Towards a formal Theory of Program Construction

Christoph Kreitz

FG Intellektik – FB Informatik, Technische Hochschule Darmstadt
Alexanderstr. 10, D-6100 Darmstadt

ABSTRACT. *A unified framework for formal reasoning about programs* and *deductive mechanisms involved in programming is developed. Within it principal approaches to program synthesis are formally investigated. We will show that a high degree of abstraction opens a way to combine their strengths, simplifies formal proofs, and leads to clearer insights into the meta-mathematics of program construction. All definitions and theorems are presented completely formal which allows to straightforwardly implement them with a proof system for the underlying calculus and derive verified implementations of programming methods from them.*

1 Introduction

Since the upcoming of the so-called software crisis efforts have been put into the production of better software. Methodologists [Gri81, Dij76] have developed a science of programming to solve the problem that products of the software production business seldomly meet the original intentions of the clients and are very difficult to maintain and modify. To a large extent programming has been identified as a reasoning process on the basis of knowledge of various kinds, an activity in which people typically do a lot of mistakes. It is therefore strongly desirable to provide machine support for program construction which in principle, not necessarily in practice, means to aim at the automation of the whole programming process. This requires a full formalization of all its 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, Hog81, BH84, Der85, BC85, Fra85, CH88, NFK89, PM89, SL89, SL90, Gal90]). The KIDS system [SL90], most mature of the realized ones, is even be-

Revue d' intelligence artificielle, 4(3):53–79, November 1990

lieved close to the point where it can be used for routine programming. Current implementations of program synthesis systems, however, more or less 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 as well. After a while many program synthesizers tend to get quite bulky and improved versions again have to be build from scratch (c.f. experiences reported in [NFK89]). Often there are even doubts about the faithfulness of the implementation and it is not clear if or why they are correct.

We believe that a solution to these problems should be approached by developing program synthesizers following the same methodologies as used for the construction of conventional programs because there are essentially no differences except for a higher level of reasoning. Thus there is a need for tools capable of formal reasoning about both programs *and* deductive methods in programming. A theoretical foundation for them does not exist so far.

In our opinion this is due to a lack of abstraction when formalizing deductive mechanisms for program development. Lots of general deductive calculi (see e.g. [ML82, CH88, Gal90]) are known to be powerful enough that all kinds of programs can be derived within them. But for treating the problem these must be considered to be at the same level as assembler languages for writing programs. Too many details in the formalization cover up the true nature of the deductive mechanisms and make proofs about their properties unneccessarily complicated. The current lack of new ideas *how* to guide and control deductive mechanisms in programming is a natural consequence of that.

What is needed, therefore, is a unified framework in which programs *and* deductive methods in programming can be investigated both completely formal and on a sufficently high level. Complete formality shall allow both to mechanically verify object knowledge (e.g. domain knowledge, individual programs and specifications) and meta-knowledge of programming (e.g. deduction methods, synthesis techniques, algorithm knowledge, general properties of programs and specifications) and to derive verified implementations of deductive methods from the formal proofs. A high level of abstraction when formalizing programming concepts shall lead to clearer insights about their properties and to simpler proofs due to the absence of superfluous context. Such a *formal theory of program construction* shall be developed in this article.

A first step to make our framework useful for program construction is an investigation of principal approaches to program synthesis and of already known individual strategies, which we will begin here as well. This will already provide a collection of verified implementations of synthesis strategies and new insights into their properties. It also enables us to unify and extend them. On the long run the framework shall be used to derive a verified implementation of a program synthesis system for a certain class of problems by methods similar to those used within program synthesis itself. Doing so the resulting program synthesizer not only is correct, easy to maintain, and easy to modify, but there is also a clear

understanding of its behaviour and capabilities. This requires, however, besides deeper insights into the mathematics of program construction a lot of research in meta reasoning and self-reflection which has not been done yet.

Directing the formal theory towards the development of verified program synthesizers, i.e. towards representing deductive mechanisms for programming, proving them correct and creating a verified implementation, distinguishes our approach from those focusing on "low level" calculi [ML82, BC85, CH88, PM89, Coq89, Gal90], on metareasoning within a such a calculus [CK86, Kno87, How88b] or on implementing a deductive mechanism by representing it in some calculus [Pau87, HRS90, Web90].

To avoid creating a new logic and get distracted from our original goal we will select one of the already established general deductive calculi to underly our theory. By expressing high-level concepts in terms of low-level constructs complex special purpose reasoning then can be reduced to a series of simple reasoning steps in the low-level formalism. They can therefore be implemented with a reasoning tool for the underlying calculus which makes the step from a formulation of the theory to its actual implementation very small. In addition to that the underlying calculus will already provide a method for deriving verified implementations of deductive mechanisms from formal proofs which allows to concentrate our efforts in formal investigations. The underlying calculus must, of course, be a higher order theory and has to include a model for computation.

We selected Intuitionistic Type Theory as general formalism to underly our formal framework. Reasons for that are discussed in the following section where we also introduce syntax and features of the NuPRL proof development system for Type Theory [BC85, CAB$^+$86]. Section 3 explains the methodology we will follow. In Section 4 then the frame for reasoning about the object level of programming will be presented and illustrated by an exemplified formalization of strategies of the LOPS system [Bib80, BH84]. We will show that even a straightforward formalization already gives some new insights about the deductive method. Since, however, in such a direct approach general principles are covered up by individual notations we go for a higher degree of abstraction in Section 5. There we will discuss formal definitions of concepts involved in the programing process and investigate properties of the principal approaches to program synthesis. It will be shown that the differences between the two main ideologies (Theorem Proving approaches and transformation based approaches) exist only superficially. They can be effectively translated into one another, a simple result which due to a lack of abstraction 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 designing systems like program synthesizers and automated theorem provers one has to incorporate a wide variety of deductive techniques and mathematical knowledge. The system should therefore be founded on 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 which is both a formulation of a *constructive* higher-order logic and a model for datatypes and computation. Mathematical statements can be translated directly into the formal language since nearly all objects of mathematics have immediate counterparts in it. Of course, one has to assign a type to each mathematical object but this just formally reflects the fact that mathematicians naturally make distinctions between different types of objects. Thus type symbols even provide important syntactic clues which are not available in untyped theories. Furthermore, the whole conceptual apparatus of programming mirrors that of intuitionistic mathematics (c.f. [ML82] p.155) and can be immediately embedded as well. Therefore Intuitionistic Type Theory not only is expressive enough for a formalization of all the activities involved in program construction but it seems to us that the view of the world one gets from using it is the one most appropriate to underly a formal theory of program development.

Type Theory still is a comparably young formalism and there are various "dialects" of it [Chu40, Bru80, ML82, Smi84, And86, CAB$^+$86, CH88, PM89, Coq89, Gal90] each emphasizing a different aspect (like minimality, maximal expressivity, or optimal extracted algorithms) and using its own syntax. As formulation to underly our theory we selected the one of NuPRL [BC85, CAB$^+$86], a descendent of Martin-Löf's Type Theory [ML82], because an interactive environment for developing completely formal theories in NuPRL is already available. It shall later serve us as a tool for implementing our theory.

TYPE	CANONICAL MEMBERS		LOGICAL EQUIVALENTS
type constructors			
$A \to B$	$\lambda x.b$	if $b \in B$	A ⇒ B
$x : A \to B$	$\lambda x.b$	if $b \in B[x]$	∀x:A.B$^+$
$A\#B$	$\langle a, b\rangle$	if $a \in A, b \in B$	A ∧ B
$x : A\#B$	$\langle a, b\rangle$	if $a \in A, b \in B[a/x]^*$	∃x:A.B
$A\|B$	$inl(a), inr(b)$	if $a \in A, b \in B$	A ∨ 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, all types that can be constructed via type constructors.		
U_2	U_1, members of U_1, all that can be constructed via type constructors.		
U_3	U_2, members of U_2, all that can be constructed via type constructors.		
⋮	⋮		

$^*B[a/x]$: In B substitute a for x, $^+$∀x:A.B: B holds for all $x \in A$.

