The Representation of Program Synthesis in Higher Order Logic

Christoph Kreitz

FG Intellektik – FB Informatik
Technische Hochschule Darmstadt
Alexanderstr. 10

D-6100 Darmstadt

Systems built for automated program construction aim at the formalization of the programming process in order to produce better software. Their implementations, however, suffer from problems similar to those they are intended to solve. Due to a lack of abstraction in the formalization of deductive mechanisms involved in programming reasoning tools for the development of program synthesizers are not yet available. For that, systems capable of formal reasoning about both programs *and* programming methods are needed.

In this paper we develop principles of a formal theory on reasoning about programs and program construction within a unified higher order framework. By an exemplified formalization of principal approaches to program synthesis we will show that a higher degree of abstraction leads to clearer insights into the meta-mathematics of program construction. Ridding the representation of deductive methods from superfluous context also results in simpler, sometimes almost trivial, proofs. Simplicity is one of the most important features of the formal theory and quite valuable if one considers the wide range of intended applications.

We present the theory in a highly formalized form built on top of Intuitionistic Type Theory. This allows us to straightforwardly implemented the concepts developed here with a proof system for Type Theory and derive verified implementations of deductive mechanisms from mechanically proven theorems.

1 Introduction

Since the upcoming of the so-called software crisis efforts have been put into the production of better software. Methodologists have developed a science of programming [Gri81, Dij76] to solve the problem that products of the software production business seldomly meet the original intentions of the clients and are very difficult to modify. To a large extent programming has been identified as a reasoning process on the basis of knowledge of various kinds, an activity where people typically do a lot of mistakes. It is, therefore, strongly desirable to provide machine support for program construction which in principle means to aim at the automation of the whole programming process. This requires a full formalization of all it parts in order to get an understanding of the mechanisms involved. Many formal approaches for the automated synthesis of programs have been developed and implemented during the last years (see e.g. [BD77, MW79, MW80, Bib80, BH84, Hog81, Der85, Smi88, SL89]) the most mature of them currently being the KIDS system [Smi88] which also aims at a strong theoretical foundation.

However, program synthesis systems built so far underly the same problems as conventional software. Despite the fact that they aim at a formalization and automatization of the programming process in order to produce better software, they themselves are difficult to maintain and modify. Often it is not even clear if or why they are correct. After a while program synthesizers tend to get quite bulky and improved versions again have to be build from scratch.

In our opinion these problems are due to a lack of abstraction when formalizing deductive mechanisms for program development. By this one is kept from getting insights into their true nature and proofs about their properties become unneccessarily complicated. The apparent lack of new ideas how to guide and control deductive mechanisms in programming is a consequence of that. A program synthesis system is nothing but a program on some higher level and should, except for a higher level of reasoning, be developed following the same methodologies as used for the construction of "conventional programs". Since a program synthesizer itself has to reason about programs there is a need for systems capable of formal reasoning about both programs *and* deductive methods in programming. A theoretical foundation for these does not exist so far.

Principles of such a *formal theory of program construction* shall be presented here. They shall allow to formulate both object knowledge (e.g. domain knowledge, programs) and meta-knowledge (e.g. synthesis techniques, algorithm knowledge) and formally prove theorems about it within some unified framework. In order to avoid creating a "new logic" the theory has to be formally represented within some already established general formalism with well known deduction rules. Complex special purpose reasoning within the theory then can be reduced to a series of simple reasoning steps in the more general formalism and thus be implemented with a reasoning tool for it. This makes the step from a formulation of the theory to its implementation very small.

In H. Marburger, ed., *14th German Workshop on Artificial Intelligence*, Informatik Fachberichte 251, pp. 171–180, © Springer Verlag, 1990.

On this level the task of automatic program synthesis can be reduced to a much more manageable problem: "Mechanically prove meta-theorems about program construction and other necessary knowledge within the formal theory and then derive a *verified implementation of a program synthesizer* from the formal proofs". Doing so the resulting program synthesis system is not only correct, easy to maintain, and easy to modify, but we also have a clear understanding of its behaviour and capabilities. A starting point would be the representation of already known approaches to systematic programming and program synthesis as theorems of the theory thus enabling us to unify and extend them. We believe that deeper insights into the mathematics of program construction will also help us to develop entirely new strategies.

As general formalism in which the formal theory of program construction shall be expressed we selected Intuitionistic Type Theory (see e.g. [Chu40, ML82, And86]). Reasons for that and highlights of Type Theory are discussed in the following section. In particular we will introduce syntax and features of the NuPRL proof development system [CAB$^+$86] for Type Theory on top of which our theory shall be implemented. Such an implementation will also help to uncover issues we may overlook when developing the theory on paper and as a major side-effect we may also get a running program synthesis system.

