Achievements and Prospects of Program Synthesis

Pierre Flener

Information Technology, Department of Computing Science
Uppsala University, Box 337, S – 751 05 Uppsala, Sweden
http://www.csd.uu.se/∼pierref/ pierref@csd.uu.se

Abstract. Program synthesis research aims at developing a program that develops correct programs from specifications, with as much or as little interaction as the specifier wants. I overview the main achievements in deploying logic for program synthesis. I also outline the prospects of such research, arguing that, while the technology scales up from toy programs to real-life software and to commercially viable tools, computational logic will continue to be a driving force behind this progress.

1 Introduction

In his seminal book *Logic for Problem Solving* [53], Bob Kowalski introduced the celebrated equation:

$$Algorithm = Logic + Control \qquad (A = L + C)$$

expressing that for an algorithm, the statement of *what* it does — the logic component — can be separated from the manner *how* it is done — the control component. Algorithms and programs in conventional languages feature a merging of these components, whereas pure logic programs only express the logic component, leaving the control component to the execution mechanism. In actual logic programming languages, such as PROLOG, some control directives can be provided as annotations by the programmer. The logic component states only the problem-specific part of an algorithm and determines only its correctness, while the control component only expresses a problem-independent execution strategy and determines only the efficiency of the algorithm.

Kowalski listed several advantages of this encapsulation, which is akin to the abstraction achieved when separating the algorithm and data-structure components of programs. These advantages include the following:

- The logic and control components of algorithms can be successively refined, and improved, independently of each other.
- A default, and thus often sub-optimal, control can be provided for less experienced programmers, who can thus focus their efforts on the logic component.

– The logic component of an algorithm can be mechanically generated from, and verified against, a formal specification, using deduction, without considering the control component. Similarly, the logic component can be mechanically transformed into another one, using deduction, without considering the control component. One thus obtains what is known as program *synthesis*, program *verification*, and program *transformation*, respectively.

The objective of this chapter — whose title is by the way subsumed by the one of Kowalski's book — is to overview the main achievements in deploying logic for program synthesis, and to outline its future prospects. As synthesis nowadays starts scaling up from toy programs to real-life software and to commercially viable tools, it can be argued that computational logic will continue to be a driving force behind these developments.

Scope of this Chapter. In contrast to Kowalski's intention, I here do not focus on the synthesis of logic programs only, but rather take a wider approach and tackle the synthesis of any kinds of programs. Indeed, the target language does not really matter, but what does matter is the use of computational logic in the synthesis process. Similarly, I shall not restrict myself to his advocated use of deductive inference for synthesis, but will also discuss the role of inductive, abductive, and analogical inference in synthesis.

Also, although there is a large overlap in concepts, notations, and techniques between program synthesis and program transformation, verification, and analysis (which is the study of the semantics and properties of programs, such as their termination), I here discuss concepts and techniques relevant to program synthesis only — assuming it can be clearly delineated from those other areas — and refer the reader to the prolific literature on these related research fields.

Having thus both widened and narrowed the scope of this chapter compared to Kowalski's original agenda, the literature to be overviewed is very voluminous and thus cannot possibly be discussed in such a single, short chapter. I have thus made a maybe subjective selection of the landmark research in program synthesis, with particular attention to seminal work and to approaches that scale up for eventual deployment in actual software development. For coverage of more approaches, I thus refer the interested reader to the numerous overviews, surveys, and paper collections periodically published before this one, such as those — in chronological order — by Barr & Feigenbaum [3], Biermann *et al.* [14, 15, 12, 13], Partsch *et al.* [73, 72], Smith [79], Balzer [2], IEEE TSE [70], Goldberg [41], Rich & Waters [74, 75], Feather [30], Lowry *et al.* [60, 61], Steier & Anderson [87], JSC [16], Deville & Lau [27], and Flener [34, 37].

Organisation of this Chapter. The rest of this chapter is organised as follows. In Section 2, I describe my viewpoint on what program synthesis actually is, and what it is not, especially in relation to other areas, such as compilation and transformation. Classification criteria are also given. The technical core of this chapter are Sections 3 to 5, where I overview past achievements of logic-based program synthesis.[1] I devote one section each to the three main streams of research, namely transformational (Section 3), constructive (Section 4), and mixed-inference (Section 5) synthesis, exhibiting one or two representative systems for each of them, in terms of their underlying machineries, their actual synthesis processes, and interesting excerpts of sample syntheses. From this sketch of the state-of-the-art, I can then outline, in Section 6, the future prospects of program synthesis, whether logic-based or not, especially in terms of the challenges it faces towards scaling up and eventual transfer of the technology to commercial software development. Finally, in Section 7, I conclude.

2 What *Is* Program Synthesis?

I now describe my viewpoint on what program synthesis actually is, and what it is not. In Section 2.1, I state the objective and rationale of program synthesis, and contrast it with program transformation. Next, in Section 2.2, I propose a classification scheme for synthesisers. Finally, in Section 2.3, I show that the goalposts of synthesis have been moving very much over the years, and that synthesis is in retrospect nothing else but compilation.

2.1 The Goal of Program Synthesis

The grand objective of *program synthesis* — also known as *automatic programming* — research is to develop a program that develops correct programs from specifications, with as much or as little interaction as the specifier wants. Nothing in this formulation is meant to imply that the focus is on programming-in-the-small. Synthesising real-life software only requires a scalable synthesis process. Just like manual programming, synthesis is thus about translating a statement from one language into *another* language, namely from the specification language into the programming language, thereby switching from a statement of *what* the program does and how it should be used to a statement of *how* the program does it, hence ideally not only establishing correctness (the program outputs satisfy the post-condition of the specification, provided the inputs meet its pre-condition) but also achieving a reasonable level of efficiency (outputs are computed within a reasonable amount of time and space).

The rationale for this objective is the notorious difficulty for most programmers of effectively developing correct and efficient programs, even when these programs are small. The benefits of a synthesiser would be higher-quality programs and the disappearance of the program validation and maintenance steps, and instead total focus on specification elaboration, validation, and maintenance, because replay of program development would become less costly. Synthesis would be especially useful in problem domains where there is a huge gap between

[1] Citations are not necessarily to the first paper on a specific approach, but to comprehensive papers that may have been published much later. In the latter case, I indicate the year of the original paper in the running text.

the end-user formulation of a problem and an efficient program for solving it, such as for constraint satisfaction problems, for instance.

The hope for synthesisers is as old as computing science itself, but it is often dismissed as a dream. Indeed, we are way off a fully automatic, general-purpose, end-user-oriented synthesiser [75], and pursuing one may well be illusory. Most of the early synthesis projects aimed at starting from informal specifications. For instance, the SAFE project [2] initially went to great efforts to do so, but eventually switched to defining GIST, a very-high-level formal language for conveying formal descriptions of specifications. Nowadays, as a simplification, virtually all synthesisers start from inputs in such formal languages. Another typical simplification through division of work is to focus on the synthesis of the logic component of programs, leaving the design of their data-structure and control components to others. In this chapter, I focus on approaches to logic-based synthesis that embody both of these usual simplifications.

