A Scheme for Defining Partial Higher-Order Functions by Recursion

William M. Farmer
McMaster University
Hamilton, Ontario, Canada

wmfarmer@mcmaster.ca

17 April 2001

Abstract

This paper describes a scheme for defining partial higher-order functions as the least fixed points of monotone functionals. The scheme can be used to define both single functions by recursion and systems of functions by mutual recursion. The scheme is implemented in the IMPS Interactive Mathematical Proof System. The IMPS implementation includes an automatic syntactic check for monotonicity that succeeds for many common recursive definitions.

1 Introduction

Recursion is a powerful technique for defining functions (and other mathematical objects). It is one of the mainstays of formal methods. Defining a function by recursion can facilitate both reasoning and computation performed with the function. Constructing a recursive definition of a function requires care: a faulty definition will not define a bona fide function and may introduce inconsistencies. For example, there is no function f on **N** (the set of natural numbers) that satisfies the recursive formula

$$\forall n \,.\, f(n) = f(n) + 1,$$

and the assumption that there is such a function implies that $0 = 1$.

Various schemes for defining functions by recursion have been proposed. A definition that is admitted by a scheme is called an *instance* of the scheme. Each scheme has a set of *instance requirements* that a proposed definition must satisfy in order to be an instance of the scheme. For some schemes a definition is required only to have a certain syntactic form, while for other schemes a definition must possess certain semantic properties. A scheme is *proper* if every instance of the scheme actually defines a function. The *domain* of a scheme is the set D of functions such that $f \in D$ iff there is some instance of the scheme that defines f.

A popular proper recursive definition scheme is the scheme of *primitive recursion* (see [10]). An instance of primitive recursion is a pair of equations satisfying certain syntactic requirements. The domain of primitive recursion is a broad, but proper, subset of the computable total[1] functions on **N**. For example, the following pair of equations constitute a primitive recursive definition of the factorial function $f : \mathbf{N} \rightarrow \mathbf{N}$:

[1]The *domain of definition* of a function f is the set D_f of values at which f is defined, and the *domain of application* of f is the set D^*_f of values to which f may be applied. A function f is *total* if $D_f = D^*_f$ and *partial* if $D_f \subseteq D^*_f$. Thus a total function is a special case of a partial function.

(1) $f(0) = 1$.

(2) $f(n+1) = h(n, f(n))$ where $h(x,y) = y * (x+1)$.

There is a family of proper recursive definition schemes that are based on *well-founded recursion*. A definition of a unary function f in this kind of scheme consists of a triple $(\mathcal{T}, \varphi, \ll)$ where $\mathcal{T}$ is a theory, φ is an formula of the form

$$\forall\, x \,.\, f(x) = A(f(a_1(x)), \ldots, f(a_k(x))),$$

and $\ll$ is a well-founded relation. The definition is an instance of the scheme if each application of f in the right side of φ is always "$\ll$-simpler" than the application of f on the left side of φ, i.e., that, for each i with $1 \leq i \leq k$,

$$a_i(x) \ll x$$

holds in $\mathcal{T}$ together with the "local context" of assumptions that govern the occurrence of $f(a_i(x))$ in φ. The domain of a well-founded recursive definition scheme can be very large including the primitive recursive functions and other computable total functions as well as possibly noncomputable total functions. Normally, the domain will not contain functions that are strictly partial. For example, $(\mathcal{A}, \varphi, \ll)$ constitutes a well-founded recursive definition of the factorial function $f : \mathbf{N} \to \mathbf{N}$ where:

(1) $\mathcal{A}$ is a standard theory of real arithmetic.

(2) φ is $\forall\, n \,.\, f(n) = \mathsf{if}(n = 0, 1, f(n-1) * n)$.

(3) $\ll$ is the usual total order on $\mathbf{N}$.

Mechanized mathematics systems—interactive computer systems for supporting and improving mathematical reasoning—usually provide their users with an implemented scheme for defining functions by recursion. The designers of a mechanized mathematics system generally choose a proper scheme with easily checked instance requirements and a large domain. For example, HOL [9] implements a generalization of primitive recursion and PVS [13] implements a scheme for defining total higher-order functions by well-founded recursion. Although strictly partial functions are ubiquitous in mathematics and computer science, nearly all implemented schemes for defining functions by recursion admit only total functions.

This paper describes a proper scheme for defining *partial* (as well as total) higher-order functions by recursion. In the scheme a function is defined as the least fixed point of a *monotone functional*, and a system of functions is defined as the simultaneous least fixed point of a *system of monotone functionals*. The scheme is derived from an approach to recursion developed by Y. Moschovakis [12]. Moschovakis presents the approach in his paper [12] using an informal second-order logic that admits undefined terms and partial functions. Our scheme is presented within a formal higher-order logic called LUTINS [2, 3, 4, 8] that admits undefined terms and partial functions and that contains a definite description operator.

The scheme has been implemented and tested in the IMPS Interactive Mathematical Proof System [7, 8] which has LUTINS as its logic. IMPS is equipped with an automatic mechanism for syntactically checking whether a functional is monotone. Many common functions can be defined in IMPS by functionals on which the monotonicity check succeeds. As a result, defining functions in IMPS by recursion is usually just a matter of writing down the appropriate functional: there are rarely any side conditions that need to be proved. Although the scheme is presented within LUTINS, it will work in other logics that admit partial higher-order functions.

