle m \rangle\!\rangle.$

Note that to ensure shorter notations, we abusively write $x \prec m$ for $x \prec^? m = \texttt{true}$ and $x \not\prec m$ for $x \prec^? m = \texttt{false}$. Using $\prec^?$, we implement the dependent decider which allows both smooth extraction and better behavior w.r.t. proof obligations.

Definition `occ_check_dec` $x\ t$: $\quad \{x \prec t\} + \{x \not\prec t\}.$

Typically, when $x \prec m$ holds, which reads "x occurs check in m", then x and m cannot be unified, i.e. no common substitution will ever make them identical.[y] On the other hand, when $x \not\prec m$, any substitution that maps x to m unifies those two terms.

A (*finite*) *substitution* is a list of type $\Sigma := \mathbb{L}\,(\mathcal{V} \times \Lambda)$ composed of substitution pairs, and for $\sigma : \Sigma$, we define the substitutions of variables $\sigma{\uparrow}(\cdot) : \mathcal{V} \to \Lambda$ and of terms $(\cdot)\{\!|\sigma|\!\} : \Lambda \to \Lambda$ with the structural recursive equations:

$$[\,]{\uparrow}x := \mu\,x \qquad \big((x,t) :: \_\big){\uparrow}x := t \qquad \big((y,\_) :: \sigma\big){\uparrow}x := \sigma{\uparrow}x \ \text{ when } x \neq y$$
$$\mu\,x\{\!|\sigma|\!\} := \sigma{\uparrow}x \qquad \varphi\,c\{\!|\sigma|\!\} := \varphi\,c \qquad (m \diamond n)\{\!|\sigma|\!\} := m\{\!|\sigma|\!\} \diamond n\{\!|\sigma|\!\}.$$

Remark that the equality decider $=^?_\mathcal{V}$ is used for comparing x and y in the definition of $\sigma{\uparrow}(\cdot)$.

We define the *composition* $\sigma \circ \nu$ of two substitutions $(\sigma\ \nu : \Sigma)$ by:

$$\sigma \circ \nu := \texttt{map}\ \big(\lambda(x,t), (x, t\{\!|\nu|\!\})\big)\ \sigma +\!\!+\ \nu$$

and the composition satisfies the following specification:

Fact `subst_comp_spec` $\sigma\ \nu\ t$: $\quad t\{\!|\sigma \circ \nu|\!\} = t\{\!|\sigma|\!\}\{\!|\nu|\!\}.$

[y]Because $x\{\!|\sigma|\!\}$ would occur strictly in $m\{\!|\sigma|\!\}$ creating a discrepancy of sizes.

8.2. *The computational graph and the domain predicate*

Given all those preliminary notions, we can at last deploy the Braga method and define the graph of the `unif` function corresponding to the set of equations of Fig. 20. The graph is described as a purely logical inductive predicate. It relates the inputs with the potential output of `unif`, and its inductive description allows to follow the nested recursive scheme quite naturally:

$$\texttt{Inductive}\ \mathbb{G}_{\texttt{unif}} : \Lambda \to \Lambda \to \texttt{option}\,\Sigma \to \mathbb{P} :=$$

$$\frac{}{\varphi\, c \ltimes m \diamond n \mapsto_{\mathrm{u}} \texttt{None}} \qquad \frac{}{m \diamond n \ltimes \varphi\, c \mapsto_{\mathrm{u}} \texttt{None}} \qquad \frac{}{\varphi\, c \ltimes \mu\, x \mapsto_{\mathrm{u}} \texttt{Some}\,[(x, \varphi\, c)]}$$

$$\frac{x \prec m \diamond n}{m \diamond n \ltimes \mu\, x \mapsto_{\mathrm{u}} \texttt{None}} \qquad \frac{x \not\prec m \diamond n}{m \diamond n \ltimes \mu\, x \mapsto_{\mathrm{u}} \texttt{Some}\,[(x, m \diamond n)]} \qquad \frac{x \prec m}{\mu\, x \ltimes m \mapsto_{\mathrm{u}} \texttt{None}}$$

$$\frac{x \not\prec m}{\mu\, x \ltimes m \mapsto_{\mathrm{u}} \texttt{Some}\,[(x, m)]} \qquad \frac{a = b}{\varphi\, a \ltimes \varphi\, b \mapsto_{\mathrm{u}} \texttt{Some}\,[\,]} \qquad \frac{a \neq b}{\varphi\, a \ltimes \varphi\, b \mapsto_{\mathrm{u}} \texttt{None}}$$

$$\frac{m \ltimes m' \mapsto_{\mathrm{u}} \texttt{None}}{m \diamond n \ltimes m' \diamond n' \mapsto_{\mathrm{u}} \texttt{None}} \qquad \frac{m \ltimes m' \mapsto_{\mathrm{u}} \texttt{Some}\,\sigma \quad n\{\!\!\{\sigma\}\!\!\} \ltimes n'\{\!\!\{\sigma\}\!\!\} \mapsto_{\mathrm{u}} \texttt{None}}{m \diamond n \ltimes m' \diamond n' \mapsto_{\mathrm{u}} \texttt{None}}$$