A few words need to be said about the relationship between synthesis and transformation. Whereas program synthesis is here defined as the translation of a statement from a possibly informal specification description language into a program in a necessarily formal programming language, with focus on correctness, *program transformation* is here defined as the equivalence-preserving modification of a program into another program of the *same* language, with focus on achieving greater efficiency, in time or space or both. This makes transformation different from synthesis in purpose, but complementary with it. In practice, they share many concepts and techniques. Optimising transformation can be achieved by changing any of the logic, control, or data-structure components of programs. This raises many interesting issues:

– One can argue that synthesis and transformation should not be a sequence of two separate but complementary tasks, because the correctness and efficiency of algorithms are inevitably intertwined, even if separated in logic and control components. But this division of work is appealing and has been useful.
– If only the text of a program enters transformation, then the rationale of its synthesis steps is lost to the transformation and may have to be rediscovered, in a costly way, in order to perform effective transformation. I am not aware of any transformation approaches that take programming rationale as input.
– In Kowalski's words [53]: "Changing the logic component is a useful short-term strategy, since the representation of the problem is generally easier to change than the problem-solver. Changing the control component, on the other hand, is a better long-term solution, since improving the problem-solver improves its performance for many different problems." A good example of the effect of suitably changing control is the switch from logic programming to constraint logic programming, thereby giving programs with a generate-and-test logic component an often spectacular speedup. Such paradigm shifts may well require a redefinition of what synthesis and transformation are.

No matter which way the purposes of synthesis and transformation are defined, there is an unclear boundary between them, made even more confusing by other considerations, examined in Section 2.3.

2.2 Classification Criteria

A huge variety of synthesis mechanisms exist, so I here propose a multi-dimensional classification scheme for them. The criteria fall into three major categories, grouping the attributes of the synthesis inputs, mechanisms, and outputs.

Synthesis Inputs. The input to synthesis is a *specification* of the informal requirements. Sometimes, a *domain theory* stating the laws of the application domain must also be provided. These inputs have the following attributes:

- **Formality**. An input to synthesis can be written in either an *informal* language (whose syntax or semantics is not predefined), or a *formal* language (whose syntax and semantics are predefined). The often encountered notion of *semi-formal* language is strictly speaking meaningless: controlled natural languages are formal, and UML and the likes are informal even though their graphical parts may have a formal syntax and semantics.
- **Language**. When using a formal input language, a specification can be either *axioms*, or input/output *examples.* Sometimes, the actual language is disguised by a suitable graphical user interface, or it is sugared.
- **Correctness wrt the Requirements**. Informally, a statement S is *correct* wrt another statement T iff S is *consistent* with T (everything that follows from S also follows from T) as well as *complete* wrt T (everything that follows from T also follows from S). Input to synthesis is usually *assumed to be consistent* with the requirements. On the other hand, the input is either *assumed to be complete* or *declared to be incomplete* wrt the requirements. In the former case, the synthesiser need only produce a program that is correct wrt the input. In the latter case, the synthesiser must try to extrapolate the actual complete requirements from the given input. In either case, actual validation against the informal requirements is done by the programmer, by changing the inputs to synthesis until the synthesised program has the desired behaviour. As opposed to the *external* consistency and completeness considered here, *internal* consistency and completeness are not classification attributes, but rather quality criteria that may be mechanically checked before synthesis begins: a statement S is *internally consistent* iff S has at least one model, and *internally complete* iff every symbol in S is either primitive to the language used or defined within S.

Synthesis Mechanisms. The mechanisms of program synthesis can also be classified along a few dimensions:

- **Level of Automation**. Having by definition excluded manual programming, synthesis is either *semi-automatic* or *fully automatic.*
- **Initiative**. In semi-automatic synthesis, the initiative in the interaction can be on either side, making the mechanism *synthesiser-guided* or *user-guided.*
- **Kinds of Inference**. There are many kinds of inference and they can all be used, and combined, towards synthesis. I here distinguish between *purely-deductive* synthesis, which performs only deductive inference and is either

transformational (see Section 3) or *constructive* (see Section 4), and *mixed-inference* synthesis, which features any appropriate mix of deductive, inductive, abductive, and analogical inference (see Section 5).

- **Kinds of Knowledge**. There is a great need for incorporating knowledge into program synthesisers. There are essentially four kinds of useful synthesis knowledge, namely knowledge about the mechanics of *algorithm design*, knowledge about the laws and refinement of *data structures*, knowledge about the laws of the *application domain* (this was called the domain theory above), and *meta-knowledge*, that is knowledge about how and when to use the other kinds of knowledge.
- **Determinism**. A *non-deterministic* synthesiser can generate a family of programs from a specification; otherwise, it is a *deterministic* synthesiser.
- **Soundness**. Synthesis should be a *sound* process, in the sense that it produces an output that is guaranteed to satisfy some pre-determined notion of correctness wrt the input.

Synthesis Outputs. The output of synthesis is a *program*, and usually only the logic component of its algorithm. The classification attribute is:

- **Language**. Technically, the synthesised program can be in any language, because any code can be generated from the chosen internal representation. In practice, the pure parts of the so-called declarative languages are usually chosen as internal and external representation of programs, because they are the highest-level languages compiled today and thus sufficient to make the point. Common target languages thus are *Horn clauses*, *recursion equations*, λ-*expressions*, etc.

These classification attributes are not independent: choices made for one of them affect the available choices for the others.

2.3 The Moving Goalposts of Program Synthesis

The first assemblers and compilers were seen as automatic programming systems, as they relieved the programmers from many of the burdens of binary programming. Ever since, program synthesis research has been trying to be one step ahead of the state-of-the-art in programming languages, but, in retrospect, it is nothing else but the quest for new programming paradigms. To paraphrase Tesler's sentence, which was originally on Artificial Intelligence: *Program synthesis deals with whatever has not been compiled yet.* Of course, as our notion of program evolves, our understanding of compilation has to evolve as well: it is not because today's compilers are largely deterministic and automatic that tomorrow's compilers, that is today's synthesisers, are not allowed to have search spaces or to be semi-automatic.

The main problem with formal inputs to program synthesis is that there is no way to construct them so that we have a formal proof that they capture our informal requirements. In fact, the phrase 'formal specification' is a contradiction in terms, as real specifications can only be informal [57]. An informal correctness proof is needed *somewhere*, as the purpose of software engineering is after all to obtain programs that implement our informal requirements. Writing such formal inputs just shifts the obligation of performing an informal proof from the program-vs-informal-requirements verification to the formal-inputs-vs-informal-requirements verification, but it does *not* eliminate that obligation.

In my opinion, programs and such formal inputs to synthesis are intrinsically the same thing. As synthesis research aims at raising the level of language in which we can interact with the computer, compilation and synthesis are intrinsically the same process. In other words, real programming and synthesis are only being done when going from informal requirements to a formal description, which is then submitted to a compiler. In this sense, focusing synthesis on starting from formal statements is not really a simplification, as claimed above, but rather a redefinition of the task, making it identical to compilation.

I am *not* saying that formal methods are useless. Of course it is important to be able to check whether a formal description is *internally* consistent and complete, and to generate prototypes from executable descriptions, because all this allows early error detection. But one cannot say that such formal descriptions are specifications, and one still knows nothing about whether they are *externally* consistent and complete, namely wrt the informal requirements. Formal inputs to program synthesis are already programs, though not in a conventional sense. But conventions change in time, and the so-called “formal specifications” of today will be perceived as programs tomorrow.

In order to stick to the contemporary terminology and make this chapter independent of agreement or disagreement on this sub-section, I shall nevertheless speak of formal specifications (without the quotes) in the following.

3 Achievements of Transformational Synthesis

In *transformational synthesis*, meaning-preserving transformation rules are applied to the specification, until a program is obtained. Usually, this is done within a so-called wide-spectrum language — such as B, GIST, VDM, Z — containing both non-executable specification constructs and executable programming constructs. I shall use the word ‘description’ to designate the software representations in such a language, be they formal specifications, programs, or hybrids in-between these two extremes.

Given a logic specification of the following form, where there is no prejudice about which parameters are inputs and which ones are outputs, at run-time:

$$\forall P \,.\, pre(P) \rightarrow (\ p(P) \leftrightarrow post(P)\ )$$

where *pre* is the pre-condition (an assertion on all the parameters P, assumed to hold when execution of a program for p starts), *post* is the post-condition (an assertion on the parameters P, to be established after execution of a program for p), and p is the specified predicate symbol, transformational synthesis iterates over a single step, namely the application of a transformation rule to some expression within the current description, until a program is obtained.

