Constructive Mathematics

and Computer-assisted Reasoning Systems

Susumu Hayashi [1]

Research Institute for Mathematical Sciences
Kyoto University,
Sakyo, Kyoto, Japan

1. Introduction

The relationship between constructive mathematics and computer science has been noticed for a long while. But it is rather recent that live interactions between constructive mathematics and computer science began. In theoretical aspects, metamathematics of constructive mathematics are utilized to give semantics of higher order functional programming languages. Researches of constructive mathematics in computer science are not just theoretical. Quite a few systems based on constructive mathematics have been designed and implemented. They are program verification systems, program extraction systems, and high-level programming languages. These developments of systems need theoretical investigations as well and so give new insights into constructive mathematics. Applications of constructive mathematics to semantics of programming languages have begun to produce significant feedbacks to constructive mathematics and related areas, see Hyland (1987), Longo and Moggi (1988+α). Although researches on constructive systems have not yet been used to unvail new aspects of constructive mathematics, there are clues of possible applications of computer systems to theoretical researches in constructive mathematics Coquand (1986), Howe (1987). It is reasonable to imagine that constructive mathematicians a decade later will be helped by computer systems as mathematicians are now helped by computer algebra systems like REDUCE. In this paper, we will give a survey of existing systems based on constructive mathematics to show the readers how constructive theories are used in actual systems.

Unfortunately my knowledge about works in Eastern bloc countries is quite limited, I regret that works in Eastern bloc countries are rather neglected in this survey.

We would like to thank Mariko Yasugi for comments on the paper.

2. Curry-Howard isomorphism

The basic principle of constructive mathematics in computer science is Curry-Howard isomorphism. We will explain it briefly. Heyting emphasized that activities in constructive mathematics are activities to construct mathematical entities. We are not God. We are finite human beings. Don't care about the paradise in the air which belongs not to us but to God. This philosophy underlies in every constructive philosophy in mathematics. A mathematician who is a finite human being must be able to construct entities when we say they exist. In Brouwer's philosophy, such a construction *must* take place in a mathematician's mind. This gives a mysterious flavor to intuitionism. Science existing only in one's mind? How subjective it is! Hetying partly blew off this mysterious mist of Brouwer's philosophy by the concept "construction." But, in his age, constructions in mathematics, which are highly abstract and intricated, could take place only in human's minds. Modern digital computers have changed the situation. Actual constructions may take place on physical machines.

[1] This paper was written while the author was visiting LFCS, Department of Computer Science, University of Edinburgh. This work is supported by Japanese Organization of Promotion of Science.

Heyting (1971) explains that the implication $A \supset B$ is asserted, if and only if we possess a construction r, which, joined to any construction proving A, would automatically effect a construction proving B. In other words, a construction of $A \supset B$ is a function mapping a construction of A to a construction of B. If P^* is the set of constructions of proposition P, then we can identify $(A \supset B)^*$ with function space $A^* \to B^*$. Similarly $(A \wedge B)^*$ is identified with cartesian product $A^* \times B^*$. Furthermore, the constructions of quantified formulas $(\forall x : A.B(x))^*$ and $(\exists x : A.B(x))^*$ are identified with *dependent types* $\Pi x \in A^*.B(x)^*$ and $\Sigma x \in A^*.B(x)^*$ defined as

$$\Pi x \in T_1.T_2(x) = \{f | f(x) \in T_2(x) \text{ holds for all } x \in T_1\},$$
$$\Sigma x \in T_1.T_2(x) = \{< x, y > | x \in T_1,\ y \in T_2(x)\}.$$

These function spaces, cartesian products, and dependent types are used in programming languages and module systems as data types. This correspondence between types and formulas (or proposition) are called Curry-Howard isomorphism, or "propositions as types" or "formulas as types" principle. Since a formula corresponds to a type, a program which computes a data of type A^* corresponds to a construction of formula A. For example, a construction of an implication may be regarded as a program which computes a data of B from a data of A. Since a construction of a formula is identified with a proof of the formula, we have the equation

$$\text{proofs} = \text{constructions} = \text{programs}.$$

Hence Curry-Howard isomorphism is also referred as "proofs as programs" principles. All in all, programming corresponds to constructive mathematics, hence the principle is also called as "constructive mathematics as programming language." Detailed explanations of Curry-Howard isomorphisms can be found in Howard (1980), Martin-Löf (1982).

3. Early developments

We will mainly concern the recent studies, but we will mention some early developments both in computer science and logic. The historical account below will give the reader a general idea how the present systems evolved, although it is far from being complete.

3.1. Early developments in computer science

The researches related to constructive mathematics in computer science began, when de Bruijn started AUTOMATH project. In the project, many versions of typed lambda calculus are designed and implemented to check mathematics on machines (see a survey de Bruijn (1980) on the project). A version of AUTOMATH has checked the entire book of Landau's "Grundlagen der Analysis". Although, it is used as a proof checker for classical mathematics, it is a system of Curry-Howard isomorphism and so it is considered as a forerunner of recent constructive type theories, e.g., Calculus of Construction.