$$\frac{m \ltimes m' \mapsto_{\mathrm{u}} \texttt{Some}\,\sigma \quad n\{\!\!\{\sigma\}\!\!\} \ltimes n'\{\!\!\{\sigma\}\!\!\} \mapsto_{\mathrm{u}} \texttt{Some}\,\nu}{m \diamond n \ltimes m' \diamond n' \mapsto_{\mathrm{u}} \texttt{Some}\,(\sigma \circ \nu)}$$

where the mixfix notation $m \ltimes n \mapsto_{\mathrm{u}} r$ is favored over the prefix notation $\mathbb{G}_{\texttt{unif}}\, m\, n\, r$. We establish the functionality of the graph $\mathbb{G}_{\texttt{unif}}$:

$$\texttt{Fact}\ \ \mathbb{G}_{\texttt{unif}}\texttt{\_fun}\ m\ n\ r\ s :\ \ m \ltimes n \mapsto_{\mathrm{u}} r\ \to\ m \ltimes n \mapsto_{\mathrm{u}} s\ \to\ r = s.$$

Proof. Quite typically, by induction on (the proof of) $m \ltimes n \mapsto_{\mathrm{u}} r$ and then inversion on $m \ltimes n \mapsto_{\mathrm{u}} s$. □

We follow with definition of the domain $\mathbb{D}_{\texttt{unif}}$ using the `Acc`essibility predicate applied to the below defined subcall relation $\preccurlyeq^{\mathrm{sc}}_{\mathrm{u}}$ of the `unif` recursive algorithm

$$\texttt{Definition}\ \ \mathbb{D}_{\texttt{unif}}\, u\, v := \texttt{Acc}\ \preccurlyeq^{\mathrm{sc}}_{\mathrm{u}}\ (u, v).$$

critically using the computational graph $\mathbb{G}_{\texttt{unif}}$ to characterize the nested recursive call in the second rule:

$$\texttt{Inductive}\ \preccurlyeq^{\mathrm{sc}}_{\mathrm{u}} : \Lambda \times \Lambda \to \Lambda \times \Lambda \to \mathbb{P} :=$$

$$\frac{}{(m, m') \preccurlyeq^{\mathrm{sc}}_{\mathrm{u}} (m \diamond n, m' \diamond n')} \qquad \frac{m \ltimes m' \mapsto_{\mathrm{u}} \texttt{Some}\,\sigma}{(n\{\!\!\{\sigma\}\!\!\}, n'\{\!\!\{\sigma\}\!\!\}) \preccurlyeq^{\mathrm{sc}}_{\mathrm{u}} (m \diamond n, m' \diamond n')}$$

8.3. *The Coq term packed with conformity*

We are now in position to build the `unification` function

$$\texttt{unif\_pwc} : \forall u\, v,\ \mathbb{D}_{\texttt{unif}}\, u\, v \to \{r \mid u \ltimes v \mapsto_{\texttt{u}} r\}$$

packed with conformity to $\mathbb{G}_{\texttt{unif}}$, of which the proof term is reported in Fig. 21. We first point out that although the two arguments u and v are packed in a pair (u, v) in the definition of the domain predicate $\mathbb{D}_{\texttt{unif}}\, u\, v$, there is no need to pack these two arguments in the definition of `unif_pwc`. This will reflect in the extracted term

```
Let Fixpoint unif_pwc $u$ $v$ $(D : \mathbb{D}_{\texttt{unif}}\, u\, v)$ {struct $D$} : $\{r \mid u \ltimes v \mapsto_{\texttt{u}} r\}$.
Proof. refine(match $u$ as $u'$ return $u = u' \to$ _ with
    | $\mu\, x$    ⇒ $\lambda E\, D$, match occ_check_dec $x$ $v$ with
      | left $H$  ⇒ exist _ None $\mathcal{O}^?_1$
      | right $H$ ⇒ exist _ Some $[x, v]$ $\mathcal{O}^?_2$
    end
    | $\varphi\, c$    ⇒ $\lambda E\, D$,
    match $v$ with
      | $\mu\, y$      ⇒ $\lambda D$, exist _ Some $[(y, u)]$ $\mathcal{O}^?_3$
      | $\varphi\, d$      ⇒ $\lambda D$, match eqC_dec $c$ $d$ with
        | left $H$  ⇒ exist _ Some [] $\mathcal{O}^?_4$
        | right $H$ ⇒ exist _ None $\mathcal{O}^?_5$
      end
      | $m' \diamond n'$ ⇒ $\lambda D$, exist _ None $\mathcal{O}^?_6$
    end $D$
    | $m \diamond n$ ⇒ $\lambda E\, D$, match $v$ with
      | $\mu\, y$      ⇒ $\lambda D$, match occ_check_dec $y$ $u$ with
        | left $H$  ⇒ exist _ None $\mathcal{O}^?_7$
        | right $H$ ⇒ exist _ Some $[(y, u)]$ $\mathcal{O}^?_8$
      end
      | $\varphi\, d$      ⇒ $\lambda D$, exist _ None $\mathcal{O}^?_9$
      | $m' \diamond n'$ ⇒ $\lambda D$, let $(r, G_r)$ := unif_pwc $m$ $m'$ $\mathcal{T}^?_1$ in
      match $r$ with
        | Some $\sigma$ ⇒ $\lambda G_r$, let $(s, G_s)$ := unif_pwc $n\{\!\{\sigma\}\!\}$ $n'\{\!\{\sigma\}\!\}$ $\mathcal{T}^?_2$ in
        match $s$ with
          | Some $\nu$ ⇒ $\lambda G_s$, exist _ Some $(\sigma \circ \nu)$ $\mathcal{O}^?_{10}$
          | None   ⇒ $\lambda G_s$, exist _ None $\mathcal{O}^?_{11}$
        end $G_s$
        | None   ⇒ $\lambda G_r$, exist _ None $\mathcal{O}^?_{12}$
      end $G_r$
    end $D$
  end eq_refl $D$).
  (* POs: termination certs $\mathcal{T}^?_{1-2}$; postconditions $\mathcal{O}^?_{1-12}$ *)
Qed.
```

