

---

# Formal Mathematics for Verifiably Correct Program Synthesis

CHRISTOPH KREITZ, *Fachgebiet Intellektik, Fachbereich Informatik,  
Technische Hochschule Darmstadt, Alexanderstraße 10,  
D-64283 Darmstadt, Germany.*  
*E-mail: kreitz@intellektik.informatik.th-darmstadt.de*

## Abstract

We describe a formalization of the meta-mathematics of programming in a higher-order logical calculus as a means to create verifiably correct implementations of program synthesis tools. Using reflected notions of programming concepts we can specify the actions of synthesis methods within the object language of the calculus and prove formal theorems about their behavior. The theorems serve as derived inference rules implementing the kernel of these methods in a flexible, safe, efficient and comprehensible way. We demonstrate the advantages of using formal mathematics in support of program development systems through an example in which we formalize a strategy for deriving global search algorithms from formal specifications.

*Keywords:* Program Synthesis, Type Theory, Reflected Reasoning

## 1 Introduction

For more than twenty years commercial software production has been in a state of endemic crisis. The crisis is caused by the property which makes software attractive: the complexity of behavior that can be produced. Its effects manifest themselves in the *cost of software* over its life cycle, emphasized by the fall in hardware cost, and the *lack of confidence* in software, which limits the extent to which digital control is adopted in safety-critical areas. Since the emergence of the crisis attempts have been undertaken to develop techniques for a production of reliable software. 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 tend to make a lot of mistakes. Therefore it is desirable to provide machine support for software development and to develop tools for *knowledge based software engineering*. Besides obtaining an accurate statement of the requirements this means *synthesizing* computer code from formal specifications. Since computerized reasoning can handle only formal objects this requires a formalization of all kinds of programming knowledge. Research in the field of *program synthesis* is active in two areas: investigations into *logical calculi* which support such a formalization of programming knowledge and *synthesis strategies* which generate programs from specifications.

Many formal calculi are, at least in principle, powerful enough to express all of mathematics and programming (see e.g. [16, 4, 7, 5, 24]). But there remains a problem of expressiveness in practical applications. Inference steps in these logics operate on a very low level and make program derivations long and difficult to comprehend. It is almost impossible for a human programmer to guide formal derivations which cannot

be found automatically. Therefore, less rigorous methods are used when developing strategies. During the last decades many approaches (see e.g. [9, 17, 18, 2, 26]) have been implemented and tested successfully for a number of examples. The KIDS system [26, 29] is believed to be close to the point where it can be used to develop small routine programs. But while the theoretical foundations of these strategies are thoroughly investigated their implementations are created ‘ad hoc’ rather than systematically. It is not clear how the systems reflect these foundations. Instead, program synthesizers have the same problems as conventional software: only specialists are able to handle them properly, unexpected errors occur, and after a while they become difficult to maintain and modify (cf. experiences reported in [19]). Most researchers are aware of these inadequacies but shy away from the amount of labour which is necessary to overcome them.

So far the most fruitful approach to bridge the gap between formal deduction and complex applications has been that of *tactics*, first introduced in Edinburgh LCF [8], and since adopted in many other systems (see e.g. [20, 4, 21, 3, 11]). Here deductive methods are written as *meta-programs* guiding the application of inference rules. Tactics are a means to combine the advantages of formality with those of high-level methods. However, all programming knowledge (about the strategies they perform and the kind of results they generate) is implicitly contained in the code which makes maintenance and modifications of deductive systems still very complicated. Thus the construction of flexible program synthesizers whose derivations are both correct and comprehensible for programmers is still an unsolved problem in both artificial intelligence and software engineering.

We believe that a solution to this problem should be approached by specifying the actions of program derivation methods in a comprehensible but completely formal logical language and proving formalized theorems about their behavior. This would allow to combine the strengths of human programmers and computers – creativity and intelligence on one side and a capability for complex formal reasoning without errors on the other – for the development of flexible and reliable software. Since the current level of abstraction in formal calculi is too low for this purpose one should first raise this level to one that can be handled by human programmers and then specify program-synthesizing strategies on that level.

These goals can be achieved by a rigorous formalization of all relevant programming knowledge – about various domains of discourse as well as about program construction *methods* – in terms of mathematical definitions and theorems which can be verified with a computerized proof system. In other words, one should raise the level of abstraction in formal reasoning by completely formalizing the object- and meta-theory of programming within the object language of some logical calculus. Formal definitions (i.e. mathematical abbreviations) should be used to represent concepts from various domains of discourse as well as (reflected notions of) programming concepts. Formal meta-theorems<sup>1</sup> should be used to make explicit *semantic* knowledge about program derivation methods, which currently is hidden within the implementation code of synthesis strategies. This allows formal reasoning about the behavior of derivation methods and provides a means for generating flexible, verifiably correct, and efficient implementations.

---

<sup>1</sup>The notion ‘meta’ refers to the meta-level of *programming* – i.e. to meta-level concepts like specifications, programs, correctness, or program development methods – which is represented within the *object* language of the formal calculus.

Although such a rigorous approach can rely on several well-known theoretical insights and techniques it requires a great amount of work before practical results will show up. Nevertheless it seems to be the only way to overcome the current difficulties. Making semantic knowledge about a deductive method explicit helps to separate it from purely syntactical search techniques. Thus writing tactics can be reduced to encoding methods to search for applicable theorems. Another advantage is that the mathematical language makes it possible to integrate approaches to program synthesis which currently appear to be incomparable. Furthermore, the gap between formal and ‘humanly comprehensible’ reasoning can be bridged by expressing formal theorems in a programmer’s terminology.

In [14] we have begun the formalization of basic programming concepts and shown how the synthesis paradigms of proofs-as-programs and synthesis by transformations are reflected in such a framework. Since then we have refined our approach by elaborating a formal meta-theory of programming and begun an implementation with the NuPRL proof development system [4]. We have investigated translations between synthesis paradigms and represented synthesis strategies of the systems LOPS [2] and KIDS [27, 28, 29]. The results, which are presented in detail in the author’s technical report [15], show that a rigorous approach does in fact lead to safe and efficient program synthesis systems.

In this presentation we want to demonstrate that it is possible to represent enough reflected semantics in an already-existing system to make deductive methods more rigorous in a practical setting and to show the advantages of using formal mathematics as a foundation for program synthesis. Section 2 summarizes the basic concepts of the formal theory of programming. Section 3 describes how to represent deductive techniques by formal theorems. In section 4 we demonstrate the capabilities of our approach by a concrete formalization of a strategy for designing global search algorithms. We conclude with a few remarks on improvements achieved and future prospects for using formal mathematics in automated software development.

## 2 Representing programming concepts by formal definitions

The formal theory of programming is structured like a typical mathematical textbook: first basic concepts and notations are defined and then lemmata and theorems are developed. The only difference is that all definitions are expressed as abbreviations for terms of some logical calculus and all theorems have to be proven in that calculus. This allows one to ‘implement’ the theory presented on paper without any modifications by using a proof system for the underlying calculus. As a basic calculus we have selected the formulation of intuitionistic type theory [16] used by the NuPRL proof development system [4]. Type theory already provides formalizations of a constructive higher order logic and low level constructs such as integers, strings, (recursive) function spaces, products, sums, lists, etc. as well as the corresponding type constructors. Conservative extensions of the formal language can be introduced by definitional equality (which roughly corresponds to a text macro):

*textual representation of the new object*  $\equiv$  *formal representation in type theory*

We use `typewriter` font to denote formal expressions (important keywords will be written in `sans serif`) and *math*-font to highlight formal parameters in a definition.