The deductive approach to program synthesis of Manna and Waldinger (1971) is another forerunner. They proposed to synthesize a program from a specification by proving the specification mechanically. Although they used classical logic, this idea was taken by Constable (1971) as a basis of "constructive mathematics as a programming language." It is apparent that some constructivists were aware of the possibility of using constructive mathematics as a programming language, but Constable seems the first person who took this idea seriously. Later, he and his team implemented Nuprl (1986) based on Martin-Löf's type theory, which is one of the most sophisticated proof checkers ever built.

Influenced by Constable (1971), Goto began to search for procedures which can realize program extraction. (Constable (1971) did not include any description of the working algorithm.) Constable (1971) criticized classical approach in Manna and Waldinger (1971). Goto (1975) tried to apply realizability interpretations to solve some anomaly of the classical approach. Later he shifted to Gödel interpretation and implemented a system based on HA (Goto, 1977, 1979). His system extracted some small LISP programs from proofs of HA. Although the extracted programs were tiny, they were rather natural programs. Sato (1979) also investigated program extraction by Gödel interpretation.

In Soviet, Nepeĭvoda (1978) gave an algorithm of program extraction for realistic programming languages, e.g., LISP, Algol, for a class of proofs.

C. Goad's thesis (Goad, 1980) is one of the most significant early works. Goad implemented an interpreter which execute first order proofs. His interpreter avoided to reduce proofs of lemmata which are Harrop formulas. His interpreter could also avoid unnecessary case distinctions looking up conditions. The second optimization increased efficiency dramatically for some cases. He reported that an application of his method to the bin packing problem increased efficiency dramatically. This could be done, because the interpreter could use logical informations in a proof In execution (reduction) of the proof. This showed "proofs as programs" approach may be used for not only verification but

also optimization. Goad also used "recursive proofs" or "circular proofs" A recursive proof is a proof whose lemma may be the same as its conclusion. Such a lemma is specially labeled to indicate that it is a recursive call of the conclusion. Since recursive proofs represent recursive definitions of functions, it may fail to terminate in a normal form. So total correctness was not guaranteed unlike the other systems of "proofs as programs." But, even if a circular argument is used in a proof, a proof may terminate in a normal form which has no recursive calls of conclusion. In a sense, his system guranteed partial correctness of recursive proofs as recursive programs. But he left the problem of verification of termination unsettled. (Cf. 4.3. below.)

Aside from these studies which are explicitly and intentionaly related to constructive mathematics, some very important studies which are implicitly related to constructive mathematics were done by Reynolds and Milner. Reynolds (1974) coined a system of second order lambda calculus for study of types in module programming languages, and Milner introduced a metalanguage ML for a proof checker Edinburgh LCF (Gordon, et al., 1979). These languages had "unusual" function spaces known as "polymorphic function spaces" which are not interpretable by classical function spaces. Later these works turned out to be closely related to Girard's functional system F^ω described below.

3.2. Early developments in logic

In the last two decades, studies of constructive mathematics made a great advance. Bishop (1967) showed that a fairly big portion of analysis can be carried out in a constructive way without exotic principles like Church's axiom and the continuity principle. After his work, logicians tried to find out correct formal theories in which Bishop's mathematics can be formalized. Theories used to formalize Bishop's mathematics can be classified in three groups, Martin-Löf's type theory (Marin-Löf, 1973), Myhill-Friedman's constructive set theories (Myhill, 1975; Friedman, 1977) and Feferman's theories of operations and classes (Feferman, 1975, 1979). (Martin-Löf's theory was not designed to formalize Bishop's constructive mathematics, but, at least, it was one of the main applications of his theory. Furthermore, it is Bishop's work that makes possible to claim that Martin-Löf's theory is enough to formalize a great portion of mathematics.) These theories were predicative so that their proof theoretic strength were very weak. They demonstrated that Bishop's constructive analysis is formalizable in those predicative theories. Even theories which are conservative extensions of constructive first order arithmetic HA were enough to formalize it (cf. Beeson, 1985). (Warning: This result depends on particular ways of formalization of mathematical concepts. Constructive logic and predicative theories cannot prove some equivalent definitions of basic concepts like compactness of topology.)

On the other hand, metamathematical study of impredicative theories also made a great advance. Girard (1972) introduced a higher order functional system F^ω and extended Gödel interpretation to intuitionistic higher order logic (IHOL). He proved strong normalization theorem of F^ω. Girard and Troelstra gave a model of F^ω, which was later used by Moggi to give a semantics of polymorphic lambda calculus (Longo and Moggi, 1988+α). Friedman and Myhill, e.g., Friedman (1973), investigated metamathematics of intuitionistic Zermelo-Frankel set theory (IZF).

Aside from these, even earlier, some attempts formulating the concept of construction were made by Scott (1970) and some others. These are regarded as forerunners of Martin-Löf's type theory.

All of these works have computational significance. In the rest of the paper, we will explain how they are related to computer science.