After fixing types representing the object language of program construction in Section 3 we will formalize the GUESS Strategy of the LOPS System [Bib80, BH84] to gather ideas how to build a general theory. We will show that a straightforward formalization already gives some new insights about the deductive method. Since, however, in such a direct approach general principles are blurred by individual notations we go for a higher degree of abstraction in Section 4. There we will discuss formal definitions of concepts involved in the programing process and formally investigate properties of the principal approaches to program synthesis. As a side-effect we will show that the differences between the two main ideologies (Theorem Proving approaches and transformation based approaches) exist only superficially. They can be translated into one another, a simple result which due to lack of abtraction has not been presented yet. All theorems given here can be mechanically proven with NuPRL.

2 Type Theory and Programming

2.1 Why Type Theory

When formalizing deductive techniques and mathematical knowledge for program synthesizers and automated theorem provers one has to make use of some universal logical language in which any mathematical statement can be expressed. The main languages that have been found adequate for this purpose are formulations of Type Theory and Axiomatic Set Theory.

Since in Axiomatic Set Theory all statements are based on forms like $x \in y$ even simple statements like the definition of functions become quite complex. Although on the surface this difficulty may be avoided by introducing abbreviations, in a computer system this would mean extending the basic language. One might as well select a more flexible language in the first place. Another difficulty arises when dealing with proofs for the existence of objects. Set Theory offers a variety of axioms on the existence of sets and considerable efforts may be required to establish the existence of objects with rather simple intuitive descriptions. Finally the notion of "algorithm" cannot be properly explained within Set Theory which makes it appear inappropriate for reasoning about programs.

None of these difficulties arise with a suitable formulation of Type Theory such as [ML82] or [CAB$^+$86]. Mathematical statements can be translated directly into the formal language. Nearly all objects of mathematics have immediate counterparts in the language. Furthermore, as Martin-Löf ([ML82] p.155) points out, even the whole conceptual apparatus of programming mirrors that of (intuitionistic) mathematics and thus can be straightforwardly embedded as well. Of course, one has to assign a type to each mathematical object but this just formally reflects the fact that mathematicians naturally do make distinctions between different types of objects. Thus type symbols provide important syntactic clues which are not available in untyped theories.

Therefore, Intuitionistic Type Theory not only is expressive enough for a formalization of all the activities involved in programming but it seems to us that the view of the world one gets from using it is the most appropriate one for expressing a formal theory of program development.

2.2 Features of Type Theory

Despite the fact that Type Theory is a well established formal theory it must be considered unknown to most of the AI community. We therefore briefly highlight some of its most important aspects.

Intuitionistic Type Theory, an extension of a typed λ-calculus, is both a formulation of a *constructive* higher-order logic and a model for datatypes and computation. As such, it is fundamental to higher type deduction and programming language design. The basic objects of reasoning are **types** and **members of types**. Type Theory does not use logical constructs explicitly. They can be expressed via the **propositions-as-types** correspondence saying that each logical construct has a type construct with the same deductive rules as an immediate counterpart. One of the most appealing features is that in Intuitionistic Type Theory every theorem has a **computational content** which can be extracted from its proof. As a consequence formal proofs about program construction give a verified implementation of a program synthesizer *for free*. This already solves the second part of the synthesis problem mentioned in the introduction and allows to concentrate our efforts in developing a theory of program development.

Since Type Theory is a comparably young formalism there are still various dialects of it each using a slightly different syntax. We will use the one of NuPRL [CAB$^+$86, BC85], a descendent of Martin-Löf's Type Theory [ML82], because for this dialect an interactive proof development system, a tool to implement our theory, is already available.

2.3 NuPRL's proof calculus for Type Theory

Types and members of types, as we already mentioned, are the basic objects of reasoning in Type Theory. NuPRL's Type Theory consists of a large set of type constructors and a few atomic types which were explicitly defined for user convenience. Associated with each atomic type and each type constructor are forms for constructing *canonical members* (like λ-abstraction $\lambda x.b$ for functions and pairing $\langle a, b\rangle$ for products) and forms for making use of members of the type (like function application $f(a)$ and projection for pairs). A cumulative hierarchy (to avoid paradoxi) of universes U_i, introduced to deal with wellformedness problems, enables higher order reasoning in a very simple and natural way. A (sorted, constructive) predicate logic, though not part of the type system itself can be expressed using the propositions-as-types correspondence. Syntax and some details of the type system are listed in Figure 1. We refer the reader to [CAB$^+$86] for a full presentation of NuPRL's Type Theory.