TYPES	Class of <i>first level</i> data types
$\overset{=}{\alpha}, \overset{\neq}{\alpha}$	Equality decision procedures for type $\alpha$
let $x$ =term in expr, expr where $x$ =term	abstraction over a term
letrec $f(x)$ =body	(Recursive) function definition <i>(predefined in [4, ch. 12])</i>
dom( $f$ )( $x$ )	$x$ is in the domain of $f$ <i>(predefined in [4, ch. 12])</i>
letrec $f(x)$ =body in $t$	abstraction over a recursive function
$\alpha \times \beta, \langle a_1, \dots, a_n \rangle, a.i$	Product type declaration, Tuple, $i$ -th projection
let $\langle a_1, \dots, a_n \rangle = p$ in expr	Local assignment of projections to the $a_i$
$\mathbb{B}, \text{true}, \text{false}$	Data type of boolean expressions, explicit truth values
$\neg, \wedge, \vee, \Rightarrow, \Leftarrow, \Leftrightarrow$	Boolean connectives
$\forall x \in S.p, \exists x \in S.p$	Limited boolean quantifiers (on finite sets and sequences)
if $p$ then $a$ else $b$	Conditional
$\mathbb{Z}, \mathbb{N}, 0, 1, -1, \dots$	Number types, explicit numbers
$+, -, *, /, \text{mod}, \text{max}, \text{min}$	Arithmetic operations
$=, \neq, \leq, <, \geq, >$	Arithmetic comparisons
Seq( $\alpha$ )	Data type of finite sequences over members of $\alpha$
null?, $\in_{\alpha}, \sqsubseteq_{\alpha}$	Decision procedures: emptiness, membership, prefix
$[], [a], [i..j], [a_1 \dots a_n]$	Empty and singleton sequence, integer subrange, literal sequence former
$a.L, L.a$	prepend $a$ , append $a$ to $L$
$[f(x) \mid x \in L \wedge p(x)],  L , L_i$	General sequence former, length of $L$ , $i$ -th element,
domain( $L$ ), range( $L$ )	The sets $\{1.. L \}$ and $\{L_i \mid i \in \text{domain}(L)\}$
nodups $_{\alpha}(L)$	Decision procedure: all the $L_i$ are distinct (no duplicates)
Set( $\alpha$ )	Data type of <i>finite</i> sets over members of $\alpha$
empty?, $\in_{\alpha}, \subseteq_{\alpha}$	Decision procedures: emptiness, membership, subset
$\emptyset, \{a\}, \{i..j\}, \{a_1 \dots a_n\}$	Empty set, singleton set, integer subset, literal set former
$S+a, S_{-}a$	element addition, element deletion
$\{f(x) \mid x \in S \wedge p(x)\},  S _{\alpha}$	General set former, cardinality
$S \cup T, S \cap T, S \setminus T$	Union, intersection, set difference
$\bigcup \text{FAMILY}, \bigcap \text{FAMILY}$	Union, intersection of a family of sets
$S \hat{=} \{x:\alpha \mid P(x)\}$	$S$ has the same elements as the NuPRL <i>set type</i> $\{x:\alpha \mid P(x)\}$

FIG. 1. Formalized Domain Vocabulary (Conservative Extensions of NuPRL)

To represent the *objects* of programming we had to extend the basic language by a collection of definitions for data types in general as well as operations on boolean expressions, natural numbers, finite sets, finite sequences, and finite maps. Some of these operations like  $\cup$  can be defined polymorphically while operations like  $\in_{\alpha}$  depend strongly on an equality decision procedure  $\overset{=}{\alpha}$  on the underlying data type<sup>2</sup>  $\alpha$ . Formally, they have to receive  $\alpha$  as parameter in an index if the theory shall be handled by a computer system. When being displayed on a computer screen, however, they should be suppressed because they make formal theorems somewhat difficult to read. Since the NuPRL system (version 4) supports such a flexible display form we shall omit the indices from now on. Figure 1 lists and explains the extensions used in this paper. Further ones (about 120) as well as the formal definitions and a few hundred lemmata stating their essential properties can be found in full detail in the author's technical report [15]. The resulting mathematical language serves both as specification and programming language.

<sup>2</sup>Since in general equality decision procedures for members of type  $\alpha$  cannot be derived from  $\alpha$  we have introduced a new class TYPES of first level data types. It denotes the collection of all tuples  $\langle \underline{\alpha}, \overset{=}{\alpha} \rangle$  where  $\underline{\alpha}$  is a type of the universe  $\mathbb{U}$  and  $\overset{=}{\alpha}$  is an equality decision procedure for members of  $\underline{\alpha}$ . All application domains whose members are finite objects (such as integers, finite sets, sequences, maps, graphs etc.) can be represented as elements of TYPES.

To support *reasoning* about specifications, programs, correctness, and program derivations we have formalized the corresponding classes in terms of appropriate higher order data types called SPEC and PROGRAMS. A program *specification* provides a declarative description of the problem. It consists of a domain type  $D$ , a range type  $R$ , a predicate  $I$  restricting  $D$  to *legal* inputs, and a predicate  $O$  relating input and feasible output. A *program* contains both a specification and a program body which operationally describes an algorithm. The latter is a *partial* function from the domain  $D$  to the range  $R$  (denoted by  $D \not\rightarrow R$ ). A program  $p$  is *correct* if the declarative specification and the operational program body describe the same behavior: for each given legal input  $x$  the computation of  $\text{body}(x)$  terminates and yields a *feasible* output value  $z$  satisfying the relation  $O(x, z)$ . We also say that  $\text{body}$  *computes* the specification and define a specification to be *satisfiable* if there is a program body which computes it.

DEFINITION 2.1 (Programming Concepts)

SPEC	$\equiv D:\text{TYPES} \times R:\text{TYPES} \times D \rightarrow \mathbb{B} \times D \times R \rightarrow \mathbb{B}$
$D(\text{spec})$	$\equiv \text{let } \langle D, R, I, O \rangle = \text{spec} \text{ in } D$
$R(\text{spec})$	$\equiv \text{let } \langle D, R, I, O \rangle = \text{spec} \text{ in } R$
PROGRAMS	$\equiv \text{spec}:\text{SPEC} \times D(\text{spec}) \not\rightarrow R(\text{spec})$
$p$ is correct	$\equiv \text{let } \langle \langle D, R, I, O \rangle, \text{body} \rangle = p$ $\quad \forall x:D. I(x) \Rightarrow \text{body}(x) \text{ halts} \wedge O(x, \text{body}(x))$
$\text{body}$ computes $\text{spec}$	$\equiv \langle \text{spec}, \text{body} \rangle$ is correct
$\text{spec}$ is satisfiable	$\equiv \exists \text{body}:D(\text{spec}) \not\rightarrow R(\text{spec}). \text{body computes spec}$

For individual programs and specifications we provide a more convenient notation separating the components by keywords. (In the following we only need the versions concerning set-valued outputs.)

DEFINITION 2.2 (Programming notation)

FUNCTION $f(x:D):R$ WHERE $I_x$ RETURNS $z$ SUCH THAT $O_{xz}$	$\equiv \langle D, R, \lambda x. I_x, \lambda x, z. O_{xz} \rangle$
FUNCTION $f(x:D):\text{Set}(R)$ WHERE $I_x$ RETURNS $\{z \mid O_{xz}\}$	$\equiv \text{FUNCTION } f(x:D):\text{Set}(R) \text{ WHERE } I_x$ RETURNS $S$ SUCH THAT $S \hat{=} \{z:R \mid O_{xz}\}$
FUNCTION $f(x:D):\text{Set}(R)$ WHERE $I_x$ RETURNS $\{z \mid O_{xz}\} = \text{body}_x$	$\equiv \langle \text{FUNCTION } f(x:D):\text{Set}(R) \text{ WHERE } I_x \text{ RETURNS } \{z \mid O_{xz}\},$ $\quad \text{letrec } f(x) = \text{body}_x \rangle$

Satisfiability of specifications is the key notion for program synthesis. Due to the constructivity of the logic proving the existence of a program body computing a given specification is the same as proving that such a body *can be constructed*. Therefore Proving satisfiability of a given specification is the same as deriving a program for it.

### 3 Representing deductive techniques by formal metatheorems

Synthesizing a verifiably correct program requires the application of formal deductions. While most deductive systems restrict themselves to ‘low-level steps’ like elementary inference rules or lemmata about domain knowledge, systems aiming at real applications like program synthesis must be able to cooperate with a user. Thus there is a need for inferences corresponding to programming concepts or synthesis methods *as a*

*whole.* To ensure the correctness of such ‘high-level’ deductions we have to formalize their essential parts as meta-theorems about a method. Formal meta-theorems therefore play a key role in the formulation of deductive systems which are safe, efficient and comprehensible.

Except for the fact that its variables must be quantified and typed a formal meta-theorem is almost identical to its informal counterpart. The following theorem, for instance, deals with the well-known relation between program synthesis and proving a ‘specification theorem’.

