HAL
archives-ouvertes.fr

Partiality and Recursion in Interactive Theorem Provers - An Overview

Ana Bove, Alexander Krauss, Matthieu Sozeau

▶ To cite this version:

Ana Bove, Alexander Krauss, Matthieu Sozeau. Partiality and Recursion in Interactive Theorem Provers - An Overview. Mathematical Structures in Computer Science, Cambridge University Press (CUP), 2012. <hal-00691459>

HAL Id: hal-00691459
https://hal.inria.fr/hal-00691459

Submitted on 26 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Under consideration for publication in Math. Struct. in Comp. Science

Partiality and Recursion in Interactive Theorem Provers — An Overview

ANA BOVE[1], ALEXANDER KRAUSS[2], and MATTHIEU SOZEAU[3]

[1] *Chalmers University of Technology, Gothenburg, Sweden, e-mail:* bove@chalmers.se
[2] *Technische Universität München, Germany, e-mail:* krauss@in.tum.de
[3] *INRIA Paris, France, e-mail:* matthieu.sozeau@inria.fr

Received January 2011

The use of interactive theorem provers to establish the correctness of critical parts of a software development or for formalising mathematics is becoming more common and feasible in practice. However, most mature theorem provers lack a direct treatment of partial and general recursive functions; overcoming this weakness has been the objective of intensive research during the last decades. In this article, we review many techniques that have been proposed in the literature to simplify the formalisation of partial and general recursive functions in interactive theorem provers. Moreover, we classify the techniques according to their theoretical basis and their practical use. This uniform presentation of the different techniques facilitates the comparison and highlights their commonalities and differences, as well as their relative advantages and limitations. We focus on theorem provers based on constructive type theory (in particular, Agda and Coq) and higher-order logic (in particular Isabelle/HOL). Other systems and logics are covered to a certain extent, but not exhaustively. In addition to the description of the techniques, we also demonstrate tools which facilitate working with the problematic functions in particular theorem provers.

1. Introduction

We are moving towards an era where the correctness of (certain parts of) complex systems has become a common practise. In order to achieve this goal, a variety of expressive logics have been developed and powerful interactive theorem provers based on those logics have been implemented. However, most of these logics can represent only total functions, and in order to ensure meta-theoretic properties of the logic—above all, consistency—they considerably restrict the use of recursion.

Clearly not many people want to trade consistency for partial functions and general recursion, but this limitation remains a disturbing weakness of many proof assistants, at least for some applications. In the last 25 years, a considerable amount of research aimed at improving this situation by studying ways of supporting partial functions and more general forms of recursion in the logics and their associated proof assistants.

This paper is a survey on the state of the art in this field of research both from a conceptual and from a practical perspective. On the conceptual side, we review a variety

of approaches from the literature, discuss their properties, commonalities and differences. On the practical side, we present a number of tools that were developed on top of some proof assistants to make working with partial and recursive functions more convenient. Both the ideas and the tools presented here witness the importance of this field as well as its progress during the last decades.

It must be clear to the reader that we do not aim at proposing new techniques or refinements to existing ones here, but at collecting and classifying the ideas spread over the literature. Our contribution in this paper is the presentation of a big picture in the field of partiality and general recursion, a picture which is often not visible when discussing just one approach, possibly confined to a particular logic or theorem prover. With this, we hope to help in understanding the connections between existing ideas and further advancing the state of the art.

1.1. *Scope of This Paper*

We mainly discuss work that has been done in the context of higher-order formalisms, notably the various flavours of (intensional) constructive type theory and classical higher-order logic. Despite many subtle but important differences in foundations, considering constructive and classical logics together yields interesting insights. Even techniques that seem specifically tailored to one logic sometimes have close relatives in the other, and such connections only become visible in direct comparison.

In addition, we concentrate on the proof assistants with which the authors are most familiar, namely, Agda, Coq, and the family of HOL systems (HOL4, HOL Light, and Isabelle/HOL), although we sometimes also comment on the situation in related systems.

When it comes to partiality, we are mainly interested in the partiality that arises from (non well-founded) recursion rather than the partiality which results from a function not being defined on a certain argument, as for example the head or tail of a list. There are well-known ways to deal with the latter class of partiality such as returning an uninteresting value of the codomain, returning a value in an option type, or restricting the domain of the function in order to exclude the values on which the function is not defined. This said, if a method is particularly suited for treating this kind of partiality we will comment on that.

Finally, we do not discuss functions over infinite objects, in other words, we do not consider the problem of corecursive functions over elements of a coinductive type. In such definitions, the notion of productivity replaces that of termination, and different (though related) approaches are required. The research in this area is much younger (and possibly not well-understood yet) and not many practical tools exist, so we leave a general overview of this topic for a later stage. On the other hand, observe that some of the methods we present here for treating recursive functions over finite objects do use coinduction.

1.2. *Terminology and Notation*

In the following, we introduce terminology and notational conventions that we use throughout this paper in the hope to ease its reading.

1.2.1. *Terminology* The term *system* refers to a proof assistant, including its logic and implementation. It defines the rules that specify which definitions and reasoning steps are allowed and which are invalid, and thus it ultimately determines what a user can do to guarantee a correct formalisation following the rules of the underlying logic. Examples of systems include Agda, Coq, and Isabelle/HOL.

Techniques are the basic units in which we organise the content of this paper. A technique can be a particular way of using a system, or a way of extending it. We use this term for grouping together approaches that are based on similar ideas. However, there is no formal definition that can tell us when some related approaches should be considered the same technique or separate ones. We merely tried to find a compromise that reflects our current understanding of the relations between the various published approaches.

One important classification criterion is the distinction between axiomatic and definitional techniques. Following traditional terminology from the HOL community, we call a technique *axiomatic* if it involves modifications to the rules of the system. This may happen by adding axioms, enriching the calculus with new constructs, or simply by generalising the criteria that specify which definitions are accepted by the system. Techniques that do not make such modifications but work entirely within the given rules of the system are called *definitional*. In essence, such techniques can be employed by the user of an unmodified system without violating the rules.

Some definitional techniques use constructions that are difficult or tedious to carry out manually. A *tool* is a program that simplifies the practical use of some technique and smooth its integration with the constrains of the logic behind the system, and which possibly also automates the generation of certain definitions and proofs. By definition, anything a tool does can also be done directly by the user, which is why tools do not increase the expressive power of a system—they merely make it more convenient to use. A consequence (and advantage) of this is that no new meta-theoretic issues can arise, since the system itself remains the same.

In particular, definitional techniques and the associated tools can never compromise the soundness of a system, even if their implementation contains errors (which is very likely). This is a big advantage, since the tools do not have to be trusted. Minimising the trusted code base is important for high-assurance applications such as the verification of critical systems.

1.2.2. *Notation* We use `typewriter` font style when giving the definition of a program as in a standard functional programming language, and sans serif font style when giving the formalisation of a program in a formal logic.

Where appropriate, we make liberal use of implicit notation and omit type annotations at binders (such as λ or $\forall$) provided the type is clear from the context. We also sometimes

omit the declaration of parameters in a type such as in $\mathtt{map} : (A \to B) \to [A] \to [B]$, where we consider types A and B as implicitly declared. In the corresponding application $\mathtt{map}\ f\ xs$, the type arguments A and B are then also implicit, since they can easily be inferred from the other arguments. We might even leave A and B implicit in the application also when they have been explicitly declared in the type.

Note that our use of implicit arguments is independent from the syntactic conventions of any particular system, which may require more explicit notation.

1.3. *Some Preliminary Notions*

We now recall some known concepts that are needed later. The reader familiar with any of them can safely skip the corresponding paragraph.

Graph of a Function. Given a function $f : A \to B$, the graph of f is the relation $G_f \subseteq A \times B$ containing the (ordered) pairs $(x, f\ x)$.

Observe that, since a function has at most one result for each argument, G_f has the property that $\forall x\ y\ z.\, (x, y) \in G_f \wedge (x, z) \in G_f \to y = z$. Hence, to state that a relation $R \subseteq A \times B$ represents the graph of a function we need to show that for any x, there exists at most one y such that $(x, y) \in R$.

Functional of a Recursive Definition. Given a (recursive) definition $f : A \to B$, we can define a second order function (or functional) $F : (A \to B) \to A \to B$ such that F is itself non-recursive and $f = F\ f$. For example, a functional for the addition of natural numbers is the following:

$$F = \lambda h\ x\ y.\, \mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ (h\ (x - 1)\ y) + 1$$

Monads. In functional programming, a monad can be seen as a datatype which is (in general but not necessarily) used to express computations with side effects. Monads were introduced by Moggi (1991) and they are now common practice in certain programming languages such as Haskell.

A monad consists of a type constructor $\mathtt{M}$ with two operations: $\mathtt{return} : A \to \mathtt{M}\ A$, which takes a value from a type an returns it inside the monadic constructor, and $\mathtt{bind} : \mathtt{M}\ A \to (A \to \mathtt{M}\ B) \to \mathtt{M}\ B$, sometimes denoted as $\gg\!=$, which extracts a value from a monadic constructor and passes it to the function that will perform the next computation.

It is common to use the so-called do-notation when working with monads, to mimic the appearance of imperative programs. Using such notation, the following programs are equivalent:

$$f \gg\!= \lambda x.\, g\ x \gg\!= \lambda y.\, \mathtt{return}\ y \qquad\qquad \begin{array}{l} \mathtt{do}\ x \leftarrow f; \\ \quad y \leftarrow g\ x; \\ \quad \mathtt{return}\ y \end{array}$$

Option Type. A useful type to deal with some kind of partiality is the well-known Option type, sometimes also called Maybe type.

$$\begin{aligned}&\mathsf{Inductive\ Option} : \forall A : \mathsf{Set}.\ \mathsf{Set}\\ &\quad \mathsf{some} : A \to \mathsf{Option}\ A\\ &\quad \mathsf{none} : \mathsf{Option}\ A\end{aligned}$$

Observe that this type naturally forms a monad by defining $\mathsf{return}\ x = \mathsf{some}\ x$, $\mathsf{bind}\ \mathsf{none}\ f = \mathsf{none}$, and $\mathsf{bind}\ (\mathsf{some}\ a)\ f = f\ a$.

Well-Foundedness. Many of the techniques and tools we present in this paper are based on the notion of *well-foundedness*. Although we expect most of our readers to be familiar with this notion in some way, we informally recall it here; a formal definition is given in §5.1.

Given a binary relation $\prec$ over a set A, the *well-founded part of* $\prec$ is defined as the set of a's in A such that there exists no infinite decreasing sequence starting from a. The relation $\prec$ is said to be *well-founded* if the well-founded part of $\prec$ is exactly the set A (Aczel 1977a).

To understand why this notion is important when we formalise recursive definitions in a logic of total functions, consider a function f which is recursive on elements of type A and let $\prec$ be a well-founded relation over A. If we can show that every recursive call of f is made on a smaller argument with respect to $\prec$, and given that any decreasing chain $a_1 \succ a_2 \succ \ldots$ must be of finite length, then we can be sure that the recursion will eventually terminate when we evaluate the function on a certain argument; in other words, f is total on A.

Chains, CPOs and Least Fixpoints. We recall here some key concepts from domain theory. Consider a set A and a partial order $\sqsubseteq \subseteq A \times A$ on A, sometimes denoted as $\sqsubseteq_A$ to make the set A explicit. A $\sqsubseteq$-*chain* is a set $C \subseteq A$ which is totally ordered, that is, $\forall x\ y \in C.\, x \sqsubseteq y \vee y \sqsubseteq x$. We say that $\sqsubseteq$ is a *complete partial order* (cpo) if every chain has a least upper bound with respect to $\sqsubseteq_A$. By definition, this least upper bound must be unique and we denote it by $\bigsqcup C$; we sometimes also use an indexed notation where $\bigsqcup_{x \in I} (f\ x)$ stands for $\bigsqcup \{f\ x \mid x \in I\}$.

A cpo $\sqsubseteq_B$ can be lifted to a cpo on the function space $A \to B$ by defining $f \sqsubseteq_{A \to B} g$ as the pointwise order $\forall x.\, f\ x \sqsubseteq_B g\ x$.

A function $f : A \to B$ is called *monotone* if $x \sqsubseteq_A y$ implies $f(x) \sqsubseteq_B f(y)$, and it is called *continuous* if it is monotone and preserves least upper bounds, that is, if $f(\bigsqcup C) = \bigsqcup_{x \in C} f(x)$.

The key result concerning recursive definitions is the following fixpoint theorem.

Theorem. Let $\sqsubseteq_A$ be a cpo and $f : A \to A$ a continuous function. Then, there exists a least fixpoint fix_f, that is,

a) $f\ (\mathrm{fix}_f) = \mathrm{fix}_f$, and
b) for all x, $f\ x = x$ implies $\mathrm{fix}_f \sqsubseteq_A x$.

1.4. *Outline of the Paper*

The rest of this paper is organised as follows. In §2 we present a small collection of examples that we use in the rest of the paper. In §3 we describe the necessary logical foundations, both constructive and classical. The next two sections comprise the main body of this paper and consist of example-guided descriptions of different techniques for dealing with partiality and general recursion, which we discuss and compare as we go along. We differentiate between axiomatic (§4) and definitional techniques (§5). Later, in §6, we demonstrate tools implemented on top of Coq and Isabelle/HOL, which simplify the practical use of some of the ideas described before. Finally, §7 presents some concluding remarks.

2. Running Examples

In the rest of the paper, we use a number of running examples. They are chosen to illustrate specific forms of function definitions that present different difficulties. This choice of simple and somewhat artificial examples is deliberate: they merely serve as vehicles for explaining the fundamental ideas we present in this paper, which equally apply to more practically relevant functions. The examples are introduced here as one would define them in a standard functional programming language.

2.1. *A Total Function: Quicksort*

The first example is the quicksort algorithm as it is found in functional programming textbooks.

$$\begin{aligned}&\texttt{qs}\ [] = []\\&\texttt{qs}\ (x :: xs) = \texttt{qs}\ (\texttt{filter}\ (\lambda y.\, y \leqslant x)\ xs) +\!\!+ x :: \texttt{qs}\ (\texttt{filter}\ (\lambda y.\, y > x)\ xs)\end{aligned}$$

To prove termination of `qs` we must reason about `filter` and show that it does not increase the size of the list. This is beyond the scope of syntactic checks that only look at the present function definition. Moreover, it requires induction, so just unfolding the definition of `filter` inside of `qs` does not help.

2.2. *A Function that Alternates the Decreasing Argument: Merge*

Our second example is a total function that merges two ordered lists.

$$\begin{aligned}&\texttt{merge}\ xs\ [] = xs\\&\texttt{merge}\ []\ ys = ys\\&\texttt{merge}\ (x :: xs)\ (y :: ys) =\\&\qquad \texttt{if}\ x < y\ \texttt{then}\ x :: \texttt{merge}\ xs\ (y :: ys)\ \texttt{else}\ y :: \texttt{merge}\ (x :: xs)\ ys\end{aligned}$$

Observe that, when both lists are not empty, the argument which gets structurally smaller on each recursive call varies depending on how the heads of the lists compare: if $x < y$ then the first list gets smaller, otherwise the second does.

2.3. *A Nested Function: McCarthy's 91-Function*

Our next example is a well-known (artificial) challenge problem: McCarthy's 91-function.

$$\texttt{f91}\ n = \texttt{if}\ n > 100\ \texttt{then}\ n - 10\ \texttt{else}\ \texttt{f91}\ (\texttt{f91}\ (n + 11))$$

Using the measure $\lambda n.101 - n$ one can easily prove that each recursive call is performed on a smaller argument (with respect to this measure) and hence the function is bound to terminate in the domain of the Natural numbers.

Even if of no practical use at all, this function is interesting because of its confusing recursive structure, which escapes immediate comprehension. In particular, it uses nested recursion, i.e., the argument of a recursive call contains another recursive call to the function being defined. Since the termination behaviour of a nested recursive function may depend on its functional behaviour, reasoning about termination and functional behaviour are often interdependent. This circularity makes certain ideas discussed in §5 more difficult or simply not possible.

2.4. *A Function with Higher-Order Recursion: Mirror*

Higher-order recursion means that some of the recursive calls are not fully applied on the right-hand side of the definition of a function. As an example, let us consider the following function, which operates on n-ary trees that are built using the single constructor $\texttt{tree} : A \to [\,\texttt{Tree}\ A\,] \to \texttt{Tree}\ A$.

$$\texttt{mirror}\ (\texttt{tree}\ a\ ts) = \texttt{tree}\ a\ (\texttt{rev}\ (\texttt{map}\ \texttt{mirror}\ ts))$$

The difficulty here is that the termination of this function critically depends on how the higher-order map function uses its argument. Again, this interdependency makes certain approaches unapplicable.

2.5. *A Partial Function: Iter*$_0$

All functions presented above are total. As a genuinely partial function we consider the following iteration combinator, which is a slightly simplified variant of the well-known unfold function. It iterates a function $f : \texttt{Nat} \to \texttt{Nat}$ on an argument x and returns a list of all the intermediate values until the iteration reaches zero.

$$\texttt{iter}_0\ f\ x = \texttt{if}\ x = 0\ \texttt{then}\ []\ \texttt{else}\ x :: \texttt{iter}_0\ f\ (f\ x)$$

This computation clearly loops when the value zero is never reached, so we cannot expect to give a termination proof in general.