TYPE	CANONICAL MEMBERS		LOGICAL EQUIVALENTS
type constructors			
$A \to B$	$\lambda x.b$	if $b \in B$	A $\Rightarrow$ B
$x : A \to B$	$\lambda x.b$	if $b \in B[x]$	$\forall$x:A.B$^+$
$A\#B$	$\langle a, b\rangle$	if $a \in A, b \in B$	A $\wedge$ B
$x : A\#B$	$\langle a, b\rangle$	if $a \in A, b \in B[a/x]^*$	$\exists$x:A.B
$A\|B$	$inl(a), inr(b)$	if $a \in A, b \in B$	A $\vee$ B
$A\ list$	$nil,\ a.l$	if $a \in A, l \in A\ list$	
$\{x : A\|B\}$	a	if $a \in A, B[a/x]$	
$x, y : A//B$	a	if $a \in A$ (Equality: $a = a'$ iff $B[a, a'/x, y]$)	
$rec(z, x.T; A)$	a	if $a \in T[\lambda x.rec(z, x.T; x), A/z, x]$	
$A \rightsquigarrow B$	(partial functions from A to B)		
propositions as types			atomic predicates
$a = a'$ `in` A	*axiom*	if $a = a'$, (no members otherwise)	
$i < j$	*axiom*	if $i, j \in int, i < j$, (no members otherwise)	
explicit types			
int	i	if i is an integer constant	
atom	"*text*"	if *text* is a character sequence	
void		no members	FALSE
Universes			
U_1	atomic types and all that can be constructed via type constructors.		
U_2	U_1, members of U_1, and all that can be constructed via type constructors.		
U_3	U_2, members of U_2, and all that can be constructed via type constructors.		
$\vdots$	$\vdots$		

* $B[a/x]$: Substitute a for x in B, $^+\forall$x:A.B : For all x in A B holds.

Figure 1: **NuPRL types and constructors**

Statements are expressed in the form of *sequents*. These are objects of the form $x_1 : T_1, \ldots, x_n : T_n \vdash C$ `[ext` m`]` which should be read as "*Under the assumption that x_i are variables of type T_i a member $m \in C$ of the type C can be constructed*". In the context of proofs sequents are also referred to as *goals*. The terms $x_i : T_i$, declaring a variable x_i of type T_i are called *hypotheses* or *assumptions*, C the *conclusion*, and m the *extract term* of the goal. The notion `[ext` m`]` reflects the fact that m usually is not known beforehand but constructed during a proof. It stays hidden up to completion of the proof. Thus sequents implicitely describe an algorithm constructing a member for the conclusion (a witness for its truth) from the assumptions. Consequently, algorithms can be specified in form of mathematical propositions implicitely asserting their existence. This so-called **proofs-as-programs** paradigm [BC85] is of particular importance for embedding a theory of program development into NuPRL.

NuPRL's proof calculus supports a *top-down* development of this algorithm. Proof rules allow to *refine* a goal, obtaining subgoals such that an algorithm for the main goal can be constructed from partial solutions for the subgoals. Refinement rules are explicitely given by rule schemes which in their formal description have been designed to reflect this behaviour (where H stands for a list of hypotheses):

$H \vdash C$ `[ext` m`]` by *rule-name*
 1. $H_1 \vdash C_1$ `[ext` m_1`]`
 $\vdots$
 n. $H_n \vdash C_n$ `[ext` m_n`]`

which corresponds to the usual bottom-up style of inference rules

rule-name: $$\frac{H_1 \vdash m_1 \in C_1,\ \ldots,\ H_n \vdash m_n \in C_n}{H \vdash m \in C}$$

Such a rule should be read as *"$H \vdash T$ is provable if the subgoals $H_i \vdash C_i$ can be proven"*. If proofs of the subgoals yield witnesses m_i for C_i being inhabited then a witness $m \in C$ for the main goal is constructed from the m_i by the rule *rule-name*. NuPRL proofs are tree structured objects whose nodes consist of a goal and a refinement rule. The children of a node are the subgoals which result from applying the rule to the goal. Because of wellformedness reasons the initial goal of a proof must have an empty hypotheses list.

It is helpful to know that the proof development system implemented for NuPRL already supports a few features which allow mechanical reasoning on nearly the same level of abstraction as mathematicians usually do. Besides a highly visual *proof-editor* for interactive development of proofs and extraction of their computational contents a *definition mechanism* allows to abstract from low-level type theoretical expressions and enhance readability of mechanical proofs. In addition to that a high-level programming language *ML*, originally developed for Edinburgh's LCF System [GMW79], serves as the *meta-language* of NuPRL allowing a user to write meta-programs guiding the application of refinement rules. This is particularly interesting for creating new deduction rules on top of NuPRL by, for instance, applying higher-order theorems about programming methods to first-order problems. Together with the expressive power of the logic these components strongly support a high level "implementation" of mathematical theories. We will make use of this knowledge when developing our formal theory in the following sections.

2.4 Notation

We will use the `typewriter` font when expressing formal constructs in our theory to indicate that these parts can directly be implemented in NuPRL. We will, though, still use special characters like $\vdash, \rightarrow$ despite the fact that in NuPRL they are simulated (by `>>, ->`) since they are not available on primitive terminals.

A definition `<New Object>` $\equiv$ `<Formal NuPRL Representation>` defines a new type-theoretical object in terms of already existing constructs. Note, that deduction rules for this new concept follow immmediately from those of the right hand side and can either be programmed in ML or even be proven as a meta-theorem.

Meta-theorems about deduction principles are written down a reverse "top down" style to reflect their intended application as top down refinement rule and they should be read accordingly.

$\forall$`<vars involved>. Main goal` $\Leftarrow$ `Subgoal`$_1$ `& ... & Subgoal`$_n$

We avoid using parentheses when a graphical separation of goals seems to be sufficient.