Transformation rules, or *transforms*, are often represented as rewrite rules with pattern variables:

$$IP \Rightarrow OP \quad [\text{ if } C\ ]$$

expressing that under the optional applicability condition C, an expression matching input pattern IP under some substitution θ may be replaced by the instance $OP\theta$ of the output pattern OP.

Transforms are either *refinements*, reducing the abstraction level of the current description by replacing a specification construct by a program construct, or *optimisations*, performing a simplification (reduction in expression size) or a reduction in runtime or space, both at the same abstraction level. Refinements can act on statements or datatype definitions, reducing non-determinism.

A sample refinement is the following unconditional transform of a high-level non-recursive array summation into a recursive expression:

$$\begin{array}{ll} & S = \sum_{i=l}^{u} A[i] \\ \Rightarrow & \\ & \Sigma(A,l,u,S) \leftarrow l > u, S = 0 \quad \%\ \Sigma(A,l,u,S) \text{ iff } S \text{ is the sum of } A[l]..A[u] \\ & \Sigma(A,l,u,S) \leftarrow \neg\ l > u, +(l,1,l'), \Sigma(A,l',u,T), +(A[l],T,S) \end{array}$$

Sample optimisations are the following conditional transform for divisions:

$$x/x \Rightarrow 1 \quad \text{if } x \neq 0$$

and the following accumulator introduction, which amounts to replacing recursion in the non-minimal case of a divide (d) and conquer (c) definition of predicate p by tail-recursion — with the minimal (m) case being solved (s) without recursion — as this can be compiled into more efficient code, like iteration:

$$\begin{array}{ll} & p(X,Y) \leftarrow m(X), s(X,Y) \\ & p(X,Y) \leftarrow \neg m(X), d(X,H,T), p(T,V), c(H,V,Y) \\ \Rightarrow & \\ & p(X,Y) \leftarrow p(X,Y,I) \\ & p(X,Y,A) \leftarrow m(X), s(X,J), c(A,J,Y) \\ & p(X,Y,A) \leftarrow \neg m(X), d(X,H,T), c(A,H,A'), p(T,Y,A') \\ \text{if} & \mathit{associative}(c) \wedge \mathit{identity}(c, \mathit{left}, I) \end{array}$$

The latter transform is applicable to the output of the refinement above, because $+/3$ is associative and has a left-identity element, namely 0. This illustrates how transforms can be chained. Of course, the refinement above could immediately have reflected such a chaining.

Other common transforms are *unfolding* (replacing a symbol by its definition), *folding* (the inverse of unfolding), *definition* (introduction of a new symbol via its definition), *instantiation* (application of a substitution), *abstraction* (introduction of a *where* clause, in functional programming), or reflect the laws of the application domain.

Several control issues arise in the rewrite cycle, because the synthesis search space is usually intractable due to the sheer number of transforms. First, who checks the applicability condition? Usually, this is considered a synthesiser responsibility, and thus becomes a task for an automatic theorem proving component thereof. Second, which transform should be applied next, and to which expression? Usually, full automation is abandoned in favour of user-guided interactive application of transforms, with the synthesiser automatically ensuring that applicability conditions are met, as well as correctly applying the chosen transform to the chosen expression, thus taking over all clerical work. Other approaches are based on rule ordering, heuristics, agendas, planning, replay, etc. Third, when to stop transforming? Indeed, many transforms can also be applied during program transformation (as defined in Section 2.1), hence blurring the transition and distinction between synthesis and transformation. Usually, one considers that synthesis *per se* has finished when the current description is entirely within the executable part of the wide-spectrum language, so that synthesis is here defined as the translation from the full wide-spectrum language into its executable subset.

When transforms are too fine-grained, they lead to very tedious and lengthy syntheses. The idea is thus to define macroscopic transforms that are higher-level in the sense that they are closer to actual programming decisions and that they are compositions of such atomic transforms. Examples are *finite differencing* (replacing expensive computations in a loop by incremental ones), *loop fusion* (merging of nested or sequentially-composed loops into one loop), *partial evaluation* (simplifying expressions for fixed arguments), *generalisation* (solving a more general, easier problem), *dynamic programming*, *memoing* (caching results of computations to avoid useless recomputations), *jittering* (preparing the application of other transforms).

To document a synthesis and ease its understanding, the applied sequence of transforms is usually recorded, ideally with the rationale of their usage. This also allows *replay*, though it remains unclear when this is suitable and when not.

I now discuss an entire product-line of representative transformational synthesisers, chosen because of the objective of scaling the technology to real-life software development tasks. Indeed, KIDS and its successors (see Section 3.1) have been successfully deployed in many real-life applications. In Section 3.2, I outline the efforts of the other research centres in transformational synthesis.

3.1 SpecWare, DesignWare, and PlanWare

At Kestrel Institute (Palo Alto, California, USA, www.kestrel.edu), Smith and his team have been designing, for over 15 years now, a series of synthesisers, all with the same philosophy, which is specific to them (see below). Their *Kestrel Interactive Development System* (KIDS) [81] extends its predecessor CYPRESS [80] and automatically synthesises correct programs within the wide-spectrum language REFINE, while leaving their transformation to a user-guided rewrite cycle. I here describe the systems of their product-line — SpecWare (for *Specification Ware*) [86], DesignWare [84], and PlanWare [18] — as well as how they relate to each other. They amount to more than just recasting, as described in [83], the synthesis and transformation calculus of KIDS in category theory.

The overall Kestrel philosophy is as follows. Consider, for instance, programs that solve constraint satisfaction problems (CSPs) by exploring the entire candidate-solution space, though with pruning of useless subspaces. They have a common structure, called global search, of which the dataflow, control-flow, and interactions between parts can be formally captured in a *program schema*. Similarly, other program schemas can be designed for capturing the methodologies leading to local search programs, divide-and-conquer programs, etc. Such program schemas can then be used in synthesis to significantly reduce the candidate-program space. Some proof obligations arise in such *schema-guided synthesis*, but they are feasible by state-of-the-art automated theorem provers. The synthesised programs are not very efficient, though, since they are just problem-specific instances of program schemas that had been designed for entire problem families, but without being able to take into account the specificities of their individual problems. The synthesised programs can thus be transformed into equivalent but more efficient ones by applying high-level transforms, in a user-guided way. However, this transformation cycle also became the bottleneck of KIDS, because the user really has to be an expert in applying these transforms in a suitable order and to the appropriate sub-expressions. Moreover, the proof obligations of synthesis are only automatable if the entire application domain knowledge is formally captured, which is an often daunting task. Smith used KIDS to rather quickly refine new, breakthrough algorithms for various CSPs [82].

The inputs to synthesis are a formal axiomatic higher-order algebraic specification, assumed to be consistent and complete wrt the requirements, and a domain theory. The synthesis mechanism is purely deductive, interactive or automatic (depending on the system), non-deterministic, and sound. Algorithm design, data structure, and application domain knowledge are exploited. The output is a program in any supported language (e.g., CommonLisp, C++).

The Transformation System. A category-theory approach to transformation is taken. Viewing specifications as finite presentations of theories, which are the closures of the specification axioms under the rules of inference, a *specification morphism* $S \rightarrow S'$ is a provability-preserving signature morphism between specifications S and S', that is a map between their sort and operator symbols, such that axioms translate into theorems.[2]

For instance, consider the specification of finite containers in Figure 1. It is parameterised on the sort E of the container elements. Containers are either empty, or singletons, or constructed by an infix binary *join* operator.