The rest of the paper is organized as follows. Section 2 briefly introduces LUTINS, the logic of IMPS. Section 3 states some of the key definitions concerning functionals and fixed points. The central theorem underlying the scheme, a fixed point theorem for monotone functionals, is proved in section 4. The notion of a recursive definition is defined in section 5. Section 6 presents some extensions to the basic scheme for defining functions by recursion. How the scheme is implemented in IMPS is the subject of section 7. The IMPS monotonicity check is described in section 8. The paper ends with a short conclusion in section 9 and an appendix which presents a fixed point theorem for continuous functionals.

2 LUTINS

LUTINS[2] is a nonconstructive version of simple type theory [1]. A formalization of the traditional approach to partial functions [5], it admits undefined terms and partial functions and has a definite description operator I. LUTINS is also equipped with a system of *sorts* for classifying terms by value which is an extension of the system of types. LUTINS closely corresponds to mathematics practice and has proven to be an effective logic for formalizing traditional mathematics (e.g., see [6]).

The application of a term denoting a partial function to a term that denotes an argument outside of the domain of the partial function is undefined. For example, $2/0$ and $\sqrt{-3}$ are undefined in a standard theory of real arithmetic. The application of a term denoting a partial function to an undefined term is also undefined. Undefined terms do not denote anything and are indiscernible from one another. The definite description operator I is used to construct *definite descriptions*, that is, terms of the form $(\mathrm{I}\, x \,.\, \varphi)$. A term $(\mathrm{I}\, x \,.\, \varphi)$ denotes the unique x that satisfies φ if there is such an x and is undefined otherwise.

Although terms may be nondenoting, LUTINS is a bivalent logic: formulas are either true or false. In particular, the application of a term denoting a predicate to an undefined term is always false. Most of the laws of classical simple type theory hold in LUTINS without modification. However, the laws dealing with instantiation and equality substitution are slightly different. For example, universal instantiation holds only for defined terms.

A *sort* is a syntactic object α that denotes a nonempty domain D_α of values. Types are the maximal sorts: every sort is a subtype of some type. Sorts are of either kind ι or kind $*$. A sort of the form

$$\alpha_1 \times \cdots \times \alpha_n \rightharpoonup \alpha_{n+1}$$

of kind ι where $n \geq 1$ denotes the domain of n-ary partial functions from $D_{\alpha_1} \times \cdots \times D_{\alpha_n}$ to $D_{\alpha_{n+1}}$. A sort of the form

$$\alpha_1 \times \cdots \times \alpha_n \rightarrow *$$

where $n \geq 1$ is of kind $*$ and it denotes the domain of n-ary total functions from $D_{\alpha_1} \times \cdots \times D_{\alpha_n}$ to $*$, the sort denoting the domain $\{\mathrm{T}, \mathrm{F}\}$ of truth values.

Every term is assigned a sort on the basis of its syntax. If a term t is assigned a sort α, then the value of t is a member of D_α *provided t is defined.* A formula of the form $t\!\downarrow$ asserts t is defined, and $(t \downarrow \alpha)$ asserts that t is defined in α, i.e., that t is defined with a value in D_α. Sorts are also used to restrict the binding operators of LUTINS: λ, $\forall$, $\exists$, and I.

In LUTINS, $=$ is a binary predicate that satisfies the usual axioms of equality. Like any other predicate, if $=$ is applied to an undefined term, the resulting expression is false. Hence, $2/0 = 5$ and $\sqrt{-3} = \sqrt{-3}$ are both false in a standard theory $\mathcal{A}$ of real arithmetic. An expression of the form $s \simeq t$ is an abbreviation

[2]Pronounced as the word in French.

for $(s{\downarrow} \vee t{\downarrow}) \supset s = t$, which asserts that either s and t denote the same value or s and t both denote no value. Hence, $2/0 \simeq 5$ is false and $\sqrt{-3} \simeq \sqrt{-3}$ is true in $\mathcal{A}$. Note that $\simeq$ is *not* a predicate, just part of an abbreviation.

A *theory* of LUTINS is a pair $\mathcal{T} = (\mathcal{L}, \Gamma)$, where $\mathcal{L}$ is a language of LUTINS and Γ is a set of sentences in $\mathcal{L}$ which serve as the axioms of $\mathcal{T}$.

For more information about LUTINS, see the references for LUTINS given above.

3 Preliminary Definitions

Let $\mathcal{L}$ be a language of LUTINS. An *expression* is an term or formula of $\mathcal{L}$. For the rest of the paper, let

$$\alpha = \alpha_1 \times \cdots \times \alpha_n \rightharpoonup \alpha_{n+1}$$

be a sort of kind ι where $n \geq 1$.

A *functional* of sort α is an expression of sort $\alpha \rightharpoonup \alpha$. A functional is *in canonical form* if it is a lambda-expression. Given functions g and h, g is a *subfunction* of h if the domain D_g of g is a subset of the domain of h and g equals h on D_g.

We define the following predicates in $\mathcal{L}$.

Definition 3.1 (Subfunction) $\forall\, g, h : \alpha \,.\, g \sqsubseteq_\alpha h \equiv$
$\forall\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, g(x_1, \ldots, x_n){\downarrow} \supset g(x_1, \ldots, x_n) = h(x_1, \ldots, x_n)$.