**THEOREM 3.1** (Proofs-as-Programs (cf. [14], Theorem 10))

$$\forall \text{spec} = \langle D, R, I, O \rangle : \text{SPEC}. \text{spec} \text{ is satisfiable} \Leftrightarrow \forall x : D. I(x) \Rightarrow \exists y : R. O(x, y)$$

This theorem is a variant of the Martin L of’s constructive axiom of choice [16, p. 50] and follows from the constructive nature of the underlying calculus. Within the context of software development systems, however, it has several additional aspects which are common to all formal meta-theorems representing deductive methods:

1. Its formulation gives a *completely formal representation* of a derivation method. Informally theorem 3.1 reads ‘to prove the satisfiability of the specification **spec** (i.e. to derive a program) it is sufficient to find a proof for the goal  $\forall x : D. I(x) \Rightarrow \exists y : R. O(x, y)$  and vice versa’.
2. A *justification* of the derivation method is provided by the proof of a meta-theorem. It shows that it is in fact sufficient to solve certain subgoals (i.e. to prove the specification theorem) in order to find a solution for the initial goal (i.e. to derive a program satisfying the specification). Thus proving theorem 3.1 yields a *formal justification* of the (reflected) *proofs-as-programs paradigm* in program synthesis.
3. A formal proof of a theorem is much more than just a proof that a certain statement is correct. Its realization with a computerized proof system like NuPRL also provides the basis for a *verified ‘implementation’* of a high-level inference rule which corresponds to derivation method represented by a formal meta-theorem. Theorem 3.1, for instance, can be viewed as *top-down inference rule* applicable to goals of the form ‘derive a program satisfying a specification **spec**’. To execute this rule one simply has to match the left hand side of the equivalence against the actual goal, instantiate the variables accordingly and – applying modus ponens – replace the goal by the instantiated right hand side (it is easy to write a tactic which performs these steps). The result of applying theorem 3.1 would thus be a specification theorem corresponding to the initial specification.

Using formal meta-theorems as derived inference rules leads to a significant efficiency improvement of the reasoning process, particularly if the theorem describes rather complex insights as in the case of theorem 4.12. Whereas a derivation without the theorem would have to execute all the steps which were necessary to prove the theorem beforehand, the same can be achieved nearly in constant time if the theorem is used as an inference rule. Thus elaborating the computerized proof of a meta-theorem beforehand – once and for all – takes most of the proof burden out of the synthesis process to be performed at derivation time.

4. In many theorems the proof contains an additional computational meaning. Realizing theorem 3.1, for instance, does not only justify the correctness of the proofs-as-programs principle but also describes how to transform a proof of a specification theorem into a correct program solving the initial specification. Thus a

formal meta-theorem also yields a *verified algorithm construction method* which constructs an algorithmic solution for its ‘main goal’ from partial solutions for the preconditions. This method, which corresponds to a validation of the top-down inference rule described by the theorem, is represented by a program term which can be extracted from the proof.

This means that using formal meta-theorems within a synthesis process can also lead to a significant efficiency improvement of the generated code compared to algorithms extracted from ‘pure’ synthesis proofs [25, 10, 23]. Theorems like theorem 4.12 contain complex algorithms which have been introduced and verified explicitly in their formal proofs in order to support a synthesis of efficient programs without having to execute long and complex derivations.

In principle, all these aspects are inherently contained in all constructive formal theories. Meta-theorems do in fact ‘behave’ like primitive inference rules of NuPRL’s type theory [4]. The true advantage of formulating and proving meta-theorems is, however, that one can turn an existing general inference system like NuPRL into a high level reasoning system specialized in the area of program development.

Within such a specialized system both the synthesis process and the generated programs will be much more efficient since a program derivation will proceed in high-level steps which are related to a programmer’s way of reasoning. Thus it will be easier for a human programmer who is not a logician as well to guide the synthesis process in order to generate verifiably correct programs. Furthermore it will be possible to provide automatic support for retrieving applicable theorems even if the system’s knowledge base of meta-theorems is large since due to the type restrictions on the parameters there will only be a small number of theorems which fit the syntactical requirements of a particular situation.

In the rest of this paper we shall focus our attention on meta-theorems about schematic solutions for a given problem. Using algorithm schemata for the development of programs makes the derivation process very efficient and can result in efficient and well-structured programs. On the other hand a schema may not be applicable at all or lead to very poor solutions. Thus strategies based on algorithm schemata can be an extremely powerful tool for program synthesis but they must be properly guided. It is the user who has to select the scheme while the system has to check the prerequisites that make it applicable and perform the elementary steps to instantiate it.

Our technique of representing deductive methods by formal meta-theorems can particularly well be illustrated by this kind of strategies. Much knowledge is required to justify the application of an efficient algorithm schema to a given problem and it is nearly impossible to verify such an algorithm *during* the derivation process. However, since correctness does not depend on the individual instantiation of the algorithm one can prove a separate theorem about the prerequisites under which the schema can be applied correctly. A design strategy can then be based on such a theorem.

The idea of developing synthesis strategies on the basis of parameterized theorems about schematic solutions to a given synthesis problem is not entirely new. Many algorithm design tactics successfully used in the KIDS system [27, 28, 26, 29, 30] rely on theorems of that kind. However, the knowledge contained in these theorems had to be incorporated by hand into the *implemented* design strategy and there is always a chance for errors and omissions.

Our approach has the advantage that the theorems are proven with the same computerized inference system which is used for designing programs. Therefore they are directly – i.e. without further encodings – applicable as key inference rule of a synthesis strategy. This drastically reduces the need for hand-coded algorithm design tactics. Furthermore the proof system makes sure that theorems will always be applied correctly or not at all. Thus we gain both efficiency and safety. We shall demonstrate this advantage by an example formalization of a synthesis strategy for the development of so-called *global-search algorithms*. This formalization is based on an algorithm design tactic developed for the KIDS System and shows that by a rigorous mathematical formalization one can create efficient implementations of synthesis methods which guide derivations within an interactive and tactics supported program development system.

## 4 Synthesis of global search algorithms

Solving a problem by enumerating candidate solutions is a well-known concept in computer science. *Global search* is a method which generalizes binary search, backtracking, branch-and-bound, and others. The basic idea is to manipulate *sets of candidates*. Starting from an *initial set* containing all solutions a global search algorithm repeatedly *extracts* candidate solutions, *splits* sets into subsets, and eliminates sets via *filters* until no sets remain to be split. Sets of candidates are represented by *descriptors* and a *satisfaction* predicate determines when a candidate solution is in the set denoted by a descriptor.

A careful analysis of the general structure of global search algorithms and the conditions under which such algorithms meet their specifications (see [28]) has shown that besides a specification  $\text{spec} = \langle D, R, I, O \rangle$  the following components are essential:

1. A type  $\boxed{\text{S}}$  of all space descriptors for sets of candidate solutions,
2. a predicate  $\boxed{\text{J}}$  on  $D \times S$  describing the *meaningful* (or legal) ones,
3. a function  $\boxed{\text{s}_0} : D \rightarrow S$  describing an initial set of candidate solutions for each input  $x$ ,
4. a predicate  $\boxed{\text{sat}}$  on  $R \times S$  where  $\text{sat}(z, s)$  means that  $z$  is in the set denoted by  $s$ ,
5. a function  $\boxed{\text{split}} : D \times S \rightarrow \text{Set}(S)$  computing subspace descriptors of a given space descriptor, and
6. a function  $\boxed{\text{ext}} : S \rightarrow \text{Set}(R)$  extracting output values from a space descriptor.

The collection of these components is called a *global search theory*. In addition *necessary filters* (predicates  $\Phi$  on  $D \times S$ ) can improve the efficiency of global search algorithms by eliminating space descriptors which do not contain feasible solutions. Given these components one can describe the general structure of global search algorithms by the following pair of programs.

```

FUNCTION F(x:D):Set(R)  WHERE I(x)  RETURNS {z | O(x,z)}
= if  $\Phi(x, \text{s}_0(x))$  then  $F_{aux}(x, \text{s}_0(x))$  else  $\emptyset$ 
FUNCTION  $F_{aux}(x, s:D \times S):Set(R)$   WHERE  $I(x) \wedge J(x, s) \wedge \Phi(x, s)$ 
  RETURNS {z |  $O(x, z) \wedge \text{sat}(z, s)$ }
= {z |  $z \in \text{ext}(s) \wedge O(x, z)$ }  $\cup \bigcup \{F_{aux}(x, t) \mid t \in \text{split}(x, s) \wedge \Phi(x, t)\}$ 

```

In other words, a global search program  $F$  calls on input  $x$  the auxiliary function  $F_{aux}$  with the initial space, if the filter holds.  $F_{aux}$  unions the set of all feasible solutions which can directly be extracted from a space  $s$  and the union of all solutions recursively found in spaces  $t$  that are obtained by splitting  $s$  and applying the filter  $\Phi$ . Note that  $\Phi$  is an input invariant of  $F_{aux}$ . Four requirements must be satisfied in order to guarantee the (partial) correctness of the above program.