3 Developing principles of the formal theory

For the sake of clarity the formal theory of program construction is divided reasoning about the object level (e.g. formulae and programs) and the meta-level (e.g. proofs, syntheses methods, transformations) of programming[1] as well as into reasoning about individual objects and classes of objects in general. Since in this paper we will focus on developing the meta-level we will consider the object level on the surface only and avoid superfluos details and notations.

3.1 Fixing classes for object language expressions

The only knowledge about the object level we will make use of is the existence of classes of objects which we have to reason about from the meta-level. Two classes need to be specified, one expressing the class of first-order formulae which we intend to use as object language and the second expressing first-order domains (or datatypes). We will represent these by types called `FORMULAE` and `TYPES`. Formulae with variables from a type `T` will be members of the type `FORMULAE(T)`. Due to the propositions-as-types principle it is reasonable to choose the most simple (intensional) definition by identifying them with the universe U_1 of first-order objects.

[1]This should not be confused with the distinction between the object language of Type Theory into which all of our theory will be embedded and NuPRL's meta-language ML.

Definition 3.1 **[Datatypes and logical formulae]**

```
TYPES        ≡  U1
FORMULAE     ≡  U1
FORMULAE(T)  ≡  T → FORMULAE
```

Note, that with the above definitions information about the syntactic structure of a type or formula can be accessed *only from the meta-level of Type Theory* (i.e. using ML) but not from a theory built within Type Theory. In order to do so an exhaustive extensional definition as a certain subclass of U_1 would have to be given via recursive types but this would lead us out of the scope of this article. In nearly all cases we can do without this syntactic information which would rather burden us with superfluous context and does not give any additional insights.

It also should be noted that in Type Theory not all types or formulae are decidable. Thus, statements like `p|¬p` or `p⇒ q ≡ ¬p|q` are not true for all `p,q∈FORMULAE`. In practice, however, nearly all the predicates involved are decidable and often use of such knowledge is made while constructing a program. In order to be able to catch this knowledge we will introduce decidable subclasses of `FORMULAE` and `TYPES` use them whenever it is necessary for the problem.

Definition 3.2 **[Decidable datatypes and logical formulae]**

```
DTYPES        ≡  {T:U1 |∀x,y:T (x=y in T | ¬x=y in T)}
DFORMULAE     ≡  {f:U1 | (f |¬f)}
DFORMULAE(T)  ≡  T → DFORMULAE
```

3.2 An example: Representing strategies of the LOPS system

By an exemplified formalization of a synthesis strategy we will now investigate how in principle deductive methods for program construction are to be represented. Our attention will be focused on developing a framework for our theory while faithfulness, the question if the chosen form actually reflects the particular strategy, is of lesser interest here. Faithfulness becomes important only when embedding particular approaches to program synthesis *into* the framework is studied. As running example we chose the LOPS-system [Bib80, BH84] which may be briefly summarized as follows:

> Starting with a specification of the form "∀i∃y (IC(i) ⇒ OC(i,y))" where i and y represent input and output variable, IC some input condition, and OC the relation between input and output (output condition) the goal is to achieve an algorithmically "better" formula which can directly be translated into a program. This goal is approached by a series of correctness-preserving transformations guided by a few strategies supported by deductive tools.

In this paper we will focus on GUESS-DOMAIN, one of the two key strategies of LOPS. It tries to find an appropriate portion of the specification which can be used to split the input into smaller pieces and compute the desired output from these pieces. The strategy consists of a transformation GUESS and a heuristic DOMAIN determining all the necessary parameters for the transformation which takes a formula of the form

$$\forall i \exists y \ (\ IC(i) \ \Rightarrow \ OC(i,y)\ )$$

and transforms it into $\quad \forall i \forall g \exists y \ dom(i,g) \ \Rightarrow \ (IC(i) \ \Rightarrow \ OC(i,y) \ \wedge \ (g{=}y \ \vee \ g{\neq}y))$

This transformation means to **guess** some hopefully correct output g or at least partial information about the output. In order to meaningfully restrict the search for g by some domain condition $dom(i,g)$ [Bib80] proposes to choose $dom(i,g)$ from among the subsets of the conjuncts in $OC(i,g)$ such that it will be possible to compute some g with $dom(i,g)$. If the output domain is not a simple datatype then the relation $g = y$ may have to be replaced by some more general predicate $t(g,y)$.

We will now give a straightforward formalization of the GUESS transformation as a meta-theorem of our theory. Note that for this we have assigned types to each variable and quantified over the formulae and types involved.

Theorem 3.3 `GUESS transformation`

```
⊢ ∀IN,OUT,A:TYPES.∀IC:FORMULAE(IN).∀OC:FORMULAE(IN#OUT).
    ∀dom:FORMULAE(IN#A).∀t:FORMULAE(A#OUT).
        ∀i:IN.∃y:OUT. IC(i) ⇒ OC(i,y)
          ⇐  ∀i:IN.∀g:A.∃y:OUT. dom(i,g) ⇒  IC(i) ⇒ OC(i,y) & (t(g,y) | ¬t(g,y))
             &∀i:IN.∃g:A. IC(i) ⇒ dom(i,g)
```

Besides the formalization of a deductive method Theorem 3.3, like all the meta-theorems presented here, has three important aspects.