4. The new generation systems

4.1. Martin-Löf's type theory as a programming language

Martin-Löf (1982) introduced a new version of his type theory and related it to programming languages. Martin-Löf's type theory is a logic free theory of typing expressions. An expression denotes a value of a data type. When an expression e has a value of data type A, it is denoted as $e \in A$. This form is called a judgement. (Strictly speaking, A itself is an expression which denotes a type.) Data types of Martin-Löf's type theory is so rich that constructive logic can be coded by means of Curry-Howard isomorphism. As Bishop identified a set as a base set with an equality on it, every type is equipped with an extensional equality on it. When two expressions e_1 and e_2 of type A are extensionally equal, Martin-Löf writes as $e_1 = e_2 \in A$. This is called an equality judgement. Besides this, Martin-Löf's type theory has a type called propositional equality or equality type. If two expressions e_1 and e_2 are of a type A, then $Eq(A,e_1,e_2)$ is a type which has a value $\mathbf{r}$ if and only if $e_1 = e_2 \in A$ holds. Namely the rules

$$\frac{\mathbf{r} \in Eq(A,a,b)}{a = b \in A}, \qquad \frac{a = b \in A}{\mathbf{r} \in Eq(A,a,b)}$$

hold for the equality type. By this type, equality of each type can be interpreted as a type. So constructive logic can be interpreted by Curry-Howard isomorphism. Martin-Löf pointed out that his

theory can be considered as a type inference system for a functional programming language. Actually, his typing rules are closely related to typing rules of some typed functional languages, e.g., ML, Pebble (Burstall and Lampson, 1984), and second order lambda calculus. Furthermore, Martin-Löf's equality type enabled to represent a specification of a program as a type. For example, a specification of a program which finds a prime larger than a natural number m is given by

$$\Pi m \in N.\Sigma p \in N.Eq(N, test(p, m), 1), \tag{1}$$

where $test(p, m)$ is 1 if and only if p is prime greater than m. If f has this type, $f(m)$ is defined and its value is $\langle p, \mathbf{r}\rangle$ where p is a prime greater than m. Namely verifying that f satisfies the specification can be done by type checking of f. Furthermore, a program development of f can be done by a proof development of the formula

$$\forall m \in N.\exists p \in N.test(p, m) = 1.$$

This synthesis of type checking, verification, and rigorous program development received attentions from computer scientists. We will look at two efforts of development of Martin-Löf's idea in the next two sections.

4.2. Göteborg type theory

A group at Göteborg, Sweden, is developing a type theory and program methodology based on Martin-Löf's type theory as well as its computer implementation. (We will call their type theory as Göteborg type theory.) They follow Martin-Löf rather faithfully. One important departure from the original Martin-Löf type theory is the introduction of a subset type. In the example of "there is a greater prime" above, a function f of the type (1) returns a value $\langle p, \mathbf{r}\rangle$, whose second component $\mathbf{r}$ is redundant. So the type (1) is not quite the specification we have in our mind. Furthermore, if $Eq(N, test(p, m), 1)$ were a very complicated type, it would be costly to compute the unnecessary second component. To solve this problem, Nordström and Petersson (1983) introduced a subset type If A is a type and $B(a)$ is a type depending on a variable a of the type A, then the subset of A of the elements a for which $B(a)$ is inhibited is denoted by

$$\{a \in A | B(a)\}.$$

By this concept of subset type, we modify the specification of (1) as

$$\Pi m \in N.\{p \in N | Eq(N, test(p, m), 1)\}. \tag{2}$$

A function of this type computes a prime which is greater than m.

The rules for the subtype are

$$\frac{a \in A \qquad b \in B(a)}{a \in \{x \in A | B(x)\}}, \qquad \frac{a \in \{x \in Z | B(x)\} \qquad c(x) \in C(x)[x \in Z, y \in B(x)]}{c(a) \in C(a),}$$

where y must not occur free in $c(x)$ or in $C(x)$. The first one is an introduction rule and the second one is an elimination rule. (Besides these a formation rule is necessary. We omitted it, since it is obvious.)

Note that an element of a subset type $\{x \in A | B(x)\}$ does not contain an element of $B(x)$. So $\forall x \in \{x \in A | B(x)\}.B(x)$ is not always provable. A sufficient condition of provability is that $B(x)$ is stable, i.e., $\forall x \in A.((\neg\neg B(x)) \supset B(x))$. (See Salvesen and Smith, 1988.)

Göteborg type theory has been used for some program developments. A detailed introduction to their type theory and program development methodology will appear in Nordström, Petersson, and Smith (1988+α).

4.3. Nuprl

The most sophisticated computer system related to constructive mathematics ever built is Nuprl by Constable's PRL group at Cornell University. After his pioneering paper, Constable seemed to shift to program verification rather than "proofs as programs." Constable and his students implemented a programming logic PL/CV2 which is a verification system for a dialect PL/I programming language. It was similar to dynamic logic so that programs explicitly appear in proofs. So it was a system of "programs as proofs". Namely, the main objects which users concern were programs and proofs were their annotations. PL/CV2 was based on first order logic and did not have enough types like function spaces, so it suffered from weak expressive power (p.271, Constable, Johnson, and Eichenlaub, 1982). Constable and his coworkers tried to cure this and began to design and implement a proof checker using Martin-Löf's type theory. The result of the effort was Nuprl.

