The formal development of non-trivial programs in constructive type theory

Silvio Valentini
Dipartimento di Matematica Pura ed Applicata
Università di Padova
via G. Belzoni n.7, I–35131 Padova, Italy
silvio@brouwer.math.unipd.it

Abstract

This paper is intended to show how non-trivial problems can be specified and solved in Martin-Löf's Type Theory. A particular class of problems is considered which contains some game problems.

Mathematics Subject Classification: 03B15, 03B20.
Keywords: Intuitionistic type theory, Program correctness, Software engineering.

1. Introduction

Since the 70s Martin-Löf has developed, in a number of successive variants, an Intuitionistic Theory of Types [7, 8, 10] (ITT for short in the following). The initial aim was to provide a formal system for constructive mathematics but the relevance of the theory also in computer science was soon recognized. In fact, from an intuitionistic perspective, to define a constructive set theory is equivalent to define a logical calculus [5] or a language for problem specification [6]. Hence the topic is of immediate relevance both to mathematicians, logicians and computer scientists. Moreover, since an element of a set can also be seen as a proof of the corresponding proposition or as a program which solves the corresponding problem, ITT is also a functional programming language with a very rich type structure and an integrated system to derive correct programs from their specification [9, 11]. These pleasant properties of the theory have certainly contributed to the interest for it arisen in the computer science community, especially among those people who believe that program correctness is a major concern in the programming activity [1].

Many are the peculiarities of the theory which justify this wide concern. As regards computing science, through very powerful type-definitions facilities and the embedded principle of "propositions as types" it primarily supplies means to support the development of proved-correct programs. Indeed here type checking achieves its very aim, namely that of avoiding *logical* errors.

The paper is organized as follows. We first recall some basic facts on type theory which can be useful to the reader which is not familiar with the topics. Many introduction on the theory can today be found in the literature (see for instance [8, 10, 12]) but we think that this introduction is sufficient to make the paper self contained. In section 7, we begin the description of the types which allow to express the problems we are interested in within type theory. First the type $Seq(A)$, of finite sequences of elements in A, and the type $Tree(A)$, of trees labeled with elements in A, are presented. Then $Graph(A)$, the type of finitary directed graphs, is introduced and it is shown how its elements can be viewed as laws for building up finite trees. In section 8 the problems we deal with are defined and their solutions developed. They are problems on "games" and clearly their specification strongly depends on how we describe a game. The results in sections 7 and 8 can be found also in [2] which contains detailed proofs of the theorems we only state here and the construction of some of the most basic programs.

2. Judgments and Propositions

In a classical approach to the development of a formal theory one usually takes care to define a formal language only to specify the syntax (s)he wants to use while as regard to the intended semantics no *a priori* tie on the used language is required. Here the situation is quite different; in fact we want to develop a *constructive* set theory and hence we can assume no *external* knowledge on sets; then we do not merely develop a suitable syntax to describe something which exists *somewhere*. Hence we have "to put our cards on the table" at the very beginning of the game and to declare the kind of judgments on sets we are going to express.

Let us show the form of the judgments we are going to use to develop our constructive set theory. The first is

$$\text{type-ness:} \quad A\ type$$

which states that A is a type. The second form of judgment is

$$\text{equal-types:} \quad A = B\ type$$

which, provided that A and B are types, states that they are equal types. The next form of judgment is

$$\text{belong-ness:} \quad a \in A$$

which states that a is an element of the type A and finally

$$\text{equal-elements:} \quad a = b \in A$$

which, provided that a and b are elements of the type A, states that a and b are equal elements of the type A.

3. Different readings of the judgments

A peculiar aspect of a constructive set theory is that we can give many different readings of the same judgment. Let us show some of them

A type	$\mathbf{a \in A}$
A is a set	a is an element of A
A is a proposition	a is a proof of A
A is a problem	a is a method which solves A

The first reading conforms to the definition of a constructive set theory. The second one, which links type theory with intuitionistic logic, is due to Heyting [4, 5]: it is based on the identification of a proposition with the set of its proofs. Finally the third reading, due to Kolmogorov [6], consists on the identification of a problem with the set of its solutions.

3.1. On the judgment $A\ set$

Let us explain the meaning of the various forms of judgment; it is convenient to commit to an epistemological approach.

What is a set? A set is defined by prescribing how its elements are formed.

Let us work out an example: the set $\mathcal{N}$ of natural numbers. We state that natural numbers form a set since we know how to construct its elements.

$$0 \in \mathcal{N} \qquad \frac{a \in \mathcal{N}}{succ(a) \in \mathcal{N}}$$

i.e. 0 is a natural number and the successor of any natural number is a natural number.

Of course in this way we only construct the *canonical* elements of the set of the natural numbers and we say nothing on elements like $3 + 2$. We can recognize also this element as a natural number if we understand that the operation $+$ is just a method such that, given two natural numbers, provides us, by means of some calculations, with a natural number in canonical form, i.e. $3 + 2 = succ(2 + 2)$; this is the reason why, besides the belong-ness judgment, we also need the equal-elements judgment. For the type of the natural numbers we put

$$0 = 0 \in \mathcal{N} \qquad \frac{a = b \in \mathcal{N}}{succ(a) = succ(b) \in \mathcal{N}}$$

Let us consider another example. Suppose that A and B are types, then we can construct the type $A \times B$, which corresponds to the cartesian product of the sets A and B, since we know what are its canonical elements:

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A \times B} \qquad \frac{a = c \in A \quad b = d \in B}{\langle a, b\rangle = \langle c, d\rangle \in A \times B}$$

So we explained the meaning of the judgment $A\ set$ but meanwhile we also explained the meaning of two other forms of judgment.

What is an element of a set? An element of a set A is a method which, when executed, yields a canonical element of A as result.

When are two elements equal? Two elements a, b of a set A are equal if, when executed, they yield equal canonical elements of A as results.

3.2. On the judgment $A\ prop$

We can now immediately answer to the question:*What is a proposition?*

Since a proposition is identified with the set of its proofs, in order to answer to this question we have only to repeat for propositions what we said for sets: a proposition is defined by laying down what counts as a proof of the proposition. Thus in the constructive approach to state that an expression is a proposition one has to state what (s)he is willing to accept as one of its proofs. Consider for instance the proposition $A\&B$: supposing A and B are propositions, then $A\&B$ is a proposition since we can state what is one of its proofs: a proof of $A\&B$ consists of a proof of A together with a proof of B, and so we put

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A\&B}$$

and then $A\&B \equiv A \times B$.

4. Hypothetical judgments

So far we have explained the basic ideas of ITT; now we want to introduce a formal system to treat with types. To this aim we need the notion of *hypothetical judgment*: a hypothetical judgment is a judgment expressed under assumptions. Let us explain what an assumption is; here we only give some basic ideas while a formal approach can be found in [3]. Let A be a type; then the assumption $x : A$ means both that the variable x has type A and that x is a hypothetical proof of A.

The previous is just the simplest form of assumption, but we will also use $y : (x : A)\ B$ which means that y is a function which maps an element $a \in A$ into the element $y(a) \in B$.

Now suppose A *type*; then the fact that B is a propositional function on elements of A can be stated by asserting the hypothetical judgment $B(x)\ prop\ [x : A]$ provided that we know what it counts as a proof of $B(a)$ for any element $a \in A$. For instance $x \neq 0 \rightarrow (\frac{3}{x} * x = 3)\ prop\ [x : \mathcal{N}]$ is a hypothetical judgment whose meaning is straightforward.

The previous is just the simplest form of hypothetical judgment. In general, supposing

$$\begin{array}{l} A_1\ type \\ A_2(x_1)\ type\ [x_1 : A_1] \\ \quad\vdots \\ A_n(x_1, \ldots, x_{n-1})\ type\ [x_1 : A_1, \ldots, x_{n-1} : A_{n-1}] \end{array}$$

we obtain the hypothetical judgment

$$J\ [x_1 : A_1, \ldots, x_n : A_n(x_1, \ldots, x_{n-1})]$$

where J is any one of the four forms of judgment we have considered.

5. The logic of types

We can now describe the full set of rules needed to describe one type. We will consider four kinds of rules: the first rule states the conditions required to *form* the type, the second one *introduces* the canonical elements of the type, the third rule explains how to use, and hence *eliminate*, the elements of the type and the last one how to *compute* using the elements of the type. For each kind of rules we will first give an abstract explanation and then we will show the actual rules for the cartesian product of two types.

Formation. How to form a new type and when two types constructed in this way are equal.

Example:

$$\frac{A\ type \quad B\ type}{A \times B\ type} \qquad \frac{A = C \quad B = D}{A \times B = C \times D}$$

Introduction. What are the canonical elements of the type and when two canonical elements are equal.

Example:

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A \times B} \qquad \frac{a = c \in A \quad b = d \in B}{\langle a, b\rangle = \langle c, d\rangle \in A \times B}$$

which state that the canonical elements of the cartesian product $A \times B$ are the couples whose first element is in A and second element is in B.

Elimination. How to define functions on the elements of the type defined by the introduction rules.

Example:

$$\frac{c \in A \times B \quad d(x, y) \in C(\langle x, y\rangle)\ [x : A, y : B]}{split(c, d) \in C(c)}$$

Equality. How to compute the function defined by the elimination rule.

Example:

$$\frac{a \in A \quad b \in B \quad d(x, y) \in C(\langle x, y\rangle)\ [x : A, y : B]}{split(\langle a, b\rangle, d) = d(a, b) \in C(\langle a, b\rangle)}$$

which states that to evaluate the function $split(c, d)$, defined by the elimination rule, one has first to evaluate c in order to obtain a canonical element $\langle a, b\rangle \in A \times B$ and then to use the method $d : (x : A)(y : B)\ C(\langle x, y\rangle)$, provided by the second premise of the elimination rule, to obtain the value $d(a, b) \in C(\langle a, b\rangle)$.

The same approach can be used to obtain the rules for a type which is not a *logical* proposition. Let us consider the case of the type $\mathcal{N}$.

Formation:

$$\mathcal{N}\ set$$

Introduction:

$$0 \in \mathcal{N} \qquad \frac{a \in \mathcal{N}}{succ(a) \in \mathcal{N}}$$

Elimination:

$$\frac{c \in \mathcal{N} \quad d \in C(0) \quad e(x, y) \in C(succ(x))[x : \mathcal{N}, y : C(x)]}{N_{rec}(c, d, e) \in C(c)}$$

Equality:

$$\frac{d \in C(0) \quad e(x, y) \in C(succ(x))\ [x : \mathcal{N}, y : C(x)]}{N_{rec}(0, d, e) = d \in C(0)}$$
$$\frac{a \in \mathcal{N} \quad d \in C(0) \quad e(x, y) \in C(succ(x))[x : \mathcal{N}, y : C(x)]}{N_{rec}(succ(a), d, e) = e(a, N_{rec}(a, d, e)) \in C(succ(a))}$$

We can also consider a new kind of rules which makes explicit the computation process which is only implicit in the equality rule.

Computation:

$$\frac{c \Rightarrow 0 \quad d \Rightarrow g}{N_{rec}(c, d, e) \Rightarrow g} \quad \frac{c \Rightarrow succ(a) \quad e(a, N_{rec}(a, d, e)) \Rightarrow g}{N_{rec}(c, d, e) \Rightarrow g}$$

5.1. A very simple program: the sum

We can develop programs on natural numbers. For instance, let $x, y \in \mathcal{N}$, then the sum of x and y is defined by means of the following deduction:

$$\frac{x \in \mathcal{N} \quad y \in \mathcal{N} \quad \dfrac{[v : \mathcal{N}]_1}{succ(v) \in \mathcal{N}}}{x + y \equiv N_{rec}(x, y, (u, v)\ succ(v)) \in \mathcal{N}}\ 1$$

This simple example suggests that ITT is a functional programming language with a strong typing system.

In general, the recursive equations with unknown f:

$$\begin{cases} f(0) = k \in A \\ f(succ(x)) = g(x, f(x)) \in A \end{cases}$$

is solved in ITT by $f(x) \equiv N_{rec}(x, k, (u, v)\ g(u, v))$ and, supposing $n : \mathcal{N}$, $k : A$ and $g : (u : \mathcal{N})(v : A)\ A$, it is possible to prove that $N_{rec}(n, k, (u, v)\ g(u, v)) \in A$.

6. Other basic types

Following the pattern we used in the previous section we can define many other types. For instance, consider the type $A \to B$ of the functions from A to B. Its formation rules are:

$$\frac{A\ type \quad B\ type}{A \to B\ type} \qquad \frac{A = C \quad B = D}{A \to B = C \to D}$$

The canonical elements of the type $A \to B$ are the functions $\lambda(b)$ such that $b(x) \in B\ [x : A]$; thus we put

$$\frac{b(x) \in B\ [x : A]}{\lambda(b) \in A \to B} \qquad \frac{b(x) = d(x) \in B\ [x : A]}{\lambda(b) = \lambda(d) \in A \to B}$$

The elimination rule is now determined; it states that the *only* elements of the type are those introduced by the introduction rule(s). Thus, besides the premise $c \in A \to B$, we have to consider only another premise, i.e. $d(y) \in C(\lambda(y))\ [y : (x : A)\ B]$ which shows how to obtain a proof of $C(\lambda(y))$ starting from a generic assumptions for the introduction rule.

$$\frac{c \in A \to B \quad d(y) \in C(\lambda(y))\ [y : (x : A)\ B]}{funsplit(c, d) \in C(c)}$$

In the following it will be handy to use also the derived rule

$$\frac{c \in A \to B \quad a \in A}{c[a] \equiv apply(c, a) \in B}$$

to mean the application of the function c to the element a where $apply(c, a) \equiv funsplit(c, (y)\ y(a))$.