- **Justification:** A formal (mechanically verified) proof of the correctness of the deductive mechanism is given. Note that the second "subgoal" is necessary for the correctness of the GUESS transformation and thus puts an effectivity condition on the selection of *dom*. This new insight too is a valuable effect of a strict formalization.

- **Implementation:** Applying the theorem means *executing* the GUESS transformation (resulting in the first subgoal) requiring effectivity of *dom* to be proven.
- **Program construction:** The extract term of the theorem is an algorithm building combining partial solutions from pieces created by the GUESS transformation into a program solving the original problem.
  Extracted Algorithm: Let `P1` for appropriate i, g calculate some $y \in OUT$ with $OC(i,y)\&(t(g,y)|\neg t(g,y))$ and `P2` compute $g \in A$ with $dom(i,g)$ from $i \in IN$ with $IC(i)$. Then return a program `P` with `P(i)=P1(i,P2(i))`.

It is important to say that Theorem 3.3 verifies correctness of the GUESS transformation but does not say anything about improvements resulting from it. This is a separate topic which appears to be much more difficult and has not been treated in literature so far.

To illustrate how simple a formal proof of such a typical meta-theorem can be we will sketch a NuPRL proof of Theorem 3.3. Hypotheses will be numbered. In subgoals we show new hypotheses and the current goal only.

Proof: *By "introduction" rules move all assumptions to the hypothesis list*
```
1.-3. IN:TYPES, OUT:TYPES, A:TYPES
4.-7. IC:FORMULAE(IN), OC:FORMULAE(IN#OUT), dom:FORMULAE(IN#A), t:FORMULAE(A#OUT)
8. ∀i:IN.∀g:A. ∃y:OUT. dom(i,g) ⇒ IC(i) ⇒ OC(i,y) & ( t(g,y) | ¬t(g,y))
9. ∀i:IN.∃g:A. IC(i) ⇒ dom(i,g)
10. i:IN
    ⊢ ∃y:OUT. IC(i) ⇒ OC(i,y)
```
Instantiate 9. on `i`
```
11. ∃g:A. IC(i) ⇒ dom(i,g)
    ⊢ ∃y:OUT. IC(i) ⇒ OC(i,y)
```
Eliminate the existential quantifier in 11. (giving a name to the object)
```
12. g:A
13. IC(i) ⇒ dom(i,g)
    ⊢ ∃y:OUT. IC(i) ⇒ OC(i,y)
```
Instantiate 8. on `i, g`*, eliminate the existential quantifier in the result.*
```
14. ∃y:OUT. dom(i,g) ⇒ IC(i) ⇒ OC(i,y) & ( t(g,y) | ¬t(g,y))
15. y:OUT
16. dom(i,g) ⇒ IC(i) ⇒ OC(i,y) & ( t(g,y) | ¬t(g,y))
    ⊢ ∃y:OUT. IC(i) ⇒ OC(i,y)
```
Choose the `y` *of hypothesis 15 as solution and move the implication assumption to the hypotheses list.*
```
17. IC(i)
    ⊢ OC(i,y)
```
Eliminate the implications of 13, 16, and of the result of this (19).
```
18. dom(i,g)
19. IC(i) ⇒ OC(i,y) & ( t(g,y) | ¬t(g,y))
20. OC(i,y) & ( t(g,y) | ¬t(g,y))
    ⊢ OC(i,y)
```
The goal follows from 20. □

4 Representing Program Synthesis

Despite the fact that the direct formalization of a particular deductive method keeping its original form already gave some insights this immediate approach does not generalize well. There is still a lot of apparently superfluous context hiding the general principles. For a general formal theory a higher degree of abstraction is desirable.

It is quite helpful to begin this with a formalization of notions which are known to be important in program development. We then will investigate the main properties of the principal approaches to program synthesis within the more abstract framework before we return to study how the above example behaves in it.

4.1 Formalizing Program Construction Concepts

Basically, the process of program development consists of the following steps. From an informal description of the problem find a formal specification, develop an algorithm how to solve it, and finally encode it in some programming language. Due to its very nature the first step is hardly formalizable but for the other ones strong automatic support is possible. Thus the task of program construction should be understood as transforming formal specifications into programs fulfilling the specification using knowledge about the domain, algorithms in general, and previously defined programs.

A programming problem typically is described by *specifying Input and Output-domain, a possible precondition IC on the input, and the relation IOR between input and output.* Using a precondition instead of restricting the input domain is not used in all approaches but quite common (see e.g. [Bib80, SL89]). Obviously, the domains (IN, OUT) should be first-order types and IC, IOR must be first-order formulae over the appropriate types. Thus the (higher-order) type of program specifications has to be represented by a type having quadruples $\langle IN, OUT, IC, IOR\rangle$ as its elements where the *type* of IC, IOR depends on the *values* of IN, OUT. This can only be expressed by using the syntax of dependent products:

```
SPECIFICATIONS          ≡    IN:TYPES # OUT:TYPES # FORMULAE (IN) # FORMULAE (IN#OUT)
```

Destructors accessing the individual components of a specification `spec=<IN,OUT,IC,IOR>`∈`SPECIFICATIONS` will be denoted by the obvious names `IN(spec)`, etc.