Figure 1: *NuPRL types and constructors*

2.2 NuPRL's proof calculus for Type Theory

Types and **members of types** 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), forms for making use of members of the type (like function application $f(a)$ and projection for pairs), and a notion of equality on the type. A cumulative hierarchy of universes U_i, introduced to deal with wellformedness problems, enables higher order reasoning in a very simple and natural way. Logical constructs are not explicitely part of the type system because each of them has a type construct with the same deductive rules as an immediate counterpart. The deductive rules of logical connective of conjunction $A\&B$, for instance, correspond to those of the type constructor for products $A\#B$, and intuitionistic disjunction $A \vee B$ corresponds to the disjoint union (or sum) of two types denoted by $A|B$. The **propositions-as-types** correspondence thus gives a constructive predicate logic with sorts for free. Syntax and some details of NuPRL's Type Theory are listed in Figure 1. For a full presentation we refer the reader to [CAB$^+$86] (chapter 8).

NuPRL statements are expressed in the form of *sequents*. These are objects of the form

$$x_1 : T_1, \ldots, x_n : T_n \vdash C \texttt{ [ext } m\texttt{]}$$

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 every theorem in Type Theory has a **computational content** since sequents implicitely describe an algorithm constructing a member of the conclusion (a witness for its truth) from the assumptions. Consequently, algorithms can be specified in form of mathematical propositions implicitely asserting their existence. The computational content of such a proposition is a program guaranteed to meet the specification and can be extracted from its proof. This so-called **proofs-as-programs** paradigm [BC85] is of particular importance for embedding our framework into NuPRL.

Unlike most other type theoretical calculi 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.

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

(where H stands for a list of hypotheses) corresponds to the following inference rule in the usual bottom-up style:

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.

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. A refinement rule must either be one of NuPRL's primitive inference rules or a *tactic*, i.e. a meta-language program controlling the application of other refinement rules. 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 provides a few features which support 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. It allows a user to write meta-programs guiding the application of refinement rules. Such tactics act as derived inference rules whose correctness is guaranteed by the fact that they have to make use of primitive inference rules to actually modify a proof. Experiments with NuPRL reported in [CAB$^+$86, How86, Kre86, Cle87, How88a, Bas89] show that together with the expressive power of Type Theory these components strongly support a flexible high level "implementation" of mathematical theories.

3 Methodology and notation

As said before the formal theory of program construction shall provide a unified framework for reasoning about programming at different levels of abstraction. For the sake of clarity we will formally separate these levels by distinguishing between reasoning about the object level and the meta-level of programming[1] and also between reasoning about individuals and classes of individuals in general which should be considered the framework for reasoning about individuals.

On the object level we have to deal with particular domain theories (like theories about numbers, lists, sets, or graphs), theories on individual programs and specifications, and the effects of applying a deductive mechanism to a concrete situation. Reasoning about object *classes* includes general properties of datatypes, logical formulae, and programs which then can be instantiated to individuals. Object classes may be considered meta-objects as well but are singled out due to their second nature as classes of first-order objects. Formalizing and investigating known deductive mechanisms and developing new ones are typical examples for reasoning about *meta-objects* which makes this part of the formal theory of program construction the most appealing one. *Meta-classes* deal with general properties of concepts like proofs, strategies, formula transformations, and syntheses. Using them, global properties and relations between meta-objects can be established on a very high level of abstraction which goes even beyond the meta-level of programming. In the development of the formal theory we will stepwisely raise the level of abstraction until on the level of meta-classes we are able to separate general properties of deductive mechanisms from those specific to a particular method.

To distinguish formal definitions and theorems from less formal explanations we will use the `typewriter` font for formal constructs. We will, though, still make use of special characters like $\vdash, \rightarrow$ despite the fact that in NuPRL they

[1] Not to be confused with the meta-level of the underlying theory which we never refer to unless explicitely stated

are simulated (by `>>, ->`) because they are not available on primitive terminals. `&`, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$, $\forall$, $\exists$ will be used for intuitionistic logical connectives which can be defined in terms of type constructors as indicated in Figure 1.

Deductive mechanisms should be considered as inference rules of some yet unexpressed calculus. Formal metatheorems intended to reflect their behaviour will therefore resemble the outer form of NuPRL proof rules stating that - given a certain context - by the deductive mechanisms some main goal may be deduced from a list (conjunction) of subgoals. We use reverse implication "$\Leftarrow$" to simulate the refinement style and avoid using parentheses if a graphical separation of goals appears sufficient:

$\vdash$ $\forall$`<vars involved in the context>.`
`Main goal`
$\Leftarrow$ `Subgoal`$_1$
$\vdots$
`&` `Subgoal`$_n$

Once proven each metatheorem representing a deductive method has four important aspects by which it becomes equal to any other NuPRL inference rule and may be used as such from then on:

Representation: A *completely formal representation* of the behaviour of the deductive method is given.

Justification: The deductive mechanism represented by the theorem is proven to be a *logically correct reasoning step*.

Implementation: Applying the theorem to a particular situation means to *execute* the mechanism. That is, if the problem to be solved matches the main goal it will be replaced by the subgoals which now have to be dealt with.

Algorithm construction: The completely formal and mechanically verified proof of the theorem contains an algorithm constructing a program solving the main goal from partial programs for the subgoals. This aspect, of course, only makes sense if the deductive method represented is used for program development. The algorithm can be accessed as the extract term of the theorem.

In most cases the four aspects of a particular metatheorem are obvious from the above description. We will only mention those who give new insights. All theorems are proven completely formal using NuPRL's primitive inference rules and tactics. Since presenting the full proof would take too much space on paper we only sketch them and verbally express the the rules applied.

New concepts will be introduced by formal definitions in the style of NuPRL's definition mechanism. `<New Object>` $\equiv$ `<Formal NuPRL Representation>` defines a new type-theoretical object having the syntax of the left-hand side in terms of already existing constructs given on the right hand side of the definition. Note, that deduction rules for a newly defined concept follow immediately from those of the right hand side and can either be proven as a metatheorem or directly be programmed as a tactic. The definition $\forall$`x:A.prop` $\equiv$ `x:A`$\rightarrow$`prop`, for instance allows to use the familiar syntax of the typed universal quantifier which

is represented by the constructor for the dependent function type and to use the deductive rules of the function type for the universal quantifier as well.

4 The frame for the object level

In this section we will develop a frame for reasoning about the object level of programming and illustrate its usage by an exemplified formal investigation of some program construction strategies and pieces of a domain theory. We restrict ourselves to formalizing object knowledge needed in later sections.

4.1 Object classes for object language expressions

The most important classes needed for reasoning about global properties of objects are the class of first-order logical formulae and the class of first-order domains or datatypes. Despite of the propositions-as-types correspondence these notions are quite different in their intuitive interpretation and in the way they are used in practice. They shall therefore be distinguished in our formal framework.

We will represent the two classes by types called `FORMULAE` and `TYPES`. Both must be subtypes of the universe U_1 of first-order objects and it is reasonable to choose the most simple definition by identifying them with U_1 itself. Formulae with variables from a type `T` will be members of the type `FORMULAE(T)`.

Definition 4.1 [**Datatypes and logical formulae**]

`TYPES`	$\equiv$ `U1`
`FORMULAE`	$\equiv$ `U1`
`FORMULAE(T)`	$\equiv$ `T` $\rightarrow$ `FORMULAE`

These definitions form the object language of programming whose semantics and deduction rules now implicitly follow from that of Type Theory. Note that due to the above intensional definition the syntactic structure of a type or formula cannot be fully accessed from a theory built within NuPRL. We would have to make use of the meta-level of Type Theory, i.e. ML-programs, to do so. Of course, one might give an exhaustive extensional definition using NuPRL's recursive types [CM85, Men87] and apply methods developed for partial reflection and formal metamathematics in NuPRL [CK86, Kno87, How88b] but this would create a big overhead. It will turn out that for investigating program construction methods we can do without this syntactic information which would rather burden us with superfluous details and does not give any additional insights.