3. Logical Foundations

In this section we introduce the logical foundations needed for understanding the ideas we present in this paper. We first describe constructive type theory, which is the basis of Agda and Coq, and then we introduce higher-order logic, which is the common basis

of the HOL systems, and which we use to explain classical techniques in general. The reader familiar with any of these logics can safely skip the corresponding section.

Our presentation of the logics abstracts over some (for the purpose of this paper) inessential details of how certain concepts are represented in a particular system, e.g., parametrised (co)inductive definitions. The interested reader can consult the documentation and the standard library of the respective systems for further details and precise definitions of some of the concepts used in this paper.

3.1. *Constructive Type Theory*

We give here a brief introduction to (intensional) constructive type theory; for further reading please refer to Martin-Löf (1984) and Coquand and Huet (1988).

In this paper, we use (intensional) constructive type theory as a programming language with dependent types. In addition, by following the Curry-Howard isomorphism (Howard 1980) a type can be seen as a proposition, and the elements of that type as the proofs of that proposition. Therefore, proof checking simply amounts to type checking. In particular, if a specification states the existence of an object with certain properties, a proof of the specification includes a program that computes such an object.

In type theory we have terms and types. Terms are denotations of mathematical or computational objects, and types are collections of terms. A type is explained by saying what its elements are and what it means for two of its elements to be equal.

We consider two basic types, Set and Prop, comprising sets and propositions respectively[†]. Both sets and propositions are inductively defined; we explain later how such definitions look like.

We now introduce some basic type formers.

Type of Elements. Given an inductively defined set (or proposition) A, the elements of A form a type denoted $\mathsf{El}(A)$. Most implementations of type theory however—and in particular the two implementations we are concerned with in this paper, namely Agda and Coq—perform this step automatically and in practice, the user does not need to deal with this type former. So in the sequel, we simply write A instead of $\mathsf{El}(A)$ and if a is an element of A we say that a has type A.

Dependent Product. (Sometimes also known as Π-*type* or as the *Cartesian product of a family of types.*) A dependent product constructs the type of (dependent) functions. Let α be a type and β be a family of types over α, that is, for every element a in α, $\beta\ a$ is a type. We write $\forall x : \alpha.\, \beta\ x$ for the type of dependent functions from α to β, and repeated dependent products as $\forall (x_1 : \alpha_1) \cdots (x_n : \alpha_n).\, \beta\ x_1\ \cdots\ x_n$. In the special case where β does not depend on x we simply write $\alpha \to \beta$ for $\forall x : \alpha.\, \beta$.

Abstractions are the canonical elements of function types and they are denoted $\lambda x\colon \alpha.\, b$ or simply $\lambda x.\, b$, if the type of x can be deduced from the context. Repeated abstractions are written as $\lambda (x_1 : \alpha_1) \cdots (x_n : \alpha_n).\, \beta$. If $f : (\forall x : \alpha.\, \beta\ x)$ and $a : \alpha$ then $f\ a : \beta\ a$.

[†] We gloss over the precise definition of universes which are irrelevant here.

As an example, consider the identity function $\lambda(A : \mathsf{Set})(x : A).\, x$ which has type $\forall A : \mathsf{Set}.\, A \to A$.

The suggestive notation using $\forall$ and $\to$ is no coincidence: Via the Curry-Howard isomorphism, the type $\forall x : \alpha.\, \beta\ x$ models universal quantification whose proofs are functions that return a proof of $\beta\ x$ for any $x : \alpha$. Similarly, the non-dependent function type $\alpha \to \beta$ models implication whose proofs are functions that map a proof of α to a proof of β.

Dependent Sum. (Sometimes also known as Σ-*type*.) If α is a type and β is a family of types over α, then $\Sigma x : \alpha.\, \beta\ x$ denotes the disjoint union of a family of types whose canonical elements are pairs (a, b), for $a : \alpha$ and $b : \beta\ a$. If $p : \Sigma x : \alpha.\, \beta\ x$, then π_1 and π_2 select the different components of p so that $\pi_1\ p : \alpha$ and $\pi_2\ p : \beta\ (\pi_1\ p)$.

Note that the special case when β does not depend on α corresponds to the standard Cartesian product of two sets and it will be denoted by $\alpha \times \beta$.

In the particular case where $\alpha : \mathsf{Set}$ and $\beta : \alpha \to \mathsf{Prop}$, the Σ-type is called a *subset type* and is written $\{x : \alpha \mid \beta\ x\}$.

Dual to products, sigma types correspond to the constructive interpretation of existential quantifiers where their elements are pairs of a witness and a proof that this witness satisfies the property. We might then denote $\Sigma x : \alpha.\, \beta\ x$ as $\exists x : \alpha.\, \beta\ x$ instead.

As we mentioned before, sets and propositions are inductively defined; we now explain such definitions in more detail.

Inductive Definitions. Given types $\alpha_1, \ldots, \alpha_n$, an inductive definition is introduced as a constant χ of type $\forall(x_1 : \alpha_1) \cdots (x_n : \alpha_n).\, \tau$, where τ is either Set or Prop. We must specify the constructors that generate the elements of $\chi\ a_1\ \cdots\ a_n$ by giving their types, for $a_1 : \alpha_1, \ldots, a_n : \alpha_n\ a_1\ \cdots\ a_{n-1}$. To guarantee that terms are well-founded, we need to impose a positivity condition on the type of the constructors. For a formal description of inductive definitions we refer to Coquand and Paulin (1990).

As an example, we give the inductive definition of the set of lists over a type A.

$$\begin{array}{l}\mathsf{Inductive}\ [\_] : \forall A : \mathsf{Set}.\, \mathsf{Set} \\ \quad [] : [A] \\ \quad \_ :: \_ : A \to [A] \to [A]\end{array}$$

Here we use a notation for lists similar to that in §2. In addition, "_" stands for a place holder which is also used to indicate the definition of mixfix operators.

Associated to each inductive definition there is an elimination rule which can be used for defining functions and proving properties over elements of the inductively defined set or proposition. The elimination rule for lists is the following:

$$\begin{array}{l}\mathsf{listrec} : \forall(xs : [A])(C : [A] \to \tau). \\ \qquad C\ [] \to (\forall(a : A)(as : [A]).\, C\ as \to C\ (a :: as)) \to C\ xs \\ \mathsf{listrec}\ []\ C\ c\ e = c \\ \mathsf{listrec}\ (a :: as)\ C\ c\ e = e\ a\ as\ (\mathsf{listrec}\ as\ C\ c\ e)\end{array}$$

where τ is either Set—if we want to use the rule for defining recursive functions over lists—or Prop—if we want to use the rule for proving properties of lists by induction.

Using $\mathsf{listrec}$, we could easily define the append function.

$$xs +\!\!+\ ys = \mathsf{listrec}\ xs\ ys\ (\lambda x\ y\ z.\, x :: z)$$

Most implementations of constructive type theory however, allow functions to be defined by pattern matching and recursion on (some of) its arguments, which is more convenient and general (see §4.1).

We omit further details on the general format of the elimination rule. Readers are encouraged to refer to Nordström et al. (1990) and Coquand and Paulin (1990).

Of particular importance is the definition of the equality proposition stating that any element in a set is equal to itself.

$$\begin{aligned}&\mathsf{Inductive}\ \_ = \_ : \forall A : \mathsf{Set}.\, A \to A \to \mathsf{Prop}\\ &\quad \mathsf{refl} : \forall a : A.\, a = a\end{aligned}$$

Given this definition (and its associated elimination rule), we can easily derive all the desired properties of equality such as the fact that it is and equivalence and congruent relation.

Coinductive Definitions. Constructive type theories often support the definition of coinductive datatypes that represent infinite structures. The type of potentially infinite lists apparently mimics the inductive definition above:

$$\begin{aligned}&\mathsf{Coinductive}\ [\![\_]\!] : \forall A : \mathsf{Set}.\, \mathsf{Set}\\ &\quad [] : [\![A]\!]\\ &\quad \_ :: \_ : A \to [\![A]\!] \to [\![A]\!]\end{aligned}$$

but it also allows the construction of infinite values such as the constant list of ones:

$$\begin{aligned}&\mathsf{ones} : [\![\mathsf{Nat}]\!]\\ &\mathsf{ones} = 1 :: \mathsf{ones}\end{aligned}$$

In order to preserve the consistency of the theory, we must impose what is known as the *productivity condition* on the definition of functions returning a coinductive value. It basically says that scrutinising a coinductive value to get its head constructor can always be performed in finite time. Just like termination, productivity is an undecidable property, so the systems rely on a sufficient syntactic criterion for determining it. The criterion requires that all corecursive calls appear as direct arguments of a coinductive constructor; such corecursive calls are said to be *guarded* (Giménez 1995). For example, ones is guarded by the $\_ :: \_$ constructor above.

Coinductive definitions will not be needed until §5.4.2, where we introduce methods to treat partial and recursive functions based on coinductive codomains. Nevertheless, in order to keep a good structure in this presentation, we introduce the notion here.

Proof Irrelevance and Program Extraction. Even though the Curry-Howard isomorphism conveys a uniform view on programs and proofs, it can make sense to keep them separate,

and use the basic type Set to represent programs and the basic type Prop to represent proofs. This distinction, which is made in Coq but not in Agda, is based on the notion of *proof irrelevance*. While two different programs satisfying a certain specification A may not be considered equivalent, for example from the efficiency point of view, two proofs of the same proposition P are interchangeable, since their only purpose is to assert the truth of P. Since proofs are irrelevant, we can remove them and keep only the computational part of a term, a process that is known as *program extraction*. This separation helps ensure that the execution of a term does not depend on the proofs it may contain, and hence the addition of proofs has no effect on the computational properties of programs. As well-foundedness proofs are part of programs, as we will see later, this is an important issue for us.

This possibility of extracting a (non-dependent, functional) program from a term by removing all proofs in it imposes some restrictions on how proofs can be used inside terms. In particular, to guarantee proof irrelevance we may not inspect (pattern match on) a proof when we are defining a term in the type Set. There is however a subtlety here since it is still possible to pattern match on certain proofs inside a term: this is permitted only when the inductive definition of the proposition we pattern match on has at most one constructor and the arguments of this constructor are all propositions—which in particular includes falsity, equality and accessibility proofs (§5.1). Intuitively, no program can behave differently depending on the shape of these proofs as they are computationally equivalent to the empty or unit type, hence it is safe to allow case analysis on them. It is moreover safe to erase these case analyses during extraction, at the condition that all the proofs are closed (i.e., there are no axioms).

3.1.1. *Agda* The Agda system (Agda 2008, Norell 2007) is primarily designed to be a programming language with dependent types, although it can also be used as an interactive proof assistant. It is implemented in Haskell (Peyton Jones 2003) and it is the latest in a sequence of similar systems developed at Chalmers University of Technology in Gothenburg, Sweden.

Agda implements a predicative extension of Martin-Löf's (constructive) type theory (Martin-Löf 1984, Nordström et al. 1990). It extends Martin-Löf's type theory with a number of features that make programming convenient, such as flexible mechanisms for defining inductive datatypes and for defining functions by pattern matching. It also extends Martin-Löf's type theory with coinductive types and simultaneous inductive-recursive definitions as defined by Dybjer (2000).

The syntax of the language resembles Haskell and has ordinary programming constructs such as datatypes and case-expressions, signatures and records, let-expressions and modules. It has also a flexible way of naming functions, datatypes and constructors, including the possibility of having mix-fix names and of using Unicode in the names. As in Haskell, function definitions are given by pattern matching on one or several arguments.

When Agda is used as a proof assistant, restrictions must be imposed on the way a function is defined in order to guarantee its totality and hence, the consistency of the logic: pattern matching must be exhaustive and recursion must be terminating. Agda's

termination checker implements a variant of size-change termination following ideas by Abel (1998) and Wahlstedt (2007); see §4.1 for a more detailed discussion on this issue.

Agda is based on the idea of direct manipulation and interactive refinement of proof-terms and not on tactics, hence proofs are just terms, not scripts.

3.1.2. *Coq* The Coq proof assistant (Coq development team 2010) is based on the Calculus of (Co)Inductive Constructions (Paulin-Mohring 1993), and it can be used both as a proof assistant for developing mathematics and as a functional programming language with dependent types. It is implemented in OCaml (OCaml 1996) and borrows some of its syntax. The system includes facilities for defining inductive datatypes and functions by pattern matching as well as an LCF-style tactic system for defining proof strategies to be used interactively or automatically. The entire system rests on a well-defined kernel that performs proof checking of terms from the strongly normalising core calculus.

Syntactic checks are used to ensure that (co)recursive definitions are terminating (respectively productive) and do not compromise the consistency of the logic. These built-in checkers are however quite restrictive; see §4.1 for a more detailed discussion on this issue.

3.2. *Higher-Order Logic*

In what follows, we introduce higher-order logic (HOL), which we will use to describe the techniques that require classical reasoning. For a complete and formal introduction to HOL, we refer to Gordon and Melham (1993).

HOL is based on simply-typed lambda calculus extended with Hindley-Milner parametric polymorphism, just like ML. Its types consist of type variables α, (non-dependent) function types $\sigma \to \tau$, and type constructors (e.g., Nat), which arise from inductive type definitions. (Other forms of type definitions can be ignored for our purposes.) Types are always inhabited and used for representing data only, not propositions. The latter are terms of the primitive type Bool, with elements true and false. Logical operations are modelled as constants, e.g., $\mathsf{implies} : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ and $\mathsf{forall} : (\alpha \to \mathsf{Bool}) \to \mathsf{Bool}$, which are governed by the standard natural deduction rules. Although proofs can be seen as another sort of terms that live on a different level, they are typically not manipulated explicitly. We write $P \to Q$ for $\mathsf{implies}\ P\ Q$ and $\forall x.\, P\ x$ for $\mathsf{forall}\ (\lambda x.\, P\ x)$. Other logical connectives can be defined in terms of $\forall$ and $\to$.

In this paper, we use the same symbol for function space and implication to gain some syntactic coherence with type theory, which unifies these two concepts. However, there is no danger of confusion: in types, $\to$ denotes functions, and in propositions it denotes implication.

Since propositions and booleans are the same thing, we can write terms of the form if *undecidable property* $\mathsf{then}\ A\ \mathsf{else}\ B$. Thus, functions do not always correspond to programs. This is even more true in the presence of Hilbert's choice operator $\mathsf{eps} : (\alpha \to \mathsf{Bool}) \to \alpha$, which is written in binder notation as $\varepsilon\, x.\, P\ x$. This operator returns any value that satisfies P, if such a value exists, and otherwise a completely arbitrary result. Here, 'any' and 'arbitrary' mean that from the rules for ε nothing about the

result can be derived. Sometimes, the weaker operator $\iota\, x.\, P$ is also used. This operator requires that P holds for exactly one element, which is then returned.

Equality is a constant $\mathsf{eq} : \alpha \to \alpha \to \mathsf{Bool}$ (written infix as $=$), with axioms specifying that it is a congruence relation and that equality on functions is extensional, i.e., $f = g \leftrightarrow (\forall x.\, f\ x = g\ x)$.

To define a new constant c, we simply add an axiom of the form $c = t$, where c does not occur in t. This restriction (together with a few others not discussed here) ensures that the new axiom cannot make the theory inconsistent. For example, the predicate $\mathsf{even} : \mathsf{Nat} \to \mathsf{Bool}$ could be defined as $\mathsf{even} = \lambda n.\, \exists m.\, n = m + m$.

Inductive definitions work similarly as in constructive type theory, except that they come in two flavours: One for predicates (on the term level) and one for datatypes (on the type level). However, in this paper we simply adopt the type-theoretic presentation for the sake of uniformity.

Recursive definitions—even structurally recursive ones—are not primitives. They are provided by definitional extensions. However, in this paper we take structural recursion for granted and refer to Berghofer and Wenzel (1999) for the definitional construction.

Two key applications of the choice operator ε should be mentioned already here. First, consider the term $\varepsilon\,(x : A).\, \mathsf{true}$. Since the predicate does not restrict which value to choose, the term denotes an arbitrary element from type A. In particular, this is a way of conjuring up an element of any type when we need it, and we abbreviate the term $\varepsilon\, x.\, \mathsf{true}$ as $\mathsf{arbitrary}$. Terms like this are often called *underspecified*.

A second notable application of the choice operator is turning a relation into a function:

$$\begin{aligned}&\mathsf{function\_of} : \forall(A, B : \mathsf{Set}).\, (A \to B \to \mathsf{Bool}) \to A \to B\\ &\mathsf{function\_of}\ R = \lambda x.\, \varepsilon\, y.\, R\ x\ y\end{aligned}$$

Although we can apply this operation to any relation, the resulting function is underspecified if the relation specifies multiple values y for some x (then the function returns any of them) or no values at all (then it returns an arbitrary result). However, if the relation is single-valued, then the following theorem relates R and $\mathsf{function\_of}\ R$.

$$\forall(x : A)(y : B).\, (\exists!z.\, R\ x\ z) \to (R\ x\ y \longleftrightarrow \mathsf{function\_of}\ R\ x = y)$$

Here, $\exists!x.\, P\ x$ abbreviates $\exists x.\, (P\ x \wedge \forall y.\, P\ y \to y = x)$.

3.2.1. *Isabelle/HOL* Isabelle/HOL (Nipkow et al. 2002) is the higher-order logic flavour of the generic theorem prover Isabelle (Paulson 1989, Wenzel et al. 2008). It implements higher-order logic roughly as described above, extended with a notion of type classes à la Haskell.