1. For all legal inputs  $x \in D$  satisfying  $I(x)$  the initial space descriptor  $s_0(x)$  must be meaningful,
2. splitting must map a meaningful space descriptor into a set of meaningful ones,
3. all feasible solutions must be contained in the initial space described by  $s_0(x)$ ,
4. an output object  $z:R$  is in the set denoted by the descriptor  $s:S$  if and only if  $z$  can be extracted after finitely many applications of `split` to  $s$  where a  $k$ -fold iteration of `split` is defined by

$$\text{split}^0(x, s) = \{s\} \text{ and } \text{split}^{k+1}(x, s) = \bigcup \{\text{split}^k(x, t) \mid t \in \text{split}(x, s)\}$$

*Termination* is guaranteed if meaningful space descriptors can be split only finitely many times. A global search theory is called *well-founded* if it has this property. In [28] it has been shown that well-founded global search theories satisfying the four axioms lead to provably correct global search algorithms of the above kind. This theorem, together with a mechanism for *refining* a predefined global search theory in order to make it applicable to a given specification, forms the basis of a powerful strategy for designing global search algorithms. The strategy requires only very few elementary global search theories to perform its task and is able to solve most of the programming problems approached in the field of program synthesis so far. It has even been demonstrated in [29] that the strategy is capable to solve a real programming problem whose solution had not been found long before.

Because of a hand-written procedural encoding, however, the *implemented* strategy does not check well-foundedness and derives global search algorithms even if the theorems supporting it are not applicable anymore. A user unaware of this fact may create nonterminating programs by selecting the wrong filters. Despite of its thorough theoretical foundation the implementation code of the strategy does not *guarantee* that the generated algorithm is correct. Obviously such omissions can be resolved once they have been discovered. But there remains a principal problem if synthesis strategies are implemented by hand: even if the encoding is done with great care there will always be a chance for errors in the implementation code of the system.

We have overcome this problem by a rigorous mathematical formalization of the knowledge contained in the derivation strategy. This formalization showed, that a few additions and refinements of the strategy described in [28] are necessary to make meta-theorems directly applicable for designing verified global search algorithms. The well-foundedness property, for instance, would put an extreme burden on the inference mechanism if it had to be proven at derivation time. Since some of the most important global search theories are not well-founded we added the concept of *well-foundedness filters*, i.e. filters which are guaranteed to make a global search theory well-founded. We have shown that well-foundedness filters can be refined together with a given global search theory and thus provided the basis for an improved version of Smith's algorithm design strategy: the problems which have to be solved during a derivation are simpler and the implementation is guaranteed to be correct. This shows that

a faithful formalization of standard programming concepts and strategies within a completely formal system *is* possible and that a major source for possible mistakes when transforming theoretical insights into implemented code of strategies can be eliminated through a rigorous use of formal mathematics. Formal proofs, because of their length, will be omitted. A reader interested in details may consult [15].

#### 4.1 *GS-theories*

In order to reason about global search algorithms we have represented the concept of global search theories by a class **GS** of all objects **G** consisting of a specification  $\text{spec} = \langle D, R, I, 0 \rangle$  and the additional components **S**, **J**,  $s_0$ , **sat**, **split**, **ext**. Such an object is a *GS-theory* (i.e. a global search theory) if its components satisfy the four axioms. It is *well-founded* if **split** can be iterated only finitely many times. The iteration  $\text{split}^k$  of **split** is defined by formal induction.

DEFINITION 4.1 (Global search theories)

$$\begin{aligned}
\mathbf{GS} &\equiv (D:\mathbf{TYPES} \times R:\mathbf{TYPES} \times I:D \rightarrow \mathbb{B} \times 0:D \times R \rightarrow \mathbb{B}) \\
&\quad \times S:\mathbf{TYPES} \times D \times S \rightarrow \mathbb{B} \times D \not\rightarrow S \times R \times S \not\rightarrow \mathbb{B} \\
&\quad \times D \times S \not\rightarrow \mathbf{Set}(S) \times S \not\rightarrow \mathbf{Set}(R) \\
D_G &\equiv \text{let } \langle D, R, I, 0 \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle = G \text{ in } D \\
\text{split}^k &\equiv \text{natind}(k; \lambda x, s. \{s\}; \\
&\quad n, \text{sp}. \lambda x, s. \bigcup \{\text{sp}(x, t) \mid t \in \text{split}(x, s)\}) \\
G \text{ is a GS-theory} &\equiv \text{let } \langle D, R, I, 0 \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle = G \text{ in} \\
&\quad \forall x:D. I(x) \\
&\quad \Rightarrow s_0(x) \text{ halts} \wedge J(x, s_0(x)) \\
&\quad \wedge \forall x:D. \forall s:S. I(x) \wedge J(x, s) \\
&\quad \Rightarrow \forall t \in \text{split}(x, s). J(x, t) \\
&\quad \wedge \forall x:D. \forall z:R. I(x) \wedge 0(x, z) \\
&\quad \Rightarrow \text{sat}(z, s_0(x)) \\
&\quad \wedge \forall x:D. \forall s:S. I(x) \wedge J(x, s) \\
&\quad \Rightarrow \forall z:R. 0(x, z) \\
&\quad \Rightarrow \text{sat}(z, s) \\
&\quad \Leftrightarrow \exists k:\mathbb{N}. \exists t \in \text{split}^k(x, s). z \in \text{ext}(t) \\
G \text{ is well-founded} &\equiv \text{let } \langle D, R, I, 0 \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle = G \text{ in} \\
&\quad \forall x:D. \forall s:S. I(x) \wedge J(x, s) \\
&\quad \Rightarrow \exists k:\mathbb{N}. \text{empty?}(\text{split}^k(x, s))
\end{aligned}$$

As an example which we shall later use in the derivation of the Costas arrays algorithm in example 4.13 we introduce a global search theory called `gs_seq_over_set` ( $\alpha$ ). It enumerates all the sequences over a given finite set **S** of elements from the type  $\alpha$  by considering sets of sequences with a common prefix. These spaces are described by their greatest common prefix **V** which makes it possible to represent splitting by appending an element from **S** onto the end of a descriptor. Extracting sequences from a space means selecting its descriptor **V**. We display the GS-theory by assigning values to the individual components (instead of using a 10-tuple).

DEFINITION 4.2 (Example GS-theory)

$$\begin{array}{lcl}
 \mathbf{gs\_seq\_over\_set}(\alpha) \equiv & \mathbf{D} & \mapsto \mathbf{Set}(\alpha) \\
 & \mathbf{R} & \mapsto \mathbf{Seq}(\alpha) \\
 & \mathbf{I} & \mapsto \lambda \mathbf{S}. \mathbf{true} \\
 & \mathbf{0} & \mapsto \lambda \mathbf{S}, \mathbf{L}. \mathbf{range}(\mathbf{L}) \subseteq_{\alpha} \mathbf{S} \\
 & \mathbf{S} & \mapsto \mathbf{Seq}(\alpha) \\
 & \mathbf{J} & \mapsto \lambda \mathbf{S}, \mathbf{V}. \mathbf{range}(\mathbf{V}) \subseteq_{\alpha} \mathbf{S} \\
 & \mathbf{s}_0 & \mapsto \lambda \mathbf{S}. \mathbf{[]} \\
 & \mathbf{sat} & \mapsto \lambda \mathbf{L}, \mathbf{V}. \mathbf{V} \subseteq_{\alpha} \mathbf{L} \\
 & \mathbf{split} & \mapsto \lambda \mathbf{S}, \mathbf{V}. \{\mathbf{V} \bullet i \mid i \in \mathbf{S}\} \\
 & \mathbf{ext} & \mapsto \lambda \mathbf{V}. \{\mathbf{V}\}
 \end{array}$$

It is not very difficult to prove that  $\mathbf{gs\_seq\_over\_set}(\alpha)$  is in fact a GS theory. The first three axioms are consequences of elementary laws about finite sets and sequences (see [15, Appendices A.4/A.5]) and the fourth is shown by induction.

LEMMA 4.3 (GS-theory for finite sequences)

$$\forall \alpha : \mathbf{TYPES}. \mathbf{gs\_seq\_over\_set}(\alpha) \text{ is a GS theory}$$

## 4.2 Filters

The theory  $\mathbf{gs\_seq\_over\_set}(\alpha)$  presented above is *not* well-founded since the split-operation is not bounded. Therefore it cannot be used for algorithm construction unless well-foundedness can be ensured by other means. An investigation of all examples where this theory has successfully been used showed that well-foundedness comes in through the filter. Thus filters not only improve the efficiency of global-search algorithms but can also turn non-terminating algorithms into terminating ones. To make use of this insight when reasoning about global search algorithms we introduce a new concept of *well-foundedness filters* (*wf-filters*) for a given global search theory  $\mathbf{G} = \langle \langle \mathbf{D}, \mathbf{R}, \mathbf{I}, \mathbf{0} \rangle, \mathbf{S}, \mathbf{J}, \mathbf{s}_0, \mathbf{sat}, \mathbf{split}, \mathbf{ext} \rangle$ .

We define  $\mathbf{Filters}(\mathbf{G})$  to be the class of predicates on  $\mathbf{D} \times \mathbf{S}$ . A filter  $\Phi$  is *necessary* for  $\mathbf{G}$  if search spaces containing feasible solutions always survive filtering.  $\Phi$  is a well-foundedness filter for  $\mathbf{G}$  if it can turn the split operation into a well-founded one. The modified split operation will be denoted by  $\mathbf{split}_{\Phi}$ .

DEFINITION 4.4 (Filters)

$$\begin{array}{lcl}
 \mathbf{Filters}(\mathbf{G}) & \equiv & \text{let } \langle \langle \mathbf{D}, \mathbf{R}, \mathbf{I}, \mathbf{0} \rangle, \mathbf{S}, \mathbf{J}, \mathbf{s}_0, \mathbf{sat}, \mathbf{split}, \mathbf{ext} \rangle = \mathbf{G} \text{ in} \\
 & & \mathbf{D} \times \mathbf{S} \rightarrow \mathbb{B} \\
 \Phi \text{ necessary for } \mathbf{G} & \equiv & \text{let } \langle \langle \mathbf{D}, \mathbf{R}, \mathbf{I}, \mathbf{0} \rangle, \mathbf{S}, \mathbf{J}, \mathbf{s}_0, \mathbf{sat}, \mathbf{split}, \mathbf{ext} \rangle = \mathbf{G} \text{ in} \\
 & & \forall \mathbf{x} : \mathbf{D}. \forall \mathbf{s} : \mathbf{S}. \exists \mathbf{z} : \mathbf{R}. \mathbf{sat}(\mathbf{z}, \mathbf{s}) \wedge \mathbf{0}(\mathbf{x}, \mathbf{z}) \Rightarrow \Phi(\mathbf{x}, \mathbf{s}) \\
 \mathbf{split}_{\Phi} & \equiv & \lambda \mathbf{x}, \mathbf{s}. \{ \mathbf{t} \mid \mathbf{t} \in \mathbf{split}(\mathbf{x}, \mathbf{s}) \wedge \Phi(\mathbf{x}, \mathbf{t}) \} \\
 \Phi \text{ wf-filter for } \mathbf{G} & \equiv & \text{let } \langle \langle \mathbf{D}, \mathbf{R}, \mathbf{I}, \mathbf{0} \rangle, \mathbf{S}, \mathbf{J}, \mathbf{s}_0, \mathbf{sat}, \mathbf{split}, \mathbf{ext} \rangle = \mathbf{G} \text{ in} \\
 & & \forall \mathbf{x} : \mathbf{D}. \forall \mathbf{s} : \mathbf{S}. \mathbf{I}(\mathbf{x}) \wedge \mathbf{J}(\mathbf{x}, \mathbf{s}) \\
 & & \Rightarrow \exists \mathbf{k} : \mathbb{N}. \mathbf{empty}?( \mathbf{split}_{\Phi}^{\mathbf{k}}(\mathbf{x}, \mathbf{s}) )
 \end{array}$$

For the theory  $\mathbf{gs\_seq\_over\_set}(\alpha)$  there are three obvious possibilities to limit splitting. One may put upper bounds to the length of the descriptor or eliminate descriptors which contain duplicate elements.

LEMMA 4.5 (Well-foundedness Filters for `gs_seq_over_set`)

1.  $\forall \alpha : \text{TYPES}. \forall k : \mathbb{N}. \lambda S, V. |V| \leq k \quad \text{wf-filter for } \text{gs\_seq\_over\_set}(\alpha)$
2.  $\forall \alpha : \text{TYPES}. \forall k : \mathbb{N}. \lambda S, V. |V| \leq k * |S| \quad \text{wf-filter for } \text{gs\_seq\_over\_set}(\alpha)$
3.  $\forall \alpha : \text{TYPES}. \lambda S, V. \text{nodups}(V) \quad \text{wf-filter for } \text{gs\_seq\_over\_set}(\alpha)$

None of the three filters is necessary for `gs_seq_over_set`( $\alpha$ ). They must be chosen depending on the problem to which `gs_seq_over_set`( $\alpha$ ) shall be applied and will turn into necessary ones after specialization (see section 4.3).

Obviously any filter is a wf-filter for a GS-theory  $G$  if  $G$  is already well-founded. Otherwise a necessary wf-filter  $\Phi$  can turn  $G$  into a well-founded GS-theory through exchanging `split` by `split $_{\Phi}$` . Using this insight we can prove an improved and completely formal version of Smith's main theorem [28, theorem 1]: GS-theories (which may not be well-founded themselves) together with necessary well-foundedness filters contain all the information required to construct provably correct algorithms.

THEOREM 4.6 (Correct Global Search Programs)

1.  $\forall G = \langle \langle D, R, I, O \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle : \text{GS}. \quad \forall \Phi : \text{Filters}(G).$   
 $\forall F_{aux} : D \times S \rightarrow \text{Set}(R).$   
 $G \text{ is a GS-theory} \wedge \Phi \text{ necessary for } G \Rightarrow$   
 $F_{aux} \text{ computes } \text{FUNCTION } Aux(x, s : D \times S) : \text{Set}(R)$   
 $\quad \text{WHERE } I(x) \wedge J(x, s) \wedge \Phi(x, s)$   
 $\quad \text{RETURNS } \{z \mid O(x, z) \wedge \text{sat}(z, s)\}$   
 $\Rightarrow \text{FUNCTION } F(x : D) : \text{Set}(R) \quad \text{WHERE } I(x) \quad \text{RETURNS } \{z \mid O(x, z)\}$   
 $\quad = \text{if } \Phi(x, s_0(x)) \text{ then } F_{aux}(x, s_0(x)) \text{ else } \emptyset$   
 $\text{is correct}$
2.  $\forall G = \langle \langle D, R, I, O \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle : \text{GS}. \quad \forall \Phi : \text{Filters}(G).$   
 $G \text{ is a GS-theory} \wedge \Phi \text{ wf-filter for } G \wedge \Phi \text{ necessary for } G$   
 $\Rightarrow$   
 $\text{FUNCTION } F_{aux}(x, s : D \times S) : \text{Set}(R) \quad \text{WHERE } I(x) \wedge J(x, s) \wedge \Phi(x, s)$   
 $\quad \text{RETURNS } \{z \mid O(x, z) \wedge \text{sat}(z, s)\}$   
 $= \{z \mid z \in \text{ext}(s) \wedge O(x, z)\} \cup \bigcup \{F_{aux}(x, t) \mid t \in \text{split}(x, s) \wedge \Phi(x, t)\}$   
 $\text{is correct}$

The proof of part 1 of this theorem is a straightforward but lengthy exercise in formal reasoning. Proving the second part is more difficult. The proof of [28, theorem 1] uses fixed point theory and cannot be formalized straightforwardly since there are no inference rules about fixed points. We therefore had to introduce a lemma about the result of a computation depending on the number  $i$  of split operations involved until termination<sup>3</sup> and to prove it formally by induction. Since it would take more than 5 pages to present the formal proof we refer the reader interested in details to [15, Appendix C.5].