Formally, a program is a function from input to output space, a view supported by [ML82, SL89] and constructive mathematicians. Including the domains as necessary information the type of all programs is represented by

```
PROGRAMS                ≡   IN:TYPES # OUT:TYPES # (IN → OUT)
```

Again, we denote the destructors of a program `p=<IN,OUT,body>`∈`PROGRAMS` by the obvious names.

A program fulfils a specification if for a given input value x satisfying the input condition the program body computes an output value such that the input-output relation holds. Of course, input- and output domain must agree:

```
FULFILS(spec,p)         ≡   IN(spec)=IN(p) in TYPES & OUT(spec)=OUT(p) in TYPES
                            & ∀x:IN(spec). IC(spec)(x)⇒ IOR(spec)(x,body(p)(x))
```

A specification is `Solved` by developing a `program` which `fulfils` it. In Type Theory this is best expressed by defining a type of all the `solutions` of a specification. Then developing a program means to construct a member of this type. Since most people, however, are more familiar with its counterpart using notation from constructive logic (i.e. developing a program by showing its existence) we use the following definition:

```
SOLVABLE(spec)          ≡   ∃p:PROGRAMS. FULFILS(spec,p)
```

With these definitions we can even give a formal (higher-order) specification for program synthesis itself. Although this cannot be a member of `SPECIFICATIONS` we will can use the same structure thus providing a way for a certain amount of self-reflection. Knowing that in principle the task of program synthesis is unsolvable we use a yet unknown precondition `problem-class` for later classification of specifications for which an automatic program synthesis will be possible.

```
SYNTHSPEC               ≡   <SPECIFICATIONS, PROGRAMS, problem-class, λp,spec.FULFILS(spec,p)>
```

Building a synthesizer thus would mean constructing a (higher-order) program fulfilling the above specification.

One may object that these definitions do not include partial or multivalued functions which typically express the behaviour of logic programs. Such an objection could be answered by referring to the possibility of choosing the powerset `P(OUT)` (represented by e.g. `OUT→U1`) of the output space as the real output domain. Since this, however, would result in a change of the corresponding specification as well, making it quite unnatural, we prefer giving separate definitions for of multivalued programs. We will use the prefix `M-` to indicate modifications of "singlevalued" definitions.

```
M-PROGRAMS              ≡   IN:TYPES # OUT:TYPES # (IN → P(OUT) )

M-FULFILS(spec,p)       ≡   IN(spec)=IN(p) in TYPES & OUT(spec)=OUT(p) in TYPES &
                            ∀x:IN(spec). IC(spec)(x)⇒ body(p)(x)={y:OUT(spec)|IOR(spec)(x,y)}
M-SOLVABLE(spec)        ≡   ∃p:M-PROGRAMS. M-FULFILS(spec,p)
```

4.2 A formal investigation of approaches to program synthesis

It has widely been held that there are essentially two different ideologies in program synthesis.

- The so-called *theorem proving* or *AE* approaches [MW80, BC85, Fra85] arose from the idea that constructing a program and proving it logically correct should be done at the same time. Thus instead of first developing program code top-down and then verifying it bottom up by investigating properties of individual statements, loops, subprograms etc. a *constructive* proof for the (AE-) theorem

  $$\forall x : IN.\exists y : OUT.\ IC(x) \Rightarrow IOR(x, y)$$

  will be build and a program of a particular (functional) language will be extracted from it.

- In the *transformation based approaches* [BD77, MW79, Bib80, Hog81, SL89] a new predicate $P(x, y)$ representing a new program P with input x and output y is defined by

$$\forall x : IN. \forall y : OUT. IC(x) \Rightarrow P(x, y) \Leftrightarrow IOR(x, y)$$

  and the body for the program P is generated by transforming $IOR(x, y)$ in the above framwork until it is computationally convenient.

In the following we will investigate the main features (representation of the problem and correctness of the deductive method) of these approaches by embedding them into the framework just formalized. Doing so, we will show that from their theoretical capabilities there are essentially no differences between these two ideologies because they may be translated into each other.

As for the AE-approaches this means only formally justifying the chosen representation since the correctness of the deduction method is unquestionable, provided a proof is correct and no nonconstructive proof methods are used.

Theorem 4.1 `Justifying the AE-representation for program synthesis`

```
 ⊢ ∀spec: SPECIFICATIONS.
        ∀x:IN(spec).∃y:OUT(spec). IC(spec)(x)  ⇒  IOR(spec)(x,y)
      ≡  SOLVABLE( spec )
 ⊢ ∀spec: SPECIFICATIONS.
        ∀x:IN(spec).∃o:P(OUT(spec)). IC(spec)(x)  ⇒  o={y:OUT(spec)|IOR(spec)(x,y)}
      ≡  M-SOLVABLE( spec )
```

These two theorems, which can be derived from a more general theorem of Type Theory, simply make explicit one of the foundational properties of Type Theory, namely that every theorem has a constructive meaning which can be extracted from its proof.