In the LCF tradition, Isabelle's architecture ensures that all proofs are checked by a small trusted inference kernel. Numerous tools automate common constructions, such as inductive types and predicates, records, quotients, or—most relevant for this paper—general recursive functions. The tool in charge, called the *function package*, will be described in §6.3. Powerful automation is available through high-level tactics such as a simplifier and a tableau prover. Apart from tactic scripts, proofs can also be written in a more declarative structured proof language.

In Isabelle, the analogue of program extraction is called *code generation* (Berghofer and Nipkow 2000, Haftmann and Nipkow 2010), and refers to the translation of equational theorems into a functional language. Unlike program extraction in constructive type theory, which always uses the definition of a function, code generation in Isabelle can work with arbitrary equations of a suitable form. It is common practise to define a function in a non-executable way (e.g., using the ε operator), and then derive recursive equations which are used for code generation. The downside of this more liberal approach is that the generated code is only partially correct with respect to the logical definition, but may have a different termination behaviour.

3.2.2. *Other HOL-Like Systems* While the difference in foundation between classical and constructive logics is often critical for the problem of function definitions, we can abstract over most of the differences among the classical systems. Most of the techniques we present here apply to all (or none) of these systems in a similar way. This includes the different implementations of higher-order logic (in particular HOL4, HOL Light, and Isabelle/HOL, which we have briefly introduced above), the ACL2 system based on first-order logic and induction, and PVS, based on a form of dependently typed higher-order logic (though we will not say much about partiality and recursion in PVS).

4. Axiomatic Techniques

This section is concerned with axiomatic techniques, which as mentioned in the introduction, are techniques that extend or modify the logical foundation of the theory.

4.1. *More Flexible Termination Checkers*

Recursive definitions using eliminators must obey a very restrictive syntactic schema. In practice, many proof assistants support a more liberal format which can still be checked syntactically.

For example, the `fix` construct in Coq allows structurally recursive definitions of a slightly more general form, where recursive calls are allowed not only on the direct subterms of the recursive argument, but on any subterm. This models course-of-value recursion and it is reducible to primitive eliminators as shown by Giménez (1995). For example, we can define division by two on natural numbers as follows:

$$\begin{aligned}&\mathsf{half}\ 0 = 0\\&\mathsf{half}\ 1 = 0\\&\mathsf{half}\ (\mathsf{S}\ (\mathsf{S}\ n)) = \mathsf{S}\ (\mathsf{half}\ n)\end{aligned}$$

The syntactic checker can see that the recursive call is made on a subterm of the initial argument $(\mathsf{S}\ (\mathsf{S}\ n))$ and is hence valid. Unfortunately, this check is very sensitive to syntactic changes and therefore requires a particular way of writing programs. For example, the following extensionally equivalent definition of half is rejected:

$$\mathsf{half}\ n = \texttt{if}\ n < 2\ \texttt{then}\ 0\ \texttt{else}\ \mathsf{half}\ (n-2)$$

Showing that $n - 2$ is actually smaller than n in the `else` branch goes beyond syntactic checking and requires a more precise analysis of the program.

Moreover, Coq's `fix` primitive only handles structural recursion on a single argument; for example, one needs two fixpoints in order to define functions that use a lexicographic order like the merge function presented in §2.2. Indeed, lexicographic recursion on pairs is disallowed in Coq since, for example, a pair of natural numbers contains no recursive subterms of type pair, so no recursive calls can be accepted on pairs of natural numbers.

Agda takes this approach even further by allowing a wider class of definitions. First, its notion of being structurally smaller goes beyond that of a deep subterm. For example, the list $x :: xs$ is considered structurally smaller than the list $x :: y :: xs$.

In addition, the termination checker of Agda tries to find a lexicographic order which makes every recursive call of a function to itself—possibly via calls to other functions when considering a mutually recursive block—strictly decreasing, This allows, for example, a straightforward formalisation of the merge function. Moreover, it makes the system more robust against minor variations: the definition of merge is accepted even when replacing its recursive equation by the following (somehow) equivalent one, where arguments are swapped in the recursive call:

$$\begin{aligned}&\texttt{merge}\ (x :: xs)\ (y :: ys) = \\ &\qquad \texttt{if}\ x < y\ \texttt{then}\ x :: \texttt{merge}\ xs\ (y :: ys)\ \texttt{else}\ y :: \texttt{merge}\ ys\ (x :: xs)\end{aligned}$$

For a better understanding on how Agda's termination checker works and how powerful it is, the reader is referred to Abel (1998) and Abel and Altenkirch (2002).

The main advantage of using more flexible termination checkers is that users do not have to deal with unnatural encodings in order to make the function fit in the restrictions of the system. Instead, they can write the algorithm directly and naturally, as long as its structure lies in the scope of the termination checker.

On the other hand, any syntactic criterion is necessarily incomplete. If the definition in question is not accepted by the termination checker, there is no way for a user to simply supply a manual termination proof as a justification. Instead, the function must be rewritten in a way that complies with the system, which might result in an overly complex definition.

It may be tempting to try to improve the usability of the system by extending the termination checker to cover even more advanced definitions. While this is convenient from a user's perspective, such extensions require great care, as the termination checker is usually part of the trusted core of the system. Improving its applicability always incurs the danger of accidentally making the system unsound.

4.2. *Implicit Use of Well-Founded Recursion*

Instead of employing a built-in termination check, the proof assistant can also pass the task of showing the termination of a function back to the user. Then the user must specify a well-founded relation together with the definition of the function, and provide proofs that all recursive calls are made on smaller values. This approach was already used in

the early Boyer-Moore system (Boyer and Moore 1979), which has evolved into ACL2, and it is also used in PVS.

More precisely, the approach works as follows:

1 The user gives the recursive equations for the function and a well-founded relation $\prec$.
2 The system analyses the equations and extracts the arguments of the recursive calls. It then produces proof obligations that state that the recursive calls are made on smaller arguments with respect to $\prec$ than the original one.
3 The user solves the proof obligations, thus establishing that the function is total.
4 The system postulates the recursive equations as axioms, which is now a safe addition given that termination has been proved.

Often, instead of a relation $\prec$ and a proof that it is well-founded, the user can provide a measure, that is, a function into the natural numbers or some other previously known well-founded set. In this case, the proof obligations state that the measure becomes smaller on the argument of each recursive call.

Generating the proof obligations (also called termination conditions) from the equations amounts to extracting recursive calls and their context, which is straightforward as long as no higher-order recursion is involved. For example, if we consider the merge function, then the following proof obligations are generated:

$$x < y \to (xs, y :: ys) \prec (x :: xs, y :: ys)$$
$$\neg(x < y) \to (x :: xs, ys) \prec (x :: xs, y :: ys)$$

The conditions arise from the fact that the recursive calls occur under an if-expression. A general description of the extraction procedure is given by Boyer and Moore (1979, p. 44).

Something that does not arise in the first-order framework of Boyer and Moore is higher-order recursion; then, the extraction of recursive calls is no longer straightforward. For example, in the mirror function (§2.4), we cannot know what the argument of the recursive call is unless we use knowledge about the implementation of the map function.

Moreover, nested recursion cannot be handled by this technique unless the theory has expressive enough types to give a characterization of the results of recursive calls at definition time, which would allow showing that nested calls are valid even though the function is not defined yet. Hence PVS can handle nested recursion such as the f91 function. We describe a use of the same technique in the definitional case in §5.2.2.

Additional Comments Compared to built-in termination checkers, this technique is more general. In principle, any total (first-order, non-nested) function can be defined with it, since the potentially hard problem of finding a termination proof is passed back to the user. However, this proof obligation can be burdensome at times. For example, it can be hard to find an explicit lexicographic order for some of the functions that pass Agda's termination checker.

To recover the ease of the use of a hard-wired termination checker, add-on tools can be provided that heuristically select a relation and automate the proofs where possible. However, even in the presence of such tools, the approach is still different in nature from plain termination checking. The key difference is that the proof obligations and

their proofs are expressed in the logic itself instead of only in the meta-theory. This means that the proofs can be checked using the normal proof checker, and hence the automated tool that produces the proofs need not be trusted. What must still be trusted is the code that generates the proof obligation from the equations and finally declares the axioms—arguably a simpler task than termination checking.

Observe that even though a termination proof for the function is built, this technique still relies on trusted code and declares axioms, hence our classification as axiomatic. The reader should compare this technique with its definitional counterpart presented in §5.1 that, when suitably automated, has a similar look-and-feel to the user, but does not rely on axioms.

4.3. *Type-Based Termination*

Another line of research explores the possibility of adding termination information directly to the underlying formalism, i.e., internalising it. The most prominent approach in this field is the use of *sized types*. Here, types are annotated with a size index which is either the precise size of the term being considered or an upper bound of it. For the purpose of this paper, we consider sizes (sometimes also called stages) as being generated by the following grammar

$$s ::= \iota \mid s + 1 \mid \infty$$

where ι is a size variable. Using quantification on size variables (explicit or implicit depending on the system), types can relate the size of the output of a function to the size of its input. When a recursive call is performed, it must be checked that the size of the argument decreases, which then entails termination of the function. This check is done during type-checking using a rule similar to the following:

$$\frac{f : A^{\iota} \to B \vdash b : A^{\iota+1} \to B[\iota := \iota + 1]}{\vdash \mathsf{fix}\ f.\, b : A^{s} \to B[\iota := s]}$$

Clearly, the body b can call f only on arguments strictly smaller than the initial argument of size $\iota + 1$. In this approach, one can assign the type $\forall A.\, (A \to \mathsf{Bool}) \to [A]^{\iota} \to [A]^{\iota}$ to the filter function, which expresses that the size of the returned list is smaller or equal to the size of the input list (recall that the size index represents an upper bound of the actual size of the term). One can then check that the quicksort function terminates with the type $\forall A.\, [A]^{\infty} \to [A]^{\infty}$ just by looking at the type of filter. Notice that the type of quicksort is not as precise as one could expect here: indeed, one needs a more complex algebra for sizes to express that quicksort is a size-preserving function. Barthe et al. (2008) explore an extension of the sized types system to deal with this issue, which poses a number of hard meta-theoretical problems, solved by restricting the shape of terms. Inference of sizes can be done, and is basically the same amount of work as checking the guardness condition in Coq.

Additional Comments The type-based termination approaches developed for theorem proving are able to deal with higher-order and nested recursion. These approaches provide a good alternative to the syntactic criteria described in §4.1, as they rely on slightly more

semantic and compositional notions. However, both approaches share the disadvantage of requiring to change the theory behind the system and accordingly adapt its meta-theory.

On the other hand, due to the automatic nature of the resolution of size constraints, systems that use type-based termination are unable to deal with problems requiring involved reasoning on size. In addition, functions requiring an ad-hoc well-founded order (like McCarthy's 91-function) are out of reach.

Finally, it must be added that no mature system uses type-based termination yet, although Abel (2010) presents a prototype implementation of a powerful variant of sized types in MiniAgda. This prototype can currently deal with the quicksort, the merge and the mirror example.

Further Reading The idea of sized types was introduced by Hughes et al. (1996) to check productivity of reactive systems and it was adapted in subsequent work by Xi (2001) for a variant of ML, this time for recursive functions. It was then taken up in the theorem proving community by Barthe et al. (2004), Barthe, Grégoire and Pastawski (2006), Blanqui (2004, 2005) and Abel (2006, 2008); the last paper includes a detailed description of related work. A tutorial on type-based termination was presented by Barthe et al. (2009).

4.4. *Logics with Partiality*

An alternative to representing partiality in a logic of total functions is to support it natively. This approach opens up a new range of design choices (and issues), which we can only touch superficially in this paper.

4.4.1. *In Classical Logics* One of the first systems to reason about recursively defined functions is the *Logic for Computable Functions (LCF)*. LCF was implemented by Milner (1972) and based on a logic devised by Scott (1993). Terms in LCF are basically typed λ-terms and the formulae are those of predicate logics. Reasoning about recursive functions is done using fixed-point induction.

To represent partiality in classical logic the basic idea is to accept the possibility that the application of a partial function to a value outside its domain may create terms that do not have a value (we say that they *do not denote*), and to extend the logic to deal with this possibility. Then, partial functions can be represented explicitly using a type $A \rightharpoonup B$.

The design choices that arise in logics with partiality mainly concern the interaction of non-denoting terms with logical connectives and quantification. These issues can already be seen in a classical first-order setting by considering the following questions: Does $t = t$ hold if t is undefined? Does $P \vee \mathsf{true}$ hold if P is undefined? Does $\forall x.x/x = 1$ hold, assuming that division by zero is undefined? It appears that whatever answer one gives to these questions, there are always a few unfamiliar consequences.

For example, in Farmer's (1993) logic PF* that is used in the IMPS system (Farmer et al. 1993), propositions cannot be undefined, so they are always either true or false. Atomic predicates (such as equality) that contain a non-denoting term are mapped to

false. This lets us retain the familiar rules of classical logic, but it also means that equality is no longer reflexive, since $1/0 = 1/0$ does not hold. However, $\forall x.\, x = x$ does hold, since quantifiers only range over defined values. When instantiating a quantifier, an explicit definedness proof is required, and IMPS includes machinery to automate such proofs.

Another route is taken in LPF (Logic of Partial Functions, Barringer et al. 1984), which is used in VDM (Jones 1990). Here, the possibility of undefinedness extends from terms to propositions, which become undefined if terms occurring in them are; this in turn means that the meaning of logical connectives must be extended to cover this case. From the various possibilities, LPF chooses the strongest (most defined) monotonic extension of the classical connectives, which naturally implies that $P \vee \mathsf{true}$ and $\mathsf{true} \vee P$ are theorems, independent from whether P denotes or not. However, this approach sacrifices the law of excluded middle, since for a non-denoting proposition P, neither P nor $\neg P$ hold.

An interesting overview of these design questions is given by Cheng and Jones (1991), who also discuss other approaches.

4.4.2. *In Constructive Logics* In constructive logic, the classical approach is not possible given that all terms must have a defined value.

Bove and Capretta (2008) extend a consistent type theory with a type of partial functions $A \rightharpoonup B$. Each recursive function in that partial type is associated to an inductively defined domain defined basically as in §5.3.2.1. The new type constructor is then justified by interpreting it into the base type theory, showing that the extended theory is also consistent.

This idea had already been used by Constable and Mendler (1985) on a slightly different constructive type theory than the one presented here. Given the definition of a function in the new type of partial functions, an inductively defined domain is constructed in order to characterise the valid inputs of the function. A difference with the work by Bove and Capretta is that the domains of Constable and Mendler play no role in the actual definition of the functions to which they are associated.

Constable and Smith (1987)—who work in the same constructive type theory as Constable and Mendler (1985)—develop a partial type theory where, for each type of the underlying total theory, there exists another type consisting of computations of the elements of the underlying type and hence, might contain diverging terms. Together with this type of partial elements, a termination predicate and an induction principle to reason about partial functions are introduced.

Based on this idea, Audebaud (1991) defines a conservative extension of the Calculus of Constructions with fixed point terms and a type of partial objects, where one can still prove strong normalisation for terms with no fixed points. From the computational point of view, we obtain an equivalent of Kleene's fixpoint theorem for partial recursive functions, but logical aspects need more examination.

5. Definitional Techniques

In this section we discuss techniques that do not require any modification to the underlying formalism or the addition of axioms. First, we show how the standard principle of

well-founded recursion can be used to construct (rather than axiomatise as in §4.2) general recursive functions. We continue by discussing how recursive definitions can benefit from the use of dependent types. Then we discuss techniques that allow the definition of our problematic functions by modifying the type of the function in some way. First, we present those approaches that change the domain of the function, then those that define a partial type for the result. We end this section with some classical techniques that do not fall into any of the previously described categories.

5.1. *Explicit Use of Well-Founded Recursion*

The negative formulation of the notion of well-foundedness that we presented in the introduction (and due to Aczel (1977a)) can be formulated positively using the notion of *accessibility* (Nordström 1988) as in the following inductive definition, which works uniformly in both constructive type theory and HOL.

$$\begin{aligned}&\mathsf{Inductive\ Acc}\ : \forall(A : \mathsf{Set})(\prec : A \to A \to \mathsf{Prop}).\, A \to \mathsf{Prop}\\ &\quad \mathsf{acc} : \forall(A : \mathsf{Set})(\prec : A \to A \to \mathsf{Prop})(a : A).\\ &\qquad (\forall(x : A).\, x \prec a \to \mathsf{Acc}\ A\ (\prec)\ x) \to \mathsf{Acc}\ A\ (\prec)\ a\end{aligned}$$

$$\begin{aligned}&\mathsf{wf} : \forall(A : \mathsf{Set})(\prec : A \to A \to \mathsf{Prop}).\, \mathsf{Prop}\\ &\mathsf{wf}\ A\ (\prec) = \forall a : A.\, \mathsf{Acc}\ A\ (\prec)\ a\end{aligned}$$

The idea behind the accessibility predicate is that $\mathsf{Acc}\ A\ (\prec)$ corresponds to the well-founded part of $\prec$ in A: a proof of $\mathsf{Acc}\ A\ (\prec)x$ shows that there are no infinite descending $(\prec)$-chains starting from x. Paulson (1986) provides a detailed study of well-founded relations. Well-founded induction (and recursion), sometimes called Noetherian induction, is a standard theorem of set theory.