Also consider the following specification of binary operators:

```
spec $BinOp$ is
  sort $T$
  op _ $bop$ _ :  $T, T \longrightarrow T$
end
```

[2] For typographic reasons, the '$\rightarrow$' symbol is thus overloaded, being used for both morphisms and logical implication. The distinction should always be clear from context. Under its morphism meaning, this symbol will be typeset here in other directions of the wind rose, to facilitate the representation of graphs of morphisms.

spec $Container$ is
  sorts $E, Cont$
  op $empty$: $\longrightarrow Cont$
  op $singleton$: $E \longrightarrow Cont$
  op $\_ \; join \; \_$: $Cont, Cont \longrightarrow Cont$
  $\ldots$ other operator declarations $\ldots$
  ops $\{empty, singleton, join\}$ construct $Cont$
  axiom $\forall X : Cont \, . \; X \; join \; empty = X$
  axiom $\forall X : Cont \, . \; empty \; join \; X = X$
  $\ldots$ axioms for the other operators $\ldots$
end

Fig. 1. A specification of finite containers

spec $ProtoSeq$ is
  sorts E, Seq
  op $empty$: $\longrightarrow Seq$
  op $singleton$: $E \longrightarrow Seq$
  op $\_ \; join \; \_$: $Seq, Seq \longrightarrow Seq$
  $\ldots$ other operator declarations $\ldots$
  ops $\{empty, singleton, join\}$ construct Seq
  axiom $\forall X : Seq \, . \; X \; join \; empty = X$
  axiom $\forall X : Seq \, . \; empty \; join \; X = X$
  axiom $\forall X, Y, Z : T \, . \; (X \; join \; Y) \; join \; Z = X \; join \; (Y \; join \; Z)$
  $\ldots$ axioms for the other operators $\ldots$
end

Fig. 2. A specification of finite sequences

The following specification of associative operators reflects the specification morphism $BinOp \rightarrow Associative$, which is $\{T \mapsto T, bop \mapsto bop\}$:

spec $Associative$ is
  import $BinOp$
  axiom $\forall X, Y, Z : T \, . \; (X \; bop \; Y) \; bop \; Z = X \; bop \; (Y \; bop \; Z)$
end

Specifications and specification morphisms form a category, called $SPEC$, in which push-outs can be computed. Informally, a *diagram* is a directed graph with specifications as vertices and specification morphisms as arcs.

For instance, the push-out of $Associative \leftarrow BinOp \rightarrow Container$ under morphisms $\{T \mapsto T, \; bop \mapsto bop\}$ and $\{T \mapsto E, \; bop \mapsto join\}$ is isomorphic to the specification of prototype finite sequences in Figure 2. Indeed, sequences are containers whose $join$ operation is associative. By another morphism, sequence-specific operators can be added to $ProtoSeq$, giving rise to a specification $Sequence$ of finite sequences. By another push-out $Commutative \leftarrow BinOp \rightarrow ProtoSeq$, we can get a specification $ProtoBag$ of prototype finite bags, to which bag-specific operators can be added, giving rise to a specification

$$\begin{tikzcd}
BinOp \arrow[r] \arrow[d] & Container \arrow[d] & \\
Associative \arrow[r] & ProtoSeq \arrow[d] & BinOp \arrow[l] \arrow[d] \\
BinOp \arrow[r] \arrow[d] & ProtoBag \arrow[d] & Commutative \arrow[l] \\
Idempotent \arrow[r] & ProtoSet &
\end{tikzcd}$$

$$\begin{tikzcd}
 & Container \arrow[d] \\
 & ProtoSeq \arrow[dl] \arrow[d] \\
Sequence & ProtoBag \arrow[dl] \arrow[d] \\
Bag & ProtoSet \arrow[dl] \\
Set &
\end{tikzcd}$$

Fig. 3. A chain of commuting diagrams (left) and a taxonomy of containers (right)

Bag of finite bags. Indeed, bags are sequences whose join operation is commutative, because element order is irrelevant. Finally, by yet another push-out $Idempotent \leftarrow BinOp \rightarrow ProtoBag$, we can obtain a specification *ProtoSet* of prototype finite sets, to which set-specific operators can be added, giving rise to a specification *Set* of finite sets. Indeed, sets are bags whose join operation is idempotent, because multiplicity of elements is irrelevant. This process can be captured in the chain of three commuting diagrams of the left of Figure 3. If we graphically add the considered additional morphisms to the central vertical chain, we obtain the *taxonomy* of containers in the right of Figure 3.

A *diagram morphism* $D \Rightarrow D'$ is a set of specification morphisms between the specifications of diagrams D and D' such that certain squares commute. It serves to preserve and extend the structure of specifications, as opposed to flattening them out via co-limits. For instance, a not shown diagram morphism $BAG \Rightarrow BAGasSEQ$ can be created to capture the refinement of bags into sequences, where *BAG* and *BAGasSEQ* are diagrams involving specifications *Bag* and *Sequence*, respectively. Diagrams and diagram morphisms also form a category, in which co-limits can be computed, using the co-limits in *SPEC*. The word 'specification' here denotes either a specification or a specification diagram, and 'refinement' refers to a diagram morphism, unless otherwise noted.

In general now, specifications — as theory representations — can capture domain models (e.g., transportation), abstract datatypes (e.g., *BAG*), software requirements (e.g., crew scheduling), algorithm theories (e.g., divide-and-conquer), etc. Tool support and a large library of reusable specifications are provided for structuring and composing new specifications. Also, specification morphisms and diagram morphisms can capture specification structuring (e.g., via imports), specification refinement (e.g., scheduling to transportation-scheduling), algorithm design (e.g., global-search to scheduling), datatype refinement (e.g., $BAG \Rightarrow BAGasSEQ$), expression optimisation (e.g., finite differencing), etc. Again, tool support is provided for creating new refinements, and a large library of useful refinements exists.

Finally, *inter-logic morphisms* are provided for translating from the specification logic into the logic of a programming language — thereby performing code generation — or of a theorem-prover or any other supporting tool.

$$\begin{tikzcd}
A \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathit{Spec}_0 \arrow[d, Rightarrow] & \\
B \arrow[r, Rightarrow] & \mathit{Spec}_1 \arrow[d, Rightarrow] & C \arrow[l, Rightarrow] \arrow[d, Rightarrow] \\
E \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathit{Spec}_2 \arrow[d, Rightarrow] & D \arrow[l, Rightarrow] \\
\cdots & \cdots \arrow[d, Rightarrow] & \cdots \arrow[d, Rightarrow] \\
 & \mathit{Spec}_n \arrow[d] & Z \arrow[l, Rightarrow] \\
 & \mathit{Code} &
\end{tikzcd}$$

Fig. 4. The synthesis process

The Synthesis Process. The refinement of a specification Spec_0 is an iterative process of calculating push-outs in commuting squares, yielding new specifications Spec_i, until the process is deemed finished and an inter-logic morphism is used to generate a program Code from the final specification Spec_n. This process is depicted in Figure 4. Here, $A \Rightarrow B$, $C \Rightarrow D$, etc, are refinements stored in a library. With push-outs being calculated automatically, the creative steps are the selection of a refinement and the construction of a *classification arrow* [83, 84] between the source diagram (A, C, etc) of a library refinement and the current specification. The leverage can be quite dramatic, with push-outs often generating many new lines, which might have been quite cumbersome, if not difficult, to write by hand.