As all theorems of our theory these two have three aspects: They *justify* the AE-representation by formally proving it correct *provided the constructive interpretation of logical formulae is used*. Applying the theorem means *executing* a transformation to switch between representations. Finally the *extracted algorithms* transform programs developed within the AE-framework into a pair $\langle program, proof \rangle$ proving solvability of the specification and vice versa.

Justifying the problem representation of transformation based approaches

The representation of the synthesis problem used by transformation based approaches can be easily justified if instead of a predicative program $P(x, y)$ we use a multivalued function p with $body(p)(x) := \{y : OUT(p)|P(x, y)\}$. A formal theorem that such a program can be constructed reads as follows

```
∃p:M-PROGRAMS.∀x:IN(p). IC(x)⇒  body(p)(x)={y:OUT(p)|IOR(x,y)}
```

This form which is nearly the same as the one of [SL89] (probably the most mature of all the approaches so far) is absolutely identical to `M-SOLVABLE(<IN, OUT, IC, IOR>)`, i.e. the synthesis problem for multivalued functions.

Correctness of the deduction method (applying transformations of logical formulae):

To make things clearer we first define a type of transformations reflecting the behaviour of the deduction method. Transformations leave all components but the input-output relation of a specification unchanged:

```
TRANSFORMATIONS         ≡  {t:SPECIFICATIONS→SPECIFICATIONS |
                            ∀s:SPECIFICATIONS. IN(T(s))=IN(s) in TYPES
                            & OUT(T(s))=OUT(s) in TYPES  & IC(T(s))=IC(s) in TYPES }
```

Theorem 4.2 `Justify the deduction method of applying transformations`

```
 ⊢ ∀T:TRANSFORMATIONS.∀spec:SPECIFICATIONS.
     M-SOLVABLE(spec)
      ⇐   M-SOLVABLE(T(spec))
          &∀x:IN(spec).∀y:OUT(spec). IC(spec)(x)⇒(IOR(spec)(x,y)≡  IOR(T(spec))(x,y))

 ⊢ ∀T:TRANSFORMATIONS.∀spec:SPECIFICATIONS.
     SOLVABLE(spec)
      ⇐   SOLVABLE(T(spec))
          &∀x:IN(spec).∀y:OUT(spec). IC(spec)(x)⇒(IOR(spec)(x,y)⇐IOR(T(spec))(x,y))
```

That is, if a transformation transforms IOR into some equivalent relation then its application is justified. For the singlevalued problem the requirements are even weaker. Obviously there principles behind these theorems which can be abstracted on some higher level. Transformation based approaches like LOPS often use the notions *correctness* or *equivalence preserving transformation.* From the above it should be clear what the meaning of these notions should be. We will capture this within a type definition and then restate the theorem in a clearer way.

```
C-TRANSFORMATIONS      ≡  {t:TRANSFORMATIONS|∀s:SPECIFICATIONS.∀x:IN(s).∀y:OUT(s).
                                 IC(s)(x) ⇒  (IOR(s)(x,y)  ⇐  IOR(T(s))(x,y)) }
EQ-TRANSFORMATIONS     ≡  {t:TRANSFORMATIONS|∀s:SPECIFICATIONS.∀x:IN(s).∀y:OUT(s).
                                 IC(s)(x) ⇒  (IOR(s)(x,y)  ≡  IOR(T(s))(x,y))}
```

The following version reveals the true principles behind Theorem 4.2. Correctness preserving transformations can be applied as one-directional deduction method to find singlevalued programs while equivalence preserving transformations can be applied back and forth for single- and multi-valued programs.

Theorem 4.3 `Correctness of transformations reformulated:`

```
 ⊢ ∀T:C-TRANSFORMATIONS.∀specification:SPECIFICATIONS.
        SOLVABLE(specification)     ⇐  SOLVABLE(T(specification))
 ⊢ ∀T:EQ-TRANSFORMATIONS.∀specification:SPECIFICATIONS.
        SOLVABLE(specification)     ≡   SOLVABLE( T(specification) )
     &  M-SOLVABLE(specification)   ≡   M-SOLVABLE(T(specification))
```

These theorems also represent an implementation method for transformation based approaches since applying them means *executing* the transformation in form of some deduction rule. Thus for a given transformation we only have to prove on a logical level that by its definition it is correctness- or equivalence preserving in order to get a valid and executable program deduction step. Combined with the theorems justifying the AE-representation this means that *every (C-/EQ-) transformation can be effectively converted into a valid proof rule for the AE-approach.*

The extract term of the theorems are algorithms creating a program for the original problem from a solution of the transformed one.