### 4.3 *Specializing a known global search theory*

Theorem 4.6 can be used for synthesizing a given specification if we can provide an appropriate GS-theory which extends it. In order to find such a GS-theory we need

<sup>3</sup>This result can be described as the collection of all the solutions directly extractable from descriptors belonging to the sets `split $_{\Phi}^j$` ( $x, s$ ) where  $0 \leq j < i$ , i.e. as  $\bigcup \{ \{z \mid z \in \text{ext}(t) \wedge O(x, z)\} \mid t \in \bigcup \{\text{split}_{\Phi}^j(x, r) \mid j \in \{0..i-1\}\} \}$ .

a method for constructing it from an already known GS-theory like  $\text{gs\_seq\_set}(\alpha)$ . This can be done by *reducing* the specification to the specification part of the GS-theory, deriving a substitution  $\theta$  from this reduction, and *specializing* the GS-theory to one extending the specification. To perform these steps we introduce a few definitions which are straightforward formalizations of the corresponding concepts in [28].

DEFINITION 4.7 (Specification Reduction)

$$\begin{aligned} \text{spec reduces to spec}' &\equiv \text{let } \langle D, R, I, O \rangle = \text{spec} \text{ and } \langle D', R', I', O' \rangle = \text{spec}' \text{ in} \\ &\quad RCR' \wedge \forall x:D. I(x) \Rightarrow \\ &\quad \exists u:D'. I'(u) \wedge \forall z:R. O(x, z) \Rightarrow O'(u, z) \\ \text{spec reduces to spec}' \text{ with } \theta & \\ &\equiv \text{let } \langle D, R, I, O \rangle = \text{spec} \text{ and } \langle D', R', I', O' \rangle = \text{spec}' \text{ in} \\ &\quad RCR' \wedge \forall x:D. I(x) \Rightarrow \\ &\quad I'(\theta(x)) \wedge \forall z:R. O(x, z) \Rightarrow O'(\theta(x), z) \end{aligned}$$

Thus a specification  $\text{spec}$  reduces to  $\text{spec}'$  with  $\theta$  if  $\text{spec}'$  modified by  $\theta$  is a generalization of  $\text{spec}$  (i.e. both input and output conditions are weaker). To find such a substitution relating a given pair of specifications we may either rely on user interaction – where the user applies an ‘understanding’ of specification reductions and considers correspondences between the two domains – or on a proof tactic which has to show that  $\text{spec}$  reduces to  $\text{spec}'$ . According to the proofs-as-programs paradigm (theorem 3.1) such a proof yields the desired substitution.

LEMMA 4.8 (Specification Reduction)

$$\begin{aligned} \forall \text{spec} = \langle D, R, I, O \rangle : \text{SPEC}. \forall \text{spec}' = \langle D', R', I', O' \rangle : \text{SPEC}. \\ \text{spec reduces to spec}' \Rightarrow \exists \theta : D \not\rightarrow D'. \text{spec reduces to spec}' \text{ with } \theta \end{aligned}$$

Using  $\not\rightarrow$  a *specialize* a GS-theory  $G$  to a GS-theory  $G_\theta(\text{spec})$  which extends  $\text{spec}$ , provided  $\text{spec}$  reduces to the specification part of  $G$ . In the same way, a filter  $\Phi$  can be specialized to a filter  $\Phi_\theta$ .

DEFINITION 4.9 (Generalization and Specialization)

$$\begin{aligned} G \text{ generalizes } \text{spec} \text{ with } \theta & \\ \equiv \text{let } \langle \text{spec}', S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle = G \text{ in} & \\ \text{spec reduces to spec}' \text{ with } \theta & \\ G_\theta(\text{spec}) & \equiv \text{let } \langle \langle D, R, I, O \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle = G \text{ in} \\ & \langle \text{spec}, S, \lambda x, s. J(\theta(x), s), \lambda x, s_0(\theta(x)), \text{sat}, \\ & \quad \lambda x, s. \text{split}(\theta(x), s), \text{ext} \rangle \\ \Phi_\theta & \equiv \lambda x, s. \Phi(\theta(x), s) \end{aligned}$$

By specializing a known global search theory  $G$  to the GS-theory  $G_\theta(\text{spec})$  we obviously get an extension of the specification  $\text{spec}$ . To make sure that this extension leads to verifiably correct global search algorithms we have to show that all the important properties are preserved during specialization: specialization turns GS-theories into GS-theories and preserves well-foundedness.

THEOREM 4.10 (Specializing Global Search Theories and Wf-Filters)

$$\begin{aligned} \forall \text{spec} : \text{SPEC}. \forall G : \text{GS}. \forall \theta : D(\text{spec}) \not\rightarrow D_G. \forall \Phi : \text{Filters}(G). \\ G \text{ generalizes } \text{spec} \text{ with } \theta \Rightarrow \\ \begin{aligned} 1. \quad G \text{ is a GS-theory} & \Rightarrow G_\theta(\text{spec}) \text{ is a GS-Theory} \\ 2. \quad \Phi \text{ wf-filter for } G & \Rightarrow \Phi_\theta \text{ wf-filter for } G_\theta(\text{spec}) \end{aligned} \end{aligned}$$

## 4.4 A strategy for the design of global search algorithms

Together with a small library of standard GS-theories and wf-filters and a tactic for proving specification reductions theorem 4.10 can be used to create a GS-theory  $G_\theta$  which extends a given specification and a wf-filter  $\Phi_\theta$  for it. After this step one only has to check whether  $\Phi_\theta$  is also necessary for  $G_\theta$ . If this is the case, theorem 4.6 can be applied to generate a verifiably correct program. To improve the efficiency of the resulting program one may *refine*  $\Phi_\theta$  by adding further conditions which follow from the existence of a value  $z$  satisfying  $\text{sat}(z, s) \wedge 0(x, z)$ . This step, which results in an additional necessary filter  $\Psi$  for  $G_\theta$ , does not affect well-foundedness and can be performed heuristically. All these insights lead to a single theorem which can easily be applied within a program derivation strategy. It improves the ones given in [28] since its requirements for generating verified global search algorithms are weaker.

**THEOREM 4.11 (Bottom-Up Synthesis of Global Search Algorithms)**

$\forall \text{spec} = \langle D, R, I, 0 \rangle : \text{SPEC}. \quad \forall G = \langle \langle D', R', I', 0' \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle : \text{GS}.$

$\forall \theta : D \not\rightarrow D'. \quad \forall \Phi : \text{Filters}(G). \quad \forall \Psi : \text{Filters}(G_\theta(\text{spec})).$

$G$  is a GS-Theory  $\wedge G$  generalizes  $\text{spec}$  with  $\theta \wedge \Phi$  wf-filter for  $G$   
 $\wedge \Phi_\theta$  necessary for  $G_\theta(\text{spec}) \wedge \Psi$  necessary for  $G_\theta(\text{spec})$

$\Rightarrow$

```

let  $\Pi = \lambda x, s. \Phi(\theta(x), s) \wedge \Psi(x, s)$  in
  letrec  $f_{aux}(x, s) = \{ z \mid z \in \text{ext}(s) \wedge 0(x, z) \}$ 
     $\cup \bigcup \{ f_{aux}(x, t) \mid t \in \text{split}(\theta(x), s) \wedge \Pi(x, t) \}$ 
  in
    FUNCTION  $f_{aux}(x, s : D \times S) : \text{Set}(R)$  WHERE  $I(x) \wedge J(\theta(x), s) \wedge \Pi(x, s)$ 
      RETURNS  $\{ z \mid 0(x, z) \wedge \text{sat}(z, s) \}$ 
      =  $f_{aux}(x, s)$ 
    is correct
   $\wedge$  FUNCTION  $F(x : D) : \text{Set}(R)$  WHERE  $I(x)$  RETURNS  $\{ z \mid 0(x, z) \}$ 
      = if  $\Pi(x, s_0(\theta(x)))$  then  $f_{aux}(x, s_0(\theta(x)))$  else  $\emptyset$ 
    is correct

```

The proof follows essentially from theorems 4.6 and 4.10.

Formalizing a theorem about program construction methods makes sense only if it is to be used within a computerized proof system ensuring its correct application to a given problem. A strategy suggested by the formulation of theorem 4.11 would lead to forward reasoning within a program derivation process and would have to determine values for all the parameters of theorem 4.11 to make it applicable. To support a more goal-oriented reasoning style we convert the theorem into a version which can directly be applied as a top down inference rule. This theorem states that for creating a verifiably correct global search algorithm for a given specification  $\text{spec}$  one has to determine the following parameters:

a GS-theory  $G$ , a substitution  $\theta$  reducing specification  $\text{spec}$  to the specification part of  $G$ , a wf-filter  $\Phi$  for  $G$  which after being transformed with  $\theta$  is necessary for the transformed GS-theory, and an optional necessary filter  $\Psi$  for the transformed GS-theory.

THEOREM 4.12 (Top-Down Synthesis of Global Search Algorithms)

$\forall \text{spec} = \langle D, R, I, O \rangle : \text{SPEC}.$

FUNCTION  $F(x:D) : \text{Set}(R)$  WHERE  $I(x)$  RETURNS  $\{z \mid O(x,z)\}$  is satisfiable

$\Leftarrow$   $\exists G : \text{GS}.$   $G$  is a GS-Theory  
 $\wedge \exists \theta : D \not\rightarrow D_G.$   $G$  generalizes  $\text{spec}$  with  $\theta$   
 $\wedge \exists \Phi : \text{Filters}(G).$   $\Phi$  wf-filter for  $G \wedge \Phi_\theta$  necessary for  $G_\theta(\text{spec})$   
 $\wedge \exists \Psi : \text{Filters}(G_\theta(\text{spec})).$   $\Psi$  necessary for  $G_\theta(\text{spec})$

This theorem can easily be proven by applying theorem 4.11 to construct the program body once the parameters  $G, \Phi, \theta$  and  $\Psi$  have been determined. It represents a high-level inference rule which can be used efficiently within a program synthesis environment and implicitly contains a method for constructing a verifiably correct global search program<sup>4</sup> for a given specification. Applied to a synthesis problem it yields a subgoal which clearly states what remains to be done. A user of the system will thus be able to guide the synthesis process interactively if a solution cannot be found automatically (or if design decisions shall be made). After the subgoal has been proven the proof contains an instantiated version of the program scheme given by theorem 4.11. This program can then be extracted by the system.

One should recall that the transition from the original synthesis problem to the subgoal requiring  $G, \Phi, \theta$  and  $\Psi$  is by far the most difficult part of the synthesis process since it contains the implicit creation and verification of an efficient global search program. By proving theorem 4.12 within a logical calculus we have separated this task from the derivations to be performed at runtime and solved it once and for all. During a synthesis process the transition can be executed *within a single step*.

Thus the advantage of formalizing and verifying such a meta-theorem is that it leads to formal program derivations which are verifiably correct, comprehensible for human programmers, and very efficient and generate efficient programs. Furthermore it gives rise to a *verified implementation of a simple synthesis strategy* designing a global search algorithm for a given specification  $\text{spec} = \langle D, R, I, O \rangle$ . This strategy refines and improves the one given in [28] since it uses theorem 4.12 as its key component.

1. State the synthesis goal as a theorem  
 $\vdash$  FUNCTION  $F(x:D) : \text{Set}(R)$  WHERE  $I(x)$  RETURNS  $\{z \mid O(x,z)\}$  is satisfiable
2. Instantiate the parameters  $D, R, I, O$  and apply theorem 4.12 as a top-down inference rule. This transforms the problem to proving the existence of appropriate  $G, \Phi, \theta$  and  $\Psi$ .
3. Sequentially solve the four parts of the subgoal
  - (a) Select a GS-theory  $G = \langle \langle D', R', I', O' \rangle, S, J, s_0, \text{sat}, \text{split}, \text{ext} \rangle$  from the library whose range type  $R'$  is  $R$  or a supertype of it. If  $R'$  contains type variables instantiate them accordingly. Prove ' $G$  is a GS-Theory' by instantiating the corresponding library lemma.
  - (b) To determine  $\theta$ , apply lemma 4.8 as top down inference rule. Prove that  $\text{spec}$  reduces to the specification part of  $G$ , i.e. that ' $\forall x:D. I(x) \Rightarrow \exists u:D'. I'(u) \wedge \forall z:R. O(x,z) \Rightarrow O'(u,z)$ ' holds. After the proof has been constructed extract the substitution  $\theta$  according to lemma 4.8.

---

<sup>4</sup>Note that this algorithm computes *all* the solutions of a given specification  $\langle D,R,I,O \rangle$  and thus satisfies the corresponding set-valued version (see definition 2.2) of the specification.

- (c) Select a wf-filter  $\Phi$  for  $G$  from the library (again use a lemma). If  $G$  is already well-founded then a trivial filter may be chosen. Try to verify that  $\Phi_\theta$  is necessary for  $G_\theta(\text{spec})$  and select another wf-filter if this cannot be done.
  - (d) To determine the additional filter  $\Psi$  select conjuncts which can *easily* be inferred from  $\text{sat}(z, s) \wedge 0(x, z)$ . This step must be controlled heuristically or by user interaction.
4. Now all the subgoals are solved, all the parameters are determined and the synthesis theorem is proven. The system will extract from the proof a program body that computes the specification. The resulting program is exactly the one given in theorem 4.11.

Automatic support could be provided for nearly all of these steps. This, however, is not even necessary since – due to the high level of abstraction – a user will be able to derive a correct global search algorithm with just a few interactive steps even without such strategic support.

We shall illustrate this strategy by the same real programming problem which has been used in [29] to demonstrate the capabilities of the KIDS system. The derivation selects the non-well-founded GS-theory `gs_seq_over_set` and can be justified *only* via the newly introduced concept of wf-filters.

EXAMPLE 4.13 (Synthesis of an Algorithm enumerating Costas Arrays)

In [6], Costas introduced a class of permutations that can be used to generate radar and sonar signals with ideal ambiguity functions. Since then, many publications have investigated combinatorial properties of these permutations, now known as Costas arrays, but no general construction has been found. A Costas array is a permutation of the set  $\{1..n\}$  such that there are no duplicate elements in any row of its *difference table* which in its first row gives the difference of adjacent elements of the permutation, in the second the difference of every other element, and so on. To state the problem we have to introduce definitions for permutations and rows in the difference table of a sequence  $p$ .

$$\begin{aligned} \text{perm}(L, S) &\equiv \text{nodups}(L) \wedge \text{range}(L)=S \\ \text{dtrow}(p, j) &\equiv [p_i - p_{i+j} \mid i \in [1..|p|-j]] \end{aligned}$$

1. Given the above definitions a program constructing all the Costas arrays of order  $n$  can be specified as follows.

```
⊢ FUNCTION Costas(n:ℤ):Set(Seq(ℤ)) WHERE n≥1
  RETURNS {p | perm(p, {1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p, j))}
```

2. After applying theorem 4.12 the subgoal is to determine appropriate parameters  $G, \Phi, \theta$  and  $\Psi$ , i.e. we have to prove the following goal.

```
⊢ ∃G:GS. G is a GS-Theory
  ∧ ∃θ:ℤ↗DG. G generalizes spec with θ
  ∧ ∃Φ:Filters(G). Phi wf-filter for G ∧ Φθ necessary for Gθ(spec)
  ∧ ∃Ψ:Filters(Gθ(spec)). Ψ necessary for Gθ(spec)
  where spec =
    ( ℤ, Seq(ℤ), λn.n≥1,
      λn, p. perm(p, {1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p, j))
    )
```

3. The four parameters can now be determined as follows

(a) The range type of  $\text{gs\_seq\_over\_set}(\mathbf{Z})$  ( $\alpha$  has been instantiated with  $\mathbf{Z}$ ) fits  $\text{Seq}(\mathbf{Z})$ , the range type of our problem. Lemma 4.3 proves that we have selected a GS theory.

(b) After applying lemma 4.8 we have to show

$$\begin{aligned} \forall n:\mathbf{Z}. n \geq 1 &\Rightarrow \exists S:\text{Set}(\mathbf{Z}). \\ \forall p:\text{Seq}(\mathbf{Z}). \text{perm}(p, \{1..n\}) \wedge \forall j \in \text{domain}(p). \text{nodups}(\text{dtrow}(p, j)) \\ &\Rightarrow \text{range}(p) \subseteq S \end{aligned}$$

Instantiating the definition of  $\text{perm}(p, \{1..n\})$  and applying a standard lemma about set equality ( $S'=S \Leftrightarrow S' \subseteq S \wedge S \subseteq S'$ ) leads to  $S=\{1..n\}$  and the substitution  $\theta=\lambda n. \{1..n\}$ .

(c) Using lemma 4.5(3) we select the wf-filter  $\Phi=\lambda S, V. \text{nodups}(V)$  and get after specialization  $\Phi_\theta=\lambda n, V. \text{nodups}(V)$ . It is not very difficult to check the condition for  $\Phi_\theta$  being necessary:

$$\begin{aligned} \exists p:\text{Seq}(\mathbf{Z}). \text{perm}(p, \{1..n\}) \wedge \forall j \in \text{domain}(p). \text{nodups}(\text{dtrow}(p, j)) \wedge \forall p \\ \Rightarrow \text{nodups}(V) \end{aligned}$$

(d) By the laws of  $\text{nodups}$  and  $\text{dtrow}$  we can infer with little effort

$$\begin{aligned} \exists p:\text{Seq}(\mathbf{Z}). \text{perm}(p, \{1..n\}) \wedge \forall j \in \text{domain}(p). \text{nodups}(\text{dtrow}(p, j)) \wedge \forall p \\ \Rightarrow \forall j \in \text{domain}(p). \text{nodups}(\text{dtrow}(V, j)) \end{aligned}$$

Thus we can add as additional necessary Filter for the specialized GS-theory:

$$\Psi:=\lambda n, V. \forall j \in \text{domain}(W). \text{nodups}(\text{dtrow}(V, j))$$

4. Now the derivation is complete and the system will extract a correct global search algorithm computing all Costas arrays of order  $n$ . The algorithm is presented below (in mathematical notation).

```

letrec Costasaux(n, V) =
  { p | p ∈ {V} ∧ perm(p, {1..n}) ∧ ∀j ∈ domain(p). nodups(dtrow(p, j)) }
  ∪ ∪ { Costasgs(n, W) | W ∈ {V•i | i ∈ {1..n}} ∧ nodups(W)
      ∧ ∀j ∈ domain(W). nodups(dtrow(W, j)) }
in
  if nodups([]) ∧ ∀j ∈ domain([]). nodups(dtrow([], j))
  then Costasgs(n, [])
  else ∅
    
```

This algorithm can obviously be optimized after it has been generated but we shall not discuss this here.

## Conclusion

We have presented a rigorous formalization of the meta-theory of programming within the object language of constructive type theory using reflected notions of programming concepts such as specifications, programs, and correctness. Our framework makes it possible to specify program-synthesizing strategies in the object language and to prove formal theorems about their behavior which can be used as derived inference rules for the underlying calculus. Our framework provides a straightforward and efficient way to create verifiably correct implementations of program synthesis strategies and to turn an existing general inference system into a high level reasoning system special-

ized in the area of program development. By a formalization of a strategy for deriving global search algorithms we have demonstrated that it is possible to use rigorous mathematical formalizations in a practical setting. We have shown that it is even possible to refine and improve existing program design tactics since the explicit formalization of the knowledge contained in them leads to clearer insights about their behavior.

Our framework can not only be used for representing synthesis strategies based on algorithm schemata (although it can most easily be explained by these) but is also applicable to strategies based on transformations (see our formalization of LOPS in [15]) or on extracting programs from constructive proofs. Because of the rigorous mathematical approach our formal framework is more general than the existing approaches to program synthesis since these depend on the peculiarities of the synthesis paradigm (i.e. proofs-as-programs, transformations, or algorithm schemata) in which they are formulated. Thus it makes it possible to integrate different approaches to program synthesis in a uniform way. Furthermore it provides a means to combine the strengths of formal logical calculi with those of ‘informally’ described derivation strategies by raising the level of abstraction of the formal object language to one comprehensible for programmers.

Compared to approaches based on logical calculi and extracting programs from formal proofs we gain efficiency in two ways. Firstly, most of the reasoning to be performed during program development is moved to the proofs of the formal meta-theorems which are elaborated once and for all while developing the program synthesis system itself. This leads to a significant efficiency improvement of the derivation process to be performed. Secondly, the generated code does not depend on the quality of a general mechanism for extracting programs from constructive proofs [25] but is an instantiation of the program which has been explicitly provided and verified in the proof of the theorem. Thus meta-theorems allow to control the structure of the generated algorithms in a much more flexible way and to generate arbitrary recursive and very efficient programs without running into undecidability problems.

Compared to less rigorous derivation strategies based on informal descriptions on paper we gain reliability which cannot be guaranteed by the hand-coded implementations of these strategies. Because of the high level of abstraction in our formal object language a formalization of an informally described concept can easily be recognized as faithful. On the other hand it can directly be used as key component of an implementation of the strategy within the underlying proof system which guarantees the correctness of all the reasoning steps performed at derivation time. Altogether the step from presenting strategy on paper to its verifiably correct realization on a computer is very small.

To realize our approach we could rely on several well-known theoretical insights and techniques. A comprehensible representation of formal theories is the main purpose of the *definition mechanism* of the NuPRL proof development systems [4]. The concept of *tactics* aims at specializing general inference systems towards a particular application. Using *meta-theorems* and *reflection* as a general technique to control the application of tactics has already been investigated in [12, 13, 1]. In *Isabelle* [22] meta-theorems about reasoning within some object logic are used to generate *derived inference rules* for this logic. These rather general techniques have been combined and elaborated in a novel way to make meta-programming more rigorous in a practical setting and to support a true cooperation between human programmers and formal

program synthesis systems.

The approach presented here is not restricted to program derivation. In principle, it also applies to algorithmic optimization, data type refinement, theorem proving strategies, planning methods, and other techniques which can be understood as logical deduction. Therefore we believe that a rigorous use of formal mathematics is a promising approach towards implementing flexible, practically useful, trustworthy, and efficient systems for knowledge based software engineering.

## References

- [1] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In John C. Mitchell, editor, *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 95–106. IEEE Computer Society Press, 1990.
- [2] Wolfgang Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14(3):243–261, 1980.
- [3] Alan Bundy. Automatic guidance of program synthesis proofs. In *Proceedings of the Workshop on Automating Software Design, IJCAI-89*, pages 57–59, 1989.
- [4] Robert L. Constable et. al. *Implementing Mathematics with the NuPRL proof development system*. Prentice Hall, 1986.
- [5] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
- [6] J. Costas. A study of a class of detection waveforms having nearly ideal range – doppler ambiguity properties. In *Proceedings of the IEEE*, volume 72, pages 996–1009, 1984.
- [7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [8] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation*. LNCS 78, Springer Verlag, 1979.
- [9] Cordell C. Green. An application of theorem proving to problem solving. In *Proceedings of the 1<sup>st</sup> International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.
- [10] S. Hayashi. PX: a system extracting programs from proofs. In *Proceedings of the IFIP Conference on Formal Description of Programming Concepts*, pages 399–424, 1986.
- [11] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In M. E. Stickel, editor, *Proceedings of the 10<sup>th</sup> Conference on Automated Deduction*, LNCS 449, pages 117–131. Springer Verlag, 1990.
- [12] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in Type Theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986.
- [13] Todd B. Knoblock. *Metamathematical extensibility in Type Theory*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY, 1987.
- [14] Christoph Kreitz. Towards a formal theory of program construction. *Revue d' intelligence artificielle*, 4(3):53–79, November 1990.
- [15] Christoph Kreitz. *METASYNTHESIS: Deriving Programs that Develop Programs*. Thesis for Habilitation, Technische Hochschule Darmstadt, FG Intellektik, 1993.
- [16] Per Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory Lecture Notes*. Bibliopolis, 1984.
- [17] Zohar Manna and Richard J. Waldinger. Synthesis: Dreams  $\Rightarrow$  programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979.
- [18] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.
- [19] G. Neugebauer, Bertram Fronhöfer, and Christoph Kreitz. XPRTS - an implementation tool for program synthesis. In D. Metzger, editor, *Proceedings of the 13<sup>th</sup> German Workshop on Artificial Intelligence*, Informatik Fachberichte 216, pages 348–357. Springer Verlag, 1989.
- [20] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

- [21] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.
- [22] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
- [23] Christine Paulin-Mohring. Extracting  $F_\omega$ 's programs from proofs in the Calculus of Constructions. In *Proc. of the 16<sup>th</sup> ACM Symposium on Principles of Programming Languages*, pages 89–104, 1989.
- [24] Robert Pollack. *The theory of LEGO – a proof checker for the extended calculus of constructions*. PhD thesis, University of Edinburgh, 1994.
- [25] James T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY, 1985.
- [26] Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, October 1990.
- [27] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, September 1985.
- [28] Douglas R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, November 1987. Revised Version, July 1988.
- [29] Douglas R. Smith. KIDS — a knowledge-based software development system. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 483–514, AAAI Press / The MIT Press, 1991.
- [30] Douglas R. Smith and Eduardo A. Parra. Transformational approach to transportation scheduling. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 60–68, 1993.

Received 1 October 1994. Revised 7 June 1995