As the size and complexity of specification and refinement libraries increase, support must be given for this approach to scale up. First, specification libraries are organised in taxonomies, such as Figure 3 above, so as to allow the incremental construction of classification arrows [84]. For instance, to apply the $\mathit{BAG} \Rightarrow \mathit{BAGasSEQ}$ refinement to the current specification S, one can first classify S as a $\mathit{Container}$, then as a $\mathit{ProtoSeq}$, next as a $\mathit{ProtoBag}$, then as a Bag, and finally as a BAG, rather than classifying S as a BAG in one go. The deeper one goes into a taxonomy, the more specification information can be exploited and the more efficient the resulting code. Second, as patterns of useful classification and refinement sequences emerge, parameterised macros, called *tactics*, can be defined to provide higher-level, if not more automatic, operations to the user. For instance, the divide-and-conquer algorithm theory admits two classification tactics, depending on whether the decomposition or the composition operator is manually selected from a library, and thus reused, in a classification step, leaving the other operator to be inferred.

SpecWare [86] is an abstract machine exporting high-level synthesis and transformation primitives that hide their low-level implementation in terms of category theory operations. Using it, one can more quickly write new synthesisers. First, a new version of kids was implemented, called DesignWare [84], extending SpecWare with domain-independent taxonomies of software design theories plus support for refining specifications using the latter. Then, on top of DesignWare, the PlanWare [18] domain-specific synthesiser of high-performance schedulers was developed. Both its synthesis and transformation processes are fully automatic, and it even automatically generates the formal specification and application domain knowledge — which are typically thousands of lines — from the information provided by the specifier, who uses a very intuitive domain-specific spreadsheet-like interface, without being aware of the underlying category theory. PlanWare extends DesignWare with libraries of design theories and refinements about scheduling, together with a specialised tactic for controlling the application of this design knowledge. Other domain-specific synthesisers are in preparation, and will also be built on top of DesignWare.

A Sample Synthesis. A synthesis of a function *sorting* that sorts bags into sequences may start from the following specification:

spec $Sorting$ is
  import $BagSeqOverLinOrd$
  op $sorted: \ Bag, Seq \longrightarrow Boolean$
  def $sorted(X, Y) = ord(Y) \wedge seqToBag(Y) = X$
  op $sorting: \ Bag \longrightarrow Seq$
  axiom $sorted(X, sorting(X))$
end

where *sorted* is used to express the post-condition on *sorting*. Universal quantification consistent with the signature declarations is assumed for unquantified variables. Suppose the specifier wants to apply a divide-and-conquer algorithm design, as embodied in the refinement $DivConq \Rightarrow DivConqScheme$, where the source specification is in Figure 5. Here, a function F from domain D into range R is specified, with post-condition O. Three mutually exclusive predicates p_i (for $i = 0..2$) are defined over D, representing conditions for the existence of decompositions, computed under post-conditions O_{Di} (for $i = 0..2$), with O_{D2} enforcing that its decompositions are smaller than the given term, under well-founded relation $\prec$. Soundness axioms require that the decompositions can be composed, under post-conditions O_{Ci} (for $i = 0..2$), to achieve the overall post-condition O. The target specification of the refinement is in Figure 6. where a schematic definition of the specified function F is introduced, together with composition operators C_i whose post-conditions are O_{Ci}.

Now, to apply the $DivConq \Rightarrow DivConqScheme$ refinement, a classification arrow $Sorting \Rightarrow DivConq$ has to be manually constructed, so that the corresponding push-out can be automatically calculated. The first part of the necessary diagram morphism is straightforward, namely $\{D \mapsto Bag,\ R \mapsto Seq,\ F \mapsto sorting,\ O \mapsto sorted,\ \ \prec\ \ \mapsto subBag,\ \ldots\}$. The remaining part gives rise to dual alternatives, which can be captured in tactics, as discussed above: either a set of simple standard decomposition operators is reused from a library and the corresponding complex composition operators are inferred, or a set of simple standard composition operators is reused and the corresponding complex decomposition operators are inferred. Following the first approach, the bag constructor set $\{emptyBag, singletonBag, bagUnion\}$ could be reused as the ba-

```
spec $DivConq$ is
  sorts $D, R, E, Unit$
  op $F :\ D \longrightarrow R$
  op $O :\ D, R \longrightarrow Boolean$
  op $\_ \prec \_ :\ D, D \longrightarrow Boolean$
  axiom $wellFounded(\prec)$
  op $p_0, p_1, p_2 :\ D \longrightarrow Boolean$
  op $O_{D0} :\ D, Unit \longrightarrow Boolean$
  op $O_{D1} :\ D, E \longrightarrow Boolean$
  op $O_{D2} :\ D, D, D \longrightarrow Boolean$
  op $O_{C0} :\ R, Unit \longrightarrow Boolean$
  op $O_{C1} :\ R, E \longrightarrow Boolean$
  op $O_{C2} :\ R, R, R \longrightarrow Boolean$
  axiom $p_0(X) \to O_{D0}(X, \langle\rangle)$
  axiom $p_1(X) \to \exists M : E \,.\, O_{D1}(X, M)$
  axiom $p_2(X) \to \exists X_1, X_2 : D \,.\, O_{D2}(X, X_1, X_2) \wedge X_1 \prec X \wedge X_2 \prec X$
  axiom $O_{D0}(X, \langle\rangle) \wedge O_{C0}(Y, \langle\rangle) \to O(X, Y)$
  axiom $O_{D1}(X, M) \wedge O_{C1}(Y, M) \to O(X, Y)$
  axiom $O_{D2}(X, X_1, X_2) \wedge O(X_1, Y_1) \wedge O(X_2, Y_2) \wedge O_{C2}(Y, Y_1, Y_2) \to O(X, Y)$
  axiom $p_0(X)\ xor\ p_1(X)\ xor\ p_2(X)$
end
```

Fig. 5. Specification of problems that have divide-and-conquer programs

sis for decomposition, giving rise to $\{\ldots,\ p_0 \mapsto emptyBag?,\ O_{D0} \mapsto \lambda X \,.\, X = emptyBag,\ p_1 \mapsto singletonBag?,\ O_{D1} \mapsto \lambda X, M \,.\, X = singletonBag(M),\ p_2 \mapsto nonSingletonBag?,\ O_{D2} \mapsto \lambda X, X_1, X_2 \,.\, X = bagUnion(X_1, X_2),\ \ldots\}$. By deductive inference, the remaining part of the morphism can be obtained, yielding translations to empty sequence construction, singleton sequence construction, and sequence merging for O_{C0}, O_{C1}, and O_{C2}, respectively, ultimately leading thus to a merge-sort algorithm. Under the second approach, the sequence constructor set $\{emptySeq, singletonSeq, seqConcat\}$ could be reused as the basis for composition, ultimately leading to a quick-sort algorithm.

Either way, after calculating the push-out, synthesis could continue by using the $BAG \Rightarrow BAGasSEQ$ datatype refinement, followed by simplification refinements, etc, progressively bringing the specification closer to a programming level, until a code-generating inter-logic morphism for translating the definition of F into a functional program can be applied.

3.2 Other Schools

Transformational synthesis is by far the dominant approach to program synthesis, and many dozens of projects have been devoted to it, so I can here only mention the seminal and dominant ones.

At the University of Edinburgh (UK), Burstall & Darlington [22,25] proposed a small, fixed set of domain-independent, low-granularity, and rather

```
spec $DivConqScheme$ is
 import $DivConq$
 op $C_0 :\ \longrightarrow R$
 axiom $O_{C0}(C_0, \langle\rangle)$
 op $C_1 :\ E \longrightarrow R$
 axiom $O_{C1}(C_1(M), M)$
 op $C_2 :\ R, R \longrightarrow R$
 axiom $O_{C2}(C_2(X_1, X_2), X_1, X_2)$
 definition of $F$ is
  axiom $p_0(X) \rightarrow O_{D0}(X, \langle\rangle) \wedge F(X) = C_0$
  axiom $p_1(X) \rightarrow \exists M : E \,.\, O_{D1}(X, M) \wedge F(X) = C_1(M)$
  axiom $p_2(X) \rightarrow \exists X_1, X_2 : D \,.\, O_{D2}(X, X_1, X_2) \wedge F(X) = C_2(F(X_1), F(X_2))$
 end
 theorem $O(X, F(X))$
end
```

Fig. 6. Specification of divide-and-conquer programs

optimisation-oriented transforms (namely folding, unfolding, definition, instantiation, and abstraction) for the synthesis and transformation of recursion equations. Laws of the application domain can also be used. They presented a strategy and a semi-automated system for transforming recursive equations, say into tail-recursive ones, with the user making the creative decisions. For synthesis, the objective of applying such transforms often is to construct, through unfolding and other rewriting, a description where recursion may be introduced through folding. The atomic transforms are proven to constitute a correct set for exploring the candidate program space.