As a final example we will return to the GUESS strategy of LOPS which we now reformulate in terms of its true nature, i.e. as a transformation controlled by a set of parameters which are the type of the new variable, its domain condition and the "tautology" predicate. Note that for this the position of predicates and quantifiers had to be changed resulting in a form equivalent to the one given in [Bib80]. (Essentially independent conditions have to be moved across quantifiers.)

```
T_guess(A,dom,t)(<IN,OUT,IC,IOR>)
        ≡       <IN, OUT, IC, λi,y.∀g:A. dom(i,g)  ⇒  IOR(i,y) & (t(g,y)|¬t(g,y))>
```

The following theorem which may be considerd a reformulation of Theorem 3.3 gives clearer insights into the deductive behaviour the GUESS transformation.

Theorem 4.4 `Deductive properties of the GUESS-transformation`

```
 ⊢ ∀spec:SPECIFICATIONS.∀A:TYPES.∀dom:FORMULAE(IN(spec)#A).
      ∀t:FORMULAE(A#OUT(spec)).  T_guess(A,dom,t) in C-TRANSFORMATIONS
     &∀t:DFORMULAE(A#OUT(spec)). T_guess(A,dom,t) in EQ-TRANSFORMATIONS
   ⇐
      SOLVABLE(<IN(spec),A,IC(spec),dom>)
```

Thus it is clearly expressed that GUESS is a correctness preserving transformation (only) if *dom* is a programmable predicate and that it is equivalence preserving only if *t* is decidable which is true in most practical cases. The previous version (Theorem 3.3) may now be considered an instantiation of the above theorem in the AE-framework which can be constructed by applying Theorems 4.3 and 4.1.

5 Conclusion

We have presented the principles of a very expressive formal theory of reasoning about both the objects of programming as well as the deductive methods involved. We have shown that by abstracting from superfluous context used in many approaches to program synthesis we are able to make formal proofs much simpler and get deeper insights into the nature of programming. Our theory has been formalized in Intuitionistic Type Theory such that the concepts developed here can be immediately implemented with the proof system NuPRL. Obviously, we could only outline the beginning of a large theory. Many parts need to be worked out in further detail. In particular a complete meta-theory about all the deductive methods involved in programming should be formalized which allows a mechanized investigation of program construction processes on an abstract level and would lead to a generic program synthesizer as well.

Another appealing path to be followed is embedding already existing approaches to program synthesis into our framework. There is, for instance, already a theory behind the KIDS system [SL89] which proves its program construction methods correct on some mathematical level. However, it lacks uniformity and formality and cannot say anything about the actual implementation. These gaps could be filled by a further formalization within our theory.

Similar work by Paulson [Pau87] representing various logics by modelling their semantics in higher order logics (but not investigating proof calculi) indicates that our formalism is not restricted to the area of programming but can be generalized to reasoning about other deductive mechanisms like logical calculi as well.

References

[And86] Peter B. Andrews. *An Introduction to mathematical logic and Type Theory: To Truth through Proof.* Academic Press, Orlando, 1986.

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[BH84] Wolfgang Bibel and K. M. Hörnig. LOPS - a system based on a strategical approach to program synthesis. In Alan W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic program construction techniques*, chapter 3, pages 69–89. MacMillan, New York, 1984.

[Bib80] Wolfgang Bibel. Syntax–directed, semantics–supported program synthesis. *Artificial Intelligence*, 14(3):243–261, October 1980.

[CAB$^+$86] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Der85] Nachum Dershowitz. Synthesis by completion. In A. Joshi, editor, *IJCAI-85 – 9th International Joint Conference on Artificial Intelligence, Los Angeles, August 1985*, pages 208–214, 1985.

[Dij76] Edsger W. Dijkstra. *A discipline of Programming.* Prentice Hall, 1976.

[Fra85] Marta Franova. A methodology for automatic programming based on the constructive matching strategy. In *EUROCAL 85*, pages 568–570. Springer Verlag, 1985.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation.* Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[Gri81] David Gries. *The science of programming.* Springer Verlag, 1981.

[Hog81] C. J. Hogger. Derivation of logic programs. *Journal of the Association for Computing Machinery*, 28(2):372–392, April 1981.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *6-th International Congress for Logic, Methodology and Philosophy of Science, 1979*, pages 153–175. North–Holland, 1982.

[MW79] Zohar Manna and Richard J. Waldinger. Synthesis: Dreams $\Rightarrow$ programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, July 1979.

[MW80] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[Pau87] Lawrence C. Paulson. The representation of logics in higher-order logic. Technical Report 113, University of Cambridge. Computer Laboratory, August 1987.

[SL89] Douglas R. Smith and Michael R. Lowry. Algorithm design and design tactics. In L. van de Snepscheut, editor, *International Conference on the Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 379–398. Springer Verlag, June 1989.

[Smi88] Douglas R. Smith. KIDS — a knowledge-based software development system. In M. R. Lowry, R. McCartney, and D. R. Smith, editors, *Workshop on Automating Software Design. AAAI-88, St. Paul, MN, August 25, 1988*, pages 182–188, August 1988. (also Technical Report KES.U.88.7, Kestrel Institute, October 1988).