Given the above definitions not all first-order formulae are decidable ($p \vee \neg p$ or $p \Rightarrow q \Leftrightarrow \neg p \vee q$ may be false) and not all first-order domains are discrete (equality on the type may not be decidable). This is due to the constructive aspect of the underlying theory. Nearly all predicates occurring in practice, however, are decidable and often use of such knowledge is made while constructing a program. In order to catch it we will introduce decidable subclasses of `FORMULAE` and `TYPES` and use them whenever it is appropriate for the problem.

Definition 4.2 [**Discrete types and decidable formulae**]

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

Discrete types and decidable formulae are closed under the most important operations, as the following lemma shows.

Lemma 4.3 `Closure properties of decidable types and formulae`

```
1.⊢∀T,S:DTYPES. (T→S∈DTYPES) & (T#S∈DTYPES)  & (T|S∈DTYPES)
              & (T list∈DTYPES)
2.⊢∀p,q:DFORMULAE.          (p&q ∈DFORMULAE) & (p∨q∈DFORMULAE)
      & (¬p∈DFORMULAE) & (p⇒q∈DFORMULAE) & (p⇔q∈DFORMULAE)
```

This can be shown by truth table proofs and, in the case of `T list`, an induction over the length of lists.

4.2 Domain theories

In a theory of program construction domain theories represent knowledge about the application domains which an algorithm design system needs to fulfil its task. Formal definitions introduce new notions. In formal theorems important properties are stated and verified. During the past years, a number of domain theories have been implemented with NuPRL. We refer the reader to [CAB+86, How86, Kre86, Cle87, How88a] and particularly to [Bas89] for accounts of how mathematical knowledge should be represented and reasoned about. Here we will present some elements of a theory about finite sets over ordered types which we will use in Section 4.3.

The Set-constructor yields for every type T the type $TSet$ of all finite sets over elements of T. Formally, sets can be simulated by the list-constructor modulo a set-equality notion. Inductive definitions on sets thus are expressed by the predefined list-induction construct. The empty set $\emptyset$ is represented by the empty list. Two lists are equal as sets if the have the same elements. $S \setminus x$ means taking out an element x from a set S. Orderings are represented by relations on a type satisfying the typical axioms of a total order.

Definition 4.4 [**Finite sets over ordered types**]

```
∅               ≡ nil
S∪{y}           ≡ y.S
{y}             ≡ ∅∪{y}
x∈S:T Set       ≡ if S=L∪{y} then (x=y∈T ∨x∈L:T Set) else false
L=S∈T Set       ≡ ∀x:T. x∈L:T Set ⇔ x∈S:T Set
T Set           ≡ L,S:T list // (L=S∈T Set)
S\x:T Set       ≡ if S=L∪{y} then (if x=y∈T then L\x else L\x∪{y})
                                 else ∅
x ≤ y           ≡ ≤(x,y)
```

```
ORDERINGS(T)      ≡ {≤:FORMULAE(T#T)| ∀a,b,c:T. ( a≤b ∨ b≤a )
                      & (a≤b & b≤a ⇒ a=b∈T) & (a≤b & b≤c ⇒ a≤c)}
x<y:T             ≡ x≤y & x≠y∈T
S≤x:T Set         ≡ ∀x:T. x∈L:T Set ⇒ y≤x
m=max(S):T set    ≡ S≠∅ ⇒ (m∈S:T Set & S≤m:T Set)
```

The following lemma is a collection of basic facts about sets and orderings. Its formal proofs are lengthy inductions although not very complicated.

Lemma 4.5 Basic properties of sets and orderings

```
1.⊢∀T:TYPES.∀≤:ORDERINGS(T).∀a,b,c:T.
   (¬a<a:T) & (a<b:T ∨ a=b∈T ∨ b<a:T) & ((a<b:T & b<c:T) ⇒ a<c:T)
2.⊢∀T:DTYPES.∀≤:ORDERINGS(T).   ≤∈DFORMULAE(T#T)
3.⊢∀T:TYPES.∀S:T Set.∀x,y:T.    ¬(y∈∅:T Set)
       &  S={x}∈DFORMULAE
       &  S={x}:T Set ⇒ S\x=∅:T Set
       &  S≠∅:T Set ⇔ S\x≠∅:T Set ∨ S={x}:T Set
       &  x∈S:T Set ⇒ (y∈S\x:T Set ⇔  y∈S:T Set & x≠y∈T)
       &  x∈S:T Set ⇒ (S≤y:T Set   ⇔  S\x≤y:T Set & x≤y)
```

4.3 An example: representing strategies of LOPS

By an exemplified investigation of some synthesis strategies we will now illustrate how deductive methods for program construction can be represented by formal theorems reflecting their behaviour. As running example we chose key strategies of the LOPS (LOgic Program Synthesis) system [Bib80, BH84, Fro85, NFK89] which may be briefly summarized as follows:

> Starting with a specification given as a first-order formula of the form
>
> $$\forall \mathrm{i}\ \exists \mathrm{y}\ (\ \mathrm{IC(i)}\ \Rightarrow \mathrm{OC(i,y)}\ )$$
>
> 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 of some particular target language. This goal is approached by a series of equivalence or correctness preserving transformations guided by a few strategies supported by deductive tools.
>
> A LOPS synthesis is centered around the strategies GUESS-DOMAIN and GET-REC. Both may require some additional pre- and postprocessing and, depending on the problem, some specialized strategies may have to support them. All LOPS strategies are designed to leave open some choices which could be made by a programmed heuristic or an assisting user.

Our formalization of LOPS strategies follows the original description of [Bib80]. Besides illustrating of the principles of our formal theory it will give us some new insights into the true nature of the LOPS approach and "mechanical" proofs for the correctness of the strategies involved. It is, however, not our intention to improve LOPS strategies or to discuss or further investigate ideas behind heuristics determining parameters for them. This will be left to future investigations.

4.3.1 GUESS-DOMAIN

The strategy GUESS-DOMAIN 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. It consists of a transformation GUESS and a heuristic DOMAIN determining all the necessary parameters for the transformation which takes a formula of the form

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

and transforms it into

$$\forall \text{i } \forall \text{g } \exists \text{y dc(i,g) } \Rightarrow (\text{IC(i) } \Rightarrow \text{OC(i,y) } \wedge (\text{g=y} \vee \text{g} \neq \text{y})).$$

This transformation means to **guess** some hopefully correct output g. In order to meaningfully restrict the search for g by some domain condition $dc(i, g)$ it is suggested to choose $dc(i, g)$ from among the subsets of the conjuncts in $OC(i, g)$ such that it will be possible to compute some g with $dc(i, g)$. $g = y$ represents a successful guess, $g \neq y$ a failure. Both alternatives, however, contain more information than the initial formula thus making the remaining task easier. If the output domain is not a simple datatype, only partial information about the output can be guessed. In such cases the relation $g = y$ has to be replaced by some more general "tautology"-predicate $t(g, y)$.

The above description of the GUESS transformation suggests it to be a proof rule for an $\forall\exists$-quantified theorem. The following formal metatheorem is a direct representation of this rule. Note that for this we have assigned types to each variable and quantified over the formulae and types involved.

Theorem 4.6 GUESS transformation

```
⊢∀IN,OUT:TYPES.∀IC:FORMULAE(IN).∀OC,dc:FORMULAE(IN#OUT).
     ∀i:IN.∃y:OUT. IC(i) ⇒ OC(i,y)
      ⇐   ∀i:IN.∀g:OUT.∃y:OUT.
            dc(i,g) ⇒ IC(i) ⇒ OC(i,y) & (g=y∈OUT ∨ g≠y∈OUT)
        & ∀i:IN.∃g:OUT. IC(i) ⇒ dc(i,g)
```

Note that the second subgoal is necessary for the correctness of the GUESS transformation and thus puts an effectivity condition on the selection of *dc*. This new insight too is a valuable effect of a strict formalization.

To illustrate how simple a formal proof of such a typical metatheorem can be we sketch a NuPRL proof of Theorem 4.6. Hypotheses will be numbered. In subgoals we show new hypotheses and the current goal only.

Proof: *By "introduction" rules move all assumptions to the hypotheses list*

```
1.-4. IN:TYPES, OUT:TYPES, IC:FORMULAE(IN), OC:FORMULAE(IN#OUT)
5. dc:FORMULAE(IN#OUT)
6. ∀i:IN.∀g:OUT.∃y:OUT. dc(i,g) ⇒ IC(i) ⇒ OC(i,y) & (g=y∈OUT ∨ g≠y∈OUT)
7. i:IN
8. ∀i:IN.∃g:OUT. IC(i) ⇒ dc(i,g)
     ⊢ ∃y:OUT. IC(i) ⇒ OC(i,y)
```

Instantiate 8. on `i`

`9. ∃g:OUT. IC(i) ⇒dc(i,g)`
`⊢ ∃y:OUT. IC(i) ⇒OC(i,y)`

Eliminate the existential quantifier in 9. giving a name to the object

`10.-11. g:OUT, IC(i) ⇒dc(i,g)`
`⊢ ∃y:OUT. IC(i) ⇒OC(i,y)`

Instantiate 6. on `i` *and* `g`*, eliminate the existential quantifier in the result.*

`12. ∃y:OUT. dc(i,g) ⇒ IC(i) ⇒ OC(i,y) & (g=y ∈ OUT ∨ g≠y ∈ OUT)`
`13.-14. y:OUT, dc(i,g) ⇒ IC(i) ⇒ OC(i,y) & (g=y ∈ OUT ∨ g≠y ∈ OUT)`
`⊢ ∃y:OUT. IC(i) ⇒OC(i,y)`

Choose `y` *from 13. as solution, move the assumption* `IC(i)` *to the hypotheses list.*

`15. IC(i)`
`⊢ OC(i,y)`

Eliminate the implications of 11, 14, and of the result of this (17).

`16.-17. dc(i,g), IC(i) ⇒ OC(i,y) & (g=y ∈ OUT ∨ g≠y ∈ OUT)`
`18. OC(i,y) & (g=y ∈ OUT ∨ g≠y ∈ OUT)`
`⊢ OC(i,y)`

The goal follows from splitting 18. □

Before executing the next strategy the result of GUESS-DOMAIN will be transformed into some normal form separating a successful guess from the failure case. This strategy, which [Bib80] calls GET-DNF, will also be represented as a theorem expressing a proof rule to be applied immediately after GUESS-DOMAIN.

Theorem 4.7 GET-DNF

```
⊢∀IN,OUT:TYPES.∀IC:FORMULAE(IN).∀OC,dc:FORMULAE(IN#OUT).
   ∀i:IN.∀g:OUT.∃y:OUT. dc(i,g) ⇒ IC(i) ⇒ OC(i,y)&(g=y in OUT∨g≠y in OUT)
      ⇐ ∀i:IN.∀g:OUT.∃y:OUT. dc(i,g) ⇒
            IC(i) ⇒ OC(i,y)& g=y in OUT  ∨  IC(i) ⇒ OC(i,y)& g≠y in OUT
```

4.3.2 The strategy GET-REC

Depending on the results of GUESS-DOMAIN, the strategy GET-REC tries to introduce recursion. For this an abstract recursion scheme providing information on possible recursions on the objects involved will be instantiated. Using knowledge about the particular domain the current formula then will be rewritten according to the instantiated scheme. In contrast to the other LOPS strategies this operation does not contain a well defined transformation. Instead, under the guidance of domain knowledge and the chosen recursion scheme, a situation specific one is created heuristically. Therefore GET-REC cannot be formalized in full generality without first having investigated abstract recursion schemes and their effects on guiding heuristics. It would lead us beyond the scope of this article to do so here. We will therefore explain the operation of GET-REC by some small standard example using one of the most typical abstract recursion schemes and leave generalizations to the future.

Example: The maximum of a finite set: (c.f. [Bib80])

An algorithm calculating the maximum `m` of a finite set `S` over some arbitrary ordered type `OUT` shall be synthesized. According to Definition 4.4 the following specification has to be considered as top goal:

```
top
⊢ ∀OUT:TYPES.∀≤:ORDERINGS(OUT). ∀S:OUT Set.∃m:OUT. S≠∅ ⇒ (m∈S & S≤m)
```

Thus `IN` is instantiated to `OUT Set`, `IC(S)` to `S≠∅`, and `OC(S,m)` to `m∈S & S≤m`. For the sake of legibility the obvious '`:OUT Set`' has been omitted. After guessing with `dc(S,g):=g∈S` and normalization (theorems 4.6, 4.7) we have the following subgoals (labelled by their path to the top goal):

```
top 1
⊢ ∀S:OUT Set.∀g:OUT. ∃m:OUT. g∈S ⇒
          S≠∅ ⇒ m∈S & S≤m & g=m  ∨  S≠∅ ⇒ m∈S & S≤m & g≠m
top 2
⊢ ∀S:OUT Set.∃g:OUT. S≠∅ ⇒ g∈S
```

Subgoal 2 follows from the constructive definition of `g∈S`. We assume that an algorithm `some-element-of` calculating `g` from `S` already exists. In the first alternative of subgoal 1 `g=max(S)` is a solution. Thus we are left with `g∈S ⇒ S≠∅ ⇒ m∈S & S≤m & g≠m` where GET-REC shall introduce recursion.

Structures like Lists, Sets, Trees, and Graphs on some ground type `A` have in common that they come with some ground object `base` (e.g. the empty list, set, tree, or graph) and a function "\" which from a structured object and one of `A` computes a new structured object with lesser complexity (e.g. takes an element out of a list or a node out of a tree) . If we give the name `A Struct` to this kind of structures then an abstract recursion scheme for all of them can be represented as a theorem about structural induction on the types involved.

Theorem 4.8 Recursion by taking out one element

```
⊢∀A:TYPES.∀P:FORMULAE(A Struct).
    ∀i:A Struct. P(i)
      ⇐  P(base)  &  ∀s:A Struct.∀g:A. ( P(s\g) ⇒ P(s) )
```

We give a proof for the case of lists. The proof can be generalized to the above form but this would involve introducing recursive types or complexity measures and lots of new terminology.

Proof: *Move all assumptions to the hypotheses list*

```
1.-2. A:TYPES, P:FORMULAE(A list)
3.-5. P(nil),  ∀s:A list.∀g:A. (P(s\g) ⇒P(s)),  i:A list
    ⊢ P(i)
```

Unroll `i` *according to its recursive definition: it is either the empty list* `nil` *or some concatenation* `a.i1` *of an element* `a` *and a list* `i1`. `P(nil)` *follows from 3.*

```
6.-8. a:A,  i1:A list,  P(i1)
    ⊢ P(a.i1)
```

Instantiate 4. on `a.i1` *and* `a`.

```
9. P(a.i1\a) ⇒P(a.i1)
    ⊢ P(a.i1)
```

Substitute `a.i1\a` *with* `i1` *in 9. and apply it to hypothesis 8.* □

Instantiated to a programming problem on sets Theorem 4.8 proves that programs over a finite set `i` can be computed recursively by expressing `P(i)` in terms of `P(i\g)` and giving some basic solution. Making use of this knowledge the strategy GET-REC tries to rewrite the result of guessing in terms of `i\g`.

Example: Rewrite $g \in S \Rightarrow S \neq \emptyset \Rightarrow m \in S \,\&\, S \leq m \,\&\, g \neq m$ in terms of $S \setminus g$:
This operation involves a heuristic component and search methods from theorem proving to find appropriate lemmata of the corresponding domain theory which rewrite the formula or parts of it. The exact behaviour of GET-REC, e.g. a general rule *how* to create or look for such lemmata, has not yet been described in literature on LOPS and is open to further investigations.

Here it finds in Lemma 4.5(3) the equivalences $S \neq \emptyset \Leftrightarrow S \setminus g \neq \emptyset \vee S = \{g\}$, $g \in S \Rightarrow (S \leq m \Leftrightarrow S \setminus g \leq m \;\&\; g \leq m)$, and $g \in S \Rightarrow (m \in S \;\&\; g \neq m \Leftrightarrow m \in S \setminus g)$ and applies them (resulting in subgoal 1 of subgoal 1):

```
top 1 1
⊢∀S:OUT Set.∀g:OUT. ∃m:OUT. g∈S ⇒
        g=max(S) ∨ ( (S\g≠∅∨S={g}) ⇒ m∈S\g & S\g≤m & g≤m & g≠m )
```

Similarly to GUESS-DOMAIN, the result of GET-REC needs to be postprocessed, which means to distribute the resulting alternatives in the input condition and then to fold back the original definition of the problem specification.

Example: The result of postprocessing is:

```
top 1 1 1
⊢∀S:OUT Set.∀g:OUT. ∃m:OUT. g∈S ⇒
  g=max(S) ∨ (m=max(S\g) & g≤m & g≠m ∨ S={g} ⇒ m∈S\g & S\g≤m & g≤m & g≠m)
```

4.3.3 Further strategies

In [Bib80] other strategies which we did not make use of have been introduced: GET-EP tries to make a predicate evaluable by replacing ineffective components. If there is more than one output variable, GET-RNV tries to reduce their number, GET-SOC separates the output conditions, CHVAR chooses an output variable for guessing. All these should be formalized in the future.

4.3.4 Program construction

Before being transformed into a program, some control information will be added. This includes dropping superfluous parts, folding definitions which are known to be computable. Again these steps will be explained using the example.

Example: Because of $S = \{g\} \Rightarrow S \setminus g = \emptyset$ (Lemma 4.5(3)) $m \in S \setminus g$ is false. Thus the alternative $S = \{g\}$ will not lead to a solution in the failure case. The contradiction is dropped adding $S \neq \{g\}$ as control information to the other alternative. After folding the definition of `<` the final result is:

```
top 1 1 1 1
⊢ ∀S:OUT Set.∀g:OUT. ∃m:OUT.g∈S ⇒ g=max(S) ∨ S≠{g} & max(S\g)=m & g<m
```

[Bib80] proposes to construct a program by an algorithmic reading of the formula. This means making use of the (meta-)knowledge that a solution of the final formula is also one for the original one and transforming the (exclusive) disjunction into a case analysis according to the decidable predicates mentioned.

Example: Making use of the above metaknowledge means to recursively define the program according to an instantiation of the recursion scheme 4.8:

```
∀OUT:TYPES.∀≤:ORDERINGS(OUT).
  ∀S:OUT Set.∃m:OUT. m=max(S)
     ⇐ ∃m:OUT. m=max(∅)
       & ∀S:OUT Set.∀g:OUT. ∃m:OUT.m=max(S\g) ⇒ ∃m:OUT.m=max(S)
```

dropping the base case which has been explicitely excluded. `S={g}` can be decided once `g` and `S` are known, `g<m` after `m` is set to `max(S\g)`. The result must be `g` or `m`. Thus we get the following algorithm (in ML-like syntax):

```
letrec max(S) = let g=some-element-of S in
                if S={g} then g
                         else let m=max(S\g) in if g<m then m else g
```

To reflect the true behaviour of LOPS, a full formalization must express the usage of meta-knowledge in program extraction as well. Thus the real frame for a LOPS synthesis must be the recursion scheme chosen by GET-REC which represents the algorithmic structure of the resulting program. All the operations performed by LOPS strategies, including the methods for program extraction described above, only reduce the induction step by operating on the right hand side of the implication. From a practical point of view LOPS is simply a shortcut in this chain of reasoning and it is only natural that later implementations of LOPS [NFK89] chose to abandon the framework of ∀∃-quantified formulae in favour of a representation fitting the LOPS approach better.

We will demonstrate this by running the maximum example in the correct framework. The control decisions now are explicitely mentioned.

Example 4.9 Complete synthesis of the maximum algorithm

```
top
⊢ ∀OUT:TYPES. ∀≤:ORDERINGS(OUT).  ∀S:OUT Set.∃m:OUT. m=max(S).
```
Apply the scheme and move assumptions to the hypotheses list

```
top 1
 1.-2. OUT:TYPES, ≤:ORDERINGS(OUT),
   ⊢ ∃m:OUT. m=max(∅)      any m solves this since the input condition ∅≠∅ is false
```

```
top 2
 1.-4. OUT:TYPES, ≤:ORDERINGS(OUT), S:OUT Set, g:OUT,
 5.     ∃m:OUT. m=max(S\g)
   ⊢ ∃m:OUT. m=max(S)
```
apply GUESS and GETREC as described

```
top 2 1
    ⊢ ∃m:OUT. g∈S ⇒   g=max(S) ∨ S≠{g} & max(S\g)=m & g<m
```
Consider cases `S={g} ∨  S≠{g}`

```
top 2 1 1
   6. S={g}
     ⊢ ∃m:OUT. g∈S ⇒  g=max(S) ∨ S≠{g} & max(S\g)=m & g<m
   choose  g because the other alternative leads to a contradiction
top 2 1 2
    6. S≠{g}
      ⊢ ∃m:OUT. g∈S ⇒  g=max(S) ∨ S≠{g} & max(S\g)=m & g<m
    eliminate 5., consider cases  ¬g<m ∨  g<m
top 2 1 2 1
    7.-9.  m:OUT, m=max(S\g), ¬g<m
       ⊢ ∃m:OUT. g∈S ⇒  g=max(S) ∨ S≠{g} & max(S\g)=m & g<m
    choose  g because the other alternative leads to a contradiction
top 2 1 2 2
    7.-9.  m:OUT, m=max(S\g), g<m
      ⊢ ∃m:OUT. g∈S ⇒  g=max(S) ∨ S≠{g} & max(S\g)=m & g<m
     choose  m, all conditions of the second alternative are satisfied
 top 2 2
    ⊢ ∃g:OUT. S≠∅ ⇒ g∈S    solved with  some-element-of
```

The algorithm NuPRL extracts from this is exactly the one described above.

5 Towards higher abstraction

Despite the fact that a direct formalization of a particular deductive method already gives some insights general principles can hardly be discovered that way. They are still hidden behind superfluous details and it is desirable to study them from a higher degree of abstraction. We begin this with a complete formalization of notions which are known to be important in the field of programming. Formal representations are obvious once we have discussed the nature of the notion to be formalized. We will then 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.

5.1 Program Construction Concepts

The task of developing a program to solve a certain problem can be divided into three major steps: from an informal description one first has to figure out what exactly the problem is, then from the formal specification develop an algorithm how to solve it, and finally write it down encoded in some programming language. Due to its very nature the first step is hardly formalizable but for the step from formal specification to written computer code strong automatic support is possible. Program synthesis has devoted itself to the latter in order to enable programmers and clients to concentrate their efforts in analyzing the problem and fixing a formal description for it. To solve its task it may use knowledge about the application domains, about algorithms in general, and previously defined programs. Since building a program synthesizer essentially is a programming problem as any other we first have to focus our attention on giving a formal description of the synthesis problem.

A program synthesizer must be able to transform formal specifications into programs fulfilling them. All approaches to systematic program development agree that a programming problem has to be described by *specifying* its input-domain IN, the output-domain OUT, and a relation IOR between input and the desired output. Many of them use an additional precondition IC on the input to exclude singular cases like division by zero (see e.g. [MW80, Bib80, SL89]). Obviously, the domains IN and 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 should have quadruples $\langle IN, OUT, IC, IOR\rangle$ as its elements where the *types* of IC, IOR depend on the *values* of IN, OUT. This is best expressed by a dependent product:

Definition 5.1 [**The Datatype of program specifications**]
`SPECIFICATIONS` $\equiv$ `IN:TYPES×OUT:TYPES×FORMULAE(IN)×FORMULAE(IN#OUT)`

Without desiring to enter philosophical discussions about the nature of programs we assume that their main purpose is to take some input and calculate some output from it. Thus a program essentially 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

Definition 5.2 [**The Datatype of programs**]
`PROGRAMS` $\equiv$ `IN:TYPES×OUT:TYPES×(IN→OUT)`

Destructors accessing individual components of a program `p`$=\langle$`IN, OUT, body`$\rangle$ or a specification `sp`$=\langle$`IN, OUT, IC, IOR`$\rangle$ will be denoted by names like `IN(p)`.

A program p fulfils a specification sp 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.

Definition 5.3 [**The relation "A program fulfils a specification"**]
`FULFILS(sp,p)` $\equiv$ `IN(sp)=IN(p)` $\in$ `TYPES & OUT(sp)=OUT(p)` $\in$ `TYPES`
`& ∀x:IN(sp). IC(sp)(x)` $\Rightarrow$ `IOR(spec)(x,body(p)(x))`

A specification is solvable if a program can be found which fulfils it. Since in Type Theory the statement "a program can be found" is identical to "a program exists" we come to the following definition:

Definition 5.4 [**The relation "A specification has a solution"**]
`SOLVABLE(sp)` $\equiv \exists$`p:PROGRAMS. FULFILS(sp,p)`

This is the key predicate for the following investigations since constructing a program for a given specification `sp` is the same as proving `SOLVABLE(sp)`.

On the long run our theory shall provide a means to derive a verified implementation of a program synthesis system for a certain class of problems. Although we cannot yet say how to do this we can already give a formal specification of the problem. Obviously this cannot be a member of the first-order

type SPECIFICATIONS but the same structure may be used to provide a way for self-reflection. Knowing that in general program synthesis is an unsolvable problem we have to include as input-condition a parameter pc describing the class of problems on which the program synthesizer shall operate. Giving a formal description of a sufficiently large class will be an interesting research topic.

SYNTHSPEC(pc) ≡ <SPECIFICATIONS, PROGRAMS, pc, FULFILS>

is a specification of a program synthesizer for the class pc and building a synthesizer means to construct a program fulfilling it (by showing its solvability):

SYNTHESIZABLE(pc) ≡ ∃synth:SPECIFICATIONS->PROGRAMS.
∀sp:SPECIFICATIONS. pc(sp) ⇒ FULFILS(sp,synth(sp))

One may object that these definitions do not include partial functions or multivalued ones which typically express the behaviour of logic programs or those used in [SL89]. Such an objection could be answered by referring to the possibility of choosing the powerset $P(OUT)$ 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 constructs into their multivalued counterparts.

Definition 5.5 [**Multivalued programming concepts**]

M-PROGRAMS ≡ IN:TYPES×OUT:TYPES×(IN→P(OUT))
M-FULFILS(sp,p) ≡ IN(sp)=IN(p)∈TYPES & OUT(sp)=OUT(p)∈TYPES
& ∀x:IN(sp). IC(sp)(x)
⇒ body(p)(x)={y:OUT(sp)|IOR(sp)(x,y)}
M-SOLVABLE(sp) ≡ ∃p:M-PROGRAMS. M-FULFILS(sp,p)

These definitions deserve further discussion. From a theoretical point of view they make the treatment of multivalued programs quite simple. If such programs shall actually run we must clarify how the output body(p)(x) shall be represented. Since we want to access individual output values we suggest to represent a set $o \in P(OUT)$ by functions enumerating its elements. This is possible as long as OUT is a denumerable domain which is true in nearly all applications. Thus the equation body(p)(x)={y:OUT(sp)|IOR(sp)(x,y)} defines the synthesis problem to finding an extensional definition of a set from a given intensional one. We will later see how this influences the approaches to solve it.

5.2 Principal approaches to program synthesis

Essentially there are two different ideologies to approach program synthesis:

- The so-called *theorem proving* or *AE* approaches [MW80, BC85, Fra85, CH88, Gal90] 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 (functional) program guaranteed to fulfil the specification $\langle IN, OUT, IC, IOR\rangle$ will be extracted from it.

The advantage of this direction is its solid theoretical foundation. A lot of research has been done in developing appropriate logical calculi (see e.g. [ML82, BC85, CH88, PM89, Coq89, Gal90]) and algorithms extracting (efficient) programs from proofs within these [Hay86, CH88, PM89].[2] However, AE approaches suffer from the fact that, if general theorem proving strategies like resolution are used, the method a program is constructed appears to be quite unnatural. Furthermore, there is only little practical experience in constructing AE proofs for a larger class of programming problems.

- In *transformation based approaches* (see e.g. [BD77, MW79, Bib80, Hog81, BH84, Der85, SL89, SL90]) a specification is treated as if it were already some kind of logic program, though an inefficient one. Instead of constructing a correct program the main aspect is to improve efficiency which shall be achieved by adapting methods from the field of program transformations.

  Starting with a specification $\langle IN, OUT, IC, IOR\rangle$ a new predicate $P(x, y)$ representing a 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 framework until it is computationally convenient.

  Practical aspects and efficiency considerations have been the driving force of these approaches. Therefore the practical results of already existing systems like KIDS [SL90] are much better than those of AE based ones. In more recent approaches also strong theoretical foundations have been incorporated (see e.g. [SL90] where domain knowledge and program construction methods have been investigated). Since, however, the degree of formalization is way below that of the AE approaches it is not clear how the implementations reflect the theoretical foundations.

So far it has not been attempted to combine the strengths of the two directions since they appeared too different. These differences, as we will show, exist only on the surface. If both approaches are embedded into a unified formal framework then they may be translated into each other and the differences disappear. If in addition to that they are stripped of their syntactic peculiarities by looking at them from a higher degree of abstraction one may also get much clearer insights into their essential properties and use that for building a verified program synthesizer unifying the approaches.

This we will begin in the rest of the article. By embedding the two directions into the framework formalized above we will be able to investigate their principal features - the representation of the problem and the correctness of the deductive method. These investigations, particularly Theorems 5.6 and 5.10, will open the way for combining the approaches so far.

[2] In NuPRL's Type Theory such an algorithm is inherent to the rules and NuPRL may therefore be counted as calculus for an AE approach as well.

5.2.1 Justifying AE approaches

Theorem 5.6 The AE representation for program synthesis

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

Theorem 5.6 gives a justification of the AE representation: provided a constructive interpretation of logical formulae, proving an AE theorem for a given specification is equivalent to solving the specification itself, i.e. constructing a program fulfilling it. Thus applying the theorems means *executing* a transformation to switch between representations and the extracted algorithms transform the results developed within the AE framework into a program solving the specification and a proof for that or vice versa. Since there is no doubt about the correctness of the deductive method (giving a completely formal proof in a well known calculus) these theorems give us a *formal justification of the AE approach* to program synthesis as such.

The proof of Theorem 5.6 is nothing but instantiating a general type theoretical theorem which we present in a first-order version to avoid introducing new notation. It makes explicit the constructive aspect of Type Theory: every theorem has a constructive meaning which can be extracted from its proof.

Theorem 5.7 The constructive aspect of type theory

```
⊢∀IN,OUT:TYPES.∀p:FORMULAE(IN#OUT).
    ∀x:IN.∃y:OUT. p(x,y)   ⇔   ∃f:IN→OUT.∀x:IN.p(x,f(x))
```

Proof: *Move all assumptions to the hypotheses list, add an identifier to the last one*

```
1.-3. IN:TYPES, OUT:TYPES, p:FORMULAE(IN#OUT)
4. id: ∀x:IN.∃y:OUT. p(x,y)
     ⊢ ∃f:IN→OUT.∀x:IN.p(x,f(x))
```

Since propositions are types `id` *is a member of the type* `x:IN→(y:OUT#p(x,y))`*, i.e. a function returning* `y` *and a proof for* `p(x,y)` *for each* `x`*.*
Introduce `λx.(y where id(x)=<y,proof>)` *for* `f` *and* β*-reduce* `f(x)`

```
5. λx.(y where id(x)=<y,proof>) ∈ IN→OUT
     ⊢ ∀x:IN.p(x, (y where id(x)=<y,proof>) )
```

Moving `x` *to the hypotheses list and instantiating 4 with* `x` *proves the goal* □

5.2.2 Justifying Transformation based approaches

The **representation of the synthesis problem** used by transformation based approaches can be justified if the new predicative program $P(x, y)$ is expressed by a multivalued program p with $body(p)(x) := \{y : OUT(p)|P(x, y)\}$. To show that such a program can be constructed means to prove the following:

```
∃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] is absolutely identical to our M-SOLVABLE(<IN,OUT,IC,IOR>) predicate, the synthesis problem for multivalued functions, and allows controlling the effects of transforming $IOR(x, y)$ by theorems of some abstract algorithm theory.

Thus, the outer representation of the synthesis problem (AE theorem versus a new predicate) is not really a distinctive feature of the two ideologies. Due to Theorem 5.6 they can be translated into each other. Since, as Theorem 5.10 will show, even the deductive methods of each direction can be embedded into the framework of the other, there are no essential differences between them except for a willful restriction to a certain kind of synthesis strategies (standard proof methods versus transformations). This is a result which due to a lack of abstraction has not been proven yet.

Starting with such a representation of the problem it appears only natural to adopt the ideology of transformation based approaches. Given an enumerable output domain one may in fact consider $p := \lambda x.\{y : OUT(p)|IOR(x, y)\}$ to already describe a solution of the specification, though a very inefficient one. The remaining problem thus would be finding a computationally more convenient term to express p, a task for which program transformations are known to be helpful. To ensure the correctness of this deductive method transformation based approaches often use the notion of a "correctness" or "equivalence preserving" transformation. The following theorem will clarify the meaning of these notions.

Theorem 5.8 The deduction method of applying transformations

```
⊢∀IN,OUT:TYPES.∀IC:FORMULAE(IN).∀IOR,ior:FORMULAE(IN#OUT)
    ( M-SOLVABLE(<IN,OUT,IC,IOR>)
      ⇐  M-SOLVABLE(<IN,OUT,IC,ior>)
        & ∀x:IN.∀y:OUT. IC(x) ⇒ (ior(x,y) ⇔ IOR(x,y))  )
 & ( SOLVABLE(<IN,OUT,IC,IOR>)
      ⇐  SOLVABLE(<IN,OUT,IC,ior>)
        & ∀x:IN.∀y:OUT. IC(x) ⇒ (ior(x,y) ⇒ IOR(x,y))  )
```

Proof: We prove the singlevalued case

```
1.-4. IN:TYPES, OUT:TYPES, IC:FORMULAE(IN), IOR: FORMULAE(IN#OUT)
5. ior:FORMULAE(IN#OUT)
6. ∀x:IN.∀y:OUT. IC(x) ⇒(ior(x,y) ⇒ IOR(x,y))
7.-8. p:PROGRAMS, FULFILS(<IN,OUT,IC,ior>,p)
    ⊢ ∃p:PROGRAMS. FULFILS(<IN,OUT,IC,IOR>,p)
```

Instantiate the definition of FULFILS *and split the conjunction*

```
9.-10. IN=IN(p)∈TYPES, OUT=OUT(p)∈TYPES
11. ∀x:IN. IC(x)⇒ior(x,body(p)(x))
    ⊢ ∃p:PROGRAMS. IN=IN(p)∈TYPES
        & OUT=OUT(p)∈TYPES & ∀x:IN. IC(x)⇒IOR(x,body(p)(x))
```

Choose p *from 7. as solution. The first two conjuncts follow from 9. and 10. Move* x *and* IC(x) *to the hypotheses list.*

```
12.-13. x:IN, IC(x)
    ⊢ IOR(x,body(p)(x))
```

The goal follows by instantiating 6. and 11. on x, body(p)(x), *and* IC(x). □

That is, if a transformation transforms the input-output relation IOR into some equivalent relation *ior* while leaving IN, OUT and IC unchanged then its application is justified. For the singlevalued problem the requirements are even weaker. Obviously there are principles behind these theorems which should be abstracted on some higher level. This will be done in the next section.

As we have seen, both directions approaching program synthesis can be properly expressed by theorems using our SOLVABLE predicate and formally be justified. From now on we will approach a synthesis problem in the frame of some formal theorem using the concepts defined in Section 5.1. Although methods for constructing programs will then be formally represented by proof rules we are not restricted to a single type of rules (like resolution). Any other approach can be translated into a rule scheme and its advantages and disadvantages can thus be investigated within our framework.

5.3 Meta-classes for deductive methods

In our opinion most insights into the mathematics of program construction can be gained by studying deductive methods at the level of meta-classes. For that one must in the end represent a complete deductive calculus for reasoning about objects in programming. This involves formal definitions of notions like sequents, logical deductions, proof rules, proofs, theorems, program derivations, syntheses etc. and formal theorems about their properties. We will demonstrate the principles of such a theory by presenting a small piece.

We will focus our attention on abstracting the transformation based approaches to program synthesis. According to Theorem 5.8 transformations shall modify only the input-output relation of a specification and leave all other components unchanged. Preserving equivalence or correctness of the input-output relation is viewed in the context of the input condition. We will capture this in definitions of the corresponding types.

Definition 5.9 [Datatype for Transformations]

```
TRANSFORMATIONS   ≡ {t:SPECIFICATIONS→SPECIFICATIONS |
                     ∀s:SPECIFICATIONS. IC(T(s))=IC(s) ∈ FORMULAE(IN)
                    & IN(T(s))=IN(s) ∈ TYPES & OUT(T(s))=OUT(s) ∈ TYPES}
C-TRANSFORMATIONS≡ {t:TRANSFORMATIONS|∀s:SPECIFICATIONS.∀x:IN(s).
                    ∀y:OUT(s). IC(s)(x) ⇒ (IOR(T(s))(x,y) ⇒ IOR(s)(x,y))}
E-TRANSFORMATIONS≡ {t:TRANSFORMATIONS|∀s:SPECIFICATIONS.∀x:IN(s).
                    ∀y:OUT(s). IC(s)(x) ⇒ (IOR(T(s))(x,y) ⇔ ,OR(s)(x,y))}
```

The following reformulation of Theorem 5.8 reveals the true principles behind it. Correctness preserving transformations can be applied as a one-directional deduction method to find singlevalued programs while equivalence preserving transformations can be applied back and forth for single- and multi-valued programs. As usual this is proven by reasoning about the effects of a transformation relative to solvability of a specification.

Theorem 5.10 Applying transformations

```
⊢∀T:C-TRANSFORMATIONS.∀sp:SPECIFICATIONS.
        SOLVABLE(sp)     ⇐ SOLVABLE( T(sp) )
⊢∀T:E-TRANSFORMATIONS.∀sp:SPECIFICATIONS.
        SOLVABLE(sp)     ⇔  SOLVABLE( T(sp) )
     &  M-SOLVABLE(sp)   ⇔  M-SOLVABLE( T(sp) )
```

The proof follows from that of Theorem 5.8. A closer look at it shows that correctness-preserving transformations *do not change the resulting program*: every solution for the transformed problem is already a solution for the original one. This perfectly reflects the ideology of transformation based approaches which is transforming a specification until it can be used as a program.

An important effect of these theorems is that they translate transformations into (verified) deduction rules for program synthesis since applying them means *executing* the transformation in form of some deduction rule. If we can prove that a logical definition represents a correctness- or equivalence preserving transformation then Theorem 5.10 gives us valid and executable program deduction step. Together with Theorem 5.6 we may effectively convert every (C-/E-) transformation into a valid proof rule for the AE approach and thus combine the two directions.

As a final example we will return to the GUESS strategy of LOPS which we now reformulate in full generality replacing equality `g=y` by a more general "tautology" predicate `t(g,y)`. This formulation will also reflect the true nature of GUESS. It is a transformation controlled by a set of parameters: the type of the new variable `g`, 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].

```
Guess(A,dc,t)(sp) ≡  <IN,OUT,IC, ∀g:A.dc(i,g) ⇒ IOR(i,y)&(t(g,y)∨¬t(g,y))>
                     - where sp=<IN,OUT,IC,IOR>
```

The following theorem which may be considerd a reformulation of Theorem 4.6 gives clearer insights into the deductive behaviour the GUESS transformation.

Theorem 5.11 Properties of the GUESS-transformation

```
⊢∀sp:SPECIFICATIONS.∀A:TYPES.∀dc:FORMULAE(IN(sp)#A).∀t:FORMULAE(A#OUT(sp))
   ( Guess(A,dc,t) ∈ C-TRANSFORMATIONS
       ⇐ SOLVABLE(<IN(sp),A,IC(sp),dc>)
 & ( Guess(A,dc,t) ∈ E-TRANSFORMATIONS
       ⇐ SOLVABLE(<IN(sp),A,IC(sp),dc>) & t ∈ DFORMULAE(A#OUT(sp))
```

Thus GUESS is a correctness preserving transformation (only) if *dc* is a computable predicate and it is equivalence preserving only if t is decidable which is true in most practical cases. The previous version (Theorem 4.6) may now be considered an instantiation of the above theorem in the AE framework which can be constructed by applying Theorems 5.10 and 5.6.

6 Conclusion

We have presented a unified framework for formal reasoning about deductive mechanisms and their application domains as well and demonstrated how to make use of such a framework. We have shown that it provides a means to combine the strengths of the existing approaches to program synthesis. Due to a complete formalization the concepts presented here can be directly implemented with a proof system for the underlying theory. Such an implementation will help to uncover issues we may have overlooked when first developing the theory on paper and also yields verified implementations of strategies which have been formally investigated.

Obviously this article can only be the starting point for a mechanized investigation of the programming process on a high level of abstraction. Many parts need to be worked out in further detail. In particular, all components involved in the programming process must be formalized to prove general correlations between them. Embedding already existing approaches to program synthesis (like LOPS [Bib80, BH84] or KIDS [SL89, SL90]) into our framework is another path to be followed. Behind the KIDS system, for instance, there is already a theory 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. Such gaps could be filled by formal investigations.

The principles discussed here are not restricted to the area of program construction. They can, as research on representing various logics in higher-order logic [Pau87] indicates, be generalized to reasoning about other deductive mechanisms as well.

Acknowledgements

We thank Robert Constable and the NuPRL group of Cornell University for helpful discussions when preparing the beginnings of this theory, particularly for hints on self-reflection mechanisms and practical limitations of implementing Type Theory.

References

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

[Bas89] D. Basin. Building theories in NuPRL. In A. R. Meyer and M. A. Taitslin, editors, *Logic at Botik 89*, pages 12–25. Springer Verlag, 1989.

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

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

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

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

[Bru80] N. G. De Bruijn. A survey of the project AUTOMATH. In J.P. Sedlin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[CAB+86] R. L. Constable et. al. *Implementing Mathematics with the NuPRL proof development system*. Prentice Hall, 1986.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

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

[CK86] R. L. Constable and T.B. Knoblock. Formalized metareasoning in Type Theory. In *Symposium on Logic in Computer Science*, 1986.

[Cle87] W. R. Cleaveland. Type-theoretic models of concurrency. Report TR 87-837, Cornell University. Department of Computer Science, 1987.

[CM85] R. L. Constable and P. Mendler. Recursive definitions in Type Theory. In *Logics of Programs Conference*, pages 61–78. Springer Verlag, 1985.

[Coq89] T. Coquand. Metamathematical investigations of a calculus of constructions. Report 1088, Institut National de Recherche en Informatique et en Automatique, 1989.

[Der85] N. Dershowitz. Synthesis by completion. In *IJCAI 85*, pages 208–214, 1985.

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

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

[Fro85] B. Fronhöfer. The LOPS-approach: Towards new syntheses of algorithms. In H. Trost and J. Retti, editors, *ÖGAI 85*, pages 164–172. Springer, 1985.

[Gal90] D. Galmiche. Constructive system for automatic program synthesis. *Theoretical Computer Science*, 71:227–239, 1990.

[GMW79] M. J. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation*. Springer Verlag, 1979.

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

[Hay86] S. Hayashi. PX: a system extracting programs from proofs. In *IFIP Conference on Formal Description of Programming Concepts*, pages 399–424, 1986.

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

[How86] D. Howe. Implementing number theory: An experiment with NuPRL. In *CADE 8*, pages 404–415. Springer, 1986.

[How88a] D. Howe. Automating reasoning in an implementation of constructive Type Theory. Report TR 88-925, Cornell University. Department of Computer Science, 1988.

[How88b] D. Howe. Computational metatheory in NuPRL. In *CADE 9*, pages 238–257, 1988.

[HRS90] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In M. Stickel, editor, *CADE 10*, pages 117–131, 1990.

[Kno87] T. B. Knoblock. Metamathematical extensibility in Type Theory. Report TR 87-892, Cornell University. Department of Computer Science, 1987.

[Kre86] C. Kreitz. Constructive automata theory implemented with the NuPRL proof development system. Report TR 86-779, Cornell University. Department of Computer Science, 1986.

[Men87] P. Mendler. Inductive definition in Type Theory. Report TR 87-870, Cornell University. Department of Computer Science, 1987.

[ML82] P. 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] Z. Manna and R. Waldinger. Synthesis: Dreams ⇒ programs. *IEEE Transactions of Software Engineering*, SE-5(4):294–328, 1979.

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

[NFK89] G. Neugebauer, B. Fronhöfer, and C. Kreitz. XPRTS - an implementation tool for program synthesis. In D. Metzing, editor, *GWAI 89*, pages 348–357. Springer Verlag, 1989.

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

[PM89] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *ACM Symposium on Principles of Programming Languages*, pages 89–104, 1989.

[SL89] D. R. Smith and M. R. Lowry. Algorithm design and design tactics. In *International Conference on the Mathematics of Program Construction*, pages 379–398. Springer Verlag, 1989.

[SL90] D. R. Smith and M. R. Lowry. KIDS - a knowledge-based software development system. In M. R. Lowry and R. McCartney, editors, *Automating Software Design*, Menlo Park, 1990.

[Smi84] J. Smith. An interpretation of Martin-Löf's Type Theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[Web90] M. Weber. Formalization of the Bird-Meertens algorithmic calculus in the DEVA meta-calculus. In *IFIP Working Conference on Programming Concepts and Methods*, 1990.