5.1.1. *Well-Founded Recursion in Constructive Type Theory*

The elimination rule associated to the inductive definition of Acc is known as the *principle of well-founded recursion* and has the following definition, where τ is either Set or Prop:

$$\begin{aligned}&\mathsf{wfr} : \forall(P : A \to \tau)(a : A).\\ &\qquad \mathsf{Acc}\ A\ (\prec)\ a \to (\forall x.\, \mathsf{Acc}\ A\ (\prec)\ x \to (\forall y.\, y \prec x \to P\ y) \to P\ x) \to P\ a\\ &\mathsf{wfr}\ P\ a\ (\mathsf{acc}\ a\ h)\ e = e\ a\ (\mathsf{acc}\ a\ h)\ (\lambda y.\, \lambda q.\, \mathsf{wfr}\ P\ y\ (h\ y\ q)\ e)\end{aligned}$$

Although this definition might seem complicated at first sight, it is nothing more than the generalisation of course-of-value induction to an arbitrary well-founded relation $\prec$.

By properly instantiating P above, we can use wfr both to define recursive functions over a well-founded set and to prove properties by well-founded induction with respect to the relation $\prec$.

Let us illustrate this technique by using wfr to define the quicksort algorithm on lists of natural numbers. First, we need to find a relation $\prec : [\mathsf{Nat}] \to [\mathsf{Nat}] \to \mathsf{Prop}$ which is well-founded on lists of natural numbers. If we take the relation that compares the lengths of two lists we can easily prove that it is well-founded on [Nat], that is, we can define a function $\mathsf{allacc} : \mathsf{wf}\ [\mathsf{Nat}]\ (\prec)$. Second, we need to prove that both recursive calls

to the quicksort algorithm are performed on lists that are smaller with respect to the relation we use; in this example, we need to prove that the lists resulting from the calls to the filter function are both shorter than the original list argument of quicksort, in other words, we need to prove the following properties.

$$\begin{aligned}&\mathsf{prlt} : \forall(x : \mathsf{Nat})(xs : [\mathsf{Nat}]).\, \mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs \prec x :: xs\\ &\mathsf{prge} : \forall(x : \mathsf{Nat})(xs : [\mathsf{Nat}]).\, \mathsf{filter}\ (\lambda y.\, y > x)\ xs \prec x :: xs\end{aligned}$$

This gives us enough to build the functional argument e above and define the quicksort algorithm by a call to wfr, now hiding the accessibility argument entirely.

$$\begin{aligned}&\mathsf{qs} : [\mathsf{Nat}] \to [\mathsf{Nat}]\\ &\mathsf{qs}\ xs = \mathsf{wfr}\ (\lambda\_.\, [\mathsf{Nat}])\ xs\ (\mathsf{allacc}\ xs)\ \mathsf{qs}_{\mathsf{wfr}}\\ &\quad \text{where } \mathsf{qs}_{\mathsf{wfr}} : \forall xs : [\mathsf{Nat}].\, \mathsf{Acc}\ [\mathsf{Nat}]\ (\prec)\ xs \to (\forall ys : [\mathsf{Nat}].\, ys \prec xs \to [\mathsf{Nat}]) \to [\mathsf{Nat}]\\ &\qquad \mathsf{qs}_{\mathsf{wfr}}\ []\ \_\ h = []\\ &\qquad \mathsf{qs}_{\mathsf{wfr}}\ (x :: xs')\ \_\ h = h\ (\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs')\ (\mathsf{prlt}\ x\ xs') +\!\!+\\ &\qquad\qquad\qquad\qquad x :: h\ (\mathsf{filter}\ (\lambda y.\, y > x)\ xs')\ (\mathsf{prge}\ x\ xs')\end{aligned}$$

5.1.2. *Well-Founded Recursion in HOL* The recursion combinator wfr above uses a dependent type to ensure that recursive calls are only made on smaller arguments with respect to $\prec$. In HOL, a similar effect is achieved with the simply-typed recursion combinator $\mathsf{wfrec} : (A \to A \to \mathsf{Bool}) \to ((A \to B) \to A \to B) \to A \to B$, together with the so-called *contraction condition*, for a functional F:

$$\forall(x : A)(f\ g : A \to B).\, (\forall y.\, y \prec x \to f\ y = g\ y) \to F\ f\ x = F\ g\ x$$

Intuitively, the contraction condition states that $F\ f\ x$ only depends on the values of f for arguments that are smaller than x with respect to $\prec$, since two functions that agree on these values must lead to the same result.

Using this condition, which we abbreviate as $\mathsf{contractive}\ F\ (\prec)$, we can obtain the following fixpoint theorem.

$$\mathsf{wf}\ (\prec) \to \mathsf{contractive}\ F\ (\prec) \to \mathsf{wfrec}\ (\prec)\ F\ x = F\ (\mathsf{wfrec}\ (\prec)\ F)\ x$$

To define the quicksort algorithm using wfrec (see the end of this section for the formal definition of wfrec) we first define the functional F_{qs} as follows.

$$\begin{aligned}&\mathsf{F}_{\mathsf{qs}}\ qs\ [] = []\\ &\mathsf{F}_{\mathsf{qs}}\ qs\ (x :: xs) = qs\ (\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs) +\!\!+ x :: qs\ (\mathsf{filter}\ (\lambda y.\, y > x)\ xs)\end{aligned}$$

Then, the function qs can be defined as $\mathsf{wfrec}\ (\prec)\ \mathsf{F}_{\mathsf{qs}}$, where $\prec$ is again the relation that compares the lengths of the lists. Now it remains to prove the contraction condition. Below is the case for $x :: xs$.

$$\begin{aligned}&(\forall ys.\, ys \prec x :: xs \to f\ ys = g\ ys) \to\\ &\quad f\ (\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs) +\!\!+ x :: f\ (\mathsf{filter}\ (\lambda y.\, y > x)\ xs) =\\ &\quad g\ (\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs) +\!\!+ x :: g\ (\mathsf{filter}\ (\lambda y.\, y > x)\ xs)\end{aligned}$$

It is easy to see that to be able to rewrite the occurrences of f to g, we must prove that its arguments are smaller than $x :: xs$ with respect to $\prec$. Thus, the properties prlt and

prgt (that we already saw in §5.1.1) are precisely what can be applied here. Finally, using the fixpoint theorem, we obtain the desired recursive equations for quicksort.

Above, we have omitted the definition of the wfrec combinator itself. It can be constructed by defining its graph inductively, proving that it is single-valued, and then using function_of to turn the relation into a function. Finally, the fixpoint theorem above can be derived.

$$\begin{aligned}&\mathsf{Inductive}\ \mathsf{wfrec\_rel} : \forall A, B : \mathsf{Set}.\ (A \to A \to \mathsf{Bool}) \to \\ &\qquad ((A \to B) \to A \to B) \to A \to B \to \mathsf{Bool} \\ &\quad \mathsf{wfrec\_rel}_1 : \forall (g : A \to B)(x : A).\ (\forall z.\, z \prec x \to \mathsf{wfrec\_rel}\ (\prec)\ F\ z\ (g\ z)) \to \\ &\qquad \mathsf{wfrec\_rel}\ (\prec)\ F\ x\ (F\ g\ x)\end{aligned}$$

$$\mathsf{wfrec}\ (\prec)\ F = \mathsf{function\_of}\ (\mathsf{wfrec\_rel}\ (\prec)\ F)$$

5.1.3. *Discussion and Comparison* We can say that function definitions by well-founded recursion have the following properties.

— A well-founded relation is required at definition time, together with proofs that the recursive calls are performed on smaller arguments in the case of constructive type theory. For HOL, this can be defered but the termination conditions will become premisses of the defining equations.
— The definition mixes computational information (the program) with logical information (the termination proof and conditions). This makes the definition less readable, and it can get in the way when reasoning (or computing, in the case of constructive type theory) with the function. For example, this might render the encoding impractical when doing proof-by-reflection where using the built-in reduction is critical.
— To hide the underlying construction, the recursive equations can be derived. These equations permit reasoning about the function in terms of the algorithmic behaviour, not its internal definition. This is the standard in HOL, but in constructive type theory, it requires the use of explicit equational reasoning rather than just relying on the built-in reduction mechanism. It then depends on the available automation whether this style of working is practical. In Coq, where rewriting tactics are available, it is relatively common, whereas Agda users prefer the built-in reduction, since equational proofs have to be spelt out explicitly.
— Partial and nested functions are not directly supported. However, some techniques discussed later, which can be seen as refinements of well-founded recursion, allow the definition of such functions.
— Functions with higher-order recursion are not trivially supported either since we might need information regarding how the higher-order function works in order to guarantee that each recursive call is performed on a smaller argument. Using the so called *congruence rules* and some techniques we explain later, which can be seen as refinements of well-founded recursion, one can deal with functions with higher-order recursion.

In constructive type theory, the following additional points are notable.

— Computing within the prover recurses over the termination proof. This imposes a slight overhead and in addition, it leads to problems when the termination proof is

not closed, i.e., contains axioms or local assumptions. Then it may happen that the function is not reduced properly.

— When code extraction is available (as it is the case in Coq), the accessibility predicate is erased completely, as it lives in Prop. Therefore, the extracted code from qs is the original functional quicksort algorithm.

Many proof assistants today come with tool support to automate definitions using well-founded recursion and make them transparent to the user by deriving the recursive equations and an induction rule. We will show the use of such tools in §6.

Further Reading The definition of Acc and its use to define general recursive functions in type theory is due to Nordström (1988). In HOL, the approach based on the contraction condition was first described by Harrison (1995).

Slind (1996) implemented a tool that automates the definition, and derives the recursive equations and an induction rule. The tool worked for HOL4 and Isabelle/HOL uniformly and it was the first purely definitional tool for general recursion.

Balaa and Bertot (2000) use well-founded recursion in Coq to prove the recursive equation defining a function. Unfortunately their approach does not have a clear separation between the computational part of an algorithm and the proof that it terminates, something that it is improved in a later work (Balaa and Bertot 2002). In any case, it is not very clear how their work can be used to formalise partial or nested recursive algorithms.

Similar automation to that in Slind (1996) is provided in Coq by a tool by Barthe, Forest, Pichardie and Rusu (2006). This work together with that of Balaa and Bertot (2000), among others, is the basis of the Function tool that we discuss in §6.2.1.

5.2. *Exploiting Dependent Types*

In this section, which applies only to systems based on constructive type theory, we show how dependent types can be leveraged to extend the scope of basic structural recursion and the well-founded recursion technique discussed above.

5.2.1. *Precise Argument Types*

5.2.1.1. *Enriching Types* Dependent types allow us to index data by arbitrary information which in turn, makes it possible to use structural recursion on the index of a term to show the termination of functions manipulating such a datatype. In other words, this approach corresponds to building the recursion argument inside the data structure. Actually, one can also see this technique as a first-class version of the sized-types idea presented in §4.3: by systematically indexing datatypes with their size one can always use this information to prove termination.

One of the simplest examples is the datatype of vectors, that is, a datatype where we index a list by its length.

$$\begin{aligned}&\mathsf{Inductive\ Vec} : \forall(A : \mathsf{Set}).\, \mathsf{Nat} \to \mathsf{Set}\\ &\quad \mathsf{nil} : \mathsf{Vec}\ A\ 0\\ &\quad \mathsf{cons} : \forall(n : \mathsf{Nat}).\, A \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{S}\ n)\end{aligned}$$

The reverse of a vector can be defined as a function of type $\forall(n : \mathsf{Nat}).\, \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ n$. Then, a function performing a recursive call on the reverse of a vector can be written by structural recursion on the length (the index) of the vector.

A prime example of the use of this approach was demonstrated by McBride (2003), who shows that the standard first-order unification algorithm is structurally recursive on the number of free variables that occur in the terms to be unified. By indexing terms by their number of free variables, it is then possible to write an obviously strucurally recursive version of unification. A possible drawback is that this technique usually requires significant changes to pre-existing programs and datatypes considered, far beyond automation capabilities. However the advantage is that more structure is made apparent in types and specifications, making the program more self-explanatory.

5.2.1.2. *Precise Domain Types* Dependent types can also be used to restrict the domains of arguments and, in that way, turn partial function definitions into total ones. Suppose that we want to define division of natural numbers by recursion on its first argument:

$$\mathtt{div}\ x\ y = \mathtt{if}\ x < y\ \mathtt{then}\ 0\ \mathtt{else}\ \mathsf{S}\ (\mathtt{div}\ (x - y)\ y)$$

This definition is not structurally recursive and worse, it is not defined for $y = 0$ since the attempt to evaluate $\mathtt{div}\ x\ 0$, for any x, will produce an infinite reduction sequence. It is possible however to show that `div` terminates if we restrict the domain of the second argument to non-zero natural numbers. Using well-founded recursion we can write a program of type $\forall x : \mathsf{Nat}.\, \{y : \mathsf{Nat} \mid y \neq 0\} \to \mathsf{Nat}$, where the restricted domain is modelled by a subset type.

Observe however that while this technique is in principle applicable to any partial function, it might not be so easy to manually identify a suitable restriction of the domain. In §5.3.2 we show a more systematic way to identify the precise domain of a function.

5.2.1.3. *Higher-Order recursion* Enriched argument types can also be used to handle higher-order recursion by restricting the application of an argument function to specific objects. For example, one can enrich the type of the usual `map` function like this: $\forall A\, B.\forall l : [A].(\forall x : A.x \in l \to B) \to [B]$. Now, the `mirror` function can be defined by well-founded recursion on the subterm order on trees. The corresponding functional becomes:

$$F\ (\mathtt{tree}\ a\ ts)\ \mathtt{mirror} = \mathtt{tree}\ a\ (\mathtt{rev}\ (\mathtt{map}\ ts\ (\lambda x\ p.\ \mathtt{mirror}\ x\ (p'\ p))))$$

where p' shows that $x \in ts \to x < \mathtt{tree}\ a\ ts$.

5.2.2. *Precise Result Types* Subset types can also be used to attach information to the codomains of functions, typically specifying their input-output relation. This allows writing nested recursive definitions naturally. In the case of McCarthy's 91-function for example, we can enrich the result type with enough information to prove that the recursion is well-founded with respect to the measure $101 - n$. Using a subset type, we can define

$$\mathsf{f91} : \forall n : \mathsf{Nat}.\, \{m : \mathsf{Nat} \mid \mathsf{if}\ n \leqslant 100\ \mathsf{then}\ m = 91\ \mathsf{else}\ m = n - 10\}$$

thereby specifying the computational behaviour of the function completely in its type. At the nested call $\mathsf{f91}\ (\mathsf{f91}\ (n+11))$, we then know either that $n+11 \leqslant 100$ and $\mathsf{f91}\ (n+11) = 91$, or that $n+11 > 100$ and $\mathsf{f91}\ (n+11) = n+1$. This gives us enough information to show that the result of the inner recursive call is indeed decreasing according to the measure; in other words, to show that $101 - \mathsf{f91}\ (n+11) < 101 - n$, which is equivalent to showing that $\mathsf{f91}\ (n+11) > n$. Then we know it is safe to allow the outer recursive call to the function, that is, the nested call $\mathsf{f91}\ (\mathsf{f91}\ (n+11))$.

More interesting examples of nested recursion can be handled similarly, for example evaluation functions of λ-calculus are often nested: reducing an application $(\lambda x.\ t)\ a$ might first require evaluating a to some a' and then continuing the evaluation with $t[a'/x]$. Showing totality of such functions requires to know that the nested call is made on a reduct of the original term and that the evaluation relation is well-founded. To show that the term $a' = \mathtt{eval}\ a$ is a reduct of a we have to specify, *in the return type* of the evaluation function, that it produces a reduct of its argument. The well-foundedness proof then corresponds to the termination of that evaluation relation.

5.2.3. *Discussion and Comparison* As we have just seen in these approaches—which are possible in all systems based on constructive type theory—the use of dependent types for the definition of functions considerably enlarges the set of functions that can be defined by well-founded recursion. However, when giving more precise types to functions, both the types of the functions and the bodies of the definitions must be decorated with more logical information, thus aggravating the issues associated with well-founded recursion. Good tool support can, on the other hand, alleviate most of these problems. This is the case for the Program tool (§6.2.2), which is particularly well-suited for giving precise types to functions given its support for subset types. It can be seen as an explicit version of the implicit use of well-founded recursion in PVS.

Another possible disadvantage is that, in addition to the well-founded order, a suitable restriction for the argument or the result type have to be known at definition time. On the other hand, by specifying more precise information in the types we can also deal with higher-order recursion.

Interestingly, the general idea of strengthening the argument and result type can also be applied in a classical setting without dependent types. The notion of inductive invariants proposed by Krstić and Matthews (2003) essentially boils down to a variant of the well-founded recursion technique in HOL, where the contraction condition is strengthened with a relation S. Charguéraud (2010) takes this even further by adding a domain condition D as well. The contraction condition then has the following form.

$$\begin{aligned}&\forall x\ f\ g.\ D\ x \to (\forall y.\ y \prec x \to D\ y \to f\ y = g\ y \wedge S\ y\ (f\ y)) \to \\ &\quad F\ f\ x = F\ g\ x \wedge S\ x\ (F\ f\ x)\end{aligned}$$

Note how this condition carries the same information as assigning the dependent type $\forall x : A.\ D\ x \to \{y : B \mid S\ x\ y\}$ to the function.