Fig. 21. Coq proof term `unif_pwc` packed with conformity.

that will not pack u and v in a pair either. And $D : \mathbb{D}_{\texttt{unif}}\, u\, v$, in which u and v are packed as the (u, v) pair, will simply be erased because its type is purely logical.

Then, we remark that proof obligations in Fig. 21 are very easy to establish and only lightly discussed here: termination certificates $\mathcal{T}_1^?$ and $\mathcal{T}_2^?$ use `Acc_inv` to safely ensure the structural decrease for the fixpoint, as in Section 4.3; postconditions $\mathcal{O}_1^?$–$\mathcal{O}_{12}^?$ have trivial proofs, basically consisting in applying the corresponding rule/constructor of $\mathbb{G}_{\texttt{unif}}$.

Then, by projecting the Σ-type $\{r \mid u \ltimes v \mapsto_{\mathtt{u}} r\}$, we get `unif` as

$$\texttt{Definition unif}\ m\ n\ (D : \mathbb{D}_{\texttt{unif}}\, m\, n) := \pi_1(\texttt{unif\_pwc}\ m\ n\ D).$$
$$\texttt{Fact unif\_spec}\ m\ n\ D :\quad m \ltimes n \mapsto_{\mathtt{u}} \texttt{unif}\ m\ n\ D$$

dependent on the domain predicate $D : \mathbb{D}_{\texttt{unif}}\, m\, n$, whereas the projection $\pi_2(\texttt{unif\_pwc}\ m\ n\ D)$ provides conformity.

8.4. *The inductive–recursive scheme*

We implement suitable constructors for the domain $\mathbb{D}_{\texttt{unif}}$ which, for the last two of them, depend on the `unif` function themselves. This is what typically happens when simulating the induction–recursion scheme of a *nested recursive* algorithm.

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{Facts}:} \\
\mathbb{D}^1_{\texttt{unif}} : \forall c\, m\, n,\ \mathbb{D}_{\texttt{unif}}\, (\varphi\, c)\, (m \diamond n). & \mathbb{D}^2_{\texttt{unif}} : \forall c\, m\, n,\ \mathbb{D}_{\texttt{unif}}\, (m \diamond n)\, (\varphi\, c). \\
\mathbb{D}^3_{\texttt{unif}} : \forall c\, x,\quad \mathbb{D}_{\texttt{unif}}\, (\varphi\, c)\, (\mu\, x). & \mathbb{D}^4_{\texttt{unif}} : \forall m\, n\, x,\ \mathbb{D}_{\texttt{unif}}\, (m \diamond n)\, (\mu\, x). \\
\mathbb{D}^5_{\texttt{unif}} : \forall x\, m,\quad \mathbb{D}_{\texttt{unif}}\, (\mu\, x)\, m. & \mathbb{D}^6_{\texttt{unif}} : \forall c\, d,\quad \mathbb{D}_{\texttt{unif}}\, (\varphi\, c)\, (\varphi\, d).
\end{array}$$
$$\begin{array}{l}
\mathbb{D}^7_{\texttt{unif}} : \forall m\, n\, m'\, n'\, D,\quad \texttt{unif}\ m\ m'\ D = \texttt{None} \quad \to \mathbb{D}_{\texttt{unif}}\, (m \diamond n)\, (m' \diamond n'). \\
\mathbb{D}^8_{\texttt{unif}} : \forall m\, n\, m'\, n'\, D\, \sigma,\ \texttt{unif}\ m\ m'\ D = \texttt{Some}\, \sigma \to \mathbb{D}_{\texttt{unif}}\, (n\{\!|\sigma|\!\})\, (n'\{\!|\sigma|\!\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathbb{D}_{\texttt{unif}}\, (m \diamond n)\, (m' \diamond n').
\end{array}$$

Proof. With `Acc_intro` for $\mathbb{D}_{\texttt{unif}}$, then `unif_spec`/$\mathbb{G}_{\texttt{unif}}$`_fun`. □