Nuprl is renowned for its sophisticated environment. Actually its window system oriented refinement logic (backward reasoning logic) is one of the most novel features of Nuprl. But we will not explain it here, since it is irrelevant to constructive logic. Basically, Nuprl's type theory is an extension of Martin-Löf's type theory. But there are some significant departures from Martin-Löf's type theory. Nuprl used quotient types as well as subset type of Göteborg type theroy. Bishop identified a set with a base set A with an equivalence relation E on it. Theoretically, it is sufficient to assume that a set is always equipped with an equivalence relation. But, in practice, it will be more convenient to make it to a quotient type. Nuprl allows to make it as a quotient type $(x,y){:}A//E$ with its own equality rather than an equivalence relation $E(x,y)$. So the equality rules for equality judgement of the quotient type can be used. The quotient type is useful only when $E(x,y)$ is a "degenerated" or "squashed" type as was pointed out in Constable et al. (1986). Namely it is useful only when $E(x,y)$ has at most one element. This is because of that the equality judgement and the equality type forget about the proof of equality. Let a and b be elements of A. Let's denote their cosets with respect to E by $[a]$ and $[b]$. When $[a]=[b]\in(x,y){:}A//E$ is derivable, then we hope retrieve an element of $E(a,b)$. Since $[a]=[b]\in(x,y){:}A//E$ is equivalent to $\mathbf{r}\in Eq((x,y){:}A//E,[a],[b])$ and $\mathbf{r}$ is the only one possible element of $Eq((x,y){:}A//E,[a],[b])$, the equality judgement does not carry any information with respect to E except that it is inhabited. So we cannot retrieve an element of E unless E is degenerated. This restriction on quotient type causes no problem in almost all cases. For example, if a pre-real number is defined as a pair of Cauchy sequence a and its modulus of convergence r, then the type representing its equivalence relation has at most one element and so its quotient type is useful.

Another departure of Nuprl type theory from Martin-Löf's type theory is inductive types and partial function spaces. Martin-Löf's type theory has some basic built-in recursive types like natural numbers and well-founded trees. Theoretically, these are enough to implement other data types. But this is not enough in practise. So Nuprl has inductive types. Furthermore, it has partial function types as well. Nuprl's semantics is based on a computation system which consists a *type free* term model and reduction rules on the terms . But not all terms in the term model is normalizable as they are untyped. Nuprl has a rule of computation by which user can evaluate his term. The evaluation is done by an evaluator using call-by-name strategy. So it is natural to extend Martin-Löf's type theory to include partial functions.

Considering partial functions, we can define functions using arbitrary recursions in the conventional way. The lack of arbitrary recursion obstacles to develop conventional and efficient programs via constructive mathematics. Although program development via constructive mathematics is neat, developed algorithms are not always natural from algorithmic point of view. For some very algorithmic problems, neatness of mathematical proofs and naturality of the extracted terms are often opposite to each other. (See 4.3.2.5 in Hayashi and Nakano, 1988.) Since program constructors and their verification rules of Martin-Löf's type theory are quite natural from mathematical point of view, this means Martin-Löf's type theory may fail to develop natural programs in some cases. So it is necessary to tune up Martin-Löf's type theory so that the conventional programming can be done by sacrificing mathematical neatness when necessary. Inductive types and partial function types of Nuprl can serve this purpose.

For example, searching a root of $f(n)=0$ for $n=0,1,\ldots$ is defined by

$$\begin{aligned} search() &= search1(0),\\ search1(x) &= if\ f(x)=0\ then\ x\ else\ search1(x+1). \end{aligned}$$

The recursion of $search1$ would not be able to be obtained by ordinary induction principles of Martin-Löf's type systems. But Nuprl can do this. Since propositions are identified with types, one can define the domain of the termination of $search1(x)$ as $\{x\in N|dom(x)\}$ by defining dom as

$$dom(x)=\begin{cases} true & \text{if } f(x)=0,\\ dom(x+1) & \text{if } f(x)\neq 0. \end{cases}$$

Assuming $dom(0)$ is inhabited, $\exists x\in N.f(x)=0$ is provable and the extracted term uses the recursion above. As subset type, recursive type does not have constructor for its elements. So the elements of $dom(0)$ is not used to compute the extracted term of the proof of $\exists x\in N.f(x)=0$. As we will explain later in the section of PX, this gives a powerful methodology to extract efficient programs and do termination proof separately. But it seems that this technique is not so used for program development in Nuprl.

The recursion associated with inductive types is *not* an arbitrary recursion. Nuprl has partial function type $A\rightharpoonup B$ so that arbitrary recursions are allowed.[2] Functions defined by arbitrary recursions

[2] PRL group uses another notation for partial function space.

may fail to terminate and so they do not belong to total function types of Martin-Löf's type theory. In Constable et al. (1986), a partial function f from A to B was considered as a total function on its domain of convergence. Inductive types are useful to characterize termination domains and graphs of a kind of recursions.

