Chapter III.2.5 of *Automated Deduction – A Basis for Applications,* pp. 105–134, Kluwer, 1998.

CHRISTOPH KREITZ

PROGRAM SYNTHESIS

1. INTRODUCTION

Since almost 30 years software production has to face two major problems: the cost of non-standard software, caused by long development times and the constant need for maintenance, and a lack of confidence in the reliability of software. Recent accidents like the crash of KAL's 747 in August 1997 or the failed launch of the Ariane 5 rocket in 1996 have partially been attributed to software problems and limit the extent to which software is adopted in safety-critical areas. Both problems have become even worse over the years since they grow with the complexity of behavior that can be produced. Reliable software development techniques therefore would immediately find application in developing economically important software systems.

Attempts to elaborate such techniques have been undertaken since the upcoming of the software crisis. 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 software engineering tools which are based on knowledge processing and logical deductions.

1.1. *The General Idea of Program Synthesis*

Program synthesis deals with the aspects of the software development process which can, at least in principle, be automated. Its goal is to mechanically *synthesize* correct and efficient computer code from declarative specifications. Although it does not address the problem of obtaining an accurate statement of the requirements in the first place, it makes the overall process faster and more reliable, as verification and maintenance can now be done on the specification rather than the code.

Like any computerized process, program synthesis relies on a formal representation of the objects it has to deal with. Research has been active in two areas: the development of expressive *calculi* which support formal reasoning about specifications and algorithms and the implementation of deduction *strategies* which derive programs from their specifications.

In principle, the synthesis of programs must be expressible in some calculus with well-known deductive rules. Thus there is a need for formal logical calculi in which mathematical specifications, algorithmic structures, and reasoning about their properties can be properly represented. This has led to a revived interest in *constructive type theories* (Martin-Löf, 1984; Constable *et al.*, 1986; Girard *et al.*, 1989; Coquand & Huet, 1988; Nordström *et al.*, 1990) and proof development systems which can perform the corresponding reasoning steps interactively or guided by proof tactics (see also chapter I.3.15). Type theories are expressive enough to represent all the fundamental concepts from mathematics and programming quite naturally. The basic inference steps in formal calculi, however, are extremely small and way below the level of reasoning used by humans during program development. To make program synthesis practically feasible the level of abstraction in formal reasoning must be raised. This research area still needs exploration.

Strategies for automated program synthesis often avoid the formal overhead of a rigorous approach. Most of them were developed independent from the above calculi. There are three basic categories. The *proofs-as-programs* paradigm understands the derivation of program as a proof process which implicitly constructs an algorithm. Strategies are based on standard theorem proving techniques, tailored towards constructive reasoning (Section 3.1), and on mechanisms for extracting algorithms from proofs. Similarly, *transformational synthesis* uses general rewriting techniques in order to transform a specification into a form that can straightforwardly be converted into a program (Section 3.2). An analysis of the structure of algorithmic classes is the basis of *knowledge based program synthesis* (Section 4). Strategies derive the parameters of program schemata and instantiate them accordingly.

1.2. *History of Program Synthesis*

The identification of algorithms with proofs has been suggested early in the history of constructive mathematics (Kolmogorov, 1932). Consequently, automatic program synthesis systems showed up shortly after the first theorem provers. Pioneer work of C. Green and R. Waldinger (Green, 1969; Waldinger, 1969; Waldinger & Lee, 1969) was followed by many approaches based on the proofs-as-programs paradigm. Transformational synthesis was introduced a few years later (Manna & Waldinger, 1975). Program synthesis was a very vivid research area until the early 80's. Some approaches were implemented and tested successfully for a number of examples but none of them scaled up very well. With the advent of logic programming the correctness problem almost appeared to be solved. The activities in the original core of the field dropped and its focus shifted towards efficiency improvements.

Large scale synthesis became possible after D. Smith began to incorporate a structural analysis of algorithmic classes into efficient synthesis strategies (Smith, 1985b; Smith, 1987b). The resulting KIDS system (Smith, 1991a) has come close to a commercial breakthrough. Its success is based on an effective use of algorithm schemata, design strategies and subsequent optimization techniques which currently have to be guided by an experienced user.

1.3. *Overview of this Chapter*

In this chapter we will describe the principal approaches to program synthesis and the related deductive techniques. We will focus mostly on general principles and derivation strategies and omit approaches which are tailored to a specific application or are only weakly related to the field of automated deduction. After introducing some notation and basic concepts in section 2 we shall discuss the proofs-as-programs paradigm and transformational synthesis in Section 3. In Section 4 we present knowledge based synthesis techniques and their relation to automated deduction. Aspects of program development which are related to program synthesis will be discussed briefly in Section 5. We conclude with a short outline of future directions.

1.4. *Other Sources of Information*

Obviously not all approaches to program synthesis can be covered equally well in this chapter and some research areas cannot be discussed at all. A reader interested in a broader overview of the field will find additional information in surveys like (Balzer, 1985; Goldberg, 1986; Steier & Anderson, 1989; Lowry & McCartney, 1991). The proceedings of the IEEE International Conference on *Automated Software Engineering* (formerly the *Knowledge Based Software Engineering Conference*), the *International Conference on Mathematics of Program Construction,* the International Workshop on *Logic Program Synthesis and Transformation,* and the Journal of *Automated Software Engineering* present recent developments on a regular basis.

2. Preliminaries and Notation

Program synthesis is a formal reasoning process which, in addition to first-order logic, often involves reasoning about computable functions and predicates describing their properties. Furthermore, type information is crucial in most modern programming languages since the data type of a variable determines which computations can be applied to it. In a logical formula

we will provide a type for each quantified variable and allow quantification over predicates and functions. We shall use $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, $\forall$, and $\exists$ for negation, conjunction, disjunction, implication, reverse implication, equivalence, universal and existential quantifier. Quantified variables will be followed by a colon and a type expression. If available, we use standard mathematical notation for data types, functions, and predicates. The formula $\forall x{:}\mathbb{N}.\,\exists z{:}\mathbb{Z}.\,(z^2 \leq x \wedge x < (z+1)^2)$, for instance, expresses that every natural number has an integer square root.

Formal specifications and programs are the basic objects of reasoning in program synthesis. A programming problem is characterized by the *domain* D of the desired program, its *range* R, an *input condition* I on *admissible* input values x, and an *output condition* O on *feasible* output values z. A program is characterized by a such specification and a possibly partial computable function *body* from D to R. Formally, these concepts are defined as follows.

DEFINITION 1. A formal *specification spec* is a 4-tuple (D, R, I, O) where D and R are data types, I is a boolean predicate on D, and O one on $D \times R$. A formal *program* p is a tuple $((D, R, I, O), body)$ where (D, R, I, O) is a specification and $body\colon D \rightharpoonup R$ a computable function.

In the literature there is a variety of formal notations for specifications and programs. The domain, range, and input-condition may be implicitly contained in the output condition. Specific aspects of the output condition may be emphasized. The above tuple form represents the essential information that is handled during the synthesis process but is sometimes difficult to read. We will often use the following more convenient notation.

$$\texttt{FUNCTION}\ f(x{:}D){:}R\ \texttt{WHERE}\ I[x]\ \texttt{RETURNS}\ z\ \texttt{SUCH THAT}\ O[x,z] \equiv body[f,x]$$

where $body[f,x]$ expresses that the variable x and the function name f may occur free in *body*. The latter is a convenient way to express recursion. We will use a functional programming language mixed with mathematical notation to describe a program body.

The above notation suggests that a program computes a *single* value satisfying the output condition O. Often, however, *all* solutions of the output condition have to be computed, assuming that there are finitely many. This problem could be expressed by the specification

$$\texttt{FUNCTION}\ f(x{:}D){:}\texttt{Set}(R)\ \texttt{WHERE}\ I[x]\ \texttt{RETURNS}\ y\ \texttt{SUCH THAT}\ y = \{z \mid O[x,z]\}$$

but we abbreviate it by the following more elegant notation

$$\texttt{FUNCTION}\ f(x{:}D){:}\texttt{Set}(R)\ \texttt{WHERE}\ I[x]\ \texttt{RETURNS}\ \{z \mid O[x,z]\}$$

Note that here $O[x,z]$ is only a part of the output condition.

Another key concept is the notion of correctness. A program is correct if its body computes a feasible output for each admissible input.

DEFINITION 2. A computable function *body:*$D \not\rightarrow R$ *satisfies* a formal specification *spec*=(D, R, I, O) if $O(x, body(x))$ holds for all $x \in D$ satisfying $I(x)$. A program *p*=(*spec*,*body*) is *correct* if its body satisfies its specification. A formal specification *spec*=(D, R, I, O) is *satisfiable* if there is a computable function *body:*$D \not\rightarrow R$ which satisfies it.

Thus from the viewpoint of deduction program synthesis is the same as proving the satisfiability of a specification and constructing a program body during the proof process.

3. SYNTHESIS IN THE SMALL

Originally, approaches to program synthesis were developed in the framework of automated deduction (Green, 1969; Manna & Waldinger, 1979; Manna & Waldinger, 1980; Bibel, 1980). Their synthesis strategies were built on general techniques from first-order theorem proving and rewriting which made them comparably easy to explain. Their fundamental ideas work very well for small programming examples and are, at least in principle, applicable to all kinds of programming problems. Essentially, there are two broad paradigms for program synthesis: "proofs-as-programs" (Bates & Constable, 1985), and "synthesis by transformations" (Burstall & Darlington, 1977).

1. In the *proofs-as-programs* paradigm a specification is expressed by the statement that a realization of the specification exists. To synthesize an algorithm, a constructive proof of this statement is produced. Because the proof is constructive, it embodies a method for realizing the specification which can be extracted from it This paradigm emphasizes the synthesis of algorithms (computable functions) from declarative specifications. Its main issue is correctness. Research focuses on the development of strong theorem provers and mechanisms for extracting algorithms from proofs.
2. In contrast, *synthesis by transformations* derives the algorithm from the specification by forward reasoning, usually based on rewrite rules which encode the logical laws of a particular domain. A specification is viewed as executable, though inefficient program and the emphasis lies on improving efficiency through program transformations. This paradigm is particularly well-suited for the synthesis of logic programs since a declarative formula can in fact be viewed as executable program which "only" has to be transformed into some restricted syntax like Horn logic.

In the following we shall present both approaches separately and discuss their principal advantages and limitations.

3.1. *Proofs as Programs*

The idea to use formal proofs as a method forconstructing programs from formal specifications arose from the observation that developing a program and proving it correct are just two aspects of the same problem (Floyd, 1967). While verifying a given program one has to formalize the same ideas which were originally used in the construction of the program. As a consequence, program verification essentially repeats the entire programming process except for a different degree of formality.

To avoid such double work when designing software the proofs-as-programs paradigm has focused on developing a program and its correctness proof at the same time (Green, 1969; Manna & Waldinger, 1980). A specification *spec*$=(D,R,I,O)$ is represented by a *specification theorem* of the form $\forall x{:}D.\exists z{:}R.(I[x] \Rightarrow O[x,z])$. To synthesize an algorithm, a *constructive* proof of this theorem is generated. In such a proof each individual step corresponds to a step in a computation. Thus the complete proof corresponds to a program satisfying the specification described by the theorem. This program can be *extracted* from the proof in a final step. The following example illustrates the main argument behind the proofs-as-programs paradigm.

EXAMPLE 3. The theorem $\forall x{:}\mathbb{Z}.\exists z{:}\mathbb{Z}.(x{\geq}0 \Rightarrow z^2{\leq}x \wedge x{<}(z{+}1)^2)$ expresses the statement that for each natural number x there is some integer z which is the square root of x. A constructive proof for this statement shows that for each x such a z can be *constructed*. It embodies a universal method for computing z from x, i.e. an algorithm for computing integer square roots.

This algorithm strongly depends on *how* the theorem was proved. Typically one would proceed by induction on x, providing 0 as solution for the base case and performing a case analysis in the step case. This means to check if the previous x is the exact square of the previous z or not and to provide a new value for z accordingly. The resulting proof implicitly contains an algorithm which computes z via primitive recursion and a conditional in the step case. It could formally be expressed as

```
FUNCTION sqrt(x:ℤ):ℤ WHERE x≥0 RETURNS z SUCH THAT z²≤x ∧ x<(z+1)²
≡ if x=0 then 0 else let z=sqrt(x-1) in if z²=x-1 then z+1 else z
```

The correspondence between programs satisfying a given specification and proofs of the specification theorem can be understood as a specific reinterpretation of the *Curry-Howard Isomorphism* (Curry *et al.*, 1958; Tait, 1967) between proofs and terms of an extended λ-calculus. For instance, *constructing* a proof for a universally quantified formula $\forall x{:}\alpha.P[x]$ corresponds to a function which takes an element x of the type α and returns (a term corresponding to) a proof for $P[x]$. *Analyzing* $\forall x{:}\alpha.P[x]$ in order to show $P[a/x]$ corresponds to applying such a function to the element a. Likewise, a proof

Table I. Proofs-as-programs correspondence for first-order logic

Connective	**Construction**	**Analysis**
Atomic Literal	Variable*	—
Conjunction	Pair	Projection
Disjunction	Injections	Conditional
Implication	Function declaration	Function application
Negation	Function declaration	Function application
Existential Quantification	Pair	Projection
Universal Quantification	Function declaration	Function application

* *The same literal will always be associated with the same variable*

for a formula $\exists x{:}\alpha.P[x]$ corresponds to a pair (a, p_a) where a is an element of α and p_a a proof for $P[a/x]$. A constructive proof for $P \vee Q$ corresponds either to a left injection $\mathtt{inl}(p)$ or a right injection $\mathtt{inr}(q)$, depending on whether P or Q is being shown. Analyzing a disjunction leads to a conditional which examines these injections.

Table I describes the correspondence between proofs and programs for first-order logic. Details depend on the individual proof calculus and the formal language in which programs are to be expressed. In sequent or natural deduction calculi the correspondence can be represented straightforwardly since programming constructs can be tied directly to proof rules. This makes it easy to define a mechanism for extracting a program from the proof of a specification theorem and to justify the proofs-as-programs principle.

THEOREM 4. *A specification* $spec{=}(D, R, I, O)$ *is satisfiable iff the theorem* $\forall x{:}D.\exists z{:}R.(I[x] \Rightarrow O[x,z])$ *can be proved (constructively).*

Proof. According to table I the proof of a specification theorem corresponds to a computable function *pf* which on input $x \in D$ returns a pair (z, p) where $z \in R$ and p is a proof term for $I[x] \Rightarrow O[x,z]$. By projecting p away, i.e. by defining $body(x) \equiv \mathtt{first}(pf(x))$ we get a correct program for the specification. Conversely, if we have a computable function $body{:}D \not\to R$ and a term *prf* corresponding to a correctness proof for the program $((D, R, I, O), body)$, then combining *body* and *prf* leads to a proof of the specification theorem. □

It should be noted that in a proof of a specification theorem only the part that shows how to build z from x needs to be constructive. The verification of $I[x] \Rightarrow O[x,z]$ does not contribute to the generated algorithm and can be performed by non-constructive means as well.

Synthesis strategies based on the proofs-as-programs principle heavily rely on the availability of theorem provers which can deal with a rich variety of application problems. In principle, any constructive proof method for which an extraction mechanism can be defined is useful for this purpose.

Skolemization and resolution, although not entirely constructive,[1] generate substitutions which can be extracted from a proof and were used in early approaches (Green, 1969; Manna & Waldinger, 1975) to program synthesis.

It has turned out that pure first-order theorem proving is too weak for program synthesis as it cannot generate more than simple substitutions which do not consider the meaning of atomic propositions. Reasoning about recursion or loops plays a vital part in the design of almost any program as well as knowledge about arithmetic and other application domains. The former has been addressed by extending theorem proving by induction techniques such as *rippling* (Bundy, 1989; Bundy *et al.*, 1992) or *constructive matching* (Franova, 1985). The latter requires specialized decision procedures since an axiomatization of the application domain would require thousands of formulas to be considered and thus be extremely inefficient.

Currently there is no integrated theorem proving system which can deal with all these proofs tasks equally well. In spite of a variety of approaches and systems which have been developed over the past 25 years the automatic syntheses of programs through proofs has only been able to create solutions for relatively small problems. Furthermore, pure proof methods do not always lead to an efficient solution. Inductive proofs of specification theorems as in example 3 can only generate primitive recursive programs with *linear* complexity although logarithmic time is possible. Faster algorithms can only be created in less rigorous frameworks which allow inductions on the output variable or by explicitly providing the algorithm structure (see Section 4.3).

3.2. *Transformational Synthesis*

While the proofs-as-programs paradigm primarily aims at the derivation of formally verified programs, efficiency considerations have been the original motivation for program synthesis by transformations. For many programming problems it is relatively easy to produce and verify a prototype implementation in a lucid, mathematical and abstract style. Correctness, comprehensibility, and easy modifications are the advantages of such programs. But a lack of efficiency makes them unacceptable for practical purposes and they can only viewed as detailed specification of the "real" program.

Initially, program development by transformations (Darlington,1975; Burstall & Darlington, 1977) focused on systematically transforming *programs* into more intricate but efficient ones. This idea was adapted to program synthesis (Manna & Waldinger, 1975; Darlington, 1975; Manna & Waldinger,

[1] Matrix-based theorem provers have recently been extended to constructive first-order logic (Otten & Kreitz, 1995). The corresponding extraction mechanism is based on a conversion of matrix proofs into sequent proofs (Schmitt & Kreitz, 1995).

1979) as a technique for transforming *specifications,* which are not necessarily executable, into expressions which can be handled by a computer. In logic programming this distinction between declarative specifications and algorithmic programs has almost disappeared. Research in transformational synthesis today aims mostly[2] at *logic program synthesis and transformations* (Clark & Sickel, 1977; Hogger, 1981), also called "*deductive program synthesis*".

Approaches to transformational synthesis usually proceed by step-wisely rewriting the output-condition of a specification into a logically equivalent or stronger formula which algorithmically can be viewed as refinement of the specification. In a final step program formation rules are applied in order to create programming constructs from the derivation tree. In the context of logic programming this step is almost superfluous since the resulting formula only needs a few syntactical adjustments to be understood as logic program.

Individual approaches vary greatly as far as notation is concerned but essentially proceed as follows. A specification *spec*$=(D,R,I,O)$ is represented by the formula $\forall x{:}D.\forall z{:}R.(I[x] \Rightarrow (P(x,z) \Leftrightarrow O[x,z]))$ which defines a new predicate P as equivalent to the given output condition. The intuition behind this is to view $O[x,z]$ as body of a program whose name P is newly introduced and to derive properties of P by *forward reasoning.* In this setting the role of x and z is not yet fixed but the transformation process will optimize $O[x,z]$ with respect to a particular designation of input and output variables. The goal of transforming $O[x,z]$ is to create a valid formula of the form

$$\forall x{:}D.\forall z{:}R.(I[x] \Rightarrow (P(x,z) \Leftarrow O_f[x,z,P]))$$

which is algorithmically satisfactory. This goal of transformational synthesis cannot be expressed within the logical formalism but only within the derivation *strategy*. A typical criterion in logic program synthesis, for instance, is that $O_f[x,z,P]$ must be a Horn formula and contains, apart from recursive instances of P, only predicates which are already known to be executable.

In order to transform $O[x,z]$ into $O_f[x,z,P]$ *conditional rewrite rules* are applied. These rules are based on (conditional) equivalences or implications which are used to define concepts of some application domain and to state their properties. In particular, the definition of P itself is considered as conditional equivalence which can be folded if recursion needs to be introduced.

Program formation rules determine how to convert the result of these transformations into a valid program. Logic programs, for instance, can be generated by simply dropping the universal quantifiers, the input condition, and any existential quantifier in the body O_f, by splitting disjunctions into two separate clauses with the same head, and by replacing functions by the

[2] Research on *program* transformations is still an active research area. Here, algebraic methods are dominating while logical deduction plays only a minor role.

corresponding predicates. Furthermore, equalities and structural decompositions in the body can be replaced by modifying the head of a clause in order to take stronger advantage of unification. The construction of functional and imperative programs is more complex since it requires a formation rule corresponding to each logical junctor and one for recursion. Some of these rules are similar to the algorithm schemata given in Sections 4.2.1 and 4.2.2.

As in the case of proofs-as-programs, the validity of transformational synthesis essentially depends on the program formation mechanism. The requirement to base the transformations on equivalences or implications only makes sure that transforming $O[x,z]$ will result in a valid formula. The correctness of the generated program is then assured by the following theorem.

THEOREM 5. *Let spec=(D,R,I,O) be a formal specification and P be a predicate defined by* $\forall x{:}D.\forall z{:}R.(I[x] \Rightarrow (P(x,z) \Leftrightarrow O[x,z]))$. *If the formula* $\forall x{:}D.\forall z{:}R.(I[x] \Rightarrow (P(x,z) \Leftarrow O_f[x,z,P]))$ *is valid then the program extracted from it satisfies the specification spec.*

Proof. (Sketch) In the case of logic programs, where the semantics is almost identical to the declarative meaning, the generated algorithm is logically a refinement of the original output condition whose correctness is easy to see. A full proof requires an induction about the structure of O_f and a precise definition of the program formation rules for each logical junctor. □

Research in transformational synthesis has mostly been concerned with developing efficient rewrite techniques and heuristics producing satisfactory practical results. Some approaches use folding and unfolding a fixed set of equivalences or equations (Darlington, 1975; Darlington, 1981), a modification of the Knuth-Bendix completion procedure (Dershowitz, 1985), or refinements according to given transformation and program formation (Manna & Waldinger, 1977; Manna & Waldinger, 1979) are closely related to each other. The LOPS-approach (Bibel, 1980) provides a small set of AI-oriented strategies which we shall describe in the following example.

EXAMPLE 6. In order to synthesize an algorithm computing the maximum $\mathtt{m}$ of a finite set $\mathtt{S}$ of integers we begin by defining a new predicate *max* to describe an element which is greater or equal to all other elements of $\mathtt{S}$.

$$\forall \mathtt{S}{:}\mathtt{Set}(\mathbb{Z}).\forall \mathtt{m}{:}\mathbb{Z}.(\mathtt{S}\neq\emptyset \Rightarrow (\mathit{max}(\mathtt{S},\mathtt{m}) \Leftrightarrow \mathtt{m}\in\mathtt{S} \wedge \forall \mathtt{x}\in\mathtt{S}.\mathtt{x}\leq\mathtt{m}))$$

In the first step we try to *guess* a solution, assuming that we will either find one or provide a way to reduce the problem. Formally, we introduce an existentially quantified variable $\mathtt{g}$ and add $\mathtt{g}=\mathtt{m} \vee \mathtt{g}\neq\mathtt{m}$ as indicator for a successful or a failing guess. To limit the possibilities for guessing we introduce a *domain condition* $\mathtt{g}\in\mathtt{S}$, which is taken from the original problem description. We add these conditions to the output condition of *max* and get

$$\forall \mathtt{S}{:}\mathtt{Set}(\mathbb{Z}).\forall \mathtt{m}{:}\mathbb{Z}.(\mathtt{S}\neq\emptyset \Rightarrow (\mathit{max}(\mathtt{S},\mathtt{m}) \Leftrightarrow \exists \mathtt{g}{:}\mathbb{Z}.(\mathtt{g}\in\mathtt{S} \wedge \mathtt{m}\in\mathtt{S} \wedge \forall \mathtt{x}\in\mathtt{S}.\mathtt{x}\leq\mathtt{m} \wedge (\mathtt{g}=\mathtt{m} \vee \mathtt{g}\neq\mathtt{m}))))$$

Next, we distribute the disjunction over the initial output condition and introduce two predicates max_1 and max_2 as abbreviations. They use g as additional input variable with input condition g∈S. This transformation results in

$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.(S\neq\emptyset \Rightarrow$
$(max(S,m) \Leftrightarrow \exists g{:}\mathbb{Z}.(g\in S \wedge (max_1(S,g,m) \vee max_2(S,g,m)))))$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.(S\neq\emptyset\wedge g\in S \Rightarrow (max_1(S,g,m)\Leftrightarrow m\in S\wedge g=m\wedge\forall x\in S.x\leq m))$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.(S\neq\emptyset\wedge g\in S \Rightarrow (max_2(S,g,m)\Leftrightarrow m\in S\wedge g\neq m\wedge\forall x\in S.x\leq m))$

The above steps are equivalence preserving and can be executed by a strategy called *GUESS-DOMAIN*. Before continuing we analyze the three subproblems. The first one is solved once we have solutions for max_1 and max_2. The second has a trivial solution and we only have to investigate the third.

The strategy *GET-REC* tries to introduce recursion by reformulating the output condition for max_2 into an instance of the output condition of *max*. It retrieves information about well-founded recursions on finite sets and attempts a rewrite towards S-g, the set resulting from removing g from S. For this purpose it searches for lemmata involving S-g≠∅, m∈S-g, and ∀x∈S-g. x≤m and rewrites the problem accordingly. This leads to a disjunction in the input condition which is split into two cases.

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.(S=\{g\} \wedge g\in S \Rightarrow$
$(max_2(S,g,m) \Leftrightarrow m\in S \wedge g\neq m \wedge\forall x\in S.x\leq m))$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.(S\text{-}g\neq\emptyset \wedge g\in S \Rightarrow$
$(max_2(S,g,m) \Leftrightarrow m\in S\text{-}g \wedge g\neq m \wedge g\leq m \wedge \forall x\in S\text{-}g.x\leq m))$

The output condition of the first case proves to be contradictory since the input condition states that S consists of the single element g. In the second we can fold back the definition of *max*, simplify g≠m ∧ g≤m, and get

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.(S\text{-}g\neq\emptyset \wedge g\in S \Rightarrow (max_2(S,g,m)\Leftrightarrow max(S\text{-}g,m)\wedge g<m))$

Now all subproblems are solved. After removing redundancies we create a logic program according to the above-mentioned formation rules and get

```
max(S,M)       :- member(G,S), max_aux(S,G,M).
max_aux(S,G,M) :- setminus(S,G,S_G), max(S_G,M), less(G,M),!.
max_aux(S,M,M) :- setless(S,M).
```

The strategies *GUESS-DOMAIN* and *GET-REC* are the center of all LOPS derivations. (Bibel, 1980) describes several strategies which support them if more complex examples shall be solved. Unfortunately many of them turned out to be difficult to formalize precisely and the LOPS approach, like many other good ideas, could not be turned into a system which was able to solve more than a few small examples.

3.3. *Discussion*

Despite their different origins there are no principal difference between the proofs-as-programs paradigm and transformational synthesis. Transformations can be simulated in a proof environment by using lemmata (i.e. the cut rule) and proof transformations if recursion is introduced. Proof rules can

be viewed as special rewrite rules. In principle, each derivation strategy in one paradigm can be converted into one for the other. The real question is rather which environment is more convenient for a particular method. The proofs-as-programs paradigm provides a clearly defined proof goal and favors analytic methods from theorem proving. Depending on the rigorousity of the proof system correctness can be guaranteed for all inferences. The latter is not the case for transformational synthesis since the transformation rules which deal with domain knowledge are usually not verified. The advantage is a smaller formal overhead which together with the focus on forward reasoning makes the derivation of programs often somewhat easier.

Both synthesis paradigms, however, rely on general reasoning methods which operate on the level of elementary logical inferences. This makes the fundamental techniques relatively easy to explain and implement and was quite successful for solving simple programming problems like sorting or the *n*-queens problem. A fully automated synthesis of larger problems, however, is almost impossible due to the explosion of the search space. Interaction is also very difficult since general methods from automated deduction have little to do with the way in which a programmer would reason when developing a program. Therefore these synthesis techniques do not scale up very well.

The main problem of general approaches to program synthesis is that they force the synthesis system to derive an algorithm almost from scratch and to re-invent algorithmic principles which are well understood in computer science. Solving complex programming problems, however, heavily relies on knowledge about application domains *and* standard programming techniques. One can hardly expect a synthesis system to be successful if such expertise is not already embedded. More recent approaches therefore make programming knowledge explicit, instead of hiding it in the code of their strategies, and develop their algorithm design strategies on the basis of such knowledge.

4. KNOWLEDGE BASED PROGRAM SYNTHESIS

Knowledge based synthesis strategies arose from the observation that the development of complex algorithms requires a deep understanding of fundamental algorithmic structures. They aim at a *cooperation* between programmer and machine. A programmer shall guide the derivation process by high-level strategic decisions which, like the selection of an appropriate program structure, require a deep understanding of the problem and the intended solution. The synthesis system will fill in the formal details and ensure the correctness of the generated algorithms.

This means that much work will be invested into the development of the synthesis *system* in order to ease the burden for the synthesis *process*. The design of knowledge based synthesis systems requires an analysis of the fundamental structure of standard algorithms and of the axioms which guarantee their correctness. It also requires the development of strategies which construct algorithms on the basis of this analysis and need only a few high-level design decisions. Since the analysis usually leads to a set of theorems stating conditions on the parameters of some well-structured algorithm schema the strategies have to derive values for these parameters which satisfy these conditions. The final step consists of instantiating the algorithm schema. The result can later be refined by program transformation techniques to be made more efficient. One advantage of this technique is that it makes the synthesis strategy independent from the programming language in which the algorithm schema is formulated and allows to use *any* programming language with a well-defined semantics for this purpose.

Synthesis strategies have been developed for a number of algorithmic classes (see (Smith & Lowry, 1990) for a general exposition). In this section we will illustrate the principle by describing a strategy for the development of so-called global search algorithms and present a variety of strategies related to other classes of algorithms. We will also discuss how to integrate knowledge based algorithm design and methods from automated deduction.

4.1. *Synthesizing Global Search Algorithms*

Solving a problem by enumerating candidate solutions is a well-known concept in computer science. *Global search* is a technique that generalizes binary search, backtracking, and other methods which explore a search space by looking at whole sets of possible solutions at once. Its basic idea is to combine enumeration and elimination processes. Usually, global search investigates the complete set of output values for a given input. It systematically enumerates a search space, which must contain this set, and tests if certain elements of the search space satisfy the output-condition. The test is necessary to guarantee correctness but rather inefficient if the search space is much bigger than the set of solutions. Therefore whole regions of the search space are filtered out during the enumeration process if it can be determined that they cannot contain output values.

A careful analysis in (Smith, 1987b), later refined and formalized in (Kreitz, 1992; Kreitz, 1996), has shown that the common structure of global search algorithms can be expressed by the pair of abstract programs presented in Figure 1. These programs contain placeholders *D, R, I,* and *O* for a specification and seven additional components $S, J, s_0, sat, split, ext, \Phi$ which are specific

```
FUNCTION f(x:D):Set(R) WHERE I[x]  RETURNS {z | O[x,z]}
≡ if Φ[x,s0[x]]  then fgs(x,s0[x])  else []
FUNCTION fgs(x,s:D×S):Set(R)  WHERE I[x]∧J[x,s]∧Φ[x,s]
   RETURNS {z | O[x,z]∧ sat[z,s]}
≡ let immediate_solutions = filter (\z.O[x,z]) (ext[s])
   and recursive_solutions =
       let filtered_subspaces = filter (\t.Φ[x,t]) (split[x,s]) in
         flatten (map (\t. fgs(x,t)) filtered_subspaces)
   in   append immediate_solutions recursive_solutions
```

Figure 1. Structure of global search algorithms

for a global search algorithm. On input x this algorithm starts investigating an *initial search space* $s_0[x]$ and passes it through the *filter* Φ which checks globally whether a search region s contains solutions. Using an auxiliary function f_{gs} the algorithm then repeatedly *extracts* ($ext[s]$) candidate solutions for testing and *splits* a search space s into a set $split[x,s]$ of subspaces which again are passed through the filter Φ. Subspaces which survive filtering contain solutions and are investigated recursively.

For the sake of efficiency search spaces are represented by *space descriptors* $s \in S$ instead of sets of values. The fact that a value $z \in R$ belongs to the search space described by s is denoted by $sat[z,s]$ while $J[x,s]$ expresses that s is a *meaningful* space descriptor for the input x. Formally, S must be a data type. J and Φ must be predicates on $D \times S$ and sat one on $R \times S$. $s_0 : D \not\to S$, $split : D \times S \not\to S$, and $ext : S \not\to \mathtt{Set}(R)$ must be computable functions.

$\forall x{:}D.\ \forall z{:}R.\ \forall s{:}S.$

1. $I(x) \Rightarrow J(x, s_0(x))$
2. $I(x) \wedge J(x,s) \Rightarrow \forall t \in split(x,s).\ J(x,t)$
3. $I(x) \wedge O(x,z) \Rightarrow sat(z, s_0(x))$
4. $I(x) \wedge J(x,s) \wedge O(x,z) \Rightarrow sat(z,s) \Leftrightarrow \exists k{:}\mathbb{N}.\ \exists t \in split^k(x,s).\ z \in ext(t)$
5. $I(x) \wedge J(x,s) \Rightarrow \Phi(x,s) \Leftarrow \exists z{:}R.\ sat(z,s) \wedge O(x,z)$
6. $I(x) \wedge J(x,s) \Rightarrow \exists k{:}\mathbb{N}.\ split^k_\Phi(x,s) = \emptyset$

where $split_\Phi(x,s) = \{ t \in split(x,s) \mid \Phi(x,t) \}$

and $split^0_\Phi(x,s) = s$, $split^{k+1}_\Phi(x,s) = \bigcup_{t \in split^k_\Phi(x,s)} split_\Phi(x,t)$.

Figure 2. Axioms of global search

Six requirements, formalized in Figure 2, ensure the correctness of the global search algorithms. The initial descriptor must be meaningful (1). Splitting must preserve meaningfulness (2). All solutions must be contained in the initial search space (3) and be extractable after splitting finitely many times (4). Subspaces containing solutions must pass the filter (5). Filtered splitting must eventually terminate (6).

THEOREM 7. *If $D,R,I,O,S,J,sat,s_0,split,ext$, and Φ fulfill the axioms of global search then the pair of programs in Figure 1 is correct.*

*Proof. (see (*Smith, 1987b*; *Kreitz, 1996*) for details)* The correctness of f follows from the correctness of f_{gs} and axioms 1 and 3. The *validity* of f_{gs} follows from axioms 2, 4, and 5. To prove this fact one has to show by induction that the result of computing $f_{gs}(x,s)$ is $\bigcup\{\ \{z \in ext(t) \mid O(x,z)\} \mid t \in \bigcup_{j<i} split^j_\Phi(x,s)\ \}$ where i is the smallest number with $split^i_\Phi(x,s) = \emptyset$, i.e. the number of steps until $f_{gs}(x,s)$ terminates. *Termination* and thus total correctness of f_{gs} follows from axiom 6. □

Thus a global search algorithm for a given specification can be synthesized by deriving seven components S, J, sat, s_0, $split$, ext, and Φ which satisfy the six axioms and instantiating the abstract programs accordingly. A direct derivation of global search algorithms, however, is still a difficult task since Theorem 7 does not show how to *find* the additional components. Also, a verification of the axioms, particularly of axioms 4 and 6, would put a heavy load on the derivation process. It is much more meaningful to base the construction of global search algorithms on additional knowledge about algorithmic structures. For each range type R, for instance, there are usually only a few general methods to enumerate search spaces. Each global search algorithm will use an instance of such a method. Therefore it makes sense to store information about generic enumeration processes in a knowledge base and to develop techniques for adapting them to a particular programming problem.

The investigations in (Smith, 1987b) have shown that standard enumeration structures for some range type R can be represented by objects of the form $((D_G,R,I_G,O_G),S,J,s_0,sat,split,ext)$ which satisfy the axioms 1 to 4. Such objects are called *GS-theories.* A *problem reduction* mechanism will make sure that these axioms are preserved when the enumeration structure is *specialized* to a specification which has the same range type.

Specializing a standard GS-theory G works as follows. Its specification $spec_G = (D_G,R_G,I_G,O_G)$ characterizes a general enumeration method f_G which explores the space R_G as far as possible. Specialization simply means to truncate the search space such that superfluous elements will not be enumerated. It is possible if $spec_G$ is more general than the given specification.

DEFINITION 8. A specification $spec_G = (D_G,R_G,I_G,O_G)$ *generalizes* the specification $spec = (D,R,I,O)$ if the following condition holds.

$$R{=}R_G \wedge \forall x{:}D.\ \exists x_G{:}D_G.(I(x) \Rightarrow (I_G(x_G) \wedge \forall z{:}R.(O(x,z) \Rightarrow O_G(x_G,z))))$$

We also say that $spec_G$ can be *specialized* to *spec*.

Thus specialization restricts the *output* of f_G to values which satisfy the stronger condition O. Furthermore, it allows to adapt the *input* x of a specified problem since the x is mapped to a value x_G which serves as input for

the search performed by f_G. According to the proofs-as-programs paradigm (theorem 4) a proof of the generalization implicitly contains a substitution $\theta{:}D{\to}D_G$ which maps x to x_G. θ can be extracted from the proof and then be used for refining f_G into a search method with inputs from D instead of D_G.

COROLLARY 9. *If* $spec_G = (D_G, R_G, I_G, O_G)$ *generalizes the specification* $spec = (D, R, I, O)$ *then there is a function* $\theta{:}D{\to}D_G$ *which satisfies the condition* $\forall x{:}D.(I(x) \Rightarrow (I_G(\theta(x)) \wedge \forall z{:}R.(O(x,z) \Rightarrow O_G(\theta(x), z))))$.

Altogether problem reduction allows us to create a global search algorithm f for *spec* by defining $f(x) = \{z \in f_G(\theta(x)) \mid O(x,z)\}$. For the sake of efficiency, the modifications caused by θ and O are moved directly into the components of the global search algorithm. By an index θ as in J_θ or $split_\theta$ we indicate that θ is applied to all arguments expecting a domain value from D, e.g. $split_\theta(x,s) = split(\theta(x), s)$.

Specializing predefined GS-theories allows us to derive six components of a global search algorithm which satisfy axioms 1 to 4 with little effort. Similarly, we can avoid having to prove the sixth axiom explicitly. For each enumeration structure there are only a few standard methods which ensure termination through an elimination process. In (Kreitz, 1992; Kreitz, 1996) it has been shown that these can be represented by filters for a GS-theory G which satisfy axiom 6. Such filters are called *well-founded wrt. G* and this property is preserved during specialization as well. Thus specialization reduces the proof burden to checking that, after specialization, the selected filter is *necessary* wrt. the GS-theory, i.e. that it satisfies axiom 5. The process of adapting the search space to the specific problem can be completely formalized and expressed in a single theorem.

THEOREM 10. *Let* $G = ((D_G, R, I_G, O_G), S, J, s_0, sat, split, ext)$ *be a GS-theory such that* (D_G, R, I_G, O_G) *generalizes* $spec = (D, R, I, O)$. *Let* θ *be the substitution extracted from the generalization proof.*
Then $G_\theta = ((D, R, I, O), S, J_\theta, s_{0\theta}, sat, split_\theta, ext)$ *is a GS-theory. Furthermore if* Φ *is a well-founded filter wrt. G then* Φ_θ *is well-founded wrt.* G_θ

Proof. Axioms 1 to 3 for G_θ can be shown directly by combining the properties of θ (Corollary 9) with the global search axioms. Axiom 4 and the well-foundedness of Φ_θ can be proved by a straightforward induction over k and the number of iterations of *split*, again using the properties of θ and the corresponding axioms for G and Φ. □

Adapting standard algorithmic knowledge to a given problem moves most of the proof burden into the creation of the knowledge base and keeps the synthesis process itself comparably easy. Information retrieved from the knowledge base will provide all the basic components and guarantee that axioms 1

to 4 and 6 are satisfied. Only the specialization property and the necessity of the specialized filter – conditions which are much easier to prove than axioms 4 and 6 – need to be checked. These insights led to the following strategy.

STRATEGY 11. *Given the specification*
FUNCTION $F(x{:}D){:}\mathtt{Set}(R)$ WHERE $I[x]$ RETURNS $\{z \mid O[x,z]\}$

1. *Select a GS-theory* $G=((D_G,R,I_G,O_G), S,J,s_0,sat,split,ext)$ *for R.*
2. *Prove that* $spec_G=(D_G,R,I_G,O_G)$ *generalizes the specification.*
   Derive a substitution $\theta : D \rightarrow D_G$ *from the proof and specialize G with* θ.
3. *Select a well-founded filter* Φ *for G and specialize it with* θ.
4. *Prove that the specialized filter is necessary for the specialized GS-theory.*
5. *Instantiate the program schema given in Figure 1.*

In steps 1 and 3 of the above strategy, the selection of the GS-theory *G* and the well-founded filter Φ is mostly a matter of design. If there are choices, the decision should be made by a human programmer rather than by a predefined heuristic. In step 4 the specialized filter could be further refined heuristically by adding conditions which preserve its necessity. In some cases this improves the efficiency of the generated algorithm drastically. The global search strategy has been successfully applied to a number of programming problems (Smith, 1987b; Smith, 1991a). In particular it has been used for the synthesis of commercial transportation scheduling algorithms (Smith & Parra, 1993; Gomes *et al.*, 1996) which turned out to be much more efficient than any hand-coded implementation.

4.2. *Design Strategies for Other Algorithmic Classes*

Schema-based synthesis strategies have been developed for a variety of algorithmic classes. The can create generate & test algorithms (Smith, 1987a), static algorithms such as the formation of conditionals (Smith, 1985a), divide & conquer algorithms (Smith, 1985b), local search algorithms (Lowry, 1991), and general problem reduction generators (Smith, 1991b). In the following we shall briefly describe the central ideas, their justification, and the corresponding synthesis strategies.

Static Algorithms
Static algorithms, i.e. algorithms without loops or recursion, are a means for adapting already existing programs to similar tasks. Such algorithms are usually designed during the synthesis of complex programming problems but can also created separately. The corresponding design strategies occur both as independent method and as part of more sophisticated synthesis techniques.

Operator Match is a reduction technique that solves a programming problem $spec = (D,R,I,O)$ by transforming its inputs into inputs of a known problem $spec' = (D',R',I',O')$ and transforming outputs of that problem back into its own range type. This technique works if, after transformation, the input condition I is stronger that I' and the output condition O is weaker than O':

$$\forall x{:}D.\, \exists x'{:}D'.\, (I(x) \Rightarrow (I'(x') \wedge \forall z'{:}R'.\exists z{:}R.(O'(x',z') \Rightarrow O(x,z))))$$

We say that *spec reduces to spec'* if the above condition can be proved. In this case, according to the proofs-as-programs paradigm, the proof contains a substitution $\theta{:}D \to D'$ which maps x to x' and a substitution $\sigma{:}D' \times R' \to D \times R$ which maps (x',z') to (x,z). Thus a program satisfying the original specification can be generated by composing θ, a solution g for *spec'*, and σ.

THEOREM 12. *Let $spec = (D,R,I,O)$ and $spec' = (D',R',I',O')$ be specifications, $\theta{:}D \to D'$, $g : D' \nrightarrow R'$, and $\sigma{:}D' \times R' \to D \times R$. If (1) spec reduces to spec', (2) θ and σ are substitutions extracted from the proof of (1), and (3) g satisfies spec' then the following program is correct*

$\texttt{FUNCTION } f(x{:}D){:}R \texttt{ WHERE } I[x] \texttt{ RETURNS } z \texttt{ SUCH THAT } O[x,z]$
$\equiv\ \sigma(\theta(x), g(\theta(x)))$

Specialization, already discussed in the context of global search (see definition 8), can be seen as a variant of operator matching. The difference is that specialization deals with set-valued problems and *restricts* the output.

THEOREM 13. *Let $spec = (D,R,I,O)$ and $spec' = (D',R',I',O')$ be specifications, $\theta{:}D \to D'$, and $g{:}D' \nrightarrow Set(R')$. If (1) spec' generalizes spec, (2) θ is the substitution extracted from the proof of (1), and (3) g satisfies the specification* $\texttt{FUNCTION } G(x'{:}D'){:}\texttt{Set}(R') \texttt{ WHERE } I'[x'] \texttt{ RETURNS } \{z' \mid O'[x',z']\}$ *then the following program is correct*

$\texttt{FUNCTION } f(x{:}D){:}\texttt{Set}(R) \texttt{ WHERE } I[x] \texttt{ RETURNS } \{z \mid O[x,z]\}$
$\equiv\ \{z \in g(\theta(x)) \mid O(x,z)\}$

Reasoning by Cases is a technique which generates conditional programs if a more direct approach to solving a programming problem fails. The essential idea (Smith, 1985a) is to split the input into two or more cases which are analyzed separately and then composed again into a correct program.

THEOREM 14. *Let $spec = (D,R,I,O)$ be a specification, $g,h{:}D \nrightarrow R$, and p be a boolean function on D. If (1) g satisfies the specification $(D,R,I \wedge p,O)$*[3] *and (2) h satisfies the specification $(D,R,I \wedge \neg p,O)$ then the following program is correct*

$\texttt{FUNCTION } f(x{:}D){:}R \texttt{ WHERE } I[x] \texttt{ RETURNS } z \texttt{ SUCH THAT } O[x,z]$
$\equiv\ \texttt{if } p(x) \texttt{ then } g(x) \texttt{ else } h(x)$

[3] We use $I \wedge p$ as abbreviation for a predicate defined by $(I \wedge p)(x) = I(x) \wedge p(x)$.

There are data types D' and R', predicates O_D on $D\times D'\times D$, O' on $D'\times R'$, and O_C on $R\times R'\times R$, and a well-founded ordering relation $\succ$ on D such that

1. *Directly-solve* satisfies the specification
   FUNCTION $F_p(x{:}D){:}R$ WHERE $I[x]\wedge prim[x]$ RETURNS z SUCH THAT $O[x,z]$
2. $\forall x,x_1{:}D.\,\forall x_2{:}D'.\,\forall z_2{:}R'.\,\forall z_1,z{:}R.$
   $O_D(x,x_2,x_1)\wedge O'(x_2,z_2)\wedge O(x_1,z_1)\wedge O_C(z_2,z_1,z)\;\Rightarrow\; O(x,z)$
3. *Decompose* satisfies the specification
   FUNCTION $F_d(x{:}D){:}D'\times D$ WHERE $I[x]\wedge\neg prim[x]$
   RETURNS x_2,x_1 SUCH THAT $I'[x_2]\wedge I[x_1]\wedge x_1\succ x\wedge O_D[x,x_2,x_1]$
4. *Compose* satisfies the specification
   FUNCTION $F_c(z_2,z_1{:}R'\times R){:}R$ RETURNS z SUCH THAT $O_C(z_2,z_1,z)$
5. g satisfies the specification
   FUNCTION $F_G(x_2{:}D'){:}R'$ WHERE $I'[x_2]$ RETURNS z_2 SUCH THAT $O'(x_2,z_2)$

Figure 3. Axioms of divide & conquer

While the justification of this method is trivial, the difficulty lies in obtaining a proper predicate which makes a solution of the individual cases feasible. One way to do this is creating *derived antecedents* (Smith, 1982; Smith, 1985a) by trying to find the requirements for the validity of a given formula. This mechanism rewrites a formula (e.g. the satisfiability condition for a specification) by applying domain lemmata until it is proved or some preconditions for its validity remain. These preconditions are used as additional input-condition for one subproblem, which now is solved, while their negation yields a second subproblem, which has to be investigated further. Eventually, this results in a cascade of conditionals which solve the original problem.

Divide & Conquer

Divide & conquer is one of the most common techniques for processing recursively defined data structures. It solves a problem by dividing it into subproblems whose solutions will be computed ('conquered') independently and composed into a *single* solution for the main problem. Divide & conquer algorithms proceed by *decomposing* an input value into "smaller" inputs, which are processed recursively, and possibly other values, which are processed by some auxiliary algorithm. The resulting output values are *composed* into an output for the original problem. Input values which cannot be decomposed anymore are considered *primitive* and will be solved *directly*. Formally, the common structure of divide & conquer algorithms can be expressed as

FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS z SUCH THAT $O[x,z]$
$\equiv$ if $prim[x]$ then *Directly-solve*$[x]$ else $(Compose\circ g\times f\circ Decompose)[x]$

This abstract program contains placeholders *prim*, *Directly-solve*, *Compose*, *g*, and *Decompose* which have to be instantiated to create a concrete divide

& conquer algorithm. The analysis in (Smith, 1985b) has shown that five requirements, formalized in Figure 3, ensure the correctness of this algorithm. *Directly-solve* must compute a correct solution for primitive inputs (1). A *Strong Problem Reduction Principle* must hold (2) saying that the output condition O can be recursively decomposed into subproblems O_D, O_C, O', and O. O_D is to be solved by *Decompose* which also "reduces" the input values[4] w.r.t. some well-founded ordering $\succ$ on D (3) to ensure the termination of f. O_C is to be solved by *Compose* (4) and O' by the auxiliary function g (5).

THEOREM 15. *If D, R, I, O, prim, Directly-solve, Compose, g, and Decompose fulfill the five axioms of divide & conquer then the program*

FUNCTION f(x:D):R WHERE $I[x]$ RETURNS z SUCH THAT $O[x,z]$
$\equiv$ if *prim*$[x]$ then *Directly-solve*$[x]$ else (*Compose* $\circ\, g \times f \circ$ *Decompose*)$[x]$

is correct.

Proof. For primitive input values f computes a correct output because of axiom 1. Otherwise f is *Compose* $\circ\, g \times f \circ$ *Decompose*. Its partial correctness is shown by composing axioms 3, 4, and 5, the induction hypotheses for smaller input values, and the strong problem reduction principle. f terminates since $\succ$ is well-founded. □

Again, the formal theorem is the foundation of a schema-based synthesis strategy. This strategy has to derive the five additional components, make sure that the axioms are satisfied, and instantiate the program schema accordingly. Obviously, it cannot fulfill its task without referring to knowledge about application domains. A knowledge base has to provide *standard decomposition techniques* on input-domains (e.g. splitting sequences in two halves or into a first element and the rest), *standard well-founded orderings* (e.g. ordering lists according to their length), and *standard composition techniques* on output-domains (e.g. appending lists or prepending an element before a list). These informations support a synthesis of divide & conquer algorithms according to the following strategy.

STRATEGY 16. *Given the specification*

FUNCTION f(x:D):R WHERE $I[x]$ RETURNS z SUCH THAT $O[x,z]$

1. *Select $\succ$ and Decompose.*
2. *Construct the auxiliary function g.*
3. *Verify Decompose and derive the input condition for its correctness. Choose prim as negation of the additional input condition.*
4. *Verify the strong problem reduction principle and construct Compose.*
5. *Construct Directly-solve.*
6. *Instantiate the divide & conquer schema.*

[4] If the domain D' of g is identical to D then *both* values generated by *Decompose* must be smaller than the original input value. This allows binary recursion.

In (Smith, 1985b) this strategy is accompanied by a series of heuristics which help solving the individual steps. Step 1, which gives us D' and O_D, will look up the knowledge base. In step 2, which yields O', I', R' and axiom 5, g is usually chosen as f if $D'=D$ and as identity function otherwise. This simplifies the synthesis process since the effects of g can always be captured in the function *Compose*. Step 3, which leads to axiom 3, uses the mechanism for deriving antecedents (page 123) to determine the requirements for the correctness of *Decompose*. An admissible input value that does not fulfill these requirements must be *primitive*. In step 4 the same mechanism is used to generate O_C as precondition for Axiom 2. Constructing *Compose* is usually the most difficult step. It requires a new synthesis on the basis of axiom 4. Often *Compose* will be another divide & conquer algorithm but in simpler cases we can also use *operator match* on the basis of known library programs. Step 5 will usually try *operator match* to satisfy axiom 1.

There are several variants of this strategy. One may proceed in reverse order by selecting *Compose* first and successively constructing g, $\succ$, *Decompose*, *prim*, and *Directly-solve*. In either of the two strategies one might construct *Compose* or *Decompose* before constructing g.

Each of these strategies will lead to different algorithms since the selected operator is usually less complicated than the constructed one. When applied to the problem of sorting lists, for instance, strategy 16 eventually yields Mergesort, if *Decompose* splits lists in halves, and Insertion Sort, if it splits into first element and rest. If the reverse strategy is used, Quicksort will be generated, if *Compose* appends two lists, and Selection Sort, if *Compose* prepends elements to lists. Thus selecting the synthesis strategy and the operations in its first step is not just a strategic issue. It is a design decision which should be made by a programmer interacting with the system.

Local search

Local search is a technique that is used for dealing with optimization problems like scheduling or travelling salesman problems. It investigates a set of possible output values for a given specification. But the focus is not on *finding* them but on selecting the *best*, e.g. a schedule that can be executed in the shortest amount of time. Formally, the specification is associated with a cost function on the range and the goal is to find a solution with minimal costs.

Since optimization is often NP-complete, *Hillclimbing* algorithms like the simplex algorithm or linear programming approximate optimal solutions by exploring the local *neighborhood* of some initial solution which was easy to compute. By incremental variations the elements of this neighborhood are searched. The algorithms proceeds until a *local minimum* has been found. In (Lowry, 1988; Lowry, 1991) the general structure of these *local search*

FUNCTION $f_{opt}(x{:}D){:}R$ WHERE $I[x]$ RETURNS z
SUCH THAT $O[x,z] \wedge \forall t{:}R.\, O[x,t] \Rightarrow cost[x,z] \leq cost[x,t]$
$\equiv f_{ls}(x, f(x))$

FUNCTION $f_{ls}(x{:}D, y{:}R){:}R$ WHERE $I[x] \wedge O[x,y]$ RETURNS z
SUCH THAT $O[x,z] \wedge \forall t \in N[x,z].\, O[x,t] \Rightarrow cost[x,z] \leq cost[x,t]$
$\equiv$ if $\forall t \in N[x,y].\, O[x,t] \Rightarrow cost[x,y] \leq cost[x,t]$
then y else $f_{ls}(x, \mathtt{arb}(\{\, t \in N[x,y] \mid O[x,t] \wedge cost[x,y] > cost[x,t] \,\}))$

Figure 4. Structure of local search algorithms

algorithms has been described as a pair of programs which is presented in Figure 4. Besides placeholders D, R, I, and O for the basic specification these programs contain three additional components. $cost{:}D \times R \to \mathcal{R}$ is a function which assigns to each pair of inputs and outputs a *cost-value* in some ordered domain $(\mathcal{R}, \leq)$. It occurs both in the (extended) specification and the program. $N{:}D \times R \to \mathtt{Set}(R)$ describes the neighborhood structure by computing a set of output values which, depending on the input, are accessible from a given output. $f{:}D \nrightarrow R$ computes the initial solution.

$\forall x{:}D.\ \forall z,t{:}R.$

1. $I[x] \wedge O[x,z] \Rightarrow z \in N(x,z)$
2. $I[x] \wedge O[x,z] \wedge O[x,t] \Rightarrow \exists k{:}\mathbb{N}.\, t \in N_O^k(x,z)$
3. $I[x] \wedge O[x,z] \Rightarrow \forall t \in N(x,z).\, O[x,t] \Rightarrow cost[x,z] \leq cost[x,t]$
   $\Rightarrow \forall t{:}R.\, O[x,t] \Rightarrow cost[x,z] \leq cost[x,t]$
4. f satisfies the base specification $spec = (D, R, I, O)$

where $N_O^0(x,z) = \{z\}$ and $N_O^{k+1}(x,z) = \bigcup_{t \in \{\, t \in N(x,z) \mid O[x,t] \,\}} N^k(x,t)$

Figure 5. Axioms of local search

Four axioms, listed in Figure 5, ensure the correctness of local search algorithms. The neighborhood structure must be reflexive (1) and *connected* w.r.t. the solutions of the base specification (2): by exploring local neighborhoods all solution must be reachable from each other. If the algorithm shall be exact, local minima must be global minima (3). Finally f has to satisfy the base specification without the optimality conditions. The following theorem has been proved in (Lowry, 1988).

THEOREM 17. *If D, R, I, O, $cost$, N, and f fulfill the four axioms of local search then the pair of programs in Figure 4 is correct.*

Theorem 17 describes the requirements on local search algorithms which compute *optimal* solutions. They are met by some algorithms but usually the third axiom is dropped to allow suboptimal solutions. In this case local search

algorithms compute local minima whose quality depends solely on the neighborhood structure. Local effects will dominate if the neighborhood structure is too fine-grained and the algorithm is likely to compute a poor approximation of the global optimum. A coarse neighborhood structure, however, may lead to inefficiency since large search spaces will be explored in each step.

Finding good neighborhood structures is the only crucial aspect in a derivation of local search algorithms. The other components are either already given (*cost*) or easy to determine (f). As usual, the construction of local search algorithms depends on general knowledge. For a given range type R, for instance, there are only a few fundamental neighborhood structures. These can be represented by formal objects, called *LS-theories*, and will be refined by specialization and filters.

The strategy for designing local search algorithms is similar to the one for global search. We first select a LS-theory and specialize it to the given specification. We then modify the neighborhood structure by filters which eliminate output values which are either no solutions for the base specification (*feasibility constraints*) or more expensive than the current one (*optimality constraints*). Both constraints roughly correspond to the notion of necessary filters in global search. If exactness is required, then a filter for *global optimality constraints* can be added as well. All filters can be derived by forward inference. Finally a solution f for the base specification is synthesized and the schematic algorithm is instantiated. A detailed description of this strategy and application examples can be found in (Lowry, 1988; Lowry, 1991).

Problem Reduction Generators

Problem reduction generators deal with problems that require a set of individual solutions to be computed independently by composing several partial solutions. Typical examples are dynamic programming and branch & bound algorithms. From a theoretical point of view they can be seen as generalization of global search and divide & conquer. Structurally they are similar to divide & conquer algorithms except for the fact that now a series of composition and decomposition functions are used and that the primitive case is contained in some of them.

$$\texttt{FUNCTION } f(x{:}D){:}\texttt{Set}(R) \texttt{ WHERE } I[x] \texttt{ RETURNS } \{z \mid O[x,z]\}$$
$$\equiv \bigcup_{i,k \in \mathbb{N}} (Compose_i \circ (F_{i_1} \times .. \times F_{i_k}) \circ Decompose_i)[x]$$

The requirements for correctness of this algorithm generalize the axioms 2 to 5 of divide & conquer (Figure 3) and include a complex version of the strong problem reduction principle. The derivation strategy is similar to the one for synthesizing divide & conquer algorithms but puts a heavier burden on the inference mechanism. A detailed exposition can be found in (Smith, 1991b).

4.3. *Integration into Automated Deduction*

In an implementation of design strategies for knowledge based program synthesis, the semi-formal descriptions leave much room for interpretation. The KIDS system (Smith, 1991a), which aims at tools for semi-automatic software development, encodes these strategies directly in a high-level programming language which includes a reasoning mechanism based on formula transformations. Many steps in these strategies, however, involve solving deductive problems such as extracting substitutions from proofs of logical properties, verifying program components, or deriving the preconditions for the correctness of some formula. Standard techniques from automated deduction could be used if knowledge based program synthesis would take place in the framework of a proof system. This would also make the derivations safer since the possibility of invalid inferences, which is always present in a hand-coded synthesis system, is ruled out by the proof environment.

Integrating design strategies into a proof based system requires a very rigorous approach. Each step in a derivation must be completely formal, such that it can be controlled by the prover, but remain on the high level of abstraction which we have used so far.

Formally verified theorems stating the requirements for the correctness of an abstract program schema (Kreitz, 1996) are the key for an integration of schema-based synthesis strategies into a deductive framework. These theorems can be applied as high-level inference rules which reduce the synthesis task to the task of proving instances of the axioms. The latter can be solved by further theorem applications, knowledge base queries, first-order theorem proving, or simple inductions. The conceptually difficult problem – generating the algorithm and proving it correct – has been solved once and for all while proving the formal theorem and requires only a single derivation step.

Technically, such an integration of knowledge based program synthesis into proof systems involves a complete formalization of all the aspects involved in the derivation of programs. Standard application domains and their laws have to be represented by formal definitions and theorems in some logical calculus. Formula transformations can then be expressed as application of verified lemmata. Next, in order to reason about programs as such, we must formalize concepts like *program*, *specification*, *correctness* and related notions. Concepts like specialization, operator match, GS-theories, filters etc. need to be formalized. Concrete GS-theories, filters, etc. have to be formalized and verified. Finally, the theorems underlying the derivation strategies have to be stated and proved. This last step may require a lot of interaction but remove the proof burden from the synthesis process.

On this basis, strategies for automated program derivation can be pro-

grammed as proof tactics which apply verified theorems and lemmata and call standard proof procedures (see e.g. chapter I.1.5) to check properties like specialization, operator match, or necessity. It has been shown in (Bibel *et al.*, 1997, Section 4) that such an integrated approach, despite its formal overhead, can make program synthesis practically feasible even in the rigorous setting of proof systems.

4.4. *Discussion*

Program synthesis based on abstract algorithm schemata requires a greater amount of preparation than the techniques described in Section 3. But these efforts pay off during the synthesis: both the synthesis process and the generated programs are much more efficient. The successful synthesis of transportation scheduling algorithms reported in (Smith & Parra, 1993; Gomes *et al.*, 1996) shows that knowledge based program synthesis, if properly guided and supported by optimization techniques (Section 5.2), can produce commercially competetive software.

Furthermore, the schematic approach to program synthesis can also be used for teaching systematic programming. Students can investigate the effects of choosing different algorithmic structures as solution for the same problem and gain insights into the consequences of their own decisions. The four sorting algorithms resulting from different design decisions in a divide & conquer synthesis (cf. Section 4.2.2) are a good example for this (a fifth algorithm, Bubble Sort, can be generated via local search). Studying the formal foundations of the corresponding strategies, on the other hand, will lead to a deeper understanding of the algorithmic structures and their specific strengths and weaknesses.

Thus knowledge based synthesis has several advantages over approaches based entirely on general proof or transformation techniques. Since it can be represented in both frameworks it should not be viewed as competitor but as the next step in the evolution of automated programming which indicates a major direction for future research.

5. Related Topics

In the previous sections we have described methods for deriving well-structured algorithms from *formal* specifications. These methods represent the core of program synthesis. But systematic program development also involves methods for obtaining proper specifications and for transforming the derived algorithm into efficient program code. We shall briefly discuss both subjects.

5.1. *Acquiring Formal Specifications*

Formally specifying a given programming problem is by no means a trivial task. It requires a deep analysis of the informal requirements, the problem domain, the environment in which the program shall be executed, implicit assumptions, etc. before a system architecture can be designed and individual modules and algorithms can be specified.

Over the past there has been a substantial amount of work on developing computer support for acquiring specifications (see (Hayes, 1987; Jones & Shaw, 1990; Kelly & Nonnenmann, 1991) and (Lowry & Duran, 1989, Sections B & D)). In particular, computers are used to visualize a design or as support for a certain design methodology. *Formal methods* such as VDM (Jones, 1990) or OMT (Rumbaugh et. al., 1991) provide guidelines and visualization techniques for specifying large software sytems and are also used for communicating design decisions.

Nevertheless, acquiring formal specifications is mostly a design process that involves creativity and design decisions in order to balance between conflicting requirements. Although informal reasoning is necessary to justify the overall design of the system the deductive aspects are comparably small. Computers can only serve as support tool when making decisions but it will be difficult to go beyond this level.

5.2. *Algorithmic Optimizations*

Automatically generated algorithms are hardly the most efficient solution for a given problem. Often a human programmer immediately detects components that can be optimized. Program expressions may be *simplified* or even completely eliminated if their context, particularly the input condition, is taken into account. Expressions containing constants may be *partially evaluated*. Expensive computations may be replaced by iterative updates of a new variable. Finally, it may pay off to select a non-standard implementation of certain data structures. All these steps can be represented by *program transformations* or *rewrite techniques* and integrated into a synthesis system. The real issue is, however, to automate these transformations such that a user only has to select an expression and a particular technique for optimizing it.

Simplification transforms a program expression into an equivalent one which is easier to compute. The transformations are based on knowledge about equivalences within some application domain. These equivalences are given a *direction* which allows to apply them as long as possible. Simplification strongly depends on an efficient organization of the knowledge base which may contain thousands of equivalences.

Partial evaluation (Bjørner *et al.*, 1988) symbolically evaluates subexpressions which contain constants or other fixed structures. Logically this corresponds to unfolding definitions and simplifying the result.

Finite differencing (Paige & Koenig, 1982) replaces a subexpression by a new variable which will be updated in each recursive call of the program. It can result in a significant speedup and supports parallel computations since depth is being replaced by breadth.

Data Type Refinement (Blaine & Goldberg, 1991) selects from several implementations of an abstract data type the one that is most efficient for the program. It depends on an analysis of the operations which refer to these data types and requires automatic conversions between different representations.

The SPECWARE approach (Srinivas & Jüllig, 1995), which implements concepts from category theory for establishing relations between algebraic theories, provides an environment in which the modular construction of formal specifications, knowledge based algorithm design stategies, algorithmic optimizations, and the construction of executable code from mathematical algorithms can be integrated. A first implementation of such an integrated program synthesis system, called PLANWARE, is currently being developed.

6. CONCLUSION

Despite its long history program synthesis is still a very active research area. Over the years its focus has shifted from the fundamental questions in the field of automated deduction to the practical aspects of systematic program development. It appears that knowledge based program synthesis, coupled with automated deduction techniques, is the most promising approach since it can be made theoretically sound and practically useful at the same time. The integration of optimization strategies, techniques for adapting generic programs (like transportation scheduling algorithms or communication systems) to specific applications, and methods for acquiring and structuring domain knowledge is becoming more and more important. Furthermore, general methodologies for creating efficient and reliable implementations of synthesis systems are an important research issue.

There is a commercial interest in computer systems which can rapidly develop efficient and reliable software, whose key properties are guaranteed but not necessarily formally verified. Program synthesis systems have come close to the point where they can be used for that purpose, provided they are guided by programmers with a strong background in formal reasoning.

REFERENCES

R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, 1985.

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

W. Bibel, D. Korn, C. Kreitz, F. Kurucz, J. Otten, S. Schmitt, G. Stolpmann. A multi-level approach to program synthesis. *7th International Workshop on Logic Program Synthesis and Transformation*, LNAI, Springer, 1998.

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

D. Bjørner, A. Ershov, N. Jones, eds. *Partial evaluation and mixed computation*. North–Holland, 1988.

L. Blaine & A. Goldberg. DTRE—a semi-automatic transformation system. *IFIP TC2 Conference on Constructing Programs from Specifications*. Elsevier, 1991.

A. Bundy, F. van Harmelen, A. Ireland, A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 1992.

A. Bundy. Automatic guidance of program synthesis proofs. In *Workshop on Automating Software Design, 11th International Joint Conference on Artificial Intelligence*, pp. 57–59. Morgan Kaufman, 1989.

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

K. Clark & S. Sickel. Predicate logic: A calculus for the formal derivation of programs. *5th International Joint Conference on Artificial Intelligence*, pp. 419–420. Morgan Kaufman, 1977.

R. Constable, et. al. *Implementing Mathematics with the NuPRL proof development system*. Prentice Hall, 1986.

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

H. Curry, R. Feys, W. Craig. *Combinatory Logic*, vol. 1. North–Holland, 1958.

J. Darlington. Application of program transformation to program synthesis. *IRIA Symposium on Proving and Improving programs*, pp. 133–144, 1975.

J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(1):1–46, 1981.

N. Dershowitz. Synthesis by completion. *9th International Joint Conference on Artificial Intelligence,*, pp. 208–214. Morgan Kaufman, 1985.

R. Floyd. Assigning meaning to programs. *Symposia in Applied Mathematics*, 19:19–32, 1967.

M. Franova. A methodology for automatic programming based on the constructive matching strategy. *EUROCAL 85*, pp. 568–570. Springer, 1985.

J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

A. Goldberg. Knowledge-based programming: A survey of program design and con-

struction techniques. *IEEE Transactions on Software Engineering*, SE-12(7):752–768, 1986.

C. Gomes, D. Smith, S. Westfold. Synthesis of schedulers for planned shutdowns of power plants. *11th Conference on Knowledge-Based Software Engineering*, 1996.

C. Green. An application of theorem proving to problem solving. *1st International Joint Conference on Artificial Intelligence*, pp. 219–239. Morgan Kaufman, 1969.

I. Hayes. *Specification Case Studies*. Prentice-Hall, 1987.

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

C. Jones & R. Shaw. *Case Studies in Systematic Software Development*. Prentice-Hall, 1990.

C. Jones. *Systematic software development using VDM*. Prentice-Hall, 1990.

V. Kelly & U. Nonnenmann. Reducing the complexity of formal specification acquisition. *Automating Software Design*, pp. 41–64, AAAI/MIT Press, 1991.

A. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

C. Kreitz. *METASYNTHESIS: Deriving Programs that Develop Programs*. Thesis for Habilitation, TU Darmstadt, 1992.

C. Kreitz. Formal mathematics for verifiably correct program synthesis. *Journal of the IGPL*, 4(1):75–94, 1996.

M. Lowry & R. Duran. Knowledge-based software engineering. *Handbook of Artificial Intelligence, Vol. IV*, chapter XX. Addison Wesley, 1989.

M. Lowry & R. McCartney. *Automating Software Design*, AAAI/MIT Press, 1991.

M. Lowry. The structure and design of local search algorithms. *Workshop on Automating Software Design. AAAI-88*, pp. 88–94, 1988.

M. Lowry. Automating the design of local search algorithms. *Automating Software Design*, pp. 515–546, AAAI/MIT Press, 1991.

Z. Manna & R. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6(2):175–208, 1975.

Z. Manna & R. Waldinger. The automatic synthesis of systems of recursive programs. *5th International Joint Conference on Artificial Intelligence*, pp. 405–411. Morgan Kaufman, 1977.

Z. Manna & R. Waldinger. Synthesis: Dreams ⇒ programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979.

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

P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

B. Nordström, K. Petersson, J. Smith. *Programming in Martin-Löfs Type Theory. An introduction*. Clarendon Press, 1990.

J. Otten & C. Kreitz. A connection based proof method for intuitionistic logic. *4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 122–137. Springer, 1995.

R. Paige & S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

J. Rumbaugh et. al. *Object-oriented Modeling and Design*. Prentice Hall, 1991.

S. Schmitt & C. Kreitz. On transforming intuitionistic matrix proofs into standard-sequent proofs. *4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 106–121. Springer, 1995.

D. Smith & M. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, 1990.

D. Smith & E. Parra. Transformational approach to transportation scheduling. *8th Knowledge-Based Software Engineering Conference*, pp. 60–68, 1993.

D. Smith. Derived preconditions and their use in program synthesis. *6th Conference on Automated Deduction*, LNCS 138, pp. 172–193. Springer, 1982.

D. Smith. Reasoning by cases and the formation of conditional programs. *9th International Joint Conference on Artificial Intelligence*, pp. 215–218. Morgan Kaufman, 1985.

D. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

D. Smith. On the design of generate-and-test algorithms: Subspace generators. *IFIP TC2 Conference on Program Specification and Transformation*, pp. 207–220. North–Holland, 1987.

D. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, November 1987.

D. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

D. Smith. KIDS — a knowledge-based software development system. *Automating Software Design*, pp. 483–514, AAAI/MIT Press, 1991.

D. Smith. Structure and design of problem reduction generators. *IFIP TC2 Conference on Constructing Programs from Specifications*, pp. 91–124. Elsevier, 1991.

Y. Srinivas & R. Jüllig. SPECWARE: Formal Support for composing software. *Conference on the Mathematics of Program Construction*, 1995.

D. Steier & A. Anderson. *Algorithm Synthesis: A comparative study*. Springer, 1989.

W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

R. Waldinger & R. Lee. PROW: A step toward automatic program writing. *1st International Joint Conference on Artificial Intelligence*, pp. 241–252. Morgan Kaufman, 1969.

R. Waldinger. Constructing programs automatically using theorem proving. PhD Thesis, Carnegie-Mellon University, 1969.