An expression $g \sqsubseteq_\alpha h$ asserts that g and h denote functions of sort α such that g is a subfunction of h.

Proposition 3.2 *$\sqsubseteq_\alpha$ is a partial order on α.*

Definition 3.3 (Monotone) $\forall\, F : \alpha \rightharpoonup \alpha \,.\, \mathsf{monotone}_\alpha(F) \equiv$
$\forall\, g, h : \alpha \,.\, g \sqsubseteq_\alpha h \supset F(g) \sqsubseteq_\alpha F(h)$.

Definition 3.4 (Fixed Point) $\forall\, f : \alpha,\ F : \alpha \rightharpoonup \alpha \,.\, \mathsf{fp}_\alpha(f, F) \equiv F(f) = f$.

Definition 3.5 (Least Fixed Point) $\forall\, f : \alpha,\ F : \alpha \rightharpoonup \alpha \,.\, \mathsf{lfp}_\alpha(f, F) \equiv$
$\mathsf{fp}_\alpha(f, F) \wedge (\forall\, g : \alpha \,.\, \mathsf{fp}_\alpha(g, F) \supset f \sqsubseteq_\alpha g)$.

Definition 3.6 (Strong Fixed Point) $\forall\, f : \alpha,\ F : \alpha \rightharpoonup \alpha \,.\, \mathsf{sfp}_\alpha(f, F) \equiv$
$\mathsf{fp}_\alpha(f, F) \wedge (\forall\, g : \alpha \,.\, F(g) \sqsubseteq_\alpha g \supset f \sqsubseteq_\alpha g)$.

Proposition 3.7 *A functional has at most one least fixed point.*

Proposition 3.8 *A strong fixed point of a functional is a least fixed point of the functional.*

4 The Fixed Point Theorem

In this section we prove that every monotone functional has a strong fixed point. We begin by showing that every monotone functional F of sort α is total, i.e., defined for every member of α.

Lemma 4.1 *The sentence*

$$\forall\, F : \alpha \rightharpoonup \alpha\,.\, \mathsf{monotone}_\alpha(F) \supset \forall\, f : \alpha\,.\, F(f)\downarrow$$

is valid in $\mathcal{L}$.

Proof Let F and f be variables of sort $\alpha \rightharpoonup \alpha$ and α, respectively. We will assume $\mathsf{monotone}_\alpha(F)$ and then derive $F(f)\downarrow$. After expanding the definition of $\mathsf{monotone}_\alpha$ and then instantiating the expanded formula with f and f, we obtain $f \sqsubseteq_\alpha f \supset F(f) \sqsubseteq_\alpha F(f)$. Since $\sqsubseteq_\alpha$ is a partial order on α (by Proposition 3.2) and f is defined in α, it follows that $F(f) \sqsubseteq_\alpha F(f)$. The latter implies $F(f)\downarrow$ since $\sqsubseteq_\alpha$ is a predicate. □

Theorem 4.2 (Fixed Point Theorem for Monotone Functionals) *The sentence*

$$\forall\, F : \alpha \rightharpoonup \alpha\,.\, \mathsf{monotone}_\alpha(F) \supset \exists\, f : \alpha\,.\, \mathsf{sfp}_\alpha(f, F)$$

is valid in $\mathcal{L}$.

Proof Fix a model $\mathcal{M}$ for $\mathcal{L}$, and let $X_\mathcal{M}$ be the denotation in $\mathcal{M}$ of an expression or sort X of $\mathcal{L}$. Let F be a functional of sort α and assume that $F_\mathcal{M}$ is monotone in $\mathcal{M}$. We must show that there is a strong fixed point of $F_\mathcal{M}$ in $\mathcal{M}$.

For a function f of sort α in $\mathcal{M}$ and an ordinal γ, define $F^\gamma_\mathcal{M}(f)$ inductively by:

(1) $F^0_\mathcal{M}(f) = f$.

(2) $F^{\gamma+1}_\mathcal{M}(f) = F_\mathcal{M}(F^\gamma_\mathcal{M}(f))$.

(3) $F^\delta_\mathcal{M}(f)$ for a limit ordinal δ is the function represented by the set of ordered pairs

$$(\langle a_1, \ldots, a_n\rangle, F^\gamma_\mathcal{M}(f)(a_1, \ldots, a_n))$$

where $\gamma < \delta$, $\langle a_1, \ldots, a_n\rangle \in (\alpha_1)_\mathcal{M} \times \cdots \times (\alpha_n)_\mathcal{M}$, and $F^\gamma_\mathcal{M}(f)(a_1, \ldots, a_n)$ is defined.

The definition of $F^\gamma_\mathcal{M}(f)$ is well-defined since $F_\mathcal{M}$ is monotone and hence total by Lemma 4.1.

Define $\triangle_\alpha$ to be the empty function of sort α in $\mathcal{M}$ and $\mathsf{card}(S)$ to be the cardinality of a given set S. Assume that $F^\gamma_\mathcal{M}(\triangle_\alpha)$ is not a fixed point of $F_\mathcal{M}$ for all ordinals γ. By this assumption, the monotonicity of $F_\mathcal{M}$, and induction on the ordinals, we can show that

$$\mathsf{card}(\gamma) \le \mathsf{card}(\mathsf{domain}(F^\gamma_\mathcal{M}(\triangle_\alpha)))$$

for all ordinals γ. Let $\kappa = \mathsf{card}((\alpha_1)_\mathcal{M} \times \cdots \times (\alpha_n)_\mathcal{M})$. Then

$$\mathsf{card}(\mathsf{domain}(F^\gamma_\mathcal{M}(\triangle_\alpha))) \le \kappa$$

for all ordinals γ. But then

$$\kappa + 1 = \mathsf{card}(\kappa + 1) \le \mathsf{card}(\mathsf{domain}(F^{\kappa+1}_\mathcal{M}(\triangle_\alpha))) \le \kappa,$$

which is a contradiction.

We have thus shown that, for some ordinal γ, $F^{\gamma}_{\mathcal{M}}(\triangle_\alpha)$ is a fixed point of $F_{\mathcal{M}}$. Let δ be the least ordinal such that $F^{\delta}_{\mathcal{M}}(\triangle_\alpha)$ is a fixed point of $F_{\mathcal{M}}$. We claim that $F^{\delta}_{\mathcal{M}}(\triangle_\alpha)$ is a strong fixed point of $F_{\mathcal{M}}$. Let g be any function of sort α in $\mathcal{M}$ such that $F_{\mathcal{M}}(g) \sqsubseteq_\alpha g$. Clearly, $\triangle_\alpha \sqsubseteq_\alpha g$, and so by the monotonicity of $F_{\mathcal{M}}$, $F^{\delta}_{\mathcal{M}}(\triangle_\alpha) \sqsubseteq_\alpha F^{\delta}_{\mathcal{M}}(g) \sqsubseteq_\alpha g$. Therefore, $F^{\delta}_{\mathcal{M}}(\triangle_\alpha)$ is a strong fixed point of $F_{\mathcal{M}}$. $\Box$

This fixed point theorem is related to the Knaster-Tarski fixed point theorem for complete partial orders [11] and the Tarski fixed point theorem for complete lattices [16].

A fixed point theorem with a stronger conclusion can be obtained if “monotone functional” is replaced with “continuous functional”. See the appendix for details. Continuous functionals are a popular device in computer science for defining functions by recursion, and in particular, they are a basic component of denotational semantics [14, 15].

5 Recursive Definitions

We can now present our scheme for defining (partial higher-order) functions in LUTINS by recursion.

A *recursive definition* in the scheme is a triple $R = (\mathcal{T}, f, F)$ where:

(1) $\mathcal{T} = (\mathcal{L}, \Gamma)$ is a theory of LUTINS.

(2) f is a constant of sort α that is not a member of $\mathcal{L}$.

(3) F is a functional of sort α that is monotone in $\mathcal{T}$.

The *defining axiom* of R is $\mathsf{sfp}_\alpha(f, F)$. The *definitional extension resulting from* R is the extension of $\mathcal{T}$ obtained by adding f to $\mathcal{L}$ and the defining axiom of R to Γ.

In the examples below, let $\mathcal{A}$ be a standard theory of real arithmetic and **N**, **Z**, and **R** be the sorts in $\mathcal{A}$ of the natural numbers, the integers, and the real numbers, respectively.

Example 5.1 The term

$$F = \lambda\, f : \mathbf{N} \rightharpoonup \mathbf{N} \,.\, \lambda\, n : \mathbf{N} \,.\, \mathsf{if}(n = 0, 1, f(n-1) * n)$$

is a monotone functional in $\mathcal{A}$ of sort $\mathbf{N} \rightharpoonup \mathbf{N}$. The recursive definition $(\mathcal{A}, !, F)$ defines the factorial function in $\mathcal{A}$ (where ! is a constant of sort $\mathbf{N} \rightharpoonup \mathbf{N}$ not in $\mathcal{L}$).

Example 5.2 The term

$$\begin{aligned} F \quad = \quad & \lambda\, \sigma : \mathbf{Z} \times \mathbf{Z} \times (\mathbf{Z} \rightharpoonup \mathbf{R}) \rightharpoonup \mathbf{R} \,. \\ & \lambda\, m, n : \mathbf{Z}, f : \mathbf{Z} \rightharpoonup \mathbf{R} \,.\, \mathsf{if}(m \leq n, \sigma(m, n-1, f) + f(n), 0) \end{aligned}$$

is a monotone functional in $\mathcal{A}$ of sort

$$\mathbf{Z} \times \mathbf{Z} \times (\mathbf{Z} \rightharpoonup \mathbf{R}) \rightharpoonup \mathbf{R}.$$

The recursive definition $(\mathcal{A}, \Sigma, F)$ defines the function in $\mathcal{A}$ that gives the summation of a function of sort $\mathbf{Z} \rightharpoonup \mathbf{R}$ over a finite segment of integers (where Σ is a constant of sort $\mathbf{Z} \times \mathbf{Z} \times (\mathbf{Z} \rightharpoonup \mathbf{R}) \rightharpoonup \mathbf{R}$ not in $\mathcal{L}$). ($\Sigma(m, n, f)$ would usually be written $\sum_{i=m}^{n} f(i)$.)

Example 5.3 The term

$$F = \lambda\, f : \mathbf{Z} \rightharpoonup \mathbf{Z} \,.\, \lambda\, n : \mathbf{Z} \,.\, f(n)$$

is a monotone functional in $\mathcal{A}$ of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$. Notice that every function of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$ is a fixed point of F. Thus, the recursive definition $(\mathcal{A}, \triangle_{\mathbf{Z} \rightharpoonup \mathbf{Z}}, F)$ defines the empty function of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$ in $\mathcal{A}$ (where $\triangle_{\mathbf{Z} \rightharpoonup \mathbf{Z}}$ is a constant of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$ not in $\mathcal{L}$).

Example 5.4 The term

$$F = \lambda\, f : \mathbf{Z} \rightharpoonup \mathbf{Z} \,.\, \lambda\, n : \mathbf{Z} \,.\, f(n) + 1$$

is a monotone functional in $\mathcal{A}$ of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$. Notice that the empty function of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$ is the only fixed point of F. Thus, the recursive definition $(\mathcal{A}, \triangle_{\mathbf{Z} \rightharpoonup \mathbf{Z}}, F)$ defines the empty function of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$ in $\mathcal{A}$ (where $\triangle_{\mathbf{Z} \rightharpoonup \mathbf{Z}}$ is a constant of sort $\mathbf{Z} \rightharpoonup \mathbf{Z}$ not in $\mathcal{L}$).

The theorem below shows that recursive definitions are merely a convenience: they do not allow any new functions to be defined that could not be defined by direct means.[3]

Theorem 5.5 *Let $\mathcal{T}$ be a* LUTINS *theory. A function can be directly defined in $\mathcal{T}$ by a term iff it can be recursively defined in $\mathcal{T}$ by a monotone functional.*

Proof Let f be a constant of sort α. Assume f is directly defined in $\mathcal{T}$ by a term t of sort α. Then $t\downarrow$ holds in $\mathcal{T}$, f does not occur in t, and the defining axiom is $f = t$. Let F be $\lambda\, f : \alpha \,.\, t$. F is a monotone functional since $t\downarrow$ holds in $\mathcal{T}$ and f does not occur in t. Clearly, t is the unique fixed point of F. Hence, f is recursively defined in $\mathcal{T}$ by F.

Now assume f is recursively defined in $\mathcal{T}$ by a monotone functional F. Then the defining axiom is $\mathsf{sfp}_\alpha(f, F)$. Let t be

$$\mathrm{I}\, f : \alpha \,.\, \mathsf{sfp}_\alpha(f, F).$$

Clearly, $t\downarrow$ holds in $\mathcal{T}$. Hence, f is directly defined in $\mathcal{T}$ by t. □

6 Extensions

Our scheme for recursively defining functions in LUTINS can be extended in three ways.

First, the notion of defining a single function by recursion can be straightforwardly generalized to the notion of defining a system of functions by mutual recursion. A *recursive definition* is redefined to be a triple

$$R = (\mathcal{T}, \langle f_1, f_2, \ldots, f_n \rangle, \langle F_1, F_2, \ldots, F_n \rangle)$$

where:

(1) $\mathcal{T} = (\mathcal{L}, \Gamma)$ is a theory of LUTINS.

(2) $n \geq 1$.

[3]Since LUTINS is a higher-order logic with a definite description operator, any function that can be defined indirectly by a formula can also be defined directly by a term that is a definite description.

(3) For all i with $1 \leq i \leq n$:

   (a) f_i is a constant of sort α_i that is not a member of $\mathcal{L}$.

   (b) F_i is an expression of sort $\alpha_1 \times \cdots \times \alpha_n \rightharpoonup \alpha_i$ that is monotone with respect to its ith argument in $\mathcal{T}$.

The *defining axiom* of R says that $\langle f_1, f_2, \ldots, f_n \rangle$ is a "simultaneous strong fixed point" of $\langle F_1, F_2, \ldots, F_n \rangle$. The *definitional extension resulting from* R is the extension of $\mathcal{T}$ obtained by adding $f_1, f_2, \ldots, f_n$ to $\mathcal{L}$ and the defining axiom of R to Γ.

Second, recursive definitions can be allowed to contain parameters. A recursive definition (of a single function) with parameters of sort $\pi_1, \ldots, \pi_m$ defines a constant f of sort $\pi_1 \times \cdots \times \pi_m \rightharpoonup \alpha$ by means of a "parameterized functional" F of sort $\alpha \times \pi_1 \times \cdots \times \pi_m \rightharpoonup \alpha$ that is monotone with respect to its first argument. The *defining axiom* of the definition is the sentence

$$\forall\, p_1 : \pi_1, \ldots, p_m : \pi_m \,.\, \mathsf{sfp}_\alpha(f(p_1, \ldots, p_m), \lambda\, g : \alpha \,.\, F(g, p_1, \ldots, p_m))$$

which says that each instance of f is a strong fixed point of the corresponding instance of F.

Example 6.1 The triple

$$(\mathcal{A}, \Sigma', F),$$

where

$$\begin{aligned} F \quad = \quad & \lambda\, \sigma : \mathbf{Z} \times (\mathbf{Z} \rightharpoonup \mathbf{R}) \rightharpoonup \mathbf{R}, m : \mathbf{Z}\,. \\ & \lambda\, n : \mathbf{Z}, f : \mathbf{Z} \rightharpoonup \mathbf{R}\,.\, \mathsf{if}(m \leq n, \sigma(n-1, f) + f(n), 0), \end{aligned}$$

is a recursive definition of the summation function Σ defined in Example 5.2 with the first argument treated as a parameter. Notice that $\Sigma(m, n, f) = \Sigma'(m)(n, f)$.

Example 6.2 Let $\mathsf{sets}(\mathbf{N})$ be the sort of sets of natural numbers in $\mathcal{A}$. The triple

$$(\mathcal{A}, \mathsf{omega\_embedding}, F),$$

where

$$\begin{aligned} F \quad = \quad & \lambda\, f : \mathbf{N} \rightharpoonup \mathbf{N}, a : \mathsf{sets}(\mathbf{N})\,.\, \lambda\, k : \mathbf{N}\,. \\ & \mathsf{if}(k = 0, \\ & \quad \mathrm{I}\, n : \mathbf{N}\,.\, n \in a \wedge (\forall\, m : \mathbf{N}\,.\, m < n \supset \neg(m \in a)), \\ & \quad \mathrm{I}\, n : \mathbf{N}\,.\, n \in a \wedge f(k-1) < n \wedge \\ & \qquad (\forall\, m : \mathbf{N}\,.\, (f(k-1) < m \wedge m < n) \supset \neg(m \in a))), \end{aligned}$$

is a recursive definition with a parameter over $\mathsf{sets}(\mathbf{N})$. Let a be an expression defined in $\mathsf{sets}(\mathbf{N})$. Then $\mathsf{omega\_embedding}(a)$ maps the natural numbers to the members of a such that i is mapped to the ith member of a for all i with with $0 \leq i < \mathsf{card}(a)$. $\mathsf{omega\_embedding}(a)$ is total iff $\mathsf{card}(a)$ is infinite.

Third, a scheme for defining predicates by recursion can be obtained by modifying the scheme for defining functions by recursion described above. Let

$$\beta = \beta_1 \times \cdots \times \beta_n \rightarrow *$$

be a sort of kind $*$ where $n \geq 1$. A *predicate* of sort β is a total function of sort β. In the recursive definition scheme for predicates, the *subpredicate* relation defined below is used in place of the *subfunction* relation defined above.

Definition 6.3 (Subpredicate) $\forall\, g, h : \beta \,.\, g \subseteq_\beta h \equiv$
$\forall\, x_1 : \beta_1, \ldots, x_n : \beta_n \,.\, g(x_1, \ldots, x_n) \supset h(x_1, \ldots, x_n)$.

Example 6.4 The recursive definition $(\mathcal{A}, \langle \mathsf{even}, \mathsf{odd} \rangle, \langle F_1, F_2 \rangle)$, where

$$F_1 = \lambda\, e, o : \mathbf{N} \rightarrow * \,.\, \mathsf{if}(n = 0, \mathsf{T}, o(n-1))$$

and

$$F_2 = \lambda\, e, o : \mathbf{N} \rightarrow * \,.\, \mathsf{if}(n = 0, \mathsf{F}, e(n-1),$$

defines in $\mathcal{A}$ the predicates even and odd on the natural numbers by mutual recursion. (T denotes true and F denotes false.)

7 Implementation in IMPS

Suppose that an IMPS user would like to define a new constant f in a theory $\mathcal{T}$ to be the function defined by a (presumably monotone) functional F in $\mathcal{T}$. The user will submit the triple $(\mathcal{T}, f, F)$ to IMPS, and then IMPS will perform the following steps:

(1) Check that f is a constant of a function sort α that is not currently in $\mathcal{T}$ or in a structural supertheory of $\mathcal{T}$.

(2) Check that F is a functional in $\mathcal{T}$ in canonical form of sort α.

(3) Check that F is known to be monotone in $\mathcal{T}$.

(4) If the checks above are successful, add the constant f to the language of $\mathcal{T}$, add the formula $\mathsf{sfp}_\alpha(f, F)$ to the axioms of $\mathcal{T}$, and install the formula $\mathsf{lfp}_\alpha(f, F)$ in $\mathcal{T}$ as a theorem.

IMPS knows that F is monotone in $\mathcal{T}$ if $\mathsf{monotone}_\alpha(F)$ has been installed in $\mathcal{T}$ as theorem or if the monotonicity check described in the next section succeeds on F.

8 The Monotonicity Check

For an expression E and variables x, y, define $E[x \mapsto y]$ to be the result of replacing each free occurrence of x in E with y. Let f, g, and h be variables of sort α, and let E be an expression that contains neither g nor h. E is *f-stable* in an IMPS theory $\mathcal{T}$ if

$$(g \sqsubseteq_\alpha h \wedge E[f \mapsto g]\!\downarrow) \supset E[f \mapsto g] = E[f \mapsto h]$$

is valid in $\mathcal{T}$. Notice that, if E is f-stable and $g \sqsubseteq_\alpha h$, then either $E[f \mapsto g]$ is undefined or $E[f \mapsto g]$ and $E[f \mapsto h]$ are equal. Notice also that f itself is not f-stable.

Lemma 8.1 (Stability Lemma) *Let*

$$F = \lambda\, f : \alpha \,.\, \lambda\, x_1 : \alpha_1, \, \ldots, \, x_n : \alpha_n \,.\, B$$

be a functional of sort α in a theory $\mathcal{T}$. Then $\mathsf{monotone}_\alpha(F)$ *is valid in $\mathcal{T}$ provided B is f-stable in $\mathcal{T}$.*

Proof Suppose B is f-stable in $\mathcal{T}$, and g and h are variables of sort α not occurring in B. Then

$$(g \sqsubseteq_\alpha h \,\wedge\, B[f \mapsto g]\!\downarrow) \,\supset\, B[f \mapsto g] = B[f \mapsto h]$$

is valid in $\mathcal{T}$. This implies

$$\begin{array}{l} g \sqsubseteq_\alpha h \supset \\ \quad \lambda\, x_1 : \alpha_1, \, \ldots, \, x_n : \alpha_n \,.\, B[f \mapsto g] \;\sqsubseteq_\alpha\; \lambda\, x_1 : \alpha_1, \, \ldots, \, x_n : \alpha_n \,.\, B[f \mapsto h] \end{array}$$

is valid in $\mathcal{T}$, and hence $g \sqsubseteq_\alpha h \supset F(g) \sqsubseteq_\alpha F(h)$ is valid in $\mathcal{T}$. Therefore, $\mathsf{monotone}_\alpha(F)$ is valid in $\mathcal{T}$. □

The following four lemmas are easy to prove:

Lemma 8.2 *If f is not free in E, then E is f-stable in $\mathcal{T}$.*

Lemma 8.3 $\mathsf{if}(\varphi, s, t)$ *is f-stable in $\mathcal{T}$ if f is not free in φ and both s and t are f-stable in $\mathcal{T}$.*

Lemma 8.4 *$f(A_1, \ldots, A_n)$ is f-stable in $\mathcal{T}$ if A_i is f-stable in $\mathcal{T}$ for all i with $1 \le i \le n$.*

Lemma 8.5 *$E(B_1, \ldots, B_m)$ is f-stable in $\mathcal{T}$ if E is f-stable in $\mathcal{T}$ and B_i is f-stable in $\mathcal{T}$ for all i with $1 \le i \le m$.*

Given a functional

$$F = \lambda\, f : \alpha \,.\, B$$

in canonical form of sort α in a theory $\mathcal{T}$, the IMPS monotonicity check works as follows. First, a functional

$$F' = \lambda\, f : \alpha \,.\, \lambda\, x_1 : \alpha_1, \, \ldots, \, x_n : \alpha_n \,.\, B(x_1, \ldots, x_n)$$

is constructed such that $F = F'$ is valid in $\mathcal{T}$. Second, the application $B(x_1, \ldots, x_n)$ is beta-reduced (in $\mathcal{T}$), yielding a possibly new expression B'. Lastly, the four lemmas above are repeatedly applied to B' in a purely syntactic manner until either B' is shown to be f-stable in $\mathcal{T}$ or else no more applications of the four lemmas are possible. In the former case, the check succeeds and the functional F is then monotone in $\mathcal{T}$ by the Stability Lemma. In the latter case, the check fails and nothing is implied about whether or not F is monotone in $\mathcal{T}$.

The IMPS monotonicity check succeeds on the functionals given in Examples 5.1, 5.2, 5.3, 5.4, and 6.1 but does not succeed on the functional given in Example 6.2 because the variable f is free in the body of the second definite description. Notice that the functional

$$\begin{array}{rcl} F' & = & \lambda\, f : \mathbf{N} \rightharpoonup \mathbf{N}, a : \mathsf{sets}(\mathbf{N}) \,.\, \lambda\, k : \mathbf{N} \,. \\ & & \mathsf{if}(k = 0, \\ & & \quad \mathrm{I}\, n : \mathbf{N} \,.\, n \in a \wedge (\forall\, m : \mathbf{N} \,.\, m < n \supset \neg(m \in a)), \\ & & \quad (\lambda\, z : \mathbf{N} \,.\, \mathrm{I}\, n : \mathbf{N} \,.\, n \in a \wedge z < n \wedge \\ & & \qquad (\forall\, m : \mathbf{N} \,.\, (z < m \wedge m < n) \supset \neg(m \in a)))(f(k-1))), \end{array}$$

is the same as the functional F given in Example 6.2 except that F' contains a non-beta-reduced lambda-application for the form

$$(\lambda\, z : \mathbf{N}\ \ldots)(f(k-1)).$$

As a result, the variable f is moved into a position so that the monotonicity check succeeds on F'. Since

$$\forall\, f\, .\ \mathsf{sfp}_\alpha(f, F) \equiv \mathsf{sfp}_\alpha(f, F')$$

is valid in $\mathcal{T}$, F' can be used to define omega_embedding instead of F.

This trick is often useful for transforming a functional on which the monotonicity check fails to an "equivalent" functional on which the check succeeds. In our experience, nearly all recursive definitions that arise naturally have functionals on which the monotonicity check succeeds directly or via a transformation by means of this trick.

There is a monotonicity check for functionals that define predicates by (mutual) recursion which is similar to the monotonicity check described in this section for functionals that define functions by (mutual) recursion.

9 Conclusion

We have presented in the logic LUTINS a proper scheme for defining partial higher-order functions by recursion. An instance of the scheme is a triple $(\mathcal{T}, f, F)$ where $\mathcal{T}$ is a theory of LUTINS, f is a constant of a function sort α, and F is a functional of sort α that is monotone in $\mathcal{T}$. The instance $(\mathcal{T}, f, F)$ defines f to be the strong fixed point of F in $\mathcal{T}$. The domain of the scheme is exactly the set of functions that can be directly defined in LUTINS. We have described three extensions of the scheme and an automatic syntactic check for monotonicity that succeeds for many common recursive definitions. The scheme, with the three extensions and the check for monotonicity, has been implemented in the IMPS Interactive Mathematical Proof System.

Appendix: Continuous Functionals

This appendix presents a fixed point theorem for continuous functionals which has a stronger conclusion than Theorem 4.2, a fixed point theorem for monotone functionals.

Let $\mathsf{sets}(\alpha)$ be the sort of sets of elements of α.

Definition 9.1 (Chain) $\forall\, S : \mathsf{sets}(\alpha)\, .\ \mathsf{chain}_\alpha(S) \equiv$
$\forall\, g, h : \alpha\, .\ (g \in S \wedge h \in S) \supset (g \sqsubseteq_\alpha h \vee h \sqsubseteq_\alpha g)$.

Definition 9.2 (Least Upper Bound) $\forall\, S : \mathsf{sets}(\alpha)\, .\ \mathsf{lub}_\alpha(S) \simeq$
$\mathrm{I}\, f : \alpha\, .\ (\forall\, g \in S \supset g \sqsubseteq_\alpha f) \wedge ((\exists\, f' : \alpha\, .\ \forall\, g \in S \supset g \sqsubseteq_\alpha f') \supset f \sqsubseteq_\alpha f')$.

Proposition 9.3 *The least upper bound of a chain is always defined.*

Definition 9.4 (Continuous) $\forall\, F : \alpha \rightharpoonup \alpha\, .\ \mathsf{continuous}_\alpha(F) \equiv$
$\mathsf{monotone}_\alpha(F) \wedge \forall\, S : \mathsf{sets}(\alpha)\, .\ \mathsf{chain}_\alpha(S) \supset F(\mathsf{lub}_\alpha(S)) = \mathsf{lub}_\alpha(\{F(g) : g \in S\})$.

For a functional F and a nonnegative integer i, let $F^i(g)$ be an abbreviation for $F(\cdots(F(g))\cdots)$ where F occurs i times.

Theorem 9.5 (Fixed Point Theorem for Continuous Functionals) *If $\triangle_\alpha$ is the term*

$$\lambda\, x_1 : \alpha_1, \, \ldots, \, x_n : \alpha_n \, . \, \mathrm{I}\, y : \alpha_{n+1} \, . \, \neg(y = y)$$

(which denotes the empty function of sort α) and f is the term

$$\mathsf{lub}(\{F^i(\triangle_\alpha) : 0 \leq i\}),$$

then the sentence

$$\forall\, F : \alpha \rightharpoonup \alpha \, . \, \mathsf{continuous}_\alpha(F) \supset \mathsf{sfp}_\alpha(f, F)$$

is valid in $\mathcal{L}$.

Proof By the definition of f,

$$F(f) = F(\mathsf{lub}(\{F^i(\triangle_\alpha) : 0 \leq i\})).$$

Since F is monotone, $\{F^i(\triangle_\alpha) : 0 \leq i\}$ is a chain, and so by the definition of a continuous functional,

$$\begin{aligned} F(\mathsf{lub}(\{F^i(\triangle_\alpha) : 0 \leq i\})) &= \mathsf{lub}(\{F(F^i(\triangle_\alpha)) : 0 \leq i\}) \\ &= \mathsf{lub}(\{F^i(\triangle_\alpha) : 1 \leq i\}) \\ &= f. \end{aligned}$$

Hence, f is a fixed point of F.

We claim that f is a strong fixed point of F. Let g be any function of sort α such that $F(g) \sqsubseteq_\alpha g$. Clearly, $\triangle_\alpha \sqsubseteq_\alpha g$, and so by the monotonicity of F, $F^i(\triangle_\alpha) \sqsubseteq_\alpha F^i(g) \sqsubseteq_\alpha g$ for all i with $0 \leq i$. Therefore,

$$f = \mathsf{lub}(\{F^i(\triangle_\alpha) : 0 \leq i\}) \sqsubseteq_\alpha g.$$

$\square$

Acknowledgments

The recursive definition scheme described in this paper was developed jointly by Dr. Joshua Guttman, Dr. F. Javier Thayer Fábrega, and the author as part of the design of IMPS.

References

[1] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.

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

[4] W. M. Farmer. Theory interpretation in simple type theory. In J. Heering et al., editor, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.

[5] W. M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43:279–294, 1995.

[6] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[7] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[8] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993. Available at `http://imps.mcmaster.ca/`.

[9] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[10] S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[11] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societeé Polonaise de Mathematique*, 6:133–134, 1928.

[12] Y. N. Moschovakis. Abstract recursion as a foundation for the theory of algorithms. In *Computation and Proof Theory, Lecture Notes in Mathematics 1104*, pages 289–364. Springer-Verlag, 1984.

[13] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.

[14] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. W. C. Brown, Dubuque, Iowa, 1986.

[15] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[16] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.