5.3. *Partiality in the Argument*

5.3.1. *Step Counter* A popular ad-hoc solution for formalising partial functions as total functions is to add a step counter as an additional argument, which is simply a natural number that decreases at each recursive call. When the counter is exhausted, a none value is returned. For example, here is the iter_0 function rewritten in this way:

$$
\begin{array}{l}
\mathsf{iter}_0\ 0\ f\ x = \mathsf{none} \\
\mathsf{iter}_0\ (\mathsf{S}\ n)\ f\ x = \\
\quad \mathsf{if}\ x = 0\ \mathsf{then\ some}\ [] \\
\qquad \mathsf{else\ case}\ \mathsf{iter}_0\ n\ f\ (f\ x)\ \mathsf{of\ none} \to \mathsf{none} \mid \mathsf{some}\ r \to \mathsf{some}\ (x :: r)
\end{array}
$$

The transformed function is now accepted in the systems, since it is structurally recursive over the counter, which acts as a bound on the number of recursive calls that will be made. It can be evaluated with little overhead, given a value for the counter. The technique is also quite expressive: it deals nicely with arbitrary recursion, including nested recursion (thinking of the index as the size of the call stack, all nested calls can use the same index) and higher-order recursion. Additionally, we do not need any knowledge about the termination behaviour initially.

However, there are two main drawbacks in this approach. First, when calling the function, a suitable value for the bound must be provided, which is normally not obvious. Second, one must always be prepared to get an empty result, either due to actual non-termination or just because the bound was too tight. In addition, reasoning about the function inevitably requires reasoning about the counter and proving that it is large enough when the function is called; this corresponds to a termination proof on the respective arguments.

Despite these complexities, this technique is quite easy to carry out and it is often used (Boyer and Moore 1996, Leroy 2006, Nipkow et al. 2006).

5.3.2. *Inductive Domains* As mentioned in §5.2.1, it is not always easy to look at a function and come up with a suitable description of its domain as a subset type. Here we explain techniques that construct the domain of a function from the informal recursive equations defining it, and that use this domain to reason about the function.

5.3.2.1. *Inductive Domains in Constructive Type Theory.* Given the recursive equations for a function, we can mechanically define an inductive predicate that characterises the domain of the function. Let us consider the quicksort algorithm again. The first equation tells us that the algorithm terminates on the empty list, hence [] belongs to the domain. The second equation tells us that if quicksort terminates on both recursive calls, then the algorithm also terminates on the list $(x :: xs)$ since the function $+\!\!+$ is a total function. Hence, $(x :: xs)$ belongs to the domain if both $\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs$ and $\mathsf{filter}\ (\lambda y.\, y > x)\ xs$ belong to it. We can formalise this information as an inductively defined predicate.

$$
\begin{array}{l}
\mathsf{Inductive}\ \mathsf{Dom}_{\mathsf{qs}} : [\mathsf{Nat}] \to \mathsf{Prop} \\
\quad \mathsf{dom}_{[]} : \mathsf{Dom}_{\mathsf{qs}}\ [] \\
\quad \mathsf{dom}_{::} : \forall (x : \mathsf{Nat})(xs : [\mathsf{Nat}]).\, \mathsf{Dom}_{\mathsf{qs}}\ (\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs) \to \\
\qquad\qquad\qquad\qquad \mathsf{Dom}_{\mathsf{qs}}\ (\mathsf{filter}\ (\lambda y.\, y > x)\ xs) \to \mathsf{Dom}_{\mathsf{qs}}\ (x :: xs)
\end{array}
$$

In constructive type theory, we can now define the function by structural recursion on the proof that the input arguments satisfies the domain predicate. Formally, the function now takes this proof as an extra argument and is defined by pattern matching on it.

$$\begin{aligned}
&\mathsf{qs} : \forall xs : [\mathsf{Nat}].\, \mathsf{Dom}_{\mathsf{qs}}\ xs \to [\mathsf{Nat}]\\
&\mathsf{qs}\ []\ \mathsf{dom}_{[]} = []\\
&\mathsf{qs}\ (x :: xs)\ (\mathsf{dom}_{::}\ x\ xs\ h_1\ h_2) = \mathsf{qs}\ (\mathsf{filter}\ (\lambda y.\, y \leqslant x)\ xs)\ h_1 +\!\!+\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x :: \mathsf{qs}\ (\mathsf{filter}\ (\lambda y.\, y > x)\ xs)\ h_2
\end{aligned}$$

Observe that, due to the dependency in the type, the list argument is determined by the type of the proof argument. Hence, the former is somehow redundant and can be made implicit (a feature that is available both in Agda and in Coq) without hiding relevant information in the subsequent uses of qs.

For a total function, we can now prove that its domain predicate is always satisfied, and then use this proof to define a total function with the desired type. In this case, we prove that $\forall xs : [\mathsf{Nat}].\, \mathsf{Dom}_{\mathsf{qs}}\ xs$ and we define a quicksort function with type $[\mathsf{Nat}] \to [\mathsf{Nat}]$. Note however, that the evaluation of a total function on a particular argument proceeds by recursion on the domain argument, creating an overhead in the computation time.

Proving that the function is total on the domain is of course not possible when dealing with a strictly partial function. However, the approach described here is still applicable and allows us to formalise strictly partial functions in a total setting. Moreover, we can reason about these partial functions and compute their results for those arguments on which the functions terminate.

Observe that this approach can also be used for non-recursive partial functions as for example the head function on lists, defined by the single equation $\mathtt{head}\ (x :: xs) = x$, which can easily be formalised in type theory as follows:

$$\begin{aligned}
&\mathsf{Inductive}\ \mathsf{Dom}_{\mathsf{head}} : \forall A : \mathsf{Set}.\, [A] \to \mathsf{Prop}\\
&\qquad \mathsf{dom} : \forall (a : A)(as : [A]).\, \mathsf{Dom}_{\mathsf{head}}\ (a :: as)\\
&\\
&\mathsf{head} : \forall (A : \mathsf{Set})(xs :: [A]).\, \mathsf{Dom}_{\mathsf{head}}\ xs \to [A]\\
&\mathsf{head}\ A\ (a :: as)\ (\mathsf{dom}\ a\ as) = a
\end{aligned}$$

In type systems which support inductive-recursive definitions (Dybjer 2000) (as it is the case of Agda but not of Coq), we can also apply this technique to define nested recursive functions. The domain of such a function must then refer to the formal version of the function and hence, both domain and function must be introduced in a mutual inductive-recursive definition. The rest of the process is the same as for the quicksort

algorithm, as it can be seen in the following definition of McCarthy's 91-function.

$$\begin{array}{l}\mathsf{Inductive}\ \mathsf{Dom_{f91}} : \mathsf{Nat} \to \mathsf{Prop} \\ \quad \mathsf{dom}_{>} : \forall n : \mathsf{Nat}.\, n > 100 \to \mathsf{Dom_{f91}}\ n \\ \quad \mathsf{dom}_{\leqslant} : \forall n : \mathsf{Nat}.\, n \leqslant 100 \to \forall h : \mathsf{Dom_{f91}}\ (n+11).\, \mathsf{Dom_{f91}}\ (\mathsf{f91}\ (n+11)\ h) \to \\ \qquad\qquad \mathsf{Dom_{f91}}\ n \end{array}$$

$$\begin{array}{l}\mathsf{f91} : \forall n : \mathsf{Nat}.\, \mathsf{Dom_{f91}}\ n \to \mathsf{Nat} \\ \mathsf{f91}\ n\ (\mathsf{dom}_{>}\ n\ \_) = n - 10 \\ \mathsf{f91}\ n\ (\mathsf{dom}_{\leqslant}\ n\ \_\ h_1\ h_2) = \mathsf{f91}\ (\mathsf{f91}\ (n+11)\ h_1)\ h_2 \end{array}$$

Note that the elimination rule associated to the domain predicate of a particular function is suitable for proving properties about the function by induction given that it follows the recursive structure of the function definition and hence, it provides us with the right inductive hypotheses.

Additional Comments The technique as presented here is formally described in Bove and Capretta (2005a), where also some limitations to the approach are presented. First, this approach does not take into account that a higher-order function can inherit partiality conditions from its arguments. In addition, given that the type of each function defined with this approach is quite ad-hoc and particular to the function itself, what would the type of the functional argument of a higher-order function look like? Another problem is that λ-abstractions in the right-hand side of an equation are translated into total functions and, if a recursive calls is made in the body of the λ-abstraction, the translation generates a domain condition which requires a proof that the domain is satisfied on all arguments. This translation is too strict since the λ-abstraction could diverge on arguments to which it is never applied during execution without jeopardising the termination behaviour of the whole program. Finally, if a function defined with this approach is applied to an insufficient number of arguments, the accessibility condition cannot even be formulated, so partial applications of general recursive functions are not allowed. In a later article (Bove and Capretta 2005b) these problems are solved by defining a type of partial functions between given types where a function is represented as a pair of a domain predicate and a function dependent on the predicate. But this solution comes at a price: it requires an impredicative type theory in order to be applicable to all functions. However, in most practical cases the idea can be adapted to work on a predicative type theory with type universes.

Based on these ideas about inductive domains, Setzer (2006, 2007) defines a datatype (of codes) of partial functions. From the code of a partial function, one can extract the domain of the function and the function itself, and one can evaluate the function on a certain argument. Nested functions and higher-order functions can also be coded as elements of this datatype of partial functions. On the other hand, this encoding is quite involved and creates additional technical overhead.

As we have already mentioned, in Coq one cannot define an element of a datatype by pattern matching on a domain proof, as they may in general have more than one constructor, hence, this technique cannot be implemented in Coq exactly as explained

here. Instead, we need to define a so-called *inversion lemma* for each recursive call in the function. These inversion lemmas must be defined in such a way that they can be seen as subterms of the proof that the function terminates on the input argument, so that the final function is accepted as structurally recursive on the proof of the domain predicate. Finally, the function must be defined by pattern matching on the input argument and not on the proof that it satisfies the domain predicate. A more detailed explanation on how to use this technique in Coq can be found in the book by Bertot and Castéran (2004, ch. 15).

It is possible to avoid inductive-recursive definitions and still allow the definition of nested recursive functions (Bove 2009). The idea is to inductively define the graph of a function and from there the domain of the function as the element which is associated to the input in the graph, if it exists. This approach seems to share the advantages of the technique described here, but it has an additional level of indirection since we need to go from the domain into the graph.

Bertot et al. (2002) present a technique to encode the method we describe above for partial and nested algorithms in type theories that do not support Dybjer's schema for simultaneous inductive-recursive definitions. They do so by combining the inductive domain predicate as presented here with step indexing and the functionals in Balaa and Bertot (2000).

5.3.2.2. *Inductive Domains in HOL.* The technique above again relies on a definition of a function by recursion on a proof. In a classical setting, this is impossible and indeed unnecessary, as the function can also be defined without it. However, inductively defined domain predicates are still very useful for reasoning. First, they give us a direct way to express the statement "f terminates on x" as a proposition, and second, they come with a tailor-made induction principle.

Before we show the construction, we list the differences to its constructive counterpart:

— The function has the simple type $A \to B$. Thus, we can always apply it to any argument, even if the domain predicate does not hold, in which case the result of the function is underspecified (see §3.2).

— The function definition does not refer to the domain, which means that the domain can be defined after the function. In particular, for nested functions, no simultaneous inductive-recursive definition is required.

These properties make this technique especially convenient for nested recursive functions, as it can be seen with McCarthy's 91-function, which is defined in the following steps.

1 Define the graph of the function as an inductive relation:

$$\begin{aligned}&\mathsf{Inductive}\ \mathsf{G_{f91}} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Prop}\\ &\quad \mathsf{g_1} : \forall n.\, n > 100 \to \mathsf{G_{f91}}\ n\ (n-10)\\ &\quad \mathsf{g_2} : \forall n\ r_1\ r_2.\, n \leqslant 100 \to \mathsf{G_{f91}}\ (n+11)\ r_1 \to \mathsf{G_{f91}}\ r_1\ r_2 \to \mathsf{G_{f91}}\ n\ r_2\end{aligned}$$

2 Turn the relation into a function by defining $\mathsf{f91} = \mathsf{function\_of}\ \mathsf{G_{f91}}$.

3 Define the domain; since the function itself is already defined, we can refer to it here.

$$\begin{array}{l}\mathsf{Inductive}\ \mathsf{Dom_{f91}} : \mathsf{Nat} \to \mathsf{Prop} \\ \quad \mathsf{dom}_{>} : \forall n.\, n > 100 \to \mathsf{Dom_{f91}}\ n \\ \quad \mathsf{dom}_{\leqslant} : \forall n.\, n \leqslant 100 \to \mathsf{Dom_{f91}}\ (n+11) \to \mathsf{Dom_{f91}}\ (\mathsf{f91}\ (n+11)) \to \mathsf{Dom_{f91}}\ n\end{array}$$

4 By induction on $\mathsf{Dom_{f91}}$, prove that the relation $\mathsf{G_{f91}}$ is single valued on the domain, that is, $\forall n.\, \mathsf{Dom_{f91}}\ n \to \exists! r.\, \mathsf{G_{f91}}\ n\ r$.

5 Derive the recursive equation, constrained by the domain:

$$\forall n.\, \mathsf{Dom_{f91}}\ n \to \mathsf{f91}\ n = \mathsf{if}\ n > 100\ \mathsf{then}\ n - 10\ \mathsf{else}\ \mathsf{f91}\ (\mathsf{f91}\ (n+11))$$

Up to now, the process was completely generic, as we have not put in any problem-specific knowledge yet. Nevertheless, we already have the function and its domain, the (constrained) recursive equation and an induction rule—which arises from the inductive domain predicate— that lets us prove properties of the form $\forall n.\, \mathsf{Dom_{f91}}\ n \to P\ n$. Now we continue by proving the key property, followed by the termination of the function.

6 Show the lemma $\forall n.\, \mathsf{Dom_{f91}}\ n \to \mathsf{f91}\ n = \mathsf{if}\ n > 100\ \mathsf{then}\ n - 10\ \mathsf{else}\ 91$, which is easily proved by induction on $\mathsf{Dom_{f91}}$.

7 Show termination of f91, that is, prove $\forall n.\, \mathsf{Dom_{f91}}\ n$. Here we can again use the measure $101 - n$. Note that the previous lemma is used to show that the measure decreases at the outer recursive call.

8 Remove the (vacuous) domain condition from the recursive equation.

This seems like a lot of effort, compared to the techniques we saw previously. However, note that steps 1–5 and 8 can be completely automated, and steps 6 and 7 are relatively easy proofs; in particular, in step 6, we have a convenient induction principle at hand, which leads to a simpler proof than the well-founded induction that would be (implicitly) used when using result strengthening via dependent types or inductive invariants (§5.2.2).

For strictly partial functions, only steps 1–5 are applicable. In this case we cannot derive the unconstrained recursive equations and must carry the domain predicate around.

The technique as presented here roughly follows Krauss (2010a), who also implemented a tool that automates the process in Isabelle/HOL. In §6.3, when we demonstrate this tool, the reader can see how little work is left to the user when suitable automation for this technique is available.

5.3.3. *Discussion and Comparison* It is worth observing that the domain predicates associated to both approaches presented above contain basically the same information which, in turn, gives us similar induction principles to reason about the function. Moreover, note that these induction principles follow the recursive structure of the function being defined.

Further Reading The idea of explicitly defining a domain predicate for a partial function seems to have arisen several times independently.

Finn et al. (1997) use a recursive domain predicate to guard non-terminating equations, which they introduce axiomatically in a classical setting. The question of why the (non-terminating) equations that specify the domain predicate are consistent is addressed by

a semantic argument. In retrospect, the simpler answer is that these predicates could in fact be inductively defined.

Dubois and Donzeau-Gouge (1998) refine the idea by Finn et al. and transfer it to type theory, replacing the recursively axiomatised domain predicate with an inductively defined one. Their approach works basically as presented in §5.3.2.1, except that nested recursion is handled with the help of a user-given post-condition (similar as in §5.2.2) instead of a simultaneous inductive-recursive definition.

Giesl (1997, 2001) discusses nested recursion and partiality in the context of automated induction theorem proving, and he points out that inductive reasoning about partial functions works just like for total ones, except that the results are restricted to the domain of the function. While his semantic arguments involving a special notion of partial truth are hard to adapt to a formal setting within type theory or HOL, it turns out that inductive domain predicates and the associated induction principles permit the same convenient reasoning style (Krauss 2006).

Greve (2009) describes a tool for function definitions in ACL2, which also constructs domain predicates. Due to the restricted logic of ACL2, which does not support inductive definitions, the domain predicate must be constructed in an intricate bootstrapping process that involves a reduction to tail-recursive form (see §5.5.2). However, the final result of Greve's construction is the same as seen above: a function whose recursive equations are constrained with a domain predicate, together with an induction rule.

5.4. *Partiality in the Result*

This section describes techniques that work by modifying the result type of the function to accommodate the possibility of non-termination. Thus, partial functions from A to B will be modelled as objects of the type $A \to \mathsf{Partial}\ B$, where the type $\mathsf{Partial}\ B$ expresses a computation which may or may not terminate but if it does, it returns an element in B. The techniques differ in the underlying model of computation, that is, in how the type constructor $\mathsf{Partial}$ is actually defined.

5.4.1. *Domain Theory in a Classical Setting*

Domain-theory (Abramsky and Jung 1994) provides a well-studied foundation for modelling the semantics of possibly non-terminating computations. Partial values are represented by simply adjoining a special undefined value to a type to model non-termination with the help of for example the Option type.

Observe that for any B, the type $\mathsf{Option}\ B$ is a cpo when we define $x \sqsubseteq y$ as $x = \mathsf{none} \vee x = y$. Such a cpo, with a distinguished bottom element and otherwise incomparable elements, is called *flat*.

Building on a formalisation of domain theory, we can exploit the fixpoint theorem to define partial recursive functions of type $A \to \mathsf{Option}\ B$:

1 Define the functional $F : (A \to \mathsf{Option}\ B) \to A \to \mathsf{Option}\ B$.
2 Prove that F is continuous.
3 Define $f = \mathrm{fix}_F$ and apply the fixpoint theorem above to derive the fixpoint equation.

As an example of this idea, we give the functional that will lead to the definition of

the $\texttt{iter}_0$ algorithm:

$$\begin{aligned}\mathsf{F}_{\mathsf{iter}_0}\ \mathit{iter}_0\ f\ x = \mathsf{if}\ x = 0\ \mathsf{then\ return}\ [] \\ \mathsf{else\ do}\ xs \leftarrow \mathit{iter}_0\ f\ (f\ x);\ \mathsf{return}\ (x :: xs)\end{aligned}$$

Note how the monadic operators nicely hide the plumbing with the underlying option type. Proving continuity of the functional is a formality: the monadic primitives return and bind as well as the if-expression preserve continuity, so the proof just follows the structure of the functional and is easily automated. The desired function then arises from the use of the fixpoint theorem.

Additional Comments Domain theory was the logical foundation of one of the first interactive theorem provers, the Edinburgh LCF system (Gordon et al. 1979). While it is an adequate model of computations, using it as a base logic creates some overhead due to the extra definedness, continuity, and admissibility conditions that arise frequently. Later descendants of the system moved to HOL instead. The most comprehensive formalisation of domain theory today is Isabelle/HOLCF, a library built on top of Isabelle/HOL. Isabelle/HOLCF is originally due to Regensburger (1995) and was later refined by Müller et al. (1999) and Huffman (2008, 2009). Using type classes, it manages to make much of the continuity reasoning implicit. Moreover, it provides a tool for defining recursive functions over cpos.

In addition to the simple definition sketched above, the general framework of domain theory naturally deals with more complicated cases, such as higher-order functions that take partial functions as arguments. It can also be used to construct general recursive datatypes, where recursion is not in a strictly positive position (Huffman 2009).

A recent tool by Krauss (2010b) automates the simple fixpoint construction above in plain Isabelle/HOL, hiding the underlying domain-theoretic concepts completely. Similar functionality was developed for Coq by Bertot and Komendantsky (2008), relying on additional classical axioms. Paulin-Mohring (2009) gives a constructive formalisation of cpos in Coq, representing flat domains coinductively, similar to what we will see in the next subsection.

5.4.2. *Coinductive Codomains* In a constructive setting, implementing Partial as the Option type as above does not really model non-termination since it is possible to define a function $\mathsf{terminates} : \mathsf{Option}\ A \to \mathsf{Bool}$ that decides whether an element in $\mathsf{Option}\ A$ actually returns a value in A or the none element. By the undecidability of the halting problem we know that such a function is not possible to write in the presence of actual non-termination.

Using a coinductive definition however, we can construct a type of computations that models partial values constructively. The idea is that a computation either directly produces a result, or it composes two sub-computations, the second depending on the result of the first.

$$\begin{aligned}&\mathsf{Coinductive\ Computation} : \forall A : \mathsf{Set}.\ \mathsf{Set} \\ &\quad \mathsf{return} : A \to \mathsf{Computation}\ A \\ &\quad \mathsf{bind} : \mathsf{Computation}\ A \to (A \to \mathsf{Computation}\ A) \to \mathsf{Computation}\ A\end{aligned}$$

We can make a few observations here. First, the constructors of this type of partial elements naturally form a monad. Second, non-termination corresponds to an infinite computation (unlike with the classical definition of Partial) which, in turn, corresponds to the intuition that we cannot detect non-termination. Finally, we can define functions returning computations corecursively; in the `iter`$_0$ example we have:

$$\begin{aligned}\mathsf{iter}_0\ f\ x = \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{return}\ []\\ \mathsf{else\ do}\ xs \leftarrow \mathsf{iter}_0\ f\ (f\ x);\ \mathsf{return}\ (x :: xs)\end{aligned}$$

Note that the recursive call is guarded by the bind constructor (here hidden by the do-notation), which makes this definition a valid corecursive one. Actually, we can always express general recursive definitions by guarded corecursive ones, possibly after wrapping an unguarded call c into a vacuous $\mathsf{bind}\ c\ (\lambda x.\ \mathsf{return}\ x)$.

This is of course too good to be true. What the function above actually returns is not a value itself but a computation trace which may be infinite. Wrapping recursive calls in binds simply adds steps to the computation trace, and relegates the burden of showing termination to the *consumer* of the computation. Then, to evaluate such a computation we must either impose a bound on the recursion depth (as in §5.3.1)

$$\mathsf{bounded\_eval} : \forall A : \mathsf{Set}.\ \mathsf{Nat} \to \mathsf{Computation}\ A \to \mathsf{Option}\ A$$

or supply a termination proof (as in §5.1 or §5.3.2)

$$\mathsf{eval} : \forall (A : \mathsf{Set})(c : \mathsf{Computation}\ A).\ \mathsf{Terminates}\ c \to A$$

where the predicate Terminates c is defined as $\exists a : A.\ c \downarrow a$ and the strengthened predicate $c \downarrow a$ expresses that c terminates with result a.

$$\begin{aligned}&\mathsf{Inductive}\ \_ \downarrow \_ : \forall A : \mathsf{Set}.\ \mathsf{Computation}\ A \to A \to \mathsf{Prop}\\ &\quad \mathsf{value\_return} : \forall a : A.\ \mathsf{return}\ a \downarrow a\\ &\quad \mathsf{value\_bind} : \forall (a\ a' : A)(c : \mathsf{Computation}\ A)(f : A \to \mathsf{Computation}\ A).\\ &\qquad\qquad c \downarrow a \to f\ a \downarrow a' \to \mathsf{bind}\ c\ f \downarrow a'\end{aligned}$$

Note that $\_ \downarrow \_$ is defined inductively, hence guaranteeing that the computation is finite.

If we use this technique for defining a total function, we now proceed by proving $\forall xs.\ \mathsf{Terminates}\ xs$. Then, with the help of eval, we can define a total function of the originally intended type. For a strictly partial function, we can still prove its termination on specific arguments.

However, there are some technical issues. First, the standard (Leibniz) equality is not appropriate to compare computations, since it not only compares the resulting values but also the size and structure of the computation. For example, the computations return 0 and bind (return 0) return are not equal, although they both return the value 0. Even two non-terminating computations are distinguished by the standard equality if they have a different branching structure. To obtain a more intuitive equality on partial values of type Computation A, we must define an equivalence relation that equates different computations with the same result, i.e. bisimulation. Support for rewriting with arbitrary relations can alleviate this technicality to some extent.

Second, the bind constructor shown above is not general enough for composing com-

putations with different result types. The general version of bind should have type $\forall B : \mathsf{Set}.\mathsf{Computation}\ B \to (B \to \mathsf{Computation}\ A) \to \mathsf{Computation}\ A$. It can indeed be defined like this, but the proofs to establish the eval function require McBride's (2002) JMEq axiom to handle type equalities (Megacz 2007). However, in the Coq library implementing this technique, this issue is hidden and the users do not need to care about it. Due to universe constraints, it is still not possible to nest computations though: $\mathsf{Computation}\ A$ lives in a higher level than A.

Additional Comments The coinductive model of partiality was first proposed by Capretta (2005). The presentation above is based on a refined version due to Megacz (2007). The main difference is that instead of the bind constructor, Capretta uses a constructor $\mathsf{step} : \mathsf{Computation}\ A \to \mathsf{Computation}\ A$ to represent the idea of a delayed computation. This looks simpler at first, but makes it much harder to express branching computations naturally. Capretta proves that all recursive functions can be represented in this way, but they often require a rather unnatural manipulation. In order to simplify definitions in Capretta's model, Bove and Capretta (2007) suggest an approach that introduces an intermediate coinductive type characterising the computation structure which is—unlike the monad above—specific to the individual function. This intermediate tree-shaped structure is called a *prophecy* and it can be seen as the dual to the inductive domain predicate. In a second step, this structure is transformed into the linear structure of Capretta's computation type. In the model by Megacz, which was developed independently, this detour is not necessary.

5.4.3. *Discussion and Comparison* If we compare these monadic approaches with the techniques described previously, the characteristic differences can be described as follows.

First, partial functions from A to B all belong to a common type $A \to \mathsf{Partial}\ B$. This is a desirable feature, since it allows reasoning about partial functions in general, not just specific ones, and it simplifies the definition of a function that takes an arbitrary partial function as argument. Compare this with the technique based on inductive domains in type theory, where a partial function f from A to B has the special-purpose type $\forall a : A.\, \mathsf{Dom_f}\ a \to B$.

Nested recursion does not pose any particular difficulties in this setting. Rewriting the definition in monadic style disentangles the nested calls, and reasoning about termination is separate from the function definition.

To accommodate higher-order recursion, the respective higher-order functions have to be rewritten to take partial functions as arguments and become partial themselves. For the mirror function, we would use a function $\mathsf{map}' : (A \to \mathsf{Partial}\ B) \to [A] \to \mathsf{Partial}\ [B]$. In the domain-theoretic model, we must then prove that map' is continuous. In the coinductive model, we must make sure that the map' is defined by corecursion, and not by recursion, to satisfy the syntactic guardedness check when defining the function mirror.

In general, there seems to be a natural trade-off between argument-based and result-based partiality. In the first case, applying the function is often harder since there is the need to provide some extra argument, but the result is directly a value. In the second case, the function can directly be applied to any argument, but it is harder to use because

the result is not directly a value. In both cases, a total function can be constructed when a termination proof is available. Note that domain theory and dependent types techniques are more comprehensive in the sense that they accomodate both argument and result partiality.

5.5. *Other Classical Techniques*

This section discusses definitional techniques in classical logics that do not exactly fit into any of the other categories.

5.5.1. *Optimal Fixpoints* We have already seen that recursive function definitions can be viewed as finding the solution to a fixpoint equation. In the case of partial functions, we are looking for a function f that satisfies the conditional fixpoint equation, where D is the domain and F is the functional that arises from the recursive equations.

$$\forall x.\, D\ x \to f\ x = F\ f\ x$$

When given just the functional F, we can ask what is the right domain for the function. Intuitively, we want the domain to be as large as possible, but without overspecifying the function with values that do not follow from the definition. Inductive domain predicates give a somewhat satisfactory answer to this question, but their support for higher-order recursion is limited.

Charguéraud (2010) gives a different answer to this question by digging out (and formalising) an old theory of optimal fixpoints by Manna and Shamir (1976), which studies the various partial solutions to fixpoint equations. A partial function is regarded as a total function $f : A \to B$ and a domain $D : A \to \mathsf{Prop}$, where the values of f outside the domain are considered meaningless. Such a pair (f, D) is considered a *partial fixpoint* of a functional $F : (A \to B) \to A \to B$ if the implication above holds. Manna and Shamir prove that any functional admits a so-called *optimal fixpoint*, which is the fixpoint with the largest domain that is still consistent with any other fixpoint. Two functions are consistent if they agree on the intersection of their domains.

By exploiting this general result, Charguéraud defines a fixpoint combinator $\mathsf{Fix} : ((A \to B) \to A \to B) \to A \to B$ which takes only the functional, that is, it does not require any other input like a well-founded relation. The definition of Fix, which we omit, relies on the ε operator and therefore makes this technique inherently classical.

A number of definition principles are then derived from this generic combinator, including a variant of well-founded recursion. But the real strength of the optimal fixpoint combinator only comes into play when we also consider corecursive function. Optimal fixpoints allow the uniform treatment of corecursive as well as mixed recursive-corecursive definitions, generalising and unifying earlier techniques by Matthews (1999) and Di Gianantonio and Miculan (2003). However, corecursive functions are beyond the scope of this paper.

Regarding recursion, the main selling point for optimal fixpoints is that no relation or domain is needed at definition time. Thus, the definition can be made rather mechanically, since the function directly arises from the functional. This property is shared by

the techniques described in §5.3, but optimal fixpoints also fully support higher-order recursion. However, even though a definition using Fix can be made easily, it does not help much when reasoning about the function. Even to derive the recursive equations, either we must specify the domain and prove termination, or we take the optimal domain that arises from the fixpoint theorem. However, that domain lacks the convenient induction rule associated to inductive domain predicates.

5.5.2. *Tail-Recursive Functions via Classical Reasoning* Many techniques discussed so far rely on a termination proof in some form, appealing to the general idea that is safe to add a terminating function to the theory.

However, the converse is not true, at least in a classical setting, where some non-terminating functions can be defined without any of the inconvenient modifications that characterised the previous techniques for partial functions (e.g., modifying the type of the function or constraining the recursive equations). An interesting class of such functions are the *tail-recursive* functions. Informally, a function is called tail-recursive if no recursive call is the argument of some other function call; the only constructs that may surround a recursive call are if- and case-expressions.

We demonstrate this technique by defining the so-called *while combinator*, which models the behaviour of a while loop. Apart from being a nice example by itself, this function is important because any tail-recursive function can be expressed as an instance of it. Its definition in a functional language would be the following:

$$\begin{aligned}&\texttt{while} : (A \to \mathsf{Bool}) \to (A \to A) \to A \to A\\&\texttt{while}\ P\ f\ x = \texttt{if}\ P\ x\ \texttt{then while}\ P\ f\ (f\ x)\ \texttt{else}\ x\end{aligned}$$

First we define the auxiliary function least, which returns the least natural number that satisfies a predicate, or an arbitrary number if the predicate is never true.

$$\begin{aligned}&\mathsf{least} : (\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Nat}\\&\mathsf{least}\ P = \varepsilon\, n.\, (P\ n \wedge (\forall m.m < n \to \neg P\ m))\end{aligned}$$

The while combinator is now defined as follows.

$$\begin{aligned}\mathsf{while}\ P\ f\ x = \mathsf{if}\ &(\exists n.\, \neg P\ (\mathsf{iter}\ f\ n\ x))\\&\mathsf{then\ iter}\ f\ (\mathsf{least}\ (\lambda n.\, \neg P\ (\mathsf{iter}\ f\ n\ x)))\ x\\&\mathsf{else\ arbitrary}\end{aligned}$$

where $\mathsf{iter}\ f\ 0\ x = x$ and $\mathsf{iter}\ f\ (\mathsf{S}\ n)\ x = \mathsf{iter}\ f\ n\ (f\ x)$. Note that the condition of the if-expression checks whether the while loop terminates. If it does not, the function returns the fixed value arbitrary.

From the rather non-computational definition above, we can now derive the intended recursive equation.

$$\mathsf{while}\ P\ f\ x = \mathsf{if}\ P\ x\ \mathsf{then\ while}\ P\ f\ (f\ x)\ \mathsf{else}\ x$$

The proof of this equality simply consists of unfolding the definition on both sides, performing case distinctions along the conditionals, and checking that both sides are equal in each case.

It is straightforward to express other tail-recursive functions in terms of while and to derive their recursive equations.

The advantage of tail-recursive functions is that we can always define them easily without worrying about termination, no proof obligations must be solved, the unmodified recursive equations hold unconditionally, and it can be used for evaluating the function and for code generation.

But not everything is as good as it looks: the functional induction rule commonly used for total functions cannot be proved for a function defined with this approach, and thus we lack a fundamental reasoning principle. To prove inductive properties about such a function, we must resort to other means, depending on the application. If all else fails, we must define an inductive domain predicate and use the induction principle that arises from that definition.

As an alternative to the construction above, tail-recursive functions can also be defined using a variant of the domain-theoretic technique (§5.4.1). We can define a cpo over any type by defining $x \sqsubseteq y$ as $x = \mathsf{arbitrary} \vee x = y$. This is a flat cpo similar to the Option cpo used previously, except that the bottom element is some arbitrary element from the type, instead of one added to it. It turns out that the functional associated with a tail-recursive function is always continuous with respect to this cpo and we can take a domain-theoretic fixpoint to define the function.

Additional Comments The observation that tail-recursive functions can be defined without a termination proof was first made by Manolios and Moore (2003), who provided a definitional tool for ACL2 that constructs a total model for any tail-recursive equation. They also pointed out that tail-recursive functions arise naturally in the definition of interpreter functions for microprocessors or programming languages. These functions, which define the main execution loop, are inherently partial, and the use of tail-recursion allows a simple treatment for them.

In HOL systems, the while combinator is typically part of the library and it can be instantiated by hand. Based on the above observation that the combinator is just an instance of the domain-theoretic construction, the tool for that technique (Krauss 2010b) can also be used to define tail-recursive functions directly in Isabelle/HOL, without the need to instantiate a combinator.

6. Tool Support

In this section we present tools implemented on top of Agda, Coq and Isabelle/HOL which simplify the formalisation of partial and general recursive functions in the respective system. When appropriate, we also present concrete examples that illustrate the tool. In order to improve readability, we have slightly adapted the syntax in some of the examples, thus it might not longer correspond to the exact syntax used by the system in question.

From the tools described below, the Function command in Coq and the function package in Isabelle/HOL are the ones that are most used among users. Next follows both the Program and the Equation package in Coq. Last comes Agda's tool agda2atp, which is also the youngest tool among the ones we present here.

6.1. *The* agda2atp *Tool in Agda*

By working in first order theories of functional programs and using Agda as a logical framework—that is, as a system which is used as a basis for the implementation of a range of special purpose logics—we can reason about mainstream general recursive functional programs. Our logic is a first order theory of combinators based on Aczel's theory of combinatory formal arithmetics (Aczel 1977b).

The agda2atp (Bove et al. 2012) tool combines interactive and automatic reasoning about programs in functional languages with higher order functions, general recursion (including nested recursion) and lazy evaluation. Agda provides support for interactive reasoning by encoding first order theories using the formulae-as-types principle. Further support is provided by off-the-shelf automatic theorem provers (ATPs) for first order logic which can be called by a program which translates Agda representations of first order formulae into the TPTP language (Sutcliffe 2009) understood by the provers.

Here, we work with an untyped language, and we postulate a domain of individuals and its elements. Using Agda inductive types, we define totality predicates over this domain specifying when an element is a total natural number, a total Boolean value, etc. We define the functions we want to reason about with equations which are then sent to the ATPs via so called pragmas. For the mirror example this would look as follows:

$$\begin{array}{l}\mathsf{postulate\ mirror\_eq} : \forall a\ ts \to \mathsf{mirror} \cdot (\mathsf{tree}\ a\ ts) \equiv \mathsf{tree}\ a\ (\mathsf{rev}(\mathsf{map\ mirror}\ ts)) \\ \{-\#\ \mathsf{ATP\ axiom\ mirror\_eq}\ \#-\}\end{array}$$

where $\equiv$ is the predicate representing equality and $\cdot$ is the application of terms in the domain of quantification.

When proving properties we combine inductive and automatic reasoning: to prove that

$$\forall t \to \mathsf{Tree}\ t \to \mathsf{mirror} \cdot (\mathsf{mirror}\ t) \equiv t$$

we use Agda to perform induction on the (inductively defined) predicate $\mathsf{Tree}\ t$ stating that t is a total tree, and we call the ATPs to perform certain parts of the proof

$$\begin{array}{l}\mathsf{postulate\ prf} : \mathsf{mirror} \cdot (\mathsf{mirror} \cdot \mathsf{tree}\ a\ []) \equiv \mathsf{tree}\ a\ [] \\ \{-\#\ \mathsf{ATP\ prove\ prf}\ \#-\}\end{array}$$

The agda2atp tool send this request off to several ATPs which simultaneously try to prove the desired property, and it communicates back which of the ATPs was able to prove it or if none of them was able to prove it within a certain default time.

By working in this way we are able to reason about basically any functional program as in mainstream functional programming languages. A disadvantage is that we loose the natural reduction one has when defining functions in systems like Agda and hence, trivial steps need to be proved—although in most cases this is trivially done by the ATPs. Another disadvantage is that we now need to prove things that we usually would obtain naturally by type-checking in Agda, as for example that the addition of two total natural numbers is a total natural number.

6.2. *Coq Tools*

Coq has currently three tools that handle the definition of recursive definitions besides the built-in fix construct. All these tools are based on an explicit, definitional use of well-founded recursion, and differ mainly in the set of programs accepted and the generated support lemmas.

6.2.1. *Function* Under the command Function, Coq includes a tool that allows the definition of both structural and well-founded recursive functions. It is based on ideas by Balaa and Bertot (2000), Barthe and Courtieu (2002), and Barthe, Forest, Pichardie and Rusu (2006).

When defining a (non-structurally) recursive function $\mathsf{f} : \mathsf{Nat} \to \mathsf{Nat}$ with the Function tool, the user is asked to provide a well-founded relation (or alternatively a measure) and to show that recursive calls are performed on smaller arguments with respect to the relation (or measure). Once this is done, the tool automatically derives a handful of definitions. First, it generates a functional F representing f and then, it defines the recursive function itself by well-founded recursion, using an auxiliary definition with a richer type:

$$\mathsf{f\_terminates} : \forall\ n : \mathsf{Nat}.\, \{m : \mathsf{Nat} \mid \exists p : \mathsf{Nat}.\, \forall\ k : \mathsf{Nat}.\, p < k \to \mathsf{iter}\ k\ F\ n = m\}$$

where iter is the standard iteration function computing $(F^k\ n)$ (see its exact definition in §5.5.2). In other words, the definition of the function includes a proof that it models the functional correctly, and this proof is in turn used to prove the recursive equation $\forall n : \mathsf{Nat}.\, f\ n = F\ f\ n$ directly. Finally, the inductive graph of the function is defined, and it is shown that the function respects it. At this point, an induction principle that follows the structure of the function, and hence provides the user with the right inductive hypotheses, is derived. This principle is actually a key feature of the Function command.

Consider for example the merge function, which can be defined by well-founded recursion if we take the sum of the lengths of the input lists as a measure:

$$\begin{array}{l}\mathsf{Function\ merge}\ (a : [\mathsf{Nat}] \times [\mathsf{Nat}])\\ \quad \{\mathsf{measure}\ (\lambda p.\, \mathsf{length}\ (\pi_1\ p) + \mathsf{length}\ (\pi_2\ p))\ a\} : [\mathsf{Nat}]\\ \quad\ \mathsf{case}\ a\ \mathsf{of}\ ([], ys) \to ys\\ \qquad\qquad\ \mid (xs, []) \to xs\\ \qquad\qquad\ \mid (x :: xs, y :: ys) \to \mathsf{if}\ x < y\ \mathsf{then}\ x :: \mathsf{merge}\ (xs, y :: ys)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathsf{else}\ y :: \mathsf{merge}\ (x :: xs, ys)\end{array}$$

The user is now asked to prove that the two recursive calls are performed on arguments that are smaller than the input argument with respect to the measure. Once the termination conditions are proved, this definition is accepted. It is then possible to reason about merge by using the induction principle provided by the tool. This principle is such that, to prove that $P\ (\mathsf{merge}\ (x :: xs, y :: ys))$ for $P : [\mathsf{Nat}] \to \mathsf{Prop}$, we know either that $x < y$ and $P\ (\mathsf{merge}\ (xs, y :: ys))$, or that $x \geqslant y$ and $P\ (\mathsf{merge}\ (x :: xs, ys))$.

This tool has however a few limitations: it is unable to handle strictly partial functions, nested or higher-order recursion, nor mutual recursion on measures. Moreover, it requires the programs to follow a restricted grammar to be able to produce the graph of the

function. In particular, it does not handle dependent pattern-matching definitions in a satisfactory way.

6.2.2. *Program* Another useful tool in Coq is the Program package developed by Sozeau (2006). Like Function, it facilitates the definition of well-founded recursive functions so that the user does not need to manipulate the wfr combinator directly.

The salient feature of Program is the ability to write programs in an extended type system that includes subtyping between subset types, close to the PVS type system. This permits to write only the computational parts of a program while giving it a strong specification. An interpretation process then decorates the code to produce a complete term that complies with the Coq typechecker. This facilitates the construction of nested recursive functions as shown in the following implementation of McCarthy's 91-function:

$$\begin{array}{l}\mathsf{Program\ Fixpoint\ f91}\ (n : \mathsf{Nat})\{\mathsf{measure}\ (101 - n)\} : \\ \quad \{m : \mathsf{Nat} \mid \mathsf{if}\ n > 100\ \mathsf{then}\ m = n - 10\ \mathsf{else}\ m = 91\} = \\ \quad\quad \mathsf{if}\ n > 100\ \mathsf{then}\ n - 10\ \mathsf{else\ f91}\ (\mathsf{f91}\ (n + 11))\end{array}$$

Note that the result type is a subset type with the specification of the function to be defined, as described in §5.2.2.

After solving the generated obligations showing that recursive calls are valid, that is, they are performed in smaller arguments with respect to the measure, and that the specification is met, that is, both branches of the if-expression satisfy the specification given by the type, the function f91 is defined.

Contrary to Function, Program allows the measure to be defined on any subset of the arguments of the function. In addition, nested and higher-order recursion can be formalised by using dependent types. Like Function, Program does not support the definition of mutually recursive functions using a well-founded relation or measure, although this can be encoded by adding an extra argument to the function definition. In Program, the user can also provide a well-founded relation instead of a measure. On the other hand, this tool does not provide any extra lemmas besides the definition of f91 itself. In particular, it does not provide any induction principle which could be used to prove properties about the function, because Program generates decorated terms that are quite different from the orignal ones which represent the computation.

6.2.3. *Equations* The Equations package (Sozeau 2010) extends the support of the Coq system to perform dependent pattern-matching and well-founded recursion on inductive families. Moreover, it provides auxiliary lemmas such as the recursive equations of the function and an induction principle, much like Function. Equation automates the definition of the subterm relation for inductive definitions. Using this relation for well-founded definitions mimics the syntactic criterion, but allows the user to provide a proof of the subterm relation manually if necessary. The tool also handles dependent pattern matching fully, based on techniques from Goguen et al. (2006), so it can be used with all the techniques based on enriching types as described in §5.2. Definitions made with Equations have a similar look-and-feel to definitions made with other systems that also handle pattern matching fully, as for example Agda and especially Epigram (McBride

and McKinna 2004, McBride 2004). In addition, arbitrary well-founded orderings can be used.

As a simple example, consider the following definition of the initial part of a non-empty vector.

```
Equations init A n (v : Vec A (S n)) : Vec A n :=
init A n v by rec v :=
  init A 0 (cons a nil) := nil
  init A (S n) (cons a v) := cons a (init v)
```

The definition is done by well-founded recursion on v using the subterm relation, as specified by the first clause of the program. The proof obligation generated by the tool for the recursive call of the example above requires to show that the vector v is a subterm of cons a v, which is trivially true—and automatically proved by the tool—by definition of the subterm relation. (Note that the relation compares vectors with potentially different indexes, here n and S n.) Once the program is accepted by Coq, the tool generates the inductive graph of the function and derives an induction principle for the definition.

6.3. *The Function Package in Isabelle/HOL*

In Isabelle/HOL, support for general recursion is provided by the so-called *function package* (Krauss 2006, Krauss 2010a). It is based on inductive domain predicates as described in §5.3.2.2, but for total functions, the internal construction is mostly hidden from the user. In particular, it includes an automated termination prover (Bulwahn et al. 2007, Krauss 2007) that often solves termination proof obligations fully automatically, obviating the need for specifying termination relations in these cases. For example, quicksort is defined conveniently by just stating its recursive equations, which the package then derives as theorems. Similarly, functions like merge and mirror are handled fully automatically.

For functions requiring a manual termination proof, users have the choice of either proving it in terms of the domain predicate (i.e., $\forall x.\, \mathsf{Dom_f}\ x$), or specifying a well-founded relation or measure and proving that the arguments of the recursive calls are decreasing—which they typically prefer.

Nested functions are first introduced as partial functions. Then the tool automatically adds the domain predicate as a condition to the recursive equations.

```
function f91 : Nat → Nat where
    f91 n = if n > 100 then n − 10 else f91 (f91 (n + 11))
```

At this point, the user can already prove inductive properties about the function, constrained by the domain predicate. One such property is the lemma f91_estimate below. Its proof uses the constrained recursive equations f91.psimps and the constrained induction rule f91.pinduct which were produced by the tool. Afterwards, the lemma is used in the termination proof, which uses the same measure function we have already seen

previously.

```
lemma f91_estimate : Dom_f91 n → f91 n = if n > 100 then n − 10 else 91
    by (induct rule : f91.pinduct) (auto simp : f91.psimps)

termination f91
    by (relation (measure (λn. 101 − n))) (auto simp : f91_estimate)
```

After the termination command, the unconstrained recursive equations and induction rule are available under the names f91.simps and f91.induct, respectively. The reader may want to compare the steps above with the steps described in §5.3.2.2.

The main limitation of the function package is inherent in the technique it uses: if the function is strictly partial, its recursive equations must always carry the domain condition. Unfortunately, Isabelle's code generator cannot handle conditional equations, a limitation which is quite fundamental. Thus, we cannot use the code generator for such functions.

7. Conclusions

After going through a colourful variety of different techniques and tools, it seems clear that despite all progress that has been made during the last decades, dealing with partiality and general recursion in proof assistants is not a solved problem and the choice of which approach to use for formalising a particular problematic function will depend on many different factors! While the formalisation of simple general recursive functions is now a fairly routine task (e.g., for an Isabelle/HOL user, defining quicksort is no harder than defining structurally recursive functions), the difficulties start as soon as the termination proof becomes harder, or once higher-order and nested recursion come into play. Although some techniques can handle such more complicated definitions in principle, this typically requires a good deal of extra (manual) effort.

When we look at which techniques made the biggest impact in practice, we must clearly say that implementation plays an important role, that is, either the approach has been implemented as part of a major system (as for example more flexible termination checkers) or as add-on tools. An extension of some logic with a new type of partial functions as discussed in §4.4 will—no matter how well it is designed— have little impact unless there is a system that really implements the extended calculus. The inductive domain technique in constructive type theory (§5.3.2.1) seems to be an interesting exception here, as it can easily be applied even with no special automation in place. Otherwise, the state of implementation usually lags behind the conceptual state of the art by a few years, and so it will be interesting to see if, e.g., sized types (§4.3) will find their way into a mature system in the future. In this respect, definitional techniques have a slight strategic advantage, since they can be implemented on top of an unchanged system, which is usually easier. On the other hand, axiomatic techniques tend to produce less overhead in the end.

We conclude this paper by showing the formalisation of a somewhat tricky function that was mentioned as a challenge problem by Owens and Slind (2008). The following

algorithm matches a string against a regular expression, which is modelled by the obvious inductive type, with constructors $\emptyset$, ϵ, $\mathtt{char}$, $|$, $\cdot$, and *:

$$\begin{aligned}
&\mathtt{match}\ \emptyset\ s\ k = \mathtt{false}\\
&\mathtt{match}\ \epsilon\ s\ k = k\ s\\
&\mathtt{match}\ (\mathtt{char}\ d)\ []\ k = \mathtt{false}\\
&\mathtt{match}\ (\mathtt{char}\ d)\ (c :: s)\ k = \mathtt{if}\ c = d\ \mathtt{then}\ k\ s\ \mathtt{else}\ \mathtt{false}\\
&\mathtt{match}\ (r_1 \mid r_2)\ s\ k = \mathtt{match}\ r_1\ s\ k\ \vee \mathtt{match}\ r_2\ s\ k\\
&\mathtt{match}\ (r_1 \cdot r_2)\ s\ k = \mathtt{match}\ r_1 s\ (\lambda s'.\mathtt{match}\ r_2\ s'\ k)\\
&\mathtt{match}\ (r^*)\ s\ k = k\ s \vee \mathtt{match}\ r\ s\ (\lambda s'.\mathtt{match}\ (r^*)\ s'\ k)
\end{aligned}$$

Note the use of the continuation k, which is applied to the remaining string whenever a prefix of the string matches the given expression. In particular, in the last two cases, the continuation given to a recursive call contains another recursive call. Moreover, the function is partial, as it terminates only on expressions in normal form. An expression is in normal form if its subexpressions occuring under a star are not nullable, that is, they do not match the empty string.

This function posed problems with the approach used by Owens and Slind, which is based on well-founded recursion in HOL (§5.1.2). However, the definition can be handled elegantly using dependent types. Then, the type of the continuation is restricted to

$$\forall s' : [A].(\mathsf{if}\ \mathsf{nullable}\ r\ \mathsf{then}\ \mathsf{length}\ s' \leqslant \mathsf{length}\ s\ \mathsf{else}\ \mathsf{length}\ s' < \mathsf{length}\ s) \to \mathsf{Bool}$$

where r and s are the first and second argument of match. Thus, the continuation can expect a string s' that is not longer than the initial string s. Moreover, when the expression r is not nullable, then s' is strictly shorter than s. The match function can then be defined, after restricting its domain to expressions in normal form. The definition uses a lexicographic combination of the size of the regular expression and the length of the string. The Program (§6.2.2) tool nicely takes care of passing around the proof arguments, and it presents the necessary proof obligations to the user, which are easily solved.

However, there seems to be no easy way to achieve the same effect without the use of dependent types. Finding an equally elegant definition in HOL remains a challenge for the moment.

Acknowledgements. We thank Arthur Charguéraud for helpful explanations on optimal fixpoints.

References

Abel, A. (1998). foetus – termination checker for simple functional programs. Programming Lab Report. http://www.tcs.informatik.uni-muenchen.de/~abel/foetus/.

Abel, A. (2006). *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*, PhD thesis, Ludwig-Maximilians-Universität München.

Abel, A. (2008). Semi-continuous sized types and termination, *Logical Methods in Computer Science* **4**(2). CSL'06 special issue.

Abel, A. (2010). MiniAgda: Integrating sized and dependent types, *in* Bove et al. (2010), pp. 14–28.

Abel, A. and Altenkirch, T. (2002). A predicative analysis of structural recursion, *Journal of Functional Programming* **12**: 1–41.

Abramsky, S. and Jung, A. (1994). Domain theory, *in* S. Abramsky, D. M. Gabbay and T. S. E. Maibaum (eds), *Handbook of Logic in Computer Science*, Vol. 3, Oxford University Press, pp. 1–168.

Aczel, P. (1977a). An Introduction to Inductive Definitions, *in* J. Barwise (ed.), *Handbook of Mathematical Logic*, North-Holland Publishing Company, pp. 739–782.

Aczel, P. (1977b). The strength of Martin-Löf's intuitionistic type theory with one universe, *in* S. Miettinen and J. Vnanen (eds), *Proc. of the Symposium on Mathematical Logic (Oulu, 1974)*, Report No. 2, Department of Philosopy, University of Helsinki, Helsinki, pp. 1–32.

Agda (2008). Agda wiki. http://wiki.portal.chalmers.se/agda/agda.php.

Ait Mohamed, O., Muñoz, C. and Tahar, S. (eds) (2008). *Theorem Proving in Higher Order Logics (TPHOLs 2008), 21st International Conference, Montreal, Canada, August 18-21, 2008, Proceedings*, Vol. 5170 of *Lecture Notes in Computer Science*, Springer Verlag.

Audebaud, P. (1991). Partial objects in the calculus of constructions, *in* G. Kahn (ed.), *Logic in Computer Science (LICS 1991)*, IEEE, pp. 86–95.

Balaa, A. and Bertot, Y. (2000). Fix-point equations for well-founded recursion in type theory, *in* M. Aagaard and J. Harrison (eds), *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, Vol. 1869 of *Lecture Notes in Computer Science*, Springer Verlag.

Balaa, A. and Bertot, Y. (2002). Fonctions récursives générales par itération en théorie des types, *Journées Francophones des Langages Applicatifs - JFLA02, INRIA* .

Barringer, H., Cheng, J. H. and Jones, C. B. (1984). A logic covering undefinedness in program proofs, *Acta Informatica* **21**: 251–269.

Barthe, G. and Courtieu, P. (2002). Efficient reasoning about executable specifications in Coq, *in* V. A. Carreno, C. Muñoz and S. Tahar (eds), *Theorem Proving in Higher-Order Logics (TPHOLs 2002)*, Vol. 2410 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 31–46.

Barthe, G., Forest, J., Pichardie, D. and Rusu, V. (2006). Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant, *in* M. Hagiya and P. Wadler (eds), *Functional and Logic Programming (FLOPS 2006)*, Vol. 3945 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 114 – 129.

Barthe, G., Frade, M. J., Giménez, E., Pinto, L. and Uustalu, T. (2004). Type-based termination of recursive definitions, *Mathematical Structures in Computer Science* **14**(1): 97–141.

Barthe, G., Grégoire, B. and Pastawski, F. (2006). CIC$\widehat{}$: Type-based termination of recursive definitions in the calculus of inductive constructions, *in* M. Hermann and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, Vol. 4246 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 257–271.

Barthe, G., Grégoire, B. and Riba, C. (2008). Type-based termination with sized products, *in* M. Kaminski and S. Martini (eds), *CSL*, Vol. 5213 of *Lecture Notes in Computer Science*, Springer, pp. 493–507.

Barthe, G., Grégoire, B. and Riba, C. (2009). A tutorial on type-based termination, *in* A. Bove, L. S. Barbosa, A. Pardo and J. S. Pinto (eds), *Language Engineering and Rigorous Software Development*, Vol. 5520 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 100–152.

Berghofer, S. and Nipkow, T. (2000). Executing higher order logic., *in* Callaghan et al. (2002), pp. 24–40.

Berghofer, S. and Wenzel, M. (1999). Inductive datatypes in HOL - lessons learned in formal-logic engineering, *in* Bertot et al. (1999), pp. 19–36.

Bertot, Y., Capretta, V. and Das Barman, K. (2002). Type-theoretic functional semantics, *in* V. A. Carreno, C. A. Muñoz and S. Tahar (eds), *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, Vol. 2410 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 83–97.

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Springer Verlag.

Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C. and Théry, L. (eds) (1999). *Theorem Proving in Higher Order Logics (TPHOLs '99)*, Vol. 1690 of *Lecture Notes in Computer Science*, Springer Verlag.

Bertot, Y. and Komendantsky, V. (2008). Fixed point semantics and partial recursion in Coq, *in* S. Antoy and E. Albert (eds), *Principles and practice of declarative programming (PPDP '08)*, ACM, New York, NY, USA, pp. 89–96.

Blanqui, F. (2004). A type-based termination criterion for dependently-typed higher-order rewrite systems, *in* V. van Oostrom (ed.), *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings*, Vol. 3091 of *LNCS*, Springer, pp. 24–39.

Blanqui, F. (2005). Decidability of type-checking in the calculus of algebraic constructions with size annotations, *in* L. Ong (ed.), *Computer Science Logic (CSL 2005)*, Vol. 3634 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 135–150.

Bove, A. (2009). Another look at function domains, *in* S. Abramsky, M. Mislove and C. Palamidessi (eds), *Mathematical Foundations of Programming Semantics (MFPS 2009)*, Vol. 249C of *ENTCS*, pp. 61–74.

Bove, A. and Capretta, V. (2005a). Modelling general recursion in type theory, *Mathematical Structures in Computer Science* **15**(4): 671–708.

Bove, A. and Capretta, V. (2005b). Recursive functions with higher-order domains, *in* P. Urzyczyn (ed.), *Typed Lambda Calculi and Applications (TLCA 2005)*, Vol. 3461 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 116–130.

Bove, A. and Capretta, V. (2007). Computation by prophecy, *in* S. R. D. Rocca (ed.), *Typed Lambda Calculi and Applications (TLCA 2007)*, Vol. 4583 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 70–83.

Bove, A. and Capretta, V. (2008). A type of partial recursive functions, *in* Ait Mohamed et al. (2008), pp. 102–117.

Bove, A., Dybjer, P. and Sicard-Ramírez, A. (2012). Combining interactive and automatic reasoning in first order theories of functional programs, *in* L. Birkedal (ed.), *15th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS 2012*, Vol. 7213 of *LNCS*.

Bove, A., Komendantskaya, E. and Niqui, M. (eds) (2010). *Workshop on Partiality and Recursion in Interative Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*, Vol. 43 of *EPTCS*.

Boyer, R. S. and Moore, J. S. (1979). *A Computational Logic*, Academic Press, New York.

Boyer, R. S. and Moore, J. S. (1996). Mechanized formal reasoning about programs and computing machines, *in* R. Veroff (ed.), *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, MIT Press.

Bulwahn, L., Krauss, A. and Nipkow, T. (2007). Finding lexicographic orders for termination proofs in Isabelle/HOL, *in* K. Schneider and J. Brandt (eds), *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, Vol. 4732 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 38–53.

Callaghan, P., Luo, Z., McKinna, J. and Pollack, R. (eds) (2002). *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, Vol. 2277 of *Lecture Notes in Computer Science*, Springer Verlag.

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

Charguéraud, A. (2010). The optimal fixed point combinator, *in* Kaufmann and Paulson (2010), pp. 195–210.

Cheng, J. H. and Jones, C. B. (1991). On the usability of logics which handle partial functions, *in* C. Morgan and J. C. P. Woodcock (eds), *3rd Refinement Workshop*, Springer Verlag, pp. 51–69.

Constable, R. L. and Mendler, N. P. (1985). Recursive definitions in type theory, *in* R. Parikh (ed.), *Logic of Programs*, Vol. 193 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 61–78.

Constable, R. L. and Smith, S. F. (1987). Partial objects in constructive type theory, *Logic in Computer Science (LICS 1987)*, IEEE, Washington, D.C., pp. 183–193.

Coq development team (2010). *Coq 8.3 Reference Manual*, INRIA. http://coq.inria.fr/refman/.

Coquand, T. and Huet, G. (1988). The Calculus of Constructions, *Information and Computation* **76**(2/3): 95–120.

Coquand, T. and Paulin, C. (1990). Inductively defined types, *in* P. Martin-Löf and G. Mints (eds), *Proceedings of Colog '88*, Vol. 417 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 50–66.

Di Gianantonio, P. and Miculan, M. (2003). A unifying approach to recursive and co-recursive definitions, *in* H. Geuvers and F. Wiedijk (eds), *Types for Proofs and Programs (TYPES 2002)*, Vol. 2646 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 148–161.

Dubois, C. and Donzeau-Gouge, V. V. (1998). A step towards the mechanization of partial functions: domains as inductive predicates, *CADE-15 Workshop on mechanization of partial functions*.

Dybjer, P. (2000). A general formulation of simultaneous inductive-recursive definitions in type theory, *Journal of Symbolic Logic* **65**(2): 525–549.

Farmer, W. M. (1993). A simple type theory with partial functions and subtypes, *Annals of Pure and Applied Logic* **64**(3): 211–240.

Farmer, W. M., Guttman, J. D. and Thayer, F. J. (1993). IMPS: an interactive mathematical proof system, *Journal of Automated Reasoning* **11**: 653–654.

Finn, S., Fourman, M. and Longley, J. (1997). Partial functions in a total setting, *Journal of Automated Reasoning* **18**(1): 85–104.

Furbach, U. and Shankar, N. (eds) (2006). *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, Vol. 4130 of *Lecture Notes in Artificial Intelligence*, Springer Verlag.

Giesl, J. (1997). Termination of nested and mutually recursive algorithms, *Journal of Automated Reasoning* **19**(1): 1–29.

Giesl, J. (2001). Induction proofs with partial functions, *Journal of Automated Reasoning* **26**(1): 1–49.

Giménez, E. (1995). Codifying guarded definitions with recursive schemes, *Types for Proofs and Programs (TYPES 1994)*, Vol. 996 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 39–59.

Goguen, H., McBride, C. and McKinna, J. (2006). Eliminating Dependent Pattern Matching, *Essays Dedicated to Joseph A. Goguen*, pp. 521–540.
URL: *http://www.cs.st-andrews.ac.uk/ james/RESEARCH/pattern-elimination-final.pdf*

Gordon, M. J. C. and Melham, T. F. (eds) (1993). *Introduction to HOL: A theorem proving environment for Higher Order Logic*, Cambridge University Press.

Gordon, M. J. C., Milner, R. and Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, Vol. 78 of *Lecture Notes in Computer Science*, Springer Verlag.

Greve, D. (2009). Assuming termination, *ACL2 Workshop Proceedings*.

Haftmann, F. and Nipkow, T. (2010). Code generation via higher-order rewrite systems, *in* M. Blume, N. Kobayashi and G. Vidal (eds), *Functional and Logic Programming (FLOPS 2010)*, Vol. 6009 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 103–117.

Harrison, J. (1995). Inductive definitions: automation and application, *in* Schubert et al. (1995), pp. 200–213.

Howard, W. A. (1980). The formulae-as-types notion of construction, *in* J. P. Seldin and J. R. Hindley (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London, pp. 479–490.

Huffman, B. (2008). Reasoning with powerdomains in Isabelle/HOLCF, *in* O. Ait Mohamed, C. Muñoz and S. Tahar (eds), *TPHOLs 2008: Emerging Trends Proceedings*, pp. 45 –56. Department of Electrical and Computer Engineering, Concordia University.

Huffman, B. (2009). A purely definitional universal domain, *in* S. Berghofer, T. Nipkow, C. Urban and M. Wenzel (eds), *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, Vol. 5674 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 260–275.

Hughes, J., Pareto, L. and Sabry, A. (1996). Proving the correctness of reactive systems using sized types, *Principles of Programming Languages (POPL 1996)*, ACM, pp. 410–423.

Jones, C. B. (1990). *Systematic Software Development using VDM*, Prentice-Hall.

Kaufmann, M. and Paulson, L. C. (eds) (2010). *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010, Proceedings*, Vol. 6172 of *Lecture Notes in Computer Science*, Springer Verlag.

Krauss, A. (2006). Partial recursive functions in higher-order logic, *in* Furbach and Shankar (2006), pp. 589–603.

Krauss, A. (2007). Certified size-change termination, *in* F. Pfenning (ed.), *Automated Deduction (CADE-21)*, Vol. 4603 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 460–476.

Krauss, A. (2010a). Partial and nested recursive function definitions in higher-order logic, *Journal of Automated Reasoning* **44**(4): 303–336.

Krauss, A. (2010b). Recursive definitions of monadic functions, *in* Bove et al. (2010), pp. 1–13.

Krstić, S. and Matthews, J. (2003). Inductive invariants for nested recursion, *in* D. A. Basin and B. Wolff (eds), *Theorem Proving in Higher Order Logics (TPHOLs 2003)*, Vol. 2758 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 253–269.

Leroy, X. (2006). Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, *in* J. G. Morrisett and S. L. Peyton Jones (eds), *Principles of Programming Languages (POPL 2006)*, ACM Press, pp. 42–54.

Manna, Z. and Shamir, A. (1976). The theoretical aspects of the optimal fixed point, *SIAM J. Comput.* **5**(3): 414–426.

Manolios, P. and Moore, J. S. (2003). Partial functions in ACL2, *Journal of Automated Reasoning* **31**(2): 107–127.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*, Bibliopolis, Napoli.

Matthews, J. (1999). Recursive function definition over coinductive types, *in* Bertot et al. (1999), pp. 73–90.

McBride, C. (2002). Elimination with a motive, *in* Callaghan et al. (2002), pp. 197–216.

McBride, C. (2003). First-order unification by structural recursion, *Journal of Functional Programming* **13**(6): 1061–1075.

McBride, C. (2004). Epigram: Practical programming with dependent types, *in* V. Vene and T. Uustalu (eds), *Advanced Functional Programming (AFP 2004)*, Vol. 3622 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 130–170.

McBride, C. and McKinna, J. (2004). The view from the left, *Journal of Functional Programming* **14**(1): 69–111.

Megacz, A. (2007). A coinductive monad for Prop-bounded recursion, *in* A. Stump and H. Xi (eds), *Programming Languages meets Program Verification (PLPV 2007)*, ACM, New York, NY, USA, pp. 11–20.

Milner, R. (1972). Logic for computable functions: description of a machine implementation., *Technical report*, Stanford, CA, USA.

Moggi, E. (1991). Notions of computation and monads, *Information and Computation* **93**(1): 55–92.

Müller, O., Nipkow, T., von Oheimb, D. and Slotosch, O. (1999). HOLCF=HOL+LCF, *Journal of Functional Programming* **9**(2): 191–223.

Nipkow, T., Bauer, G. and Schultz, P. (2006). Flyspeck I: Tame graphs, *in* Furbach and Shankar (2006), pp. 21–35.

Nipkow, T., Paulson, L. C. and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *Lecture Notes in Computer Science*, Springer Verlag.

Nordström, B. (1988). Terminating General Recursion, *BIT* **28**(3): 605–619.

Nordström, B., Petersson, K. and Smith, J. (1990). *Programming in Martin-Löf's Type Theory. An Introduction.*, Oxford University Press.

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

OCaml (1996). Ocaml web page. http://caml.inria.fr/ocaml/.

Owens, S. and Slind, K. (2008). Adapting functional programs to higher-order logic, *Higher-Order and Symbolic Computation* **21**(4): 377–409.

Paulin-Mohring, C. (1993). Inductive definitions in the system Coq - rules and properties, *Typed Lambda Calculi and Applications (TLCA 1993)*, Vol. 664 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 328–345.

Paulin-Mohring, C. (2009). A constructive denotational semantics for Kahn networks in Coq, *in* Y. Bertot, G. Huet, J.-J. Levy and G. Plotkin (eds), *From Semantics and Computer Science: Essays in Honor of Gilles Kahn*, Cambridge University Press.

Paulson, L. C. (1986). Constructing recursion operators in intuitionistic type theory, *J. Symb. Comput.* **2**(4): 325–355.

Paulson, L. C. (1989). The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5**(3): 363–397.

Peyton Jones, S. (ed.) (2003). *Haskell 98 Language and Libraries The Revised Report*, Cambridge University Press.

Regensburger, F. (1995). HOLCF: higher order logic of computable functions, *in* Schubert et al. (1995), pp. 293–307.

Schubert, E. T., Windley, P. J. and Alves-Foss, J. (eds) (1995). *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*, Vol. 971 of *Lecture Notes in Computer Science*, Springer Verlag.

Scott, D. S. (1993). A type-theoretical alternative to iswim, cuch, owhy, *Theor. Comput. Sci.* **121**: 411–440.

Setzer, A. (2006). Partial recursive functions in Martin-Löf Type Theory, *in* A. Beckmann, U. Berger, B. Löwe and J. V. Tucker (eds), *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30-July 5, 2006. Proceedings.*, LNCS, Springer, pp. 505 – 515.

Setzer, A. (2007). A data type of partial recursive functions in Martin-Löf Type Theory. 35pp, submitted.

Slind, K. (1996). Function definition in higher-order logic, *in* J. von Wright, J. Grundy and J. Harrison (eds), *Theorem Proving in Higher Order Logics (TPHOLs '96)*, Vol. 1125 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 381–397.

Sozeau, M. (2006). Subset coercions in Coq, *in* T. Altenkirch and C. McBride (eds), *TYPES*, Vol. 4502 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 237–252.

Sozeau, M. (2010). Equations: A dependent pattern matching compiler, *in* Kaufmann and Paulson (2010), pp. 419–434.

Sutcliffe, G. (2009). The TPTP problem library and associated infrastructure. The FOT and CNF parts, v.3.5.0, *Journal of Automated Reasoning* **43**(4): 337–362.

Wahlstedt, D. (2007). *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*, PhD thesis, Chalmers University of Technology.

Wenzel, M., Paulson, L. C. and Nipkow, T. (2008). The Isabelle framework, *in* Ait Mohamed et al. (2008), pp. 33–38.

Xi, H. (2001). Dependent types for program termination verification, *Logic in Computer Science (LICS 2001)*, IEEE, pp. 231–242.