Higher Order Functions in First Order Logics

Andrei Voronkov

ECRC, Arabellastr.17, 8000 Munich 81, Germany (voronkov@ecrc.de)*

Abstract. Most of approaches to extracting programs from (constructive) proofs use type theories. Usually it is argued that first order logic has many drawbacks to be used as a language for programming in logic. In particular, higher order functions are not directly expressible in first order logic. Here we show how to use proof schemes in first order logic for representing higher order functions. We generalize the semantics introduced in [Vor 90] to the proof schemes and show how it is related to extraction of higher order functions from proofs in first order logic.

1 Introduction

Several approaches can be used to derive programs from specifications. One of them is based on extracting programs from constructive proofs. The essential idea of this approach is that under some restrictions a proof of a specification implicitly contains information sufficient to construct a program meeting the specification. Different restrictions give different logics. Most of such logics are *logics of higher types* or *type theories* [Bas 89, Chi 87, Con 86, Coq 88, Hen 92, Nor 84]. Type theories are very helpful in many aspects:

- Rich expressive power. It means that type theories usually can be used for the representation of programming data types as well as for the representation of extracted programs.
- The language of extracted programs is the part of the language of type theories which allows one to use the theories also for manipulations of programs.
- Independently of the language of proofs extracted programs are usually written in a variant of the lambda calculus, i.e. they are essentially pure functional programs. Functional programming greatly benefits from the use of higher order functions. Nevertheless higher order functions which are easily expressible in type theories are not directly representable in first order logic.

Besides it the notion of type theory is sufficiently flexible and allows different modifications which can be used to represent some ideas from programming, e.g. typed vs. untyped programming languages.

The first step in the design of a proof extraction system is to define the logical system in which formal proofs will be written. For writing formal proofs type theories have serious disadvantages. It is much easier to construct proofs in first order theories then in type theories. First order logic has a natural model theoretical semantics, which allows easier understanding of proofs. On the contrary, type theories have much more complicated semantics compared to first order logics. As a consequence, type theories are unnatural for specifying input-output conditions of programs. The

* On leave from the International Laboratory of Intelligent Systems (SINTEL), 630090, Universitetski Prospect 4, Novosibirsk 90, Russia (voronkov@sintel.nsk.su).

specification of a function f, which by any x satisfying $A(x)$ gives y such that $B(x,y)$ is true, in first order logic is written in the form $\forall x(A(x) \supset \exists y B(x,y))$, which is intuitively a natural way to express existence of such a function. In type theories such a specification is usually written in the form $f \in \Pi x : A.\Sigma y : B(x)$. To understand the meaning of this specification one needs to know quite complicated semantics of type theories. Even if such a specification is proved in type theory, it is not trivial to understand that the function of the type $\Pi x : A.\Sigma y : B(x)$ satisfies the specification $\forall x(A(x) \supset B(x, f(x)))$.

The computer implementations of type theories [Chi 87] are also quite complicated and the proofs in this systems are difficult to understand. At the same time there are quite successful implementations of first order constructive logics [Goa 80a, Goa 80b, Hay 88]. In our previous papers we gave a systematic approach to the theory of programming in constructive logic [SV 90, Vor 88, Vor 90, Vor 91b, Vor 91a]. This approach has been used to give a formal definition of constructive logic. The approach is quite general: for example among the results proved in our previous papers are

- The proofs that well-known constructive logics are constructive in the strict sense of our definitions.
- The proofs connecting two notions of constructivism: that of constructive logic and that of the recursion theory.
- The proofs of some relations between the classical model theory and the model theory based on the constructive notion of truth.

The systems described in [Goa 80a, Goa 80b] had already shown the practical importance of first order logics for program extraction. In a forthcoming paper we are planning to show that first order logic can be also used for the purposes claimed to be advantages of type theories, namely the introduction of new types.

In this paper we show how to use *proof schemes* to extract higher order functions from the proofs in first order logic. In Section 2 we give an intuitive example of the use of parameters in proofs to represent higher order functions. In Section 3 we define a constructive notion of truth for parametric formulae. Then in Section 4 we show how to define constructive logics using constructive truth. The notion of constructive logic justifies the use of parameters in the example from Section 2

2 An example

Suppose that we want to specify the function `mapcar` which applies a given function `f` to all elements of a given list `ln`. The usual specification of this higher order function can be written in the form of recursive equations:

```
mapcar f nil ⇒ nil
mapcar f (cons x l) ⇒ cons (f x) (mapcar f l)
```

However our approach is to use first order logic without any additional features like recursion. Hayashi [Hay 88] handles recursive equations in his (essentially first order) system, but the logic he uses is quite complicated compared to usual first order logic.

To specify `mapcar` we need to use a language of a constructive theory for proving properties of lists. Such a system can be found e.g. in [Vor 91a]. Here we do not give a full axiomatization of the list theory, but we provide only some essential details instead.

The list theory has three sorts: the sort α of atoms, the sort λ of lists and the sort ν which is the union of α and λ. Some of the special functions and predicates used in the language of the theory are

`nil` - the constant for empty list;
`cons` - the list constructor;
$l_1 \sqsubseteq l_2$ - the predicate between lists which is true if l_1 can be obtained from l_2 by deleting several elements from the beginning.

We shall also occasionally use functions `car` giving the first element of a list and `length` giving the length of a list. One of the possible ways to write the specification of `mapcar` in form of a predicate is the following:[2]

```
mapcar(In,Out) ⇔ length(In) = length( Out) ∧
                 (∀X⊑In)¬¬(∃Y⊑Out)(length(X) = length(Y) ∧
                 ¬X=[] ⊃ car(Y) = f(car(X)) )
```

We can in principle use any other specification, because the scheme of the proof given below remains valid for all of them.

Our aim is to prove the following formula specifying the existence of this function:

```
(∀In)(∃Out)mapcar(In,Out).
```

The only rule for lists that will be used in the proof below is *list induction:*

$$\dfrac{\varphi(\mathrm{nil}) \qquad \begin{matrix}[\varphi(l)]\\ \vdots \\ \varphi(\mathrm{cons}(u,l))\end{matrix}}{\forall l \varphi(l)}$$

where u is a variable of the sort ν and l is a variable of the the sort λ. The constructiveness of this rule is easy to prove (see e.g. [Vor 91a]).

Harrop formulae [Har 60] are defined as follows:

1. Any atomic formula is a Harrop formula;
2. If φ, ψ are Harrop formulae, χ an arbitrary formula, then the following formulae are Harrop formulae: $\varphi \wedge \psi$, $\forall x \varphi$, $\neg \chi$, $\chi \supset \varphi$.

We do not provide proofs of Harrop formulae in the example given below, because Harrop formulae have no computational content. It means that the proof of these formulae is necessary for proving correctness of the extracted function, but it does not effect the definition of the function.

A (partial) proof of the above specification in the sequent natural deduction system [Pra 65] is as follows:

(1) `mapcar(In,Out) ⊢ mapcar([Z|In],[f(Z)|Out])`	[Harrop]
(2) `(∃Out)mapcar(In,Out) ⊢ (∃Out)mapcar(In,Out)`	[axiom]
(3) `mapcar(In,Out) ⊢ (∃Out)mapcar([Z|In],Out)`	[∃-intro 1]
(4) `⊢ mapcar([],[])`	[Harrop]
(5) `⊢ (∃Out)(mapcar([],Out))`	[∃-intro 4]
(6) `(∃Out)mapcar(In,Out) ⊢ (∃Out)mapcar([Z|In],Out)`	[∃-elim 2,3]
(7) `⊢ (∀In)(∃Out)mapcar(In,Out)`	[ListInd 5,6]

[2] The reason to use double negation is that we want `mapcar` to be Harrop formula (see below).

The annotation [Harrop] is used to denote sequents with Harrop formulae in the succedent.

It is easy to see that the proof does not depend on particular properties of the function `f`. Hence we may consider it as a specification of a higher order function which takes `f` and `In` as the input and produces the output `Out = mapcar(f,In)`. This is the main idea of how to synthesize higher order functions from the proofs in first order logic. However to make this idea precise we need to give formal definitions and to prove theorems about correctness of thus extracted functions. We do it in the subsequent sections.

Usually the specifications for program extraction are written in one of the following forms: $(\forall x)(\exists y)Spec(x,y)$ or $(\forall x)(In(x)\supset(\exists y)Out(x,y))$. Suppose that the functional symbol f occurs in this formulae. If there are no occurences of f in the set of axioms of the theory in which proofs are written then the above formulae serve as a specification of an algorithm which takes x <u>and</u> f as the input and gives y as the output.

To give the definition of *truth* of a formula on a model the functions and predicates must be interpreted on this model. After we fixed the model we may construct calculi for proving properties of this model, for instance Peano arithmetic for natural numbers. In the above example we used the symbol `f` which had no interpretation on the intended model. So the above proof can be considered as a scheme in which one can substitute concrete interpreted functions for `f`. To justify the use of uninterpreted functions and such *proof schemes* in a constructive framework we shall change the basic definitions from [Vor 90] in the next sections.

3 Constructive truth for formulae with parameters

This section contains auxiliary definitions concerning the theory of enumerated models and higher type functionals and the definitions of constructive truth for formulae with parameters.

First of all we introduce the definition of *enumerated models* [Ers 80]. Enumerated models and computable functionals of higher types form the basis of our theory of extracting programs from proofs. (More details may be found in [Vor 88, Vor 90]). The motivation to use enumerated models was explained in [Vor 90]. We want to represent programming data types as *models* in the sense of the model theory [Cha 77]. However as we intend to formalize computations over these data types, we need a notion of computability over the models. One of possible ways is to enumerate the elements of the models by natural numbers and to use this enumeration for defining computable functions on the data types. In a way it corresponds to the implementation of data types in programming languages: finally all elements of data types are encoded by sequences of bits in the computer memory. This encoding is invisible to the user of the language, but it is needed to compile programs. In our approach the enumeration of the model is also invisible — all we have is the axioms of some theory describing properties of the data type. However to prove that we can extract programs from proofs (which corresponds to compilation in programming) such encoding is needed.

Definition 1. Let S be a set. An <u>enumeration</u> [Ers 77] of S is any mapping of the set of all natural numbers $\mathfrak{N}$ onto S. An <u>enumerated set</u> is any pair (S,ν) where S is a set and ν is an enumeration of S.

Let $\mathfrak{M} = \langle M, P_0, P_1, \ldots, h_0, h_1, \ldots \rangle$ be a model of a signature σ with predicates $P_0, P_1 \ldots$ and function symbols $h_0, h_1 \ldots$ We shall use $\mathfrak{N}$ to denote the set of all natural numbers.

Definition 2. An <u>enumeration</u> of the model $\mathfrak{M}$ is any enumeration $\nu : \mathfrak{N} \to M$ of the domain M of $\mathfrak{M}$ such that there exists a binary total recursive function H such that for any $n, y_1, \ldots, y_{m_n} \in \mathfrak{N}$ we have $h_n(\nu y_1, \ldots, \nu y_{m_n}) = \nu H(n, \langle y_1, \ldots, y_{m_n} \rangle)$, where $\langle y_1, \ldots, y_{m_n} \rangle$ is the Gödel number of the tuple $y_1, \ldots, y_{m_n}$. The pair $(\mathfrak{M}, \nu)$ where $\mathfrak{M}$ is a model of the signature σ, and ν is its enumeration is called <u>enumerated model</u> of the signature σ.

We can consider an enumeration of a model $\mathfrak{M}$ as an implementation of the data type corresponding to this model. The restriction on enumerations means that all functions from the signature of the model are computable in this enumeration.

To define the constructive semantics we use the functionals of higher types. For representation of such functionals we choose the Scott's notion of *information systems* [Sco 82]. There is no space to write all the formal definitions concerning the information systems, so here we present only the most essential for us properties.

Information systems allow one to define the functionals of higher types. For any information systems $\mathbf{A}$,$\mathbf{B}$ there exist the information systems $\mathbf{A} \oplus \mathbf{B}$, $\mathbf{A} \otimes \mathbf{B}$ and $\mathbf{A} \to \mathbf{B}$. We refer reader to [Sco 82] for the complete definitions. The set of all elements of an information system $\mathbf{A}$ is denoted by $\mid \mathbf{A} \mid$. In any information system $\mathbf{A}$ there always exists the least element $\perp_{\mathbf{A}}$. The system $\mathbf{A} \to \mathbf{B}$ is the domain of all *approximable mappings* (or *continuous functions*) from $\mathbf{A}$ to $\mathbf{B}$. The domain $\mathbf{A} \oplus \mathbf{B}$ is essentially the *disjoint union* of the domains $\mathbf{A}$ and $\mathbf{B}$. Its elements are of the form $\mathrm{inl}(a)$ with $a \in \mid \mathbf{A} \mid$, or $\mathrm{inr}(b)$ with $b \in \mid \mathbf{B} \mid$ or $\perp_{\mathbf{A} \oplus \mathbf{B}}$. $\mathbf{A} \otimes \mathbf{B}$ is the *Cartesian product* of $\mathbf{A}$ and $\mathbf{B}$. Elements of $\mathbf{A} \otimes \mathbf{B}$ are pairs of the form $\langle a, b \rangle$ with $a \in \mid \mathbf{A} \mid$ and $b \in \mid \mathbf{B} \mid$.

The information system $\mathbf{N}$ [Vor 88] is such that the elements of $\mathbf{N}$ are either $\perp_{\mathbf{N}}$ or natural numbers $0, 1, \ldots$. The information system $\mathbf{1}$ is defined in such a way that it has only the bottom element $\perp_{\mathbf{1}}$. As it is done in [Vor 88], one can define the notion of a computable element for any information system constructed from $\mathbf{N}$ and $\mathbf{1}$ by repeated applications of the constructors $\to, \otimes, \oplus$.

The notion of *constructive truth* was defined in [Vor 88, Vor 90] for any enumerated model. The definition relies upon the fact that all functions and predicates are interpreted in the enumerated models. However if we are to define constructive truth for the proof schemes containing uninterpreted symbols we have to change the definitions to the case of uninterpreted functions.

As in [Vor 90] we start from the classical truth.

Let $(\mathfrak{M}, \nu)$ be an enumerated model of the signature σ. Let δ is obtained from σ by adding new functional symbols $f_0, \ldots, f_n$. In what follows we call $f_0, \ldots, f_n$ *parameters*. An *interpretation* $\Im$ is any mapping which maps each $f_i, i = 0, \ldots, n$ to a function on $\mathfrak{M}$. Let the signature Δ is obtained from δ by adding constants νi for all natural numbers i with the standard interpretation. We assign to each formula φ of the extended signature Δ an information system $\mathbf{A}_\varphi$ and a relation $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$ (a classically realizes φ under the interpretation $\Im$) where $a \in \mid \mathbf{A}_\varphi \mid$.

If φ is a formula with free variables $x_0, \ldots, x_n$ then $\forall \varphi$ will denote the formula $\forall x_0, \ldots, x_n \varphi$. If there are no free variables in φ then $\forall \varphi \Leftrightarrow \varphi$.

Definition 3 (The relation $\underline{\mathfrak{cl}}_{\Im}$). In (1)-(7) all formulae are closed.

1. $\mathbf{A}_\varphi \Leftrightarrow \mathbf{1}$ for atomic φ and $\perp \underline{\mathfrak{cl}}_{\Im}\, \varphi$ iff φ is true in $\mathfrak{M}$ under the interpretation $\Im$.

(Strictly speaking the definition of $\underline{\mathfrak{cl}}_{\Im}$ depends of the model $(\mathfrak{M},\nu)$, so we ought to write $\underline{\mathfrak{cl}}_{(\mathfrak{M},\nu),\Im}$ instead of $\underline{\mathfrak{cl}}_{\Im}$, but we omit indices where it causes no ambiguities).

2. $\mathbf{A}_{\varphi\wedge\psi} \Leftrightarrow \mathbf{A}_{\varphi} \otimes \mathbf{A}_{\psi}$ and $\langle a,b\rangle \,\underline{\mathfrak{cl}}_{\Im}\, \varphi\wedge\psi$ iff $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$ and $b \,\underline{\mathfrak{cl}}_{\Im}\, \psi$.
3. $\mathbf{A}_{\varphi\vee\psi} \Leftrightarrow \mathbf{A}_{\varphi} \oplus \mathbf{A}_{\psi}$ and for $r \in \mid \mathbf{A}_{\varphi\vee\psi} \mid$ we have $r \,\underline{\mathfrak{cl}}_{\Im}\, \varphi\vee\psi$ iff one of the following is true:
   (a) for some $a \in \mathbf{A}_{\varphi}$, $r = \mathtt{inl}(a)$ and $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$;
   (b) for some $b \in \mathbf{A}_{\psi}$, $r = \mathtt{inr}(b)$ and $b \,\underline{\mathfrak{cl}}_{\Im}\, \psi$.
   We remind that inl and inr are natural embeddings of $\mid \mathbf{A}_{\varphi} \mid$ and $\mid \mathbf{A}_{\psi} \mid$ into $\mid \mathbf{A}_{\varphi\vee\psi} \mid$.
4. $\mathbf{A}_{\varphi\supset\psi} \Leftrightarrow \mathbf{A}_{\varphi} \to \mathbf{A}_{\psi}$ and for $f \in \mid \mathbf{A}_{\varphi\supset\psi} \mid$, $f \,\underline{\mathfrak{cl}}_{\Im}\, \varphi\supset\psi$ iff for every $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$ we have $f(a) \,\underline{\mathfrak{cl}}_{\Im}\, \psi$.
5. $\mathbf{A}_{\neg\varphi} \Leftrightarrow \mathbf{1}$ and $\perp_{\mathbf{1}} \underline{\mathfrak{cl}}_{\Im}\, \neg\varphi$ iff for every $a \in \mathbf{A}_{\varphi}$ it is not true that $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$.
6. $\mathbf{A}_{\exists x\varphi(x)} \Leftrightarrow \mathbf{N} \otimes \mathbf{A}_{\varphi(t)}$, where t is an arbitrary closed term and $\langle n,a\rangle \,\underline{\mathfrak{cl}}_{\Im}\, \exists x\varphi(x)$ iff $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi(\nu n)$.
7. $\mathbf{A}_{\forall x\varphi(x)} \Leftrightarrow \mathbf{N} \to \mathbf{A}_{\varphi(t)}$ and for $f \in \mid \mathbf{A}_{\forall x\varphi(x)} \mid$, $f \,\underline{\mathfrak{cl}}_{\Im}\, \forall x\varphi(x)$ iff for every $n \in \mathfrak{N}$ we have $f(n) \,\underline{\mathfrak{cl}}_{\Im}\, \varphi(\nu n)$.
8. If there are free variables in φ, then $\mathbf{A}_{\varphi} \Leftrightarrow \mathbf{A}_{\forall\varphi}$, and $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$ iff $a \,\underline{\mathfrak{cl}}_{\Im}\, \forall\varphi$.

A formula φ is <u>classically realizable</u> under the interpretation $\Im$ iff for some $a \in \mid A_{\varphi} \mid$ we have $a \,\underline{\mathfrak{cl}}_{\Im}\, \varphi$.

The following theorem proved in [Vor 90] for the language without parameters, explains why this semantics is called "classical realizability":

Theorem 4. *There exists a such that $a \,\underline{\mathfrak{cl}}_{(\mathfrak{M},\nu),\Im} \varphi$ iff φ is true in $\mathfrak{M}$ under the interpretation $\Im$.*

This semantics is called classical realizability because the set of all realizable formulae coincides with the set of all classically true formulae. The following definition introduce the set of constructively true formulae as a subset of classically true ones.

Definition 5. Let $a \in \mid A_{\varphi} \mid$. Then $a \,\underline{\mathfrak{con}}_{\Im}\, \varphi$ iff $a \,\underline{\mathfrak{cl}}_{\Im} \varphi$ and a is computable. If $a \,\underline{\mathfrak{con}}_{\Im}\, \varphi$ then we will say that a <u>constructively realizes</u> φ. A formula φ is said to be <u>constructively true</u> in the enumerated model $(\mathfrak{M},\nu)$ under the interpretation $\Im$ (denoted by $(\mathfrak{M},\nu) \models_{\underline{\mathfrak{con}},\Im} \varphi$ iff there is a with $a \,\underline{\mathfrak{con}}_{(\mathfrak{M},\nu),\Im} \varphi$.

Up to now this definition differs from that of [Vor 90] only by adding words "under interpretation" when needed. Now we extend it to the parametric case in an obvious way.

Let f_i be n_i-place functions and $\mathbf{F}_i$ be the information systems $\underbrace{\mathbf{N} \otimes \ldots \otimes \mathbf{N}}_{n_i \text{ times}} \to \mathbf{N}$.

Definition 6 (Relations $\underline{\mathfrak{cl}}$ and $\underline{\mathfrak{con}}$ for the parametric case). Let $a \in \mathbf{F}_1 \otimes \ldots \otimes \mathbf{F}_n \to \mathbf{A}_{\varphi}$. Then $a \,\underline{\mathfrak{cl}}\, \varphi$ iff for any interpretation $\Im$ we have $a(\langle \Im(f_1), \ldots, \Im(f_n)\rangle)$ $\underline{\mathfrak{cl}}_{\Im}\, \varphi$. Similar, $a \,\underline{\mathfrak{con}}\, \varphi$ iff for any interpretation $\Im$ we have $a(\langle \Im(f_1), \ldots, \Im(f_n)\rangle) \,\underline{\mathfrak{con}}_{\Im}\, \varphi$.

Note. It is easy to modify all our definitions to many-sorted logics, which better correspond to the practice of programming — every programming languages has several different data types and functions/relations working on these data types. However for simplicity we give all the definitions only for the one-sorted case.

The first consequence of this definition is that the classical realizability (the relation $\underline{\mathfrak{cl}}$) does not coinside with the classical truth any more (Theorem 8 below). However we can prove soundness: if a formula is realizable, then this formula is true. Soundness is sufficient for using the theory for program extraction: when we extract a program from a proof we need to be sure that the program satisfies the specification given by the end formula of the proof.

Theorem 7. *If there exists a such that $a \, \underline{\mathfrak{cl}}_{(\mathfrak{M},\nu)} \varphi$ then φ is true in $\mathfrak{M}$ under any interpretation.*

The following theorem shows that the inverse is not true, unlike the first order case: when one adds function parameters to the language, classical realizability does not coinside with truth in general. In fact we prove even a stronger statement - there exists a formula which is provable in classical first order predicate logic, but which is not realizable.

Theorem 8. *There is an enumerated model $(\mathfrak{M}, \nu)$ with a binary predicate P such that the formula $\psi \Leftrightarrow \forall x P(x, f(x)) \vee \neg \forall x P(x, f(x))$ is not classically realizable on $(\mathfrak{M}, \nu)$.*

Proof. To give the complete proof one needs to give the complete definitions of how the higher types functionals are represented in information systems, for which we have no space here. So we shall only describe the idea of the proof. Let $\mathfrak{M}$ be the standard model of natural numbers with the identity mapping as the enumeration. Let P be the equality on natural numbers. Assume that the formula ψ is realizable and $a \, \underline{\mathfrak{cl}} \, \psi$. Then for every function $g : \mathfrak{N} \to \mathfrak{N}$, $a(g) \, \underline{\mathfrak{cl}} \forall x P(x, g(x)) \vee \neg \forall x P(x, g(x))$. Let g be the identity mapping, i.e. $g(x) = x$ for all $x \in \mathfrak{N}$. Since $a(g)$ realizes a disjunction, then there is a *finite* piece of information in $a(g)$, which is sufficient to say what part of the disjunction is true. Hence there is a *finite* set S of natural numbers, such that for every function h with $h(x) = x$ on S, $h(x) = x$ for all natural numbers, which is false.

Theorem 8 is not desirable for program synthesis, because systems for extracting programs from proofs often use classical logic in some computationally irrelevant parts of proofs. One can consider for instance the use of Harrop formulae in [Goa 80a, Goa 80b] or the mixed Markov's principle from [Vor 90, Vor 91b]. However below we will show that in both these cases classical logic can be used also in the parametric case. First we shall formulate an interesting property of Harrop formulae.

Theorem 9. *If φ is a Harrop formula, then $\mathbf{A}_\varphi$ has one element $\perp$ which is computable.*

Corollary 10. *Let φ be a Harrop formula. Then the following conditions are equivalent.*

1. *φ is true on $\mathfrak{M}$.*
2. *For every enumeration ν of $\mathfrak{M}$, φ is constructively true on $(\mathfrak{M}, \nu)$.*

The same remains true if we change truth to truth under interpretation.

4 Constructive logics with proof schemes

The most important definitions are connected with the notion of *constructive calculus*. Intuitively, the calculus is constructive for some enumerated model iff by any inference rule of this calculus and any tuple of elements realizing its premises one can effectively construct an element realizing its conclusion. Below we give such definitions.

In what follows we will sometimes use informal notions (e.g. saying that a set of formulae is effectively enumerable). But all corresponding formal notions can of course be given.

Definition 11. An inference figure of the signature σ is a finite sequence of formulae $\varphi_1, \dots, \varphi_n, \varphi$, where $n \geq 0$, of this signature. A calculus $\mathcal{L}$ is any effectively enumerable set of inference figures.

Provability of a formula in a calculus $\mathcal{L}$ is defined as usual: φ is provable iff there is an inference figure of the form $\varphi_1, \dots, \varphi_n, \varphi$ in $\mathcal{L}$ such that for all $i \in \{1, \dots, n\}$ φ_i is provable in $\mathcal{L}$.

From now on we assume that an enumerated model $(\mathfrak{M}, \nu)$ of a signature σ is fixed and that δ is the extended signature obtained from σ by adding function symbols $f_0, \dots, f_n$.

Definition 12. Parametric calculus is any calculus in the signature δ. Parametric proof is a proof in a parametric calculus. Functional symbols $f_0, \dots, f_n$ are called parameters of such a proof.

Definition 13. A parametric calculus $\mathcal{L}$ is constructive for an enumerated model $(\mathfrak{M}, \nu)$ iff there exists an algorithm α which by any inference figure $\varphi_1, \dots, \varphi_n, \varphi$ of $\mathcal{L}$ and any $a_1, \dots, a_n$ such that $a_1 \,\underline{\mathfrak{cl}}_{(\mathfrak{M},\nu)} \varphi_1, \dots, a_n \,\underline{\mathfrak{cl}}_{(\mathfrak{M},\nu)} \varphi_n$ gives a such that $a \,\underline{\mathfrak{cl}}_{(\mathfrak{M},\nu)} \varphi$.

The definition of constructive calculus is given in such terms that the the notion of constructive calculus is sound w.r.t. constructive truth. Intuitively, the constructive calculi for an enumerated model $(\mathfrak{M}, \nu)$ are exactly the calculi that we can use for program extraction, or for describing computations over the data type represented by $(\mathfrak{M}, \nu)$.

Theorem 14. *Let $\mathcal{L}$ be constructive for $(\mathfrak{M}, \nu)$ and $\vdash_{\mathcal{L}} \varphi$. Then $(\mathfrak{M}, \nu) \models_{\mathrm{con}} \varphi$. Moreover, an element a with $a \,\underline{\mathtt{con}}_{(\mathfrak{M},\nu)} \varphi$ can be found effectively by a proof of φ in $\mathcal{L}$.*

Proof. Trivial by the length of a proof of φ in $\mathcal{L}$. It is easy to see that we can construct in a computable way some a with $a \,\underline{\mathfrak{cl}}_{(\mathfrak{M},\nu)} \varphi$. Thus this a is computable and $a \,\underline{\mathtt{con}}_{(\mathfrak{M},\nu)} \varphi$.

Now we shall formulate the main theorem on the use of constructive parametric calculi, which justifies the extraction of higher order functions from proofs:

Theorem 15. *Let $\mathcal{L}$ be a constructive parametric calculus for $(\mathfrak{M}, \nu)$. Then given any proof Π of a closed formula $\forall \bar{x} \exists y \varphi(x, y)$ one can effectively construct a general recursive function g with the following property:*

> *If $\Im$ is an interpretation such that there exist general recursive functions $g_0, \dots, g_n$ with $\Im(f_i)(\nu m_1, \dots, \nu m_k) = \nu g_i(m_1, \dots, m_k)$[3] and $l_1, \dots, l_n$ are Kleene numbers of $g_1, \dots, g_n$, then for any tuple $\bar{p}$ of natural numbers the formula $\varphi(\nu \bar{p}, \nu g(l_1, \dots, l_n, \bar{p}))$ is true on $\mathfrak{M}$ under the interpretation $\Im$.*

[3] This condition means that all parameters f_i are computable in this interpretation.

Proof. Straightforward but tedious. It is sufficient to show how to encode computable elements of the information systems by recursive functions and then to apply Theorem 14.

It is easy to see that this formal statement informally means the following: there exists a (second-order) computable function g meeting the specification

$$(\forall\bar{x})(\forall f_0)\dots(\forall f_n)\varphi(\bar{x}, g(f_0,\dots,f_n,\bar{x}))$$

It is also possible to formulate the analogs of Theorem 15 for specifications of partial functions, predicates and partial predicates, as in [Vor 91b].

However the general definition of constructive calculus with the soundness theorem is not sufficient to show the general applicability of the introduced notions. We need to show that the well-known examples of constructive calculi are constructive according to our definitions. We start from the intuitionistic predicate calculus and the intuitionistic (Heyting) arithmetic.

Theorem 16. *The intuitionistic predicate calculus is constructive for every enumerated model* $(\mathfrak{M},\nu)$.

Proof. Copies the proof of Kleene's realizability of formulae provable in the intuitionistic predicate calculus [Kle 52]. The same functionals that are used in the original proof given by Nelson [Nel 47] can be used in our proof. The representation of the functionals in the form of elements of the information systems is a trivial exercise (the techniques for such a representation may be found in [Sco 82]).

Corollary 17. *Let* $\mathcal{L}$ *be a (non-parametric) calculus in the signature* σ *constructive for* $(\mathfrak{M},\nu)$ *and* $\mathcal{L}'$ *be obtained from* $\mathcal{L}$ *by adding the rules of the intuitionistic predicate calculus of the signature* δ *to* $\mathcal{L}$. *Then* $\mathcal{L}'$ *is constructive for* $(\mathfrak{M},\nu)$.

Theorem 18. *The intuitionistic (Heyting) arithmetic is constructive for the standard model of arithmetic* $(\mathfrak{N},\mathrm{id}_{\mathfrak{N}})$.

Corollary 19. *Let* $\mathcal{L}$ *be a calculus in the signature* σ *constructive for* $(\mathfrak{N},\mathrm{id}_{\mathfrak{N}})$ *and* $\mathcal{L}'$ *be obtained from* $\mathcal{L}$ *by adding the rules of the intuitionistic predicate calculus plus the induction scheme for parametric formulas to* $\mathcal{L}$. *Then* $\mathcal{L}'$ *is constructive for* $(\mathfrak{N},\mathrm{id}_{\mathfrak{N}})$.

Theorem 20. *The calculus consisting of all instances of Markov's principle* $\forall x(\varphi\vee\neg\varphi)\supset(\neg\neg\exists x\varphi\supset\exists x\varphi)$ *is constructive for any enumerated model* $(\mathfrak{M},\nu)$.

Markov's principle is interesting because the intuitionistic arithmetic with Markov's principle is Turing complete: one can extract all computable functions from proofs in this calculus [Vor 91b]. We had already shown that in the parametric case there may be problems with the classical truth. However, we can still use *mixed inference rules* which combine classical and constructive provability, like *mixed Markov's principle* [Vor 90, Vor 91b]:

$$\frac{\vdash_{\mathrm{con}}\forall x(\varphi\vee\neg\varphi) \qquad \vdash_{\mathrm{cl}}\exists x\varphi}{\vdash_{\mathrm{con}}\exists x\varphi}$$

where $\vdash_{\mathrm{cl}}$ and $\vdash_{\mathrm{con}}$ means the classical and the constructive provabilities. We shall not define the semantics of such mixed inference rules here, but we shall prove constructiveness of a stronger form of Markov's principle [Vor 91b]:

Theorem 21. *Let $\mathcal{L}$ be a calculus consisting of any number of formulae of the form $\forall x(\varphi\vee\neg\varphi)\supset\exists x\varphi$, such that the formula $\exists x\varphi$ is true in the model $\mathfrak{M}$. Then $\mathcal{L}$ is constructive for $(\mathfrak{M}, \nu)$, where ν is an arbitrary enumeration of $\mathfrak{M}$.*

In the example from Section 2 we proved the existence of the function `mapcar`. The proof was constructed in the language of list theory [Vor 91a]. In the same way we can easily prove the existence of many other higher order functions. Actually we made such proofs for all built-in higher order functions of the programming language MULISP [MUL 89]. The other higher order functions of MULISP can be defined in MULISP by using higher order function `apply`, which can be defined by the equation `apply` $fx \;\Leftrightarrow\; fx$. In our approach it is a trivial exercise to prove the existence of `apply`: the following proof scheme, where f is a parameter, specifies `apply`:

$$\cfrac{\cfrac{f(x) = f(x)}{\exists y(f(x) = y)}}{\forall x\exists y(f(x) = y)}$$

We could also easily prove higher-order functions from the book of Sokolowski [Sok 89] however with several modifications, connected with the type system of Standard ML.

Lists comprise a very useful data type in functional programming. The list theory is important for specifying facts about higher order functions, so we shall formulate here a theorem about constructiveness of this theory. This theorem serves as a foundation of using proof schemes in the example from Section 2.

Theorem 22. *The intuitionistic list theory [Vor 91a] is constructive for the model of finite lists.*

Proof. It is sufficient to use functionals defined by recursion in [Vor 91a] as the realizers and to prove that they are computable elements of the information systems.

To stress the generality of our approach to extracting higher order programs from proofs we will formulate the constructiveness of *well-order induction.* Both list induction ListInd, *foundation axiom* for lists from [Vor 91a] and induction on natural numbers Ind can be derived from a very general induction scheme:

$$\frac{(\forall x)((\forall y\prec x)\Phi(y)\supset\Phi(x))}{(\forall x)\Phi(x)}$$

where $\prec$ is any Harrop formula without parameters representing a relation with no infinite descending chains $\ldots\prec a_1\prec a_0$. A similar induction scheme had been used in [KP 90], as well as in many mathematical books (see e.g. [Bar 75]). This induction scheme can be used to represent many kinds of recursion. It will be the core of the program synthesis system which is currently under development in the Laboratory of Intelligent Systems, Novosibirsk. The following theorem states that the instances of this induction scheme form a constructive calculus.

Theorem 23. *Let $(\mathfrak{M}, \nu)$ be an enumerated model and $\prec$ be a relation on M with the above properties. Then the calculus $\mathcal{L}$ consisting of all instances of the above induction scheme is constructive for $(\mathfrak{M}, \nu)$.*

Proof. For simplicity we consider only the case when φ has no other variables except x. Let a realizes the premise of the rule. We define functionals b, c by the following recursion:

$$b\ x \ \Leftrightarrow\ a\ x\ c$$
$$c\ y\ \bot\ \Leftrightarrow\ b\ y$$

We prove that b realizes the conclusion of the rule under any interpretation of parameters of φ. To this end we have to prove that for every x, $(b\ x)\,\underline{\mathfrak{cl}}\,\Phi(x)$, or $(a\ x\ c)\,\underline{\mathfrak{cl}}\,\Phi(x)$. Since the relation $\prec$ is well-ordered one can assume that for every $y \prec x$ we have $(b\ y)\,\underline{\mathfrak{cl}}\,\Phi(y)$. Since a realizes the premise of the rule, then $(a\ x)\,\underline{\mathfrak{cl}}\,\forall y \prec x \Phi(y) \supset \Phi(x)$. Thus it is sufficient to prove $c\,\underline{\mathfrak{cl}}\,\forall y \prec x \Phi(y)$, or that for every y, we have $(c\ y)\,\underline{\mathfrak{cl}}\,(y \prec x \supset \Phi(y))$. Consider two cases.

1. $y \prec x$. Then $\bot\,\underline{\mathfrak{cl}}\,y \prec x$ and $(b\ y)\,\underline{\mathfrak{cl}}\,\Phi(y)$. It remains to apply the definition of c.
2. $y \prec x$ is not true. In this case any element realizes $y \prec x \supset \Phi(y)$.

As one can see from the proof, the proof remains valid if the formula expressing the relation $\prec$ contains parameters, but satisfies the well-foundness property for each interpretation of the parameters.

Related work

There are not so many papers related to this paper. In most of the approaches type theories are used. The treatment of higher order functions in type theories is different - as a matter of fact there is no difference between functions and other objects in type theories. A very similar to ours treatment of constructive logics had been used in [Goa 80a, Goa 80b]. However in [Goa 80a, Goa 80b] all functions are interpreted and there is no formal definition of what is constructive logic in general. In [Hay 88] higher order functions can be simulated as follows. The definitions of functions are terms of the logic **PX** from [Hay 88]. There is also a function applying definitions of functions to arguments. In this way it is easy to represent any functions, expressible in pure LISP, including higher order functions. However the resulted logic is complicated compared to usual first order logic.

In the field of mathematical logic there were some similar to ours definition of realizability [NS 82, Pli 78, Pli 86]. However neither of them considered the problem of higher order functions. To this end the most interesting are variants of so called *uniform realizability* intensively studied in e.g. [Pli 78]. Intuitively, a formula containing predicate P is uniformly realizable iff there is a functional which realizes this formula independently of the concrete value of P. Thus the uniform realizability is very similar to the definition that we use. The essential difference is that the uniform realizability had been defined only for natural numbers, which does not allow to use its definition for data types different from natural numbers.

Acknowledgements

I am gratefully acknowledged to Susumu Hayashi and Michel Parigot for interesting discussions. A remark of Alan Bundy about higher order functions in type theories had also stimulated this work.

References

[Bas 89] D.A. Basin. Building theories in Nuprl. In *Logic at Botic'89*, volume 363 of *Lecture Notes in Computer Science*, pages 12–25, 1989.

[Cha 77] C.C.Chang and H.J.Keisler. *Model theory.* North Holland, 1977.

[Chi 87] P. Chisholm. Derivation of a parsing algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8(1):1–42, 1987.

[Con 86] R.L. Constable and e.a. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Coq 88] T. Coquand and G Huet. The calculus of constructions. *Information and computation*, 74(213):95–120, 1988.

[Ers 77] Yu.L. Ershov. *The Theory of Enumerations (in Russian)*. Nauka, Moscow, 1977.

[Ers 80] Yu.L. Ershov. *Decidability Problems and Constructive Models (in Russian)*. Nauka, Moscow, 1980.

[Goa 80a] C.A. Goad. Proofs as descriptions of computation. In *5th CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 39–52, 1980.

[Goa 80b] C.A. Goad. Computational uses of the manipulation of formal proofs. Technical Report TR no. STAN-CS-80-819, Stanford Univ. Department of Computer Science, 1980.

[Har 60] R Harrop. Concerning formulas of the types $A \to B \wedge C$, $A \to (Ex)B(x)$ in intuitionistic formal system. *J. of Symb. Logic*, 17:27–32, 1960.

[Hay 88] S. Hayashi and H. Nakano. **PX***: a Computational Logic*. MIT Press, 1988.

[Hen 92] M.C. Henson. Safe positive induction in the programming logic TK. In A.Voronkov, editor, *1st and 2nd Russian Conferences on Logic Programming*, Lecture Notes in Artificial Intelligence, pages 215–231. Springer Verlag, 1992.

[Bar 75] J.Barwise. *Admissible Sets and Structures*. Springer Verlag, 1975.

[Kle 52] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand P.C., Amsterdam, 1952.

[KP 90] J.-L. Krivine and M. Parigot. Programming with proofs. *J.Inf.Proces. Cybern. EIK*, 26(3):149–167, 1990.

[Nel 47] D. Nelson. Recursive functions and intuitionistic number theory. *Trans. Amer. Math. Soc.*, 61:307–368, 1947.

[NS 82] N.N. Nepeivoda and D.I. Sviridenko. *Towards the Theory for Program Synthesis (in Russian)*, volume 2 of *Trudy Instituta Matematiki*, pages 159–175. Nauka, Novosibirsk, 1982.

[Nor 84] B. Nordström and J.M. Smith. Propositions, types and specifications of programs in Martin-Löf's type theory. *BIT*, 24(3):288–301, 1984.

[Pli 78] V.E. Plisko. Some variants of realizability for predicate formulas. *Izvestiya Akademii Nauk SSSR, Mathematics*, 42(3):636–653, 1978.

[Pli 86] V.E. Plisko. Towards a model theory for languages with constructive semantics (in Russian). In *Soviet Conference on Applications of Mathematical Logic*, pages 154–156, Tallinn, 1986. Institute of Cybernetics.

[Pra 65] D. Prawitz. *Natural deduction*. Almquist and Wicksell, Stockholm, 1965.

[Sco 82] D.S. Scott. Domains for denotational semantics. In *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–612, 1982.

[MUL 89] Soft Warehouse Inc. *MULISP Reference Manual*, 1989.

[Sok 89] S. Sokolowski. Applicative high-order programming or standard ML in the battlefield. Technical report, Universität Passau, Fakultät für Mathematik und Informatik, 1989.

[SV 90] S.S. Starchenko and A. Voronkov. On connections between classical and constructive semantics. In *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, 1990.

[Vor 88] A.A. Voronkov. *Model theory based on a constructive notion of truth, (in Russian)*, volume 8 of *Trudy Instituta Matematiki*, pages 25–42. Nauka, Novosibirsk, 1988. This volume was cover-to-cover translated to English by American Mathematical Society.

[Vor 90] A.A. Voronkov. Towards the theory of programming in constructive logic. In N.Jones, editor, *Proc. of the European Symp. on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 421–435. Springer Verlag, 1990.

[Vor 91a] A.A. Voronkov. N-realizability: one more constructive semantics. Technical Report 71, Department of Mathematics, Monash University, 1991.

[Vor 91b] A.A. Voronkov. On completeness of program synthesis systems. Technical Report 29, Univ. Paris 7, Equipe de Logique Mathematique, 1991.