At Stanford University (California, USA), at the same time, but independently, Manna & Waldinger [63] discovered the same atomic rules and automatically synthesised LISP programs with their *DEDuctive ALgorithm Ur-Synthesiser* (DEDALUS). The system has over 100 rules, and also generates correctness and termination proofs. See Section 4.1 for a detailed discussion of a redesign of DEDALUS as a constructive synthesiser.

In the UK, much of the early efforts on the synthesis of logic programs were conducted, based on the foundational fold/unfold work mentioned above. Under a first approach, Clark *et al.* [23] execute the specification with symbolic values that cover all possible forms of the type of the chosen induction parameter. For instance, if that parameter is a list, then the empty and non-empty lists are considered. A similar approach was taken by Hogger [49], though with slight differences. Induction on some parameter was only introduced as the need arises. A highly structured top-down strategy for applying folding and unfolding, guided by a recursion schema provided by the specifier, as well as the notion of specification framework for synthesis, were proposed by Lau *et al.* [55,56]. This approach is amenable to mechanisation. Specification frameworks enabled a first-order logic reconstruction of KIDS-like schema-guided synthesis [36,35,38].

Several researchers tried to make synthesis a deterministic process, akin to compilation. For instance, implication formulas with arbitrary bodies may be normalised into normal clauses by the Lloyd-Topor translation [59]. However, this does not always yield useful logic programs, due to the deficiencies of SLDNF resolution, such as floundering. Also, the obtained programs are sometimes hopelessly inefficient. Overcoming these flaws is the objective of program transformation. Another approach was taken by Sato & Tamaki's first-order compiler [77], whose synthesis of partially correct definite programs is fully automatic and deterministic, but may fail, for lack of logical power.

At TU Munich and TU Darmstadt (Germany), Bibel leads synthesis projects since 1974. Their LOPS (*LOgical Program Synthesis*) system [8–10], although presented as being a constructive synthesiser, was actually transformational. Synthesis consisted of a four-phased application of heuristics that control special transformations. A novel feature is the breaking of inputs into parts so as to discover in what way they contribute to the construction of the outputs; in this way, loops can be discovered without the need for recursively-expressed background axioms, which would be essentially identical to the synthesised programs. The current MAPS project [11] takes a multi-level approach to synthesis, and is essentially a re-implementation of KIDS within NuPRL, but without optimising transformations yet.

At Stanford University (California, USA), the PSI project led by Green [45] included the transformational engine PECOS [4], which is based on a large, fixed catalog of domain-specific transforms. Cooperation with an efficiency expert, called LIBRA [52], ensured efficient synthesis of efficient programs. A successor system, called CHI [46], was partly developed at Kestrel Institute.

At the University of Southern California (USA), the 15-year-project SAFE/TI (*Specification Acquisition From Experts*, and *Transformational Implementation*) headed by Balzer [2] provided a fixed catalog of domain-specific transforms for refining specifications within the wide-spectrum language GIST, via a knowledge-based approach. Automation issues were tackled by the GLITTER sub-system [31].

At TU Munich (Germany), the long-term CIP (*Computer-aided Intuition-guided Programming*) project of Bauer and co-workers [6, 72] led, since 1975, to the wide-spectrum algebraic specification language CIP-L and the interactive environment CIP-S. The main emphasis was on a user-extensible catalog of transforms, starting from a small set of generative rules.

The *Vienna Development Method* (VDM) by Bjørner & Jones [17] is an ISO-standardised comprehensive software development methodology, proceeding by refinement from formal specifications of abstract datatypes in the META-IV wide-spectrum language. Many tools are available, from different sources, but they are not integrated. See www.csr.ncl.ac.uk/vdm for more details.

From Oxford University (UK) comes Z [85], a very successful and soon-to-be-ISO-standardised notation for formal specifications, based on set theory. There is third-party tool support, though not integrated, on top of the HOL theorem prover. Award-winning applications include the IBM CICS project and a specification of the IEEE standard for floating-point arithmetic. See www.afm.sbu.ac.uk/z.

The B formal method was developed by Abrial [1]. A first-order logic specification language with sets is provided to specify and refine systems that are modelled as abstract machines. Tool support for refinement and discharging many of its proof obligations exists. See www.afm.sbu.ac.uk/b.

At the University of California at San Diego (USA), the OBJ language family of Goguen and his team [40] provides wide-spectrum algebraic languages, based on order-sorted equational logic, possibly enriched with other logics. Tool support for refinement exists. See www.cs.ucsd.edu/users/goguen/sys/obj.html.

At the Universities of Edinburgh (UK) and Warsaw (Poland), Sannella & Tarlecki [78] propose EXTENDEDML as a wide-spectrum language for specification and formal development of STANDARDML programs, through refinement. See www.dcs.ed.ac.uk/home/dts/eml.

4 Achievements of Constructive Synthesis

Constructive synthesis — also known as *proofs-as-programs synthesis*, and, a bit misleadingly, as *deductive synthesis* — is based on the *Curry-Howard isomorphism* [50], which says that there is a one-to-one relationship between a constructive proof [7, 68] of an existence theorem and a program that computes witnesses of the existentially quantified variables of the theorem. Indeed, the use of induction in proofs corresponds to the use of recursive or iterative composition in programs, while case analysis corresponds to a conditional composition, and lemma invocation to a procedure call.

Assume given a logic specification of the following form:

$$\forall X \,.\, \exists Y \,.\, pre(X) \rightarrow post(X, Y) \tag{1}$$

where *pre* is the pre-condition (an assertion on the input parameters X, assumed to hold when execution of the program starts), and *post* is the post-condition (an assertion on X and the output parameters Y, to be established after execution of the program). Note that this specification form naturally leads to the synthesis of total functions, but not of relations. A solution to this is to view relations as functions into Booleans [20]. Constructive synthesis proceeds in two steps:

1. Constructively *prove* the satisfiability of the specification.
2. *Obtain* the procedure, embodied in the proof, of realising the specification.

For the second step, there are two approaches:

– The *interpretative* approach directly interprets the proof as a program, by means of an operational semantics defined on proofs.
– The *extractive* approach mechanically extracts — or: compiles — a program, in a given target language, from the proof.

The two approaches have complementary advantages and drawbacks: interpretation is not as efficient as the execution of a compiled version, but the choice of a target language might obscure computational properties of proofs.

The idea of exploiting constructive proofs as programs is actually way older than its naming as the Curry-Howard isomorphism in 1980: the idea is inherent to intuitionistic logic — see the work of Kleene in the 1940s — and the oldest synthesisers of this approach are QA3 (*Question-Answering system*) by Green [44], and ProW (*PROgram Writer*) by Waldinger & Lee [90], both from the late 1960s. The terminology 'proofs-as-programs' seems to have been coined by Constable in the early 1970s, according to [5].