We continue with the eliminator/recursion principle which expresses that any proof-irrelevant predicate $P : \forall m\, n,\ \mathbb{D}_{\texttt{unif}}\, m\, n \to$ `Type` holds over the whole domain $\mathbb{D}_{\texttt{unif}}$ when it is closed for the constructors:

$$
\begin{array}{l}
\texttt{Theorem}\ \mathbb{D}_{\texttt{unif}}\texttt{\_rect}\ (P : \forall m\,n,\ \mathbb{D}_{\texttt{unif}}\,m\,n \to \texttt{Type}) :\\
\quad \big(\forall m\,n\,D_1\,D_2,\ P\ m\ n\ D_1 \to P\ m\ n\ D_2\big)\\
\to \big(\forall c\,m\,n,\ P\ \_\ \_\ (\mathbb{D}^1_{\texttt{unif}}\ c\ m\ n)\big)\\
\to \big(\forall c\,m\,n,\ P\ \_\ \_\ (\mathbb{D}^2_{\texttt{unif}}\ c\ m\ n)\big)\\
\to \big(\forall c\,x,\ P\ \_\ \_\ (\mathbb{D}^3_{\texttt{unif}}\ c\ x)\big)\\
\to \big(\forall m\,n\,x,\ P\ \_\ \_\ (\mathbb{D}^4_{\texttt{unif}}\ m\ n\ x)\big)\\
\to \big(\forall x\,m,\ P\ \_\ \_\ (\mathbb{D}^5_{\texttt{unif}}\ x\ m)\big)\\
\to \big(\forall a\,b,\ P\ \_\ \_\ (\mathbb{D}^6_{\texttt{unif}}\ a\ b)\big)\\
\to \big(\forall m\,n\,m'\,n'\,D_1\,(\_ : P\ \_\ \_\ D_1)\,H,\ P\ \_\ \_\ (\mathbb{D}^7_{\texttt{unif}}\ \_\ \_\ \_\ \_\ D_1\,H)\big)\\
\to \big(\forall m\,n\,m'\,n'\,D_1\,(\_ : P\ \_\ \_\ D_1)\,\sigma\,H\,D_2,\\
\qquad\qquad P\ \_\ \_\ D_2 \to P\ \_\ \_\ (\mathbb{D}^8_{\texttt{unif}}\ \_\ \_\ \_\ \_\ D_1\ \_\ H\,D_2)\big)\\
\to \big(\forall m\,n\,D,\ P\ m\ n\ D\big).
\end{array}
$$

We finish the construction of the induction–recursion scheme of `unif` and establish proof-irrelevance and fixpoint equations:

`Facts` :

`unif_pirr`	$: \forall m\,n\,D_1\,D_2,\ \texttt{unif}\ m\ n\ D_1 = \texttt{unif}\ m\ n\ D_2.$
`unif_fix_1`	$: \forall c\,m\,n,\ \texttt{unif}\ \_\ \_\ (\mathbb{D}^1_{\texttt{unif}}\ c\ m\ n) = \texttt{None}.$
`unif_fix_2`	$: \forall c\,m\,n,\ \texttt{unif}\ \_\ \_\ (\mathbb{D}^2_{\texttt{unif}}\ c\ m\ n) = \texttt{None}.$
`unif_fix_3`	$: \forall c\,x,\ \texttt{unif}\ \_\ \_\ (\mathbb{D}^3_{\texttt{unif}}\ c\ x) = \texttt{Some}\,[(x, \varphi\, c)].$
`unif_fix_4`	$: \forall m\,n\,x,\ x \prec m \diamond n \to \texttt{unif}\ \_\ \_\ (\mathbb{D}^4_{\texttt{unif}}\ m\ n\ x) = \texttt{None}.$
`unif_fix_4'`	$: \forall m\,n\,x,\ x \not\prec m \diamond n \to \texttt{unif}\ \_\ \_\ (\mathbb{D}^4_{\texttt{unif}}\ m\ n\ x) = \texttt{Some}\,[(x, m \diamond n)].$
`unif_fix_5`	$: \forall x\,m,\ x \prec m \to \texttt{unif}\ \_\ \_\ (\mathbb{D}^5_{\texttt{unif}}\ x\ m) = \texttt{None}.$
`unif_fix_5'`	$: \forall x\,m,\ x \not\prec m \to \texttt{unif}\ \_\ \_\ (\mathbb{D}^5_{\texttt{unif}}\ x\ m) = \texttt{Some}\,[(x, m)].$
`unif_fix_6`	$: \forall c,\ \texttt{unif}\ \_\ \_\ (\mathbb{D}^6_{\texttt{unif}}\ c\ c) = \texttt{Some}\,[\,].$
`unif_fix_6'`	$: \forall c\,d,\ c \neq d \to \texttt{unif}\ \_\ \_\ (\mathbb{D}^6_{\texttt{unif}}\ c\ d) = \texttt{None}.$
`unif_fix_7`	$: \forall m\,n\,m'\,n'\,D\,H,\ \texttt{unif}\ \_\ \_\ (\mathbb{D}^7_{\texttt{unif}}\ m\ n\ m'\ n'\ D\ H) = \texttt{None}.$
`unif_fix_8`	$: \forall m\,n\,m'\,n'\,D_1\,\sigma\,H\,D_2,\ \texttt{unif}\ \_\ \_\ D_2 = \texttt{None}$ $\to \texttt{unif}\ \_\ \_\ (\mathbb{D}^8_{\texttt{unif}}\ m\ n\ m'\ n'\ D_1\ \sigma\ H\,D_2) = \texttt{None}.$
`unif_fix_8'`	$: \forall m\,n\,m'\,n'\,D_1\,\sigma\,H\,D_2\,\nu,\ \texttt{unif}\ \_\ \_\ D_2 = \texttt{Some}\,\nu$ $\to \texttt{unif}\ \_\ \_\ (\mathbb{D}^8_{\texttt{unif}}\ m\ n\ m'\ n'\ D_1\ \sigma\ H\,D_2) = \texttt{Some}\,(\sigma \circ \nu).$

8.5. *High-level partial correctness*

Once the inductive–recursive schemes in place, we can establish the partial correctness of `unif`, i.e. an *abstract specification* of what it computes on its domain. By abstract, we mean that we would get more information on `unif` $m\,n\,D$ than just the low-level result `unif_spec` that expresses conformity with the computational graph, i.e. that $m \ltimes n \mapsto_{\mathrm{u}} \texttt{unif}\,m\,n\,D$ holds.

Equivalence denoted $\sigma \approx \nu$ means that the two lists σ and ν of substitution pairs, despite being two potentially different lists, have the same extensional behavior:

$$\texttt{Infix} \approx : \Sigma \to \Sigma \to \mathbb{P}. \qquad \forall \sigma\, \nu : \Sigma,\ \sigma \approx \nu \leftrightarrow \forall t : \Lambda,\ t\{\!\!\{\sigma\}\!\!\} = t\{\!\!\{\nu\}\!\!\}.$$

Non-unifiability denoted $m \between n$ means no substitution can unify m and n:

$$\texttt{Infix} \between : \Lambda \to \Lambda \to \mathbb{P}. \quad \forall m\, n : \Lambda,\ m \between n \leftrightarrow \forall \sigma : \Sigma,\ m\{\!\!\{\sigma\}\!\!\} \neq n\{\!\!\{\sigma\}\!\!\}$$

and mgu m n σ means σ is a *most general unifier* for m and n:

$$\begin{aligned}&\texttt{Definition mgu } (m : \Lambda)\ (n : \Lambda)\ (\sigma : \Sigma) : \mathbb{P} := \\ &\quad m\{\!\!\{\sigma\}\!\!\} = n\{\!\!\{\sigma\}\!\!\} \wedge \forall \nu : \Sigma,\ m\{\!\!\{\nu\}\!\!\} = n\{\!\!\{\nu\}\!\!\} \to \exists \tau : \Sigma,\ \nu \approx \sigma \circ \tau.\end{aligned}$$

Note that two mgus need not be (extensionally) equivalent (i.e. w.r.t. $\approx$) because the definition of mgu does not characterize their behavior for the variables not occurring inside of m or n, hence one can freely permute those outside variables while preserving the mgu property.

The mechanized proof below follows the script described by Krauss [14] which first establishes partial correctness results to conclude with totality/termination. This feature is recurrent with nested algorithms: proving termination involves some knowledge of what the function computes, a vicious cycle for Coq that can be broken with the Braga method.

Hence we first establish partial correctness: on its domain of termination $\mathbb{D}_{\texttt{unif}}$, unif outputs either Some σ where σ is an mgu of m and n, or else None in which case m and n cannot be unified.

$$\begin{aligned}&\texttt{Theorem unif\_partial\_correct } m\ n\ (D : \mathbb{D}_{\texttt{unif}}\, m\, n) : \\ &\texttt{match unif } m\ n\ D \texttt{ with Some } \sigma \Rightarrow \texttt{mgu } m\ n\ \sigma \mid \texttt{None} \Rightarrow m \between n \texttt{ end}.\end{aligned}$$

Proof. By direct induction on D using $\mathbb{D}_{\texttt{unif}}$_rect and the other components of the proof-irrelevant inductive–recursive scheme of unif. □

This illustrates that we can study the output value of unif in Coq, without and independently of having to establish termination/ totality. Moreover, we can also get refined partial correctness results such as, the output of `unif` $m\ n$, if it is `Some` σ, then applying the substitution σ does not produce any new variable:

```
Lemma mgu_trm_vars_incl $m\ n\ (D : \mathbb{D}_{\texttt{unif}}\, m\, n)$ :
          match unif $m\ n\ D$ with
            | Some $\sigma$  $\Rightarrow \forall t,\ \langle\!\langle t\{\![\sigma]\!\}\rangle\!\rangle \subseteq \langle\!\langle m\rangle\!\rangle + \!\! + \langle\!\langle n\rangle\!\rangle + \!\! + \langle\!\langle t\rangle\!\rangle$
            | None    $\Rightarrow \top$
          end.
```

Another important partial correctness result states that the output of `unif` $m\ n$, if it is `Some` σ (extensionally) different from the identity substitution $[\,]$, then σ erases at least one variable from those of m or n:

```
Lemma mgu_trm_vars_dec $m\ n\ (D : \mathbb{D}_{\texttt{unif}}\, m\, n)$ :
   match unif $m\ n\ D$ with
     | Some $\sigma$  $\Rightarrow \sigma \approx [\,] \vee \exists x : \mathcal{V},\ x \in \langle\!\langle m\rangle\!\rangle + \!\! + \langle\!\langle n\rangle\!\rangle \wedge \forall t : \Lambda,\ x \notin \langle\!\langle t\{\![\sigma]\!\}\rangle\!\rangle$
     | None    $\Rightarrow \top$
   end.
```

These two partial correctness lemmas are both established by induction on $D : \mathbb{D}_{\texttt{unif}}\, m\, n$ using $\mathbb{D}_{\texttt{unif}}$`_rect`.

8.6. *Termination*

These three partial correctness results give us enough feedback properties to allow the proof of totality for $\mathbb{D}_{\texttt{unif}}$, i.e. termination of `unif` $m\, n$ for any input values m and n:

```
Theorem unif_total :   $\forall m\, n,\ \mathbb{D}_{\texttt{unif}}\, m\, n$.
```

Proof. By a lexicographic (or nested) induction on:

(a) first, the list $\langle\!\langle m\rangle\!\rangle + \!\! + \langle\!\langle n\rangle\!\rangle$ ordered by strict list inclusion;
(b) second, the size $[\![m]\!]$ ordered by the strictly less relation $<$.

Starting from the call `unif` $(m \diamond n)\ (m' \diamond n')$, the termination of the first subcall `unif` $m\ m'$ is ensured by (b). Then, thanks to `mgu_trm_vars_dec`, in the case `unif` $m\ m' =$ `Some` σ where there is a second (nested) subcall `unif` $n\{\![\sigma]\!\}\ n'\{\![\sigma]\!\}$:

- either $\sigma \approx [\,]$ in which case the subcall is identical to `unif` $n\ n'$, terminating because of (b) again;

```
let rec unify u v =
  match u with
  | Var x -> if occ_check_b x v then None else Some [(x,v)]
  | Cst c -> (match v with
     | Var y -> Some [(y,u)]
     | Cst d -> if eqC c d then Some [] else None
     | App (_,_) -> None)
  | App (m, n) -> (match v with
     | Var y -> if occ_check_b y u
                then None
                else Some [(y,u)]
     | Cst _ -> None
     | App (m',n') -> (match unify m m' with
        | Some r -> (match unify (subst r n) (subst r n') with
           | Some s -> Some (subst_comp r s)
           | None -> None)
        | None -> None))
```

Fig. 22. Extracted OCaml code for the `unify` algorithm.

- or there is a variable x, outside of both $n\{\!\!\{\sigma\}\!\!\}$ and $n'\{\!\!\{\sigma\}\!\!\}$, ensuring that condition (a) holds and we get termination again.

In any case, termination is ensured by the induction hypotheses. □

We trivially derive the fully specified terminating unification algorithm

$$\texttt{Definition unify}\ m\ n : \{r \mid \texttt{match}\ r\ \texttt{with}\ \texttt{Some}\,\sigma \Rightarrow \texttt{mgu}\ m\ n\ \sigma \mid \texttt{None} \Rightarrow m \between n\ \texttt{end}\}.$$

which extracts gracefully in Fig. 22 as the expected OCaml code that reflects faithfully on the equations of Fig. 20. Note that the identity deciders for variables `eqV` : $\alpha\to\alpha\to$`bool` and constants `eqC` : $\beta\to\beta\to$ `bool` are not extracted in the OCaml code because they are global `Parameters` for the whole project of this section and thus should be properly instantiated before using `unify`. Alternatively, they could be declared as `Variables`, in which case they would appear as extra arguments for `unify`, `occ_check_b`, `subst` and `subst_comp`.

9. Related Works

In this chapter, we have described the Braga method. Mostly through examples, we explain how to systematically encode partial recursive schemes into Coq while, at the same time, ensuring a tight control over the computational contents of terms. The method is friendly to extraction while allowing to build the tools to define and reason about partial recursive functions in Coq.

Our own contribution is based on a very rich literature that originates in the mid-1990s and concerned with the *mechanized study* of recursive algorithms. Of course, the formal study of the properties of recursive algorithms is much older with, e.g. the work of Manna and Pnueli [15] in the early 1970s. Also, the mechanization of reasoning and the verification of proofs of mathematical theorems by computers can be traced back in the 1970s with the work of de Bruijn on Automath [26]. But here, we only collect and briefly describe some of the references that were influential in the design of the Braga method.

Foremost, maybe it is the seminal paper of Giesl [22] that gave us the good foundation for approaching the difficult cases of nested algorithms where the properties of the output have an impact on the study of the domain. Hence separating the study of termination from the study of correctness is a critical insight. Building on this idea, Krauss [14] gave an approach to be able to define and manipulate functions implementing algorithms, independently of their termination or correctness properties. His approach however relies on Hilbert's description operator in HOL, a highly non-constructive feature that typical users of Coq extraction mechanism want to avoid because there is no way to extract this operator. Moreover, as it is incompatible with many propositional axioms, assuming it makes it easy to silently corrupt the internal logic of Coq. Nonetheless, the examples we develop in this chapter mostly come from Giesl [22] and Krauss [14].

These two previous authors do not consider constructive frameworks like type theory or Coq, and in this context, the landmark reference is Bove and Capretta [10] who use inductive–recursive schemes to model partiality. However, we do not really follow their approach, but we can retrieve their tools as convenient ways to manipulate termination domains and partial functions in one of the variants of the Braga method. In contrast, our custom domain (or accessibility) predicates are critically implemented as non-informative propositions, allowing their erasure at extraction. Moreover, we also remark that induction on the computational graph can often be used as a cheaper alternative to inductive–recursive schemes, provided one accepts working with relations in place of equations. Actually, by reasoning on the computational graph, one could prove properties of the partial function and its domain without even writing the function.[z] In that context, the Coq implementation of the function would only matter for extraction purposes.

The idea of defining the domain as the projection of the computational graph on its inputs can at least be traced back to Dubois and Donzeau-Gouge [9]. This idea is revisited by Bove [19] but there, the domain predicate is informative. Hence the way termination is proved would leak into the extracted program, thus failing to separate code definition from correctness and termination study. By projecting the computational graph on its inputs to get the domain predicate, these two references pick up an approach that does not naturally capture the structure of recursive calls over the domain.

Bove *et al.* [27] propose a quite recent overview of recursion in the context of interactive theorem provers, illustrated with typical examples. They focus mainly on higher-order logics (Hols), either the constructive type theories of Agda and Coq, or the more classical HOL. Putting aside co-inductive examples, we have successfully tested the Braga method on most of the examples they list. It is our intention to complement our distributed code with these examples later on.

Concerning Coq, Sozeau and Mangin [28] propose the "equations" package that allows the definition of recursive functions with a much more flexible syntax. `Equations` has many advantages over the `Fixpoint` primitive or the more elaborate `Program Fixpoint` declaration. However, it is difficult to tightly control its behavior w.r.t. extraction when dealing with somewhat complicated schemes [12]. Also for termination, it is based on well-founded recursion and thus, not always suitable for partial algorithms or else algorithms that are better manipulated as partial, typically nested ones. That said, `Equations` can perfectly be used when deploying the Braga method and it is our hope that the method will one day find its way for

[z]This idea can be pushed further to functions written with non-existent features in Coq and OCaml, such as a pattern-matching on virtual constructors, as illustrated with our reference "fold-left from the tail" function.

full integration in the equations framework, thus allowing a seamless treatment of partial recursive functions.

At TYPES 2018, Andreas Abel pointed us to the contemporary work of Wieczorek and Biernacki [29] on normalization by evaluation implemented in Coq. In there, independently of our work, they use some tools belonging to the herein called Braga method like custom inductive domains and induction on the computational graph. In their Section 3.2 on page 269, they compare their approach to the existing literature at the time, mostly the work of Bove and Capretta [10, 19]. As they also aim at extraction, they make similar observations to our own w.r.t. induction–recursion and informative domain predicates.

They only reason on the computational graph, actual definitions of partial functions are there only for program extraction. Additionally, they do not note that inductive–recursive schemes can be inferred in Coq using the restriction to proof-irrelevant predicates illustrated here on `dfs`, `nm` and `unif`, so that the two approaches — induction on the computational graph and equational reasoning using inductive–recursive schemes — turn out to be equivalent.

Moreover Section 3.3 of Wieczorek and Biernacki [29] don't explain how their projection/inversion functions actually provide structurally smaller arguments in recursive calls though this is a key aspect of the method. We consider that this structural decrease can be shown very clearly in different situations, as illustrated from our introduction to custom inductive domain predicates in 3.1, then more typically in Fig. 14, or in the encoding of Paulson's `nm`. Because they aim at solving a complex problem with an algorithm, their recursive scheme reflects this complexity and (to us) is not ideal as an illustration of their method. They seem to consider it somehow ad-hoc while on the contrary, we have the conviction that the Braga method is very versatile.

More recently, Jan Bessai kindly wrote us to explain how the Braga method, as outlined in the two pages TYPES 2018 abstract [2] and the accompanying code, helped him to implement his correct by construction algorithm for fast BCD subtyping [30]. On this example, he also extended the method to be able to capture some properties related to a measure of complexity of his algorithm. This gives us even more conviction that simple/short examples help at the understanding of the Braga method. That is why we insisted on these examples in this chapter, and in the future, we intend to populate our available Coq code with additional well-documented illustrations of the method.

Acknowledgments

The research of the first author was partly supported by the TICAMORE joint ANR-FWF project (ANR grant 16-CE91-0002).

The research of the second author was partly supported by the French national research organization ANR (ANR grant 15-CE25-0008).

References

[1] X. Leroy (2006). Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, ACM, pp. 42–54, ISBN 1-59593-027-2.

[2] D. Larchey-Wendling and J.-F. Monin (2018). Simulating induction–recursion for partial algorithms. In *24th Int. Conf. Types for Proofs and Programs, TYPES 2018*, Braga, Portugal (https://hal.archives-ouvertes.fr/hal-02333374).

[3] Y. Bertot and P. Castéran (2004). *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science. An EATCS Series. Springer, ISBN 978-3-642-05880-6.

[4] G. Gilbert, J. Cockx, M. Sozeau and N. Tabareau (2019). Definitional proof-irrelevance without K. In *Proc. ACM Programming Languages*, pp. 1–28 (https://doi.org/10.1145/3290316).

[5] C. Paulin-Mohring (1989). *Extraction de programmes dans le Calcul des Constructions*, Thèse d'université, Paris 7.

[6] P. Letouzey (2008). Extraction in Coq: An overview. In A. Beckmann, C. Dimitracopoulos and B. Löwe (eds.), *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008*, Athens, Greece, June 15–20, 2008, *Proc.* Vol. 5028, Lecture Notes in Computer Science, Springer, pp. 359–369.

[7] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau and T. Winterhalter (2019). Coq Coq correct! verification of type checking and erasure for Coq, in Coq, *Proc. ACM Program. Lang.* **4** (POPL) (https://doi.org/10.1145/3371076).

[8] T. Altenkirch (1994). Proving strong normalization of CC by modifying realizability semantics. In H. Barendregt and T. Nipkow (eds.), *Types for Proofs and Programs*, LNCS 806, pp. 3–18.

[9] C. Dubois and V. Viguié Donzeau-Gouge (1998). A step towards the mechanization of partial functions: Domains as inductive predicates (1998). In: Presented at *CADE-15, Workshop on the Mechanization of Partial Functions.*

[10] A. Bove and V. Capretta (2005). Modelling general recursion in type theory. *Math. Struct. Comput. Sci.* **15**(4):671–708 (https://doi.org/10.1017/S0960129505004822).

[11] J.-F. Monin and X. Shi (2013). Handcrafted inversions made operational on operational semantics. In S. Blazy, C. Paulin-Mohring and D. Pichardie (eds.), *Interactive Theorem Proving*, Springer Berlin Heidelberg, pp. 338–353, ISBN 978-3-642-39634-2 (https://doi.org/10.1007/978-3-642-39634-2_25).

[12] D. Larchey-Wendling and R. Matthes (2019). Certification of breadth-first algorithms by extraction. In G. Hutton (ed.), *Mathematics of Program Construction*, Cham: Springer International Publishing, pp. 45–75, ISBN 978-3-030-33636-3 (https://doi.org/10.1007/978-3-030-33636-3_3).

[13] D. Larchey-Wendling (2018). Proof pearl: Constructive extraction of cycle finding algorithms. In J. Avigad and A. Mahboubi (eds.), *Interactive Theorem Proving*, Cham: Springer International Publishing, pp. 370–387, ISBN 978-3-319-94821-8 (https://doi.org/10.1007/978-3-319-94821-8_22).

[14] A. Krauss (2010). Partial and nested recursive function definitions in higher-order logic. *J. Automa. Reason.* **44**:303–336 (https://doi.org/10.1007/s10817-009-9157-2).

[15] Z. Manna and A. Pnueli (1970). Formalization of properties of functional programs, *J. ACM.* **17**(3):555–569, ISSN 0004-5411, doi:10.1145/321592.321606 (https://doi.org/10.1145/321592.321606).

[16] J. Lagarias (2010). *The Ultimate Challenge: The* $3x{+}1$ *Problem.* American Mathematical Society, ISBN 9780821849408 (http://bookstore.ams.org/mbk-78).

[17] P. Dybjer (2000). A General formulation of simultaneous inductive-recursive definitions in type theory. *The J. Symbolic Logic.* **65**(2):525–549, ISSN 00224812 (http://www.jstor.org/stable/2586554).

[18] U. Norell, N. A. Danielsson, A. Abel and J. Cockx. The Agda Wiki (https://wiki.portal.chalmers.se/agda/Main/History).

[19] A. Bove (2009). Another look at function domains. *Electron. Notes Theoret. Comput. Sci.* **249**:61–74, ISSN 1571-0661 (https://doi.org/10.1016/j.entcs.2009.07.084). (*Proc. 25th Conf. Mathematical Foundations of Programming Semantics (MFPS 2009)*).

[20] J.-F. Monin (2010). Proof trick: Small inversions. In Y. Bertot (ed.), *Second Coq Workshop*. Edinburgh Royaume-Uni (http://hal.inria.fr/inria-00489412/en/).

[21] Wikipedia. Depth-first search (https://en.wikipedia.org/wiki/Depth-first_search).

[22] J. Giesl (1997). Termination of nested and mutually recursive algorithms. *J. Autom. Reason.* **19**:1–29 (https://doi.org/10.1023/A:1005797629953).

[23] Z. Manna and R. Waldinger (1981). Deductive synthesis of the unification algorithm. *Sci. Comput. Program.* **1**(1):5–48, ISSN 0167-6423 (https://doi.org/10.1016/0167-6423(81)90004-6).

[24] K. Slind (2000). Another look at nested recursion. In M. Aagaard and J. Harrison (eds.), *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, pp. 498–518, ISBN 978-3-540-44659-0 (https://doi.org/10.1007/3-540-44659-1_31).

[25] J.-F. Monin (1996). Exceptions considered harmless. *Sci. Comput. Program.* **26**:179–196.

[26] F. D. Kamareddine (2011). *Thirty Five Years of Automating Mathematics*, 1st edn. Springer Publishing Company, Incorporated, ISBN 9048164400.

[27] A. Bove, A. Krauss and M. Sozeau (2016). Partiality and recursion in interactive theorem provers — An overview. *Math. Struct. Comput. Sci.* **26**(1):38–88 (https://doi.org/10.1017/S0960129514000115).

[28] M. Sozeau and C. Mangin (2019). Equations reloaded: High-level dependently-typed functional programming and Proving in Coq. *Proc. ACM Program. Lang.* **3**(ICFP) (https://doi.org/10.1145/3341690).

[29] P. Wieczorek and D. Biernacki (2018). A Coq formalization of normalization by evaluation for Martin-Löf type theory. In *Proc. 7th ACM SIGPLAN Int. Conf. Certified Programs and Proofs*, CPP 2018, Association for Computing Machinery, New York, NY, USA, pp. 266–279, ISBN 9781450355865 (https://doi.org/10.1145/3167091).

[30] J. Bessai, J. Rehof and B. Düdder (2019). *Fast verified BCD subtyping*. In T. Margaria, S. Graf and K. G. Larsen (eds.), *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, Springer International Publishing, Cham, pp. 356–371, ISBN 978-3-030-22348-9 (https://doi.org/10.1007/978-3-030-22348-9_21).