Constable and Smith (1987) have presented another approach to partial functions in Nuprl. This approach uses a type of computation $\overline{A}$. $\overline{A}$ is a type of computation whose possible value belongs to the type A. So the typing rule for the fixpoint operator fix is

$$\frac{f \in A \rightarrow A}{fix(f) \in \overline{A}.}$$

Partial function type $A \rightharpoonup B$ is defined by $A \rightarrow \overline{B}$. Since a formula $\exists y \in B.P(x,y)$ is understood as a type, $\overline{\exists y \in B.P(x,y)}$ is at the same time a type and a formula. So $\forall x \in A.\overline{\exists y \in B.P(x,y)}$ is a formula and this is a specification of partially correct programs of the problem $\forall x \in A.\exists y \in B.P(x,y)$, i.e., if a function of the type terminates for a value of A, then the value satisfies the specification. Namely, if A is a formula, $\overline{A}$ is the type of "partial proofs" of A. The semantics of bar type $\overline{A}$ is a collection of terms whose values belong to type A if they exist. Equality on bar types are Kleene's equalities. So all divergent terms belong to any bar type and are equal. Note that this approach resembles Goad's "recursive proofs". Nuprl's approach has a significant superiority. Nuprl can reason about the properties of "proofs" which are Nuprl's terms. So Nuprl can reason about terminations of "partial proofs". Goad had to argue about termination metamathematically, since proofs are not terms in his system. Basin (1988) has shown how to develop partial programs using bar types.

4.4. PX

PX is a system developed by Hayashi and Nakano at Kyoto University (Hayashi, 1987: Hayashi and Nakano, 1988). The underlying theory of PX is a constructive formal theory based on Feferman's T_0. In type theories, logic is defined by means of Curry-Howard isomorphism. In PX, logic is fixed in advance and programs are extracted from proofs by Curry-Howard isomorphism, and data types called classes are defined by means of logic.

PX is a first oreder two sorted theory based on logic of partial terms (LPT). LPT is a logic whose terms may fail to denote values (cf. Beeson, 1985; Hayashi and Nakano, 1988; Moggi, 1988). In PX, arbitrary recursive definition is allowed. This does not cause a contradiction, for terms may fail to denote values. By a definedness predicate $E(t)$, we can express "term t has a value." The equality of PX is Kleene's equality. LPT of PX axiomatizes LISP. It is an axiomatization of call-by-value intensional partial recursive functions over S-expressions. The axiomatization resembles that of axiomatic recursion theory.

The sorts of PX are the sorts of individuals and classes. Individuals are S-expressions in the sense of LISP. Since the sort of classes is a subsort of individuals, every value in PX is an individual. PX is thus untyped. Data types are values in PX, since data types are defined as classes.

Furthermore, every function has a canonical representation as an individual.

Data types of Martin-Löf's type theory except universes can be interpreted as classes. Since classes are objects, it is easy to define a class of all classes generated from given finitely many classes by finite many class forming operators. For example, a cartesian closed category generated from finite many classes is again a class. Total and partial intensional function spaces can be defined as classes. The partial function space $A \rightharpoonup B$ is defined as

$$\{f | \forall a : A.(E(app*(f,a)) \supset app*(f,a) : B)\},$$

where $app*(f,a)$ is the result of the application of f as a function to a. E is the definedness predicate mentioned above. This reads that "if the application of f to a from A has value, then it belongs to B." Bar type $\overline{A}$ is defined as $1 \rightharpoonup A$ or

$$\{p | E(app*(p)) \supset app*(p) : A\}.$$

A least fixpoint operator with the same typing rule as that of Nuprl is easily definable.[3]

Since the equality of PX is the intensional equality on S-expressions, these function spaces are intensional. Extensional types must be defined as partial equivalence relations (PER). A PER is a binary class C such that (i) $[x,y] : C \supset [y,x] : C$, (ii) $[x,y] : C \supset [y,z] : C \supset [x,z] : C$. (See Mitchell,

[3] MGA in Hayashi and Nakano (1988) is necessary to show the typing rule.

1986.) PER C is interpreted as equality on the class $\{x|[x,x] : C\}$ and extensional spaces can be defined by using PER, e.g., an extensional total function space is defined as

$$\{f|\forall[a,a'] : A.[app(f,a), app(f,a')] : B)\}.$$

Curry-Howard isomorphism in PX is given by a formalized realizability. If A is a theorem of PX, then there is a term e for which the fact "e realizes A" is provable in PX again. It is different from ordinary realizability so that if A is realizable, then A holds. Namely, if A is realizable, then it holds in the standard model of PX. Since the standard model is given *classically*, this makes the following difference from Martin-Löf's and Nuprl's approach. Nuprl contradicts classical logic (see Theorem 3, Constable and Smith, 1987). Contrary to this, PX is consistent with classical logic. Hence, in Martin-Löf's type theory and Nuprl, if $\forall x : A.\exists y : B.F(x,y)$ is proved and a term f is extracted from it, then f meets the specification $\forall x : A.F(x,f(x))$ in the sense of constructive logic. In PX, f meets the specification in the sense of classical logic. (Note that it meets the specification in the sense of constructive logic of PX as well.) Namely, PX allows to interpret the input-output predicate $F(x,y)$ classically. Furthermore, this compatibility allows to prove many lemmata classically. A family of formulas, called rank 0 formulas, obeys classical logic in PX. The properties expressible by rank 0 formulas are properties about data types and termination conditions of extracted programs. Furthermore, PX has a modal operator $\diamondsuit$ which resembles the squash operator of Nuprl. If A is a formula $\diamondsuit A$ holds iff A holds classically. Namely, putting $\diamondsuit$, every formula can be classicalized and loose all informations except A holds *classically*. In the case of squash operator, a squashed proposition holds, if and only if the original proposition holds constructively.

The realizability and the extractor of PX are optimized so that efficient and natural programs are extracted. For example, if A is a rank 0 formula, a realizer of $A \wedge B$ is just a realizer of B. Note that the interpretation of logical operators of the usual Curry-Howard isomorphism is defined uniformly from the types of realizers of arguments of logical operators. In PX, the interpretation of a formula depends not only on its outermost logical operator but also on the *ranks* of immediate subformulas. (The rank of a formula is the length of realizers of the formula.)

We have done some program developments with PX, e.g., a solution of the Chinese remainder theorem, a deterministic theorem prover for classical propositional sequent calculus (see Hayashi and Nakano, 1988). A technique which resembles the one used for the example of *search* in Nuprl described above were extensively used in these program developments. We define termination domains as classes and so termination problems are stated as rank 0 formulas. We can then extract programs before we settle termination problems. Those termination problems may be proved by classical logic or even metamathematical considerations on the standard semantics. PX can prove these by a classical proof checker based on higher order logic in which the semantics of PX is embedded. This resembles Cook's idea of the relative completeness of Hoare logic. We have given a translation of Hoare logic to PX using this idea.

4.5. Calculus of Constructions

Calculus of Construcion (CC) is a system developed by Coquand and Huet at INRIA, Roquencourt. It is a decent of AUTOMATH, Martin-Löf's type theory, and Girard's F^ω. It uses de Bruijn notation to avoid renaming of variables. It is a impredicative system as F^ω. Its types are defined simultaneoulsy with terms as in Martin-Löf's type theory. In these sense, CC is an impredicative extension of Martin-Löf's type theory. Basic differences besides impredicativity from Martin-Lóf's type theory are

1. CC does not have equality judgement in the sense of Martin-Löf's type theory,
2. CC does not have basic types as equality types, natural number type, and finite set types, etc,
3. CC does not have either of quotient types or subset types,
4. CC does not have strong dependent sums.

The only type constructor of CC is the dependent product and the only basic type is $Prop$.[5] Impredicativity allows to define many types by these. Equality type is defined as Libnitz's equality. A type of natural number can be defined, although ordinary properties cannot proved in the basic CC. Empty type can be defined as $\forall\alpha \in Prop.\alpha$. The lack of basic types is not quite important, for CC is an open system so that such types are added when they are necessary.

Quotient types would be treated as PER's as we did in PX. Subset types would be treated as propositional functions. These types are meaningful when equivalence relations and subset conditions are "degenerated." Namely, a concept as squashed types or rank 0 formula is necessary. So these would need distinction of $Prop$ and $Spec$ by Pauling-Mohring described below.

[5] CC has many versions. In some versions, CC has more basic types or kinds.

The lack of strong dependent sum is essential. A dependent sum $\Sigma x : A.B(x)$ is called strong, if the first projection of elements of the type is available. Martin-Löf's dependent sum is strong. It is known that this contradicts to impredicativity (see Coquand, 1986).

A good point of CC is a beautiful combination of minimality and strong expressive power. Since CC has a small numbers of basic concepts, it looks as if it is an abstract machine. It can express many complicated types from small numbers of basic concepts. The higher order impredicative types seems enough to formalize almost all concepts of abstract mathematics like abstract algebra and topology. Since it extends F^{ω}, all of provably computable functions of higher order arithmetic are representable as terms of CC without recursions.

An early implementation and experiments with it was described in Coquand and Huet (1985). Now there are some different implementations of CC at INRIA, Cornell University, and Edinburgh University. Griffin (1987) has impelmented CC in his EFS system. Randy Pollack at Edinburgh University has implemented a proof checker of CC called LEGO and has done many interesting examples on it. The implementation of LEGO is based on higher order unification and so resembles Isablle described below. J. Despeyroux (1988) has implemented a window oriented proof checker for CC on CENTAUR. Coquand is implementing a new version of his proof checker using LCF-like tactics.

So far CC has been mainly used as a framework to implement mathematics. It is interesting how CC serves as a program extraction system. Pauling-Mohring (1986, 1987) is developing program extraction methodologies with CC and a new version of CC for program extraction. She extended CC so that there are two kinds of types. One is *Prop* and another is *Spec*. *Spec* corresponds to *Prop* of the original CC. Types of *Prop* resemble Harrop formulas. So types of *Prop* are used as squashed types of Nuprl and rank 0 formulas of PX to avoid unnecessary informations.[6] Typing rules for *Spec* and *Prop* are isomorphic to those of original CC, so the extension still retains CC's minimality. This was possible since CC has only dependent product as type constructors.

There are some attempts to extend CC to stronger type theories. Luo (1988, 1988a) extended CC using Martin-Löf like predicative universes. His type system has strong dependent sum at higher predicative types. He proved that his system is strongly normalizable and gave a mathematical model. Although the predicative hierarchy resembles hierarchies of small models of ZF set theory, its proof theoretic strength is much weaker than ZF. It seems that Luo's system is a right framework to formalize category theory. His type theory has been implemented in LEGO.

4.6. ICOT-QJ

QJ is a system introduced by Sato (1985). It is a type theory based on a concurrent functional language Quty based on unification. QJ has a minimal fixpoint operator by which even inductive types are definable. Implementation of QJ is being carried at ICOT by Takayama (1987). Takayama (1988) has presented a new idea to avoid unnecessary informations automatically by "proof analysis" which resembles program analysis.

4.6. ZFR

As we saw above, Martin-Löf's type theory and Feferman's untyped theory have been implemented and used from computational point of view in Göteborg type theory, Nuprl, and PX. But there has not been such an attempt for Myhill and Friedman's theories mentioned in 3.2. ZFR by Beeson (1987) aims to fill this gap. ZFR is a mixture of lambda calculus and ZF set theory. Basic idea is to use sets as data types as infinite S-expressions and give an interpreter of lambda calculus on it. Since set recursion can be defined by lambda calculus, some recursive structures like transitive closure of a set are easily defined. ZFR has not been implemented or used for program development.

4.7. LF, Isabelle, λProlog

Curry-Howard isomorphism shows that type theories can define constructive logic. LF (logical framework) group at Edinburgh university has extended the "formulas as types" notion to the "judgements as types" notion, so that many logics can be defined by means of a Martin-Löf-like type theory called Edinburgh LF (Harper, R., Honsell, F., and Plotkin, G., 1987). LF enables us to "code up" logics easily without changing the basic proof checker. All systems of Despeyroux, Griffin, and Pollack can be used as systems for LF.

It is not necessary to keep constructions (proofs) to code up logics, if the provability alone is concerned. Paulson's Isabelle system (Paulson, 1987) uses intuitionistic higher order logic to code up logics. A similar approach is used by Felty and Miller (1988) to code up logics in λProlog. λProlog is a higher order Prolog based on hereditary Harrop formulas, which form a fragment of Harrop formulas of

[6] Salvesen and Smith (1987) presented a similar idea.

intuitionistic higher order logic. In λProlog, it is possible to code up logics so that proofs are preserved. An interesting point is that an extension of the existence property under Harrop formulas is used to make search space tractable in λProlog.

References

Basin, D.A., 1988, An Environment for Automated Reasoning About Partial Functions, in: Springer Lecture Notes in Computer Science 310.

Beeson, M.J., 1985, "The Foundations of Constructive Mathematics", Springer, Berlin.

Beeson, M.J., 1987, Towards a Computation System Based on Set Theory, preprint.

Bishop, E., 1967, "Foundations of Constructive Analysis," McGraw-Hill.

de Bruijn, N.G., 1980, A survey of the project AUTOMATH, in: "Essays in Combinatory Logic, Lambda Calculus, and Formalism," Seldin, J.P. and Hindley, J.R., eds., Academic Press, New York.

Burstall, R., and Lampson, B., 1984, A kernel language for abstract data types and modules, in: Springer Lecture Note in Computer Science 173.

Constable, R.L., 1971, Constructive mathematics and automatics programs writers, in: "Proceedings of IFIP Congress," Ljubljana.

Constable, R.L., et al., 1986, "Implementing Mathematics with the Nuprl Proof Development System," Prentice-Hall.

Constable, R.L., Johnson, S.D., and Eichenlaub, C.D., 1982, *Introduction to the PL/CV2 Programming Logic*, Springer Lecture Notes in Computer Science 135.

Constable, R.L., and Smith, S.F., 1987, Partial Objects in Constructive Type Theory, in: "Second Annual Symposium on Logic in Computer Science," IEEE.

Coquand, T., 1986, An analysis of Girard's paradox, in: "First Annual Symposium on Logic in Computer Science," IEEE.

Coquand, T., and Huet, G., 1985, Constructions: A higher order proof system of mechanizing mathematics, in: Springer Lecture Notes in Computer Science 203.

Coquand, T., and Huet, G., 1988, The Calculus of Constructions, *Information and Computation 76.*

Despeyroux, J., 1988, First experiments with theorem proving in Centaur: the Calculus of Construction, the Edinburgh Logical Framework, and Theo, in: "GIPE, ESPRIT project 348, Third annual review report."

Feferman, S., 1975, A language and axioms for explicit mathematics, in: Springer Lecture Notes in Computer Science 450.

Feferman, S., 1979, Constructive theories of functions and classes, in: "Logic Colloquium '78," North-Holland, Amsterdam.

Felty, A. and Miller, D., Specifying Theorem Provers in a Higher-order Logic Programming Language, in: Springer Lecture Notes in Computer Science 310.

Friedman, H., 1973, Some applications of Kleene's methods for intuitionistic systems, in: Springer Lecture Note in Mathematics 337, Springer, Berlin.

Friedman, H., 1977, Set theoretic foundations for constructive analysis, *Annals of Mathematics 105.*

Girard, J.Y., 1972, "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur", Ph.D. thesis, University of Paris, VII.

Goad, C., 1980, "Computational uses of the manipulation of formal proofs", Ph.D. thesis, Stanford University.

Gordon, M.J., Milner, R., and Wadsworth, C.P., 1979, "Edinburgh LCF", Springer Lecture Notes in Computer Science, Springer-Verlag 78.

Goto, S., 1975, Recursive Realizability and Automatic Program Synthesis (in Japanese), IECE (The Institute of Electrical and Communication Engineers, Japan), Shingaku-Giho AL (Automata and Language) AL-75-54.

Goto, S., 1977, Foundations of Automatic Program Synthesis: an Application of Godel's Interpretation (in Japanese), Shingaku-Giho AL (Automata and Language) AL-77-16, English version appeared in: Springer Lecture Note in Computer Science 75 (1979).

Goto, S., 1979, Program synthesis from natural deduction proofs, in: "Proceedings of the Sixth International Joint Conference on Artificial Intelligence, vol. 1."

Griffin, T.G., 1987, An environment for formal systems, Technical Report 87-846, Department of Computer Science, Cornell University.

Harper, R., Honsell, F., and Plotkin, G., 1987, A Framework for Defining Logics, in: "Second Annual Symposium on Logic in Computer Science," IEEE.

Hayashi, S., 1987, **PX**: a system extracting programs from proofs, in "Formal Description of Programming Concepts-III," Wirsing, M., ed., North-Holland, Amsterdam.

Hayashi, S., and Nakano, H., 1988, **PX**: A Computational Logic, The MIT Press, Cambridge, Massachusetts.

Heyting, A., 1971, *Intuitionism, An Introduction, third revised edition*, North-Holland, Amsterdam.

Howard, W., 1980, The formulas-as-types notion of construction, in: "Essays on Combinatory Logic, Lambda Calculus and Formalism," Seldin, J.P. and Hindley, J.R., eds., Academic Press, New York.

Howe, D.J.., 1987, The Computational Behaviour of Girard's Paradox, in: "Second Annual Symposium on Logic in Computer Science," IEEE.

Hyland, J.M.E., 1987, A small complete category, preprint.

Longo, J., and Moggi, E., 1988+α, Constructive Natural Deduction and its "modest" Interpretation, in: "Semantics of Natural and Computer Languages," Meseguer et al. eds., The MIT Press, Cambridge, Massachusetts.

Luo, Z., 1988, A Higher-order Calculus and Theory Abstraction, LFCS Report ECS-LFCS-88-57, LFCS, Edinburgh University.

Luo, Z., 1988a, CC$^\omega$ and its Meta Theory, LFCS Report ECS-LFCS-88-58, LFCS, Edinburgh University.

Manna, Z., and Waldinger, R., 1971, Towards automatic program synthesis, *Communications of ACM 14.*

Martin-Löf, P., 1974, An intuitionistic theory of types: Predicative Part, in: "Logic Colloquium 73," Rose, H., and Shepherdson, J., eds., North-Holland, Amsterdam.

Martin-Löf, P., 1982, Constructive mathematics and computer programming, in: "Logic, Methodology, and Philosophy of Science VI," Cohen, L.J., et al., eds., North-Holland, Amsterdam.

Mitchell, J.C., 1986, A type-inference approach to reduction properties and semantics of polymorphic expressions, in: "1986 ACM Symposium on Lisp and Functional Programming."

Moggi, E., 1988, "The Partial Lambda-Calculus," Ph.D. thesis, University of Edinburgh.

Myhill, J., 1975, Constructive set theory, J. Symbolic Logic 40.

Nepeĭvoda, N.N., 1978, A relation between the natural deduction rules and operators of higher level algorithmic languages, *Soviet Mathematics Doklady 19.*

Nordström, B., and Petersson, K., 1983, Types and specifications, in: *Proceedings IFIP '83*, Elsevier, Amsterdam.

Nordström, Petersson, and Smith, 1988+α, *Programming in Martin-Löf's type theory. An introduction*, outcoming from Oxford University Press.

Paulson, L.C., 1987, The representation of logics in higher-order logic, Cambridge Technical Report 113.

Paulin-Mohring, C., 1986, Algorithm development in the calculus of constructions, in: "First Annual Symposium on Logic in Computer Science," IEEE.

Paulin-Mohring, C., 1987, Extraction de programmes dans le Calcul des Constructions, preprint.

Reynolds, J.C., 1974, Towards a theory of type structure, in: Springer Lecture Note in Computer Science 19.

Salvesen, A. and Smith, J.M., 1988, The strength of the subset type in Martin-Löf's type theroy, in: "Third Annual Symposium on Logic in Computer Science," IEEE.

Sato, M., 1979, Towards a mathematical theory of program synthesis, in: "Proceedings of the Sixth International Joint Conference on Artificial Intelligence, vol. 2."

Sato, M., 1985, Typed Logical Calculus, Department of Information Science, Faculty of Science, University of Tokyo, Technical Report 85-13.

Scott, D., 1970, Constructive validity, in: Springer Lecture Notes in Computer Science 125, Springer, Berlin.

Takayama, Y., 1987, Writing Programs as QJ-Proofs and Compiling into PROLOG Programs, in: "Proceedings of 4th Symposium on Logic Programming."

Takayama, Y., 1988, Proof Theoretic Approach to the Extraction of Redundancy-free Realizer Codes, preprint.