The bottleneck is of course the state-of-the-art in automated theorem proving (ATP). In essence, the hard problem of synthesis has been translated into the other hard — if not harder! — problem of ATP. The proof space for most conjectures is indeed intractable, and formal specifications tend to be quite complex conjectures. Solutions are thus being worked out to control the navigation through this search space, namely synthesisers with reuse, interactive provers, tactical provers, etc.

I here discuss two representative constructive synthesisers, chosen due to their interesting relationship to each other. Indeed, AMPHION (see Section 4.2) can be seen as an outgrowth of DEDALUS (see Section 4.1), with the objective of scaling the technology to real-life software development tasks, and this was the decisive criterion in my selection. In Section 4.3, I outline the efforts of the other main research centres in constructive synthesis.

4.1 DEDALUS

The *DEDuctive ALgorithm Ur-Synthesiser* (DEDALUS) system of Manna & Waldinger (at Stanford and SRI, California, USA) was originally developed as a transformational synthesiser [63] (see Section 3.2), and then re-designed within the proofs-as-programs paradigm, in a considerably more elegant manner [64, 67].

The inputs to synthesis are a formal axiomatic first-order logic specification, assumed to be consistent and complete wrt the requirements, as well as a domain theory. The synthesis mechanism is purely deductive and fully automatable, but an interactive interface with user guidance exists. Only application domain knowledge is exploited. Synthesis is non-deterministic and sound. The outputs of synthesis are a side-effect-free applicative program, as well as implicitly a proof of its correctness and termination.

The Proof System. Constructive logics are not necessarily required for *all* of a constructive synthesis. Indeed, many derivation steps during synthesis actually are only verification steps, and need thus not be constructive at all. Classical logic is thus sufficient, provided it is sufficiently constructive when needed.

Their deductive tableau proof system was developed especially for proofs-as-program synthesis. A *deductive tableau* is a two-dimensional structure, where each row is a *sentence* of the form $\langle a, -, o\rangle$ or $\langle -, g, o\rangle$, where a is an *assertion* and g a *goal*, both in classical first-order logic, while o is an optional *output term* in LISP. The symbol '$-$' denotes the absence of an entry in that column, and is equivalent to *true* for assertions, $false$ for goals, and any new variable for output terms. For simplicity, I assume there is only one output parameter in specifications. For instance,

$$\langle -, M \in S \wedge (\forall X \,.\, X \in S \rightarrow M \leq X), M \rangle$$

is a sentence capturing a pre-condition-free specification of the $minimum(S)$ function, which returns the minimum element M of integer-set S.

The semantics of a sentence $\langle a, g, o \rangle$, in an interpretation $\mathcal{I}$, is the set of closed terms t that, for some substitution θ, are equal to instance $o\theta$ of the output term, if any, and either the instance $a\theta$ of the assertion, if any, is closed and false or the instance $g\theta$ of the goal, if any, is closed and true, in $\mathcal{I}$.

The semantics of a tableau is the union of the semantics of its sentences. There is thus an implicit conjunction between the assertions of a tableau, and an implicit disjunction between its goals. Note the dual role of assertions and goals: a formula can be transferred between the assertions and goals columns by negating it. Nevertheless, the distinction between assertions and goals provides intuitive and strategic power, and is thus kept.

A set of deduction rules is provided to add new sentences to a tableau, not necessarily in an equivalent way, but at least preserving the set of *computable expressions* (which are quantifier-free expressions in terms of the basic functions of the theory, plus the functions for which programs have already been synthesised, including the function for which a program is currently being synthesised, as this enables recursion formation). Hence the program denoted by a tableau remains unchanged through application of these rules. Each user-provided new rule needs to be first proven *sound* according to this precept.

A *deduction rule* has a set of *required sentences* in the old tableau, representing the applicability condition of the rule, and a set of *generated sentences* in the new tableau, representing the difference between the old and new tableaus.

For instance, the *if-split* rule breaks required sentence $\langle -, \text{if } a \text{ then } g, t \rangle$ into the generated sentences $\langle a, -, t \rangle$ and $\langle -, g, t \rangle$. There are dual splitting rules.

Conditional output terms are normally introduced by four non-clausal resolution rules, reflecting case analysis in informal proofs. For instance, the *goal-goal resolution* rule is as follows:

$$\frac{\langle -, g_1[p], s \rangle \qquad \langle -, g_2[q], t \rangle}{\langle -,\ g_1\theta[false] \ \wedge\ g_2\theta[true],\ \text{if } p\theta \text{ then } t\theta \text{ else } s\theta \rangle} \qquad (GG)$$

where, assuming the required sentences are standardised apart, θ is the most-general unifier for formulas p and q. See below for an example. Similarly, there are the dual *assertion-assertion* (AA), *goal-assertion* (GA), and *assertion-goal* (AG) resolution rules.

There are also rules for *equivalence* (replacing a formula by an equivalent one), theory-independent *equality* (replacing a term by an equal one, using a non-clausal version of paramodulation), *skolemisation* (eliminating existential quantifiers), and *well-founded induction* (allowing formation of terminating recursion in the output term, when the induction hypothesis is actually used).

The Synthesis Process. Synthesis goes as follows, starting from a specification of the form (1), for a function f, in a theory $\mathcal{T}$:

1. Form the initial tableau, with the sentence $\langle -, pre(X) \to post(X,Y), Y \rangle$ built from the specification, and assertion-only sentences for the axioms of $\mathcal{T}$. Add f to the set of functions of $\mathcal{T}$ and those already synthesised in $\mathcal{T}$.
2. Apply deduction rules to add new sentences to the tableau.
3. Stop with the final tableau when a sentence of the form $\langle false, -, t \rangle$ or $\langle -, true, t \rangle$ appears, where t is a computable expression.

The extracted program then is the function definition $f(X) = t[X]$. It is *correct* wrt specification (1) in the sense that the formula $\forall X \,.\, pre(X) \to post(X, f(X))$ is valid in theory $\mathcal{T}$ augmented with the axiom $\forall X \,.\, f(X) = t[X]$. The program is also guaranteed to terminate.

Equivalence-preserving simplification of sentences is automatically performed, as a terminating rewrite process, before synthesis starts and after application of any deduction rule. There are theory-independent logical simplifications, such as replacing formula $a \wedge a$ by a, and theory-specific simplifications, such as replacing integer expression $n+0$ by n.

The resolution rules have a symmetric nature. For instance, applying the AG rule to an assertion a and a goal g could be replaced by applying the GA rule to g and a. However, typically, one of the two symmetric applications will not advance the proof. The *polarity search control strategy* (not explained here) tries to prevent such unsuitable applications of the resolution rules, and always does so without lengthening the proof nor compromising the completion of the proof.

Two issues around recursion formation deserve discussion. First, there are mechanisms for constructing new well-founded relations (wfr) from old ones, for use in application of the induction rule. However, this makes the wfr search space rather large, and, worse, it is usually difficult to choose in advance the most suitable wfr, which only becomes apparent several steps later. To overcome this, *middle-out reasoning* (originally explored in [48, 54]) is performed, here replacing the required wfr by a variable, so as to wait until its desired properties become apparent. Second, there is a *recurrence search control strategy* that tries to match goals and sub-goals so as to form recursion.

Specification-based reuse of existing programs within a theory $\mathcal{T}$ — such as, but not exclusively, already synthesised programs — becomes possible through the addition of formulas of the form $\forall X \,.\, pre(X) \to post(X, f(X))$ to the axioms of $\mathcal{T}$, when starting a new synthesis.

Finally, it is worth stating that the deduction rules are powerful enough to also perform program transformation.

A Sample Synthesis. Rather than showing a full synthesis for a toy function, where the final program is virtually identical to the specification or to some of the necessary axioms in the theory, I decided to exhibit an interesting passage from a more difficult synthesis [66], highlighting the power of the resolution rules.