In an analogous way we can introduce a new type which will play a main role in the following: the type $W(A, B)$ of the trees whose nodes are labeled with elements a of type A and whose branching width is determined by $B(a)$.

Formation:

$$\frac{A\ type \quad B(x)\ type\ [x : A]}{W(A, B)\ type}$$

Introduction:

$$\frac{a \in A \quad b(y) \in W(A, B)\ [y : B(a)]}{sup(a, b) \in W(A, B)}$$

The justification of the introduction rule is that the element $sup(a, b) \in W(A, B)$ is a node which has the label $a \in A$ and a branch which arrives at the tree $b(y) \in W(A, B)$ in correspondence with each element $y \in B(a)$.

As above, the elimination rule is completely determined by the introduction rule.

Elimination:

$$\frac{c \in W(A, B) \quad \begin{array}{c} [x : A, w : (y : B(x))W(A, B), z : (y : B(x))C(w(y))]_1 \\ \vdots \\ e(x, w, z) \in C(sup(x, w)) \end{array}}{T_{rec}(c, e) \in C(c)}\ 1$$

7. New types for new problems

In this section we will introduce some general types we will use in the next of the paper. Let us recall that, given $a \in \mathcal{N}$, the type $\mathcal{N}_<(a)$ is the type of all the natural numbers less than a such that $\mathcal{N}_<(0) = \emptyset$.

7.1. The set $Seq(A)$

Instead of introducing the type of the lists on A directly we will implement them as pairs since this approach makes the following easier. The first component of a pair is the length $n \in \mathcal{N}$ of the list and the second a function mapping an element $i \in \mathcal{N}_<(n)$ to the i-th element of the list. Thus, we put

$$Seq(A) \equiv (\exists x \in \mathcal{N})\ \mathcal{N}_<(x) \to A$$

and, supposing $a \in A$ and $s \in Seq(A)$, we make the following definitions:

$$\begin{array}{rcl} nil & \equiv & \langle 0, empty \rangle \\ a \bullet s & \equiv & \langle succ(fst(s)), \\ & & \quad \lambda x.\ \text{if } x <^{\mathcal{N}} fst(s) \text{ then } snd(s)[x] \text{ else } a \rangle \end{array}$$

which suggest that, supposing $C(x)\ prop\ [x : Seq(A)]$, $d \in C(nil)$ and $e(x, y, z) \in C(x \bullet y)\ [x : A, y : Seq(A), z : C(y)]$,

$$\begin{array}{l} L_{rec}(s, d, e) \equiv \\ \qquad split(s, (n, f)\ N_{rec}(n, d, (u, v)\ e(f[u], \langle u, f \rangle, v))) \end{array}$$

is a correct proof-method to find a proof of $C(s)$.

We will use the abbreviations

$$\begin{array}{rcll} \sharp s & \equiv & fst(s) & \text{for the length of } s \text{ and} \\ s\{i\} & \equiv & snd(s)[i] & \text{for the } i\text{-th element of } s \end{array}$$

If $a \in A$ and $s \in Seq(A)$ then the proposition $a\ InSeq\ s$ holds if and only if a is an element of the sequence s. Then the following equations have to hold

$$\begin{cases} a\ InSeq\ nil & = & \perp \\ a\ InSeq\ b \bullet t & = & (a =^A b) \vee (a\ InSeq\ t) \end{cases}$$

and their solution is

$$a\ InSeq\ s \equiv L_{rec}(s, \perp, (x, y, z)\ (a =^A x)\ \vee\ z)$$

To filter out, from a given sequence s, all the elements which do not satisfy a given condition $f(x) \in Bool\ [x : A]$ we define the function

$$filter(f, s) \in Seq(A)\ [f : (x : A)\ Bool, s : Seq(A)];$$

the recursive equations for the function $filter$ are

$$\begin{cases} filter(f, nil) = nil \\ filter(f, a \bullet s) = \\ \qquad \text{if } f(a) \text{ then } a \bullet filter(f, s) \text{ else } filter(f, s) \end{cases}$$

and they can be solved by making the explicit definition

$$\begin{array}{l} filter(f, s) \equiv \\ \quad L_{rec}(s, nil, (x, y, z) \text{ if } f(x) \text{ then } x \bullet z \text{ else } z) \end{array}$$

Theorem 7.1 *If $a \in A$, $s \in Seq(A)$, $f(x) \in Bool\ [x : A]$, then the proposition $a\ InSeq\ filter(f, s)$ is true if and only if $(f(a) =^{Bool} true)$ and $(a\ InSeq\ s)$.*

7.2. The set $Tree(A)$

The set $Tree(A)$ is the set of all finite trees whose nodes are labeled with elements in the set A. As we already said, the well-ordering type has labeled trees of finite depth as elements, so to obtain finite trees we have only to add the constrain that any node has a finite set of predecessors. We make the following definition

$$Tree(A) \equiv W(A \times \mathcal{N}, (x)\ \mathcal{N}_<(snd(x)))$$

and thus $Tree(A)$ is the set of finitely branched trees with nodes of the form $\langle a, n\rangle$ where a is an element in A and n is the number of immediate successors. For instance, the singleton tree with only one node labeled by $a_0 \in A$, can be defined as

$$leaf(a_0) \equiv sup(\langle a_0, 0\rangle, empty).$$

Let $t \in Tree(A)$; then the root of the tree t and its i-th subtree are defined by

$$\begin{array}{rcl} root(t) & \equiv & T_{rec}(t, (x, y, z)\ x) \\ subtree(t, i) & \equiv & T_{rec}(t, (x, y, z)\ y(i)) \end{array}$$

A useful function on trees is the function

$$travel(t) \in Seq(A)\ [t : Tree(A)]$$

which associates to a given tree t the sequence of all the labels present in t. It follows the recursive equations

$$travel(sup(\langle a, n\rangle, f)) = a \bullet append(n, (i)\ travel(f(i)))$$

where the function

$$append(n, f) \in Seq(A)\ [n : \mathcal{N}, f : (i : \mathcal{N}_<(n))\ Seq(A)]$$

associates to n sequences on A the sequence obtained by appending them. Hence, supposing $append2(s_1, s_2) \in Seq(A)\ [s_1, s_2 : Seq(A)]$ is the standard function to append two sequences, we put

$$\begin{array}{l} append(n, f) \equiv N_{rec}(n, nil, (x, y)\ append2(f(x), y)) \\ travel(w) \equiv \\ \qquad T_{rec}(w, (x, y, z)\ fst(x) \bullet append(snd(x), z)) \end{array}$$

Supposing $a \in A$ and $t \in Tree(A)$, we can use the proposition $a\ InSeq\ s$ to define the proposition $a\ InTree\ t$ that holds if and only if a is the label of a node in t.

$$a\ InTree\ t \equiv a\ InSeq\ travel(t)$$

Suppose now that $b(x) \in Bool\ [x : A]$ is a boolean function, and that we are interested to know if in the finite tree t there is a node, whose label is a, such that $b(a) =^{Bool} true$. The recursive equation for a solution of this problem is

$$(find(sup(\langle a, n\rangle, h), b) = b(a)) \vee \bigvee(n, (i)\ find(h(i), b))$$

where the value of the function $\bigvee(n, f) \in Bool\ [n : \mathcal{N}, f : (i : \mathcal{N}_<(n))\ Bool]$ is the disjunction of n elements. The solution of the previous equation is

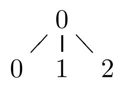
$$find(w, b) \equiv T_{rec}(w, (x, y, z)\ b(fst(x)) \vee \bigvee(snd(x), z)).$$

Theorem 7.2 *If $t \in Tree(A)$ and $b(x) \in Bool\ [x : A]$, then the proposition $(find(t, b) =^{Bool} true)$ is true if and only if $(\exists x \in A)\ (f(x) =^{Bool} true)\ \&\ (x\ InTree\ t)$.*

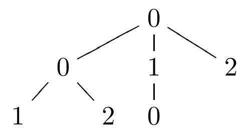
7.3. Expanding a finite tree

Let $preds \in A \to Seq(A)$ and $t \in Tree(A)$. We want to define a function $expand(preds, t)$ whose value is the tree t expanded in all leaves, $leaf(a)$, such that the subtrees of a are all singleton trees with nodes whose labels are taken from the sequence $preds[a]$. For instance, if $preds \in \mathcal{N} \to Seq(\mathcal{N})$ maps 0 to the sequence $1 \bullet 2 \bullet nil$, 1 to the sequence $0 \bullet nil$ and 2 to the sequence nil, and if $t \in Tree(\mathcal{N})$ is the tree

$$sup(\langle 0, 3\rangle, \lambda x.\ leaf(x))$$



then $expand(preds, t)$ will be the tree



The following recursive equations holds for the function $expand$

$$\begin{cases} expand(preds, sup(\langle a, 0\rangle, f)) = \\ \qquad sup(\langle a, \sharp preds[a]\rangle, (i)\ leaf(preds[a]\{i\})) \\ expand(preds, sup(\langle a, succ(n)\rangle, f)) = \\ \qquad sup(\langle a, succ(n)\rangle, (i)\ expand(preds, f(i))) \end{cases}$$

This equations can be solved in type theory by

$$\begin{array}{ll} expand(preds, t) \equiv & \\ \quad T_{rec}(t, (x, y, z) & \text{if } snd(x) \simeq^{\mathcal{N}} 0 \\ & \text{then } sup(\langle fst(x), \sharp preds[fst(x)]\rangle, \\ & \qquad (i)\ leaf(preds[fst(x)]\{i\})) \\ & \text{else } sup(x, z)) \end{array}$$

7.4. Finitary Graphs

We will identify a finitary graph on A with a function which maps an element to its neighbors:

$$Graph(A) \equiv A \to Seq(A)$$

A tree t is expanded with respect to a graph g if the children of any node a in the tree are exactly the neighbors of the node a in the graph, i.e. the following equality holds for the predicate $Expanded$:

$$\begin{array}{l} Expanded(g, sup(\langle a, n\rangle, f)) = (\sharp g[a] =^{\mathcal{N}} n)\ \& \\ (\forall i < n)\ ((g[a]\{i\} =^{A} root(f(i)))\ \&\ Expanded(g, f(i))) \end{array}$$

This equation can be solved by transfinite recursion. The following set will be useful:

$$\begin{array}{l} ExpandedTree(A, g, a) \equiv \\ \quad \{t \in Tree(A) |\ Expanded(g, t)\ \&\ root(t) =^{A} a\} \end{array}$$

i.e. the set of all the expanded trees with label in A and root a. The first observation is that there is at most one element in $ExpandedTree(A, g, a)$.

Theorem 7.3 *If A is a set, $g \in Graph(A)$, $a \in A$ then there is at most one element in $ExpandedTree(A, g, a)$.*

Note that if for some $m \in \mathcal{N}$

$$(expand(g))^{m}(leaf(a)) = (expand(g))^{m+1}(leaf(a))$$

then $(expand(g))^{m}(leaf(a))$ is the "only" tree in $ExpandedTree(A, g, a)$.

Indeed, it is possible to show that the existence of such an m is also a necessary condition for $ExpandedTree(A, g, a)$ being inhabited.

Theorem 7.4 *If A is a set, $a \in A$, $g \in Graph(A)$ then*

$$\begin{array}{l} (\forall t \in ExpandedTree(A, g, a))(\exists k \in \mathcal{N}) \\ \quad t =^{Tree(A)} (expand(g))^{k}(leaf(root(t)))) \end{array}$$

8. Games and Games trees

8.1. Game description

When we want to explain a game to somebody we often start by describing the states of the game. Then we describe the moves of the game, i.e. we describe all the different ways a game can continue from one state to the next. Finally we describe the initial state and how to recognize a winning state. If there is only one player that is all, if there are two or more players let us think that the information about who is the player in turn is part of the state of the game and that, for each player, we can explain if a state is winning for him or not. We will consider games which are characterized by the following entities:

$$\begin{array}{l} \langle numPlayers, State, s_0, \sharp, next, \\ \qquad\qquad playerInTurn, winning\rangle \end{array}$$

where

$numPlayers \in \mathcal{N}$	the number of players;
$State\ set$	the set of states of the game;
$s_0 \in State$	the initial state;
$\sharp(s) \in \mathcal{N}$ $[s : State]$	the number of alternative moves in the state s;
$next(s, i) \in State$ $[s : State,$ $i : \mathcal{N}_{<}(\sharp(s))]$	the state after making the i-th alternative move in the state s;
$winning(s, k) \in Bool$ $[s : State,$ $k : \mathcal{N}_{<}(numPlayers)]$	is $true$ if the state s is a winning state for the k-th player, false otherwise;
$playerInTurn(s)$ $\in \mathcal{N}_{<}(numPlayers)$ $[s : State]$	the number of the next player to make a move in the state s.

The course of a game starts in an initial state s_0. It proceeds by moving from one state to the next according to the rules of the game. The state of the game is an instantaneous description of the game. In choosing this way of characterizing a game, we have made the following restrictions:

1. The number of alternative moves in a certain state s is always finite and it can be computed from s.
2. All possible next states can be computed from the current state.
3. It is decidable if a state is winning for a player or not.

The reasons for choosing the first two restrictions is that we want to have a general algorithm which decides if it is possible to reach a winning position or not. The third restriction is not a severe restriction; it is just a concrete way of expressing that there should be no doubt if a game is won or not.

8.2. The set of game trees.

Let $Game$ be the game

$$\langle numPlayers, State, s_0, \sharp, next, \\ playerInTurn, winning\rangle$$

Then the functions $\sharp$ and $next$ define a graph, namely

$$g_0 \equiv \lambda s.\ \langle \sharp(s), \lambda i.\ next(s,i)\rangle$$

This is the graph which defines the rules of the game in the sense that the neighbors of any node s in the graph are exactly all possible states immediately following s in the game. In section 7.4 we saw how to associate a tree with a graph: the set $ExpandedTree(State, g_0, s_0)$ contains at most one element, i.e. the game tree associated with $Game$. This is a tree with a game state in each node and the initial state s_0 in the root. Furthermore, for each node s in the tree, the i-th child of s is the resulting state after choosing the i-th alternative move from s. We saw also (cfr. theorem 7.4) that if there is a game tree, it can be written as $(expand(g_0))^m(leaf(s_0))$ for some $m \in \mathcal{N}$.

8.3. Some general game problems

Given a game there are many different questions we could ask. For instance, let us consider the following ones:

1. Is it possible for the k-th player to win the game?
2. Is there a winning strategy for the k-th player?
3. Let s be a situation. Give me the list of all the winning moves for the k-th player.

It should be clear that all these questions can be formalized by using the set $ExpandedTree(State, g_0, s_0)$.

As a first example, let us consider the question: "Is it possible for the k-th player to win the game?" We have to find a boolean function which associates to the k-th player $true$ if and only if there exists the game tree and it has a node labeled with a state which is winning for the k-th player. The problem can be expressed by the following type:

$$(\forall k < numPlayers)\ \{x \in Bool|\ (x = true) \leftrightarrow \\ (\exists g \in ExpandedTree(State, g_0, s_0)) \\ HasWinning(g,k)\}$$

where

$$HasWinning(g,k) \equiv \\ (\exists a \in State)\ (a\ InTree\ g)\ \&\ (winning(a,k) = true).$$

To solve the second problem we have to find a boolean function which associates to the k-th player $true$ if and only if there exists the game tree and it is winning for the k-th player. A game tree is winning for the player k if:

1. k is in turn
   - and the situation of the game is winning for k or
   - there exists a move for k to a tree that is winning for k
2. k is not in turn
   - and the situation of the game is winning for k or
   - the situation is not terminal and every move that the player in turn can do turns out in a game tree that is winning for k.

Then the recursive equation defining the proposition

$$IsWinningTree(t,k)\ [t : Tree(State), k < numPlayers]$$

is:

$$\begin{aligned} & IsWinningTree(sup(\langle a,n\rangle, f), k) = \\ & \quad \text{if } (playerInTurn(a) = k) \\ & \quad \text{then } (winning(a,k) = true) \vee \\ & \qquad (\exists i < n)\ IsWinningTree(f(i),k)) \\ & \quad \text{else } (winning(a,k) = true) \vee \\ & \qquad ((n \neq 0)\ \&\ (\forall i < n)\ IsWinningTree(f(i),k)) \end{aligned}$$

This recursive equation can be solved in type theory by a standard technique; then the second problem can be expressed as:

$$(\forall k < numPlayers)\ \{x \in Bool|\ (x = true) \leftrightarrow \\ (\exists t \in ExpandedTree(State, g_0, s_0)) \\ IsWinningTree(t,k)\}$$

To solve the problem "Give me the list of all the winning moves for the k-th player in a situation s" we have to find a function which associates to a player k and a state s the sequence of the winning moves for the k-th player from the state s. This is an element w of $Seq(\mathcal{N}_<(\sharp s))$ which satisfies the condition:

$$\begin{aligned}&(\exists t \in ExpandedTree(State, g_0, s))(\forall i < \sharp s)\\ &\quad (i\, InSeq\, w) \leftrightarrow IsWinningTree(subtree(t,i),k)\end{aligned}$$

Thus the problem is expressed by the type:

$$\begin{aligned}&(\forall k < numPlayers)(\forall s \in State)\ \{w \in Seq(\mathcal{N}_<(\sharp s))|\\ &(\exists t \in ExpandedTree(State, g_0, s))\ (\forall i < \sharp s)\\ &(i\, InSeq\, w) \leftrightarrow IsWinningTree(subtree(t,i),k)\}\end{aligned}$$

8.4. Some general solutions

We can observe that in the types expressing the three considered problems there is always a construction of the form:

$$(\exists t \in ExpandedTree(State, g_0, s_0))\ Q(t)$$

for a suitable proposition $Q(t)$.

A consequence of theorem 7.4 is that

$$\begin{aligned}&(\exists t \in ExpandedTree(State, g_0, s_0))\ Q(t) \leftrightarrow\\ &(\exists m \in \mathcal{N})\\ &(expand(g_0))^{m+1}(leaf(s_0)) = (expand(g_0))^m(leaf(s_0))\\ &\&\ Q((expand(g_0))^m(leaf(s_0)))\end{aligned}$$

Hence whenever $\bar{t} \equiv (expand(g_0))^m(leaf(s_0))$ we can give equivalent formalizations of the problems in the previous section.

1. $(\forall k < numPlayers)$ $\{x \in Bool|\ (x = true) \leftrightarrow HasWinning(\bar{t},k)\}$
2. $(\forall k < numPlayers)$ $\{x \in Bool|\ (x = true) \leftrightarrow IsWinningTree(\bar{t},k)\}$
3. $(\forall k < numPlayers)(\forall s \in State)$ $\{w \in Seq(\mathcal{N}_<(\sharp s))|\ (\forall i < \sharp s)\ (i\, InSeq\, w) \leftrightarrow$ $IsWinningTree(subtree(\bar{t},i),k)\}$

A general solution of problem (1) is then given by the function

$$\lambda k.\ find((expand(g_0))^m(leaf(s_0)), (s)\ winning(s,k))$$

since we proved (see theorem 7.2) that

$$\begin{aligned}&find(t,p) \in \{x \in Bool|\ (x = true) \leftrightarrow\\ &\quad (\exists a \in State)\ (a\, InTree\, g)\ \&\ (p(a) = true)\}\end{aligned}$$

For problem (2) we can easily find a function

$$\begin{aligned}&winningTree(t,k) \in Bool\\ &\qquad [t : Tree(State), k < numPlayers]\end{aligned}$$

satisfying the condition:

$$\begin{aligned}&(winningTree(t,k) =^{Bool} true) \leftrightarrow\\ &\qquad IsWinningTree(t,k).\end{aligned}$$

Its recursive definition mimics the definition of the type $IsWinningTree$ and can be solved in a similar way. Hence

$$\lambda k.\ winningTree((expand(g_0))^m(leaf(s_0)),k)$$

is a general solution of the problem (2).

Finally, a general solution for the problem (3) is

$$\lambda k.\ filter((i)\ winningTree(subtree(t,i),k), \langle \sharp a, \lambda x.\ x\rangle)$$

whose correctness follows from Theorem 7.1.

References

[1] R. Backhouse, P. Chisholm, G. Malcom, and E. Saaman. Do-it-yourself type theory. *Formal Aspect of Computing*, 1:19–84, 1989.

[2] A. Bossi, B. Nordstrom, and S. Valentini. General program specifications in Martin-Löf's type theory. Technical Report 7, Dip. Matematica Pura e Applicata, 1988.

[3] A. Bossi and S. Valentini. An intuitionistic theory of types with assumptions of high-arity variables. *Annals of Pure and Applied Logic*, 57:93–149, 1992.

[4] A. Heyting. *Intuitionism, an introduction*. North-Holland, Amsterdam, 1956.

[5] W. Howard. The formula-as-types notion of construction. In R. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[6] A. Kolmogorov. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

[7] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, IV*, pages 153–175, Amsterdam, 1982. 6th International Congress, North-Holland.

[8] P. Martin-Löf. *Intuitionistic Type Theory, notes by G. Sambin of a series of lectures given in Padua*. Bibliopolis, Naples, 1984.

[9] B. Nordström. Programming in constructive set theory: some examples. In *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 141–153, 1981.

[10] B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's Type Theory, An introduction*. Clarendon Press, Oxford, 1990.

[11] K. Peterson and J. Smith. Program derivation in type theory: a partitioning problem. *Comput. Languages*, 11 (3/4):161–172, 1986.

[12] S. Valentini. Another introduction to Martin-Löf's intuitionistic type theory. In *Workshop on new Trands in Theoretical Informatics*, pages 107–126. Innsbruck University Publication, 1996.