Consider the specification of a function returning the square-root R of a non-negative rational number N, within a positive rational tolerance ϵ:

$$\epsilon > 0 \to R^2 \leq N \wedge N < (R+\epsilon)^2$$

within a theory $\mathcal{R}$ for non-negative rationals, including addition $(+)$, squaring (x^2), inequalities $(<, >, \leq, \geq)$, etc.

Suppose synthesis leads to a tableau with the following sentence, after an *if-split* in the initial sentence built from the specification, and after application of the equivalence rule $a < b \leftrightarrow \neg(b \leq a)$:

$$\langle -, R^2 \leq N \wedge \neg \boxed{(R+\epsilon)^2 \leq N}, R \rangle \tag{2}$$

Let us apply resolution rule (GG) to this sentence and the following standardised-apart copy of itself:

$$\langle -, \boxed{S^2 \leq N} \wedge \neg[(S+\epsilon)^2 \leq N], S \rangle$$

The boxed sub-goals unify under most-general substitution $\{S/R+\epsilon\}$, so the generated sentence is:

$$\begin{array}{l} \langle -, \\ R^2 \leq N \wedge \neg \mathit{false} \wedge \mathit{true} \wedge \neg[((R+\epsilon)+\epsilon)^2 \leq N], \\ \text{if } (R+\epsilon)^2 \leq N \text{ then } R+\epsilon \text{ else } R \rangle \end{array}$$

which is automatically simplified into:

$$\langle -, R^2 \leq N \wedge \neg[(R+2\epsilon)^2 \leq N], \text{if } (R+\epsilon)^2 \leq N \text{ then } R+\epsilon \text{ else } R \rangle \tag{3}$$

Whereas (2) expresses that the square-root of N is in the half-open interval $[R..R+\epsilon[$, in which case R is a suitable output, sentence (3) expresses that the square-root of N is in the wider half-open interval $[R..R+2\epsilon[$, in which case conditional term 'if $(R+\epsilon)^2 \leq N$ then $R+\epsilon$ else R' is a suitable output. Noting that $R+\epsilon$ is the midpoint of that wider interval, sentence (3) simply says that if a square-root is known to be in wide interval $[R..R+2\epsilon[$, then it is the first element of either its right half or its left half. In other words, sentence (3) provides an idea for a binary search program, whereas sentence (2) does not. This is very interesting, as this discovery can thus be made mechanically, by a simple application of a resolution rule.

Using DEDALUS, rather intricate programs were synthesised, such as unification [65], as well as interesting new ones [66].

4.2 AMPHION

AMPHION [88] (ase.arc.nasa.gov/docs/amphion.html) was developed by Lowry and his team at NASA Ames and SRI (California, USA). It is of particular interest due to its attention to real-life software engineering considerations, and because it is actually deployed at NASA JPL.

The inputs to synthesis are a formal axiomatic first-order logic specification, assumed to be consistent and complete wrt the requirements, as well as a domain theory. The novelty is that specifications can be conveyed through a menu-driven, domain-independent graphical user-interface. The synthesis mechanism is purely deductive, fully automatic, non-deterministic (though there is no practical difference between alternate programs), and sound. Only application domain knowledge is exploited. The output of synthesis is a side-effect-free applicative program, which can be automatically translated into any other currently supported language (e.g., FORTRAN-77).

The Proof System. The proof system of AMPHION is essentially the deductive tableau system of DEDALUS (see Section 4.1). The automated theorem prover SNARK (*SRI's New Automated Reasoning Kit*) of Stickel and his colleagues was chosen to carry out the proofs. Its initial lack of an induction rule was unproblematic, as discussed below.

The Synthesis Process. AMPHION is domain-independent, but was first deployed in the domain of interplanetary mission planning and data analysis. An axiomatic theory, called NAIF, was formalised for this domain, comprising basic properties of solar-system astronomy as well as formal specifications of the reusable routines of a solar-system kinematics library, developed in FORTRAN-77 at NASA JPL. Synthesised programs in the resulting AMPHION/NAIF are therefore compiled into FORTRAN-77. The options in the graphical user-interface for capturing specifications also depend on the provided domain theory.

Library routines are often difficult to reuse, because of the time needed to master their sheer number, if not because of inadequate specifications, and because competent library consultants may be in short supply. Reluctant or careless programmers may thus well duplicate functionality in the library, thereby losing time and being at the risk of errors. Automated support for correct reuse and composition of library routines would thus come in very handy. But this is precisely what a DEDALUS-like system such as AMPHION can achieve, because reuse is supported, as we have seen in the previous section. Synthesis need thus not bottom out in the primitives of the target language.

Another practical insight concerns the choice of the composition mechanisms — such as conditions and recursion — used during synthesis. Although constructive synthesis can generate them all, recursion formation is by far the most difficult composition. If sufficiently many library routines performing sophisticated calculations are provided, then synthesis need not really "lift" recursion from them but may rather amount to generating an adequate straight-line program — with just sequential and conditional composition — from the specification. AMPHION was designed to synthesise only straight-line code, on the assumption that not too sophisticated proofs would be performed in theories with a large number of axioms. Synthesis is then not bottlenecked by recursion formation.

The synthesised programs can be optimised using the transforms of KIDS (see Section 3.1). Heuristic considerations need to be dealt with when finetuning the domain theory. For instance, a suitable recursive-path ordering and a suitable agenda-ordering function have to be supplied. Also, heuristics, such as the set-of-support strategy, may turn out very beneficial to the prover.

METAAMPHION [62] is a synthesiser synthesiser (*sic*) assisting domain experts in the creation and maintenance of a new instance of AMPHION, starting from a domain theory, and this without requiring any substantial training in deductive inference. This is done by applying AMPHION at the meta-level.

A Sample Synthesis. Considering the scale of synthesis tasks that can be handled by AMPHION, I can here only point to the two on-line sample syntheses at ase.arc.nasa.gov/docs/amphion-naif.html. One of them computes the solar incidence angle at the point on Jupiter pointed to by a camera on the Galileo sonde. A NAIF expert could construct such a program within half an hour, but may not be available to do so. However, after a one-hour tutorial, non-programmer planetary scientists can specify such problems within a few minutes, and synthesis of a correct program usually takes less than three minutes. The synthesised programs are indeed mostly straight-line code, which would however have been quite hard to program for non NAIF-experts.

Other results are the Saturn viewer, developed for use during the time Saturn's ring plane crossed the Earth, or an animation visualising Saturn and its moon Titan as seen from the Cassini sonde on its fly-by, with stars in the background. The latter helped planetary scientists evaluate whether proposed tours of Cassini could satisfy their observational requirements.

4.3 Other Schools

A large number of additional constructive synthesis projects exist, so I can here only skim over the most seminal and important ones.

At Cornell University (New York, USA), Constable and his group designed the PRL [5] and NuPrl [24] interactive proof and functional program development systems, the latter being based on the intuitionistic second-order type theory of Martin-Löf [68].

At the University of Edinburgh (UK), NuPrl was used for the synthesis of deterministic logic programs by Bundy and his team [19]. A first-order subset of the OYSTER proof development system, which is a re-implementation of NuPrl in PROLOG, was also used for logic program synthesis, with special focus on the synthesis of programs that compute relations, and not just total functions. A proof-planner called CLAM was adjoined to OYSTER [21], making it a tactical prover, using Edinburgh LCF [42], which is based on Scott's Logic for Computable Functions. The overall effort also resulted in the WHELK proof development system [91], which performs proofs in the Gentzen sequent calculus and extracts logic programs, the PERIWINKLE synthesiser [54], which systematises the use of middle-out reasoning in logic program synthesis, and many other systems, as the group spawns around the world.

At Uppsala University (Sweden), the logic programming calculus of Tärnlund [89], based on Prawitz' natural deduction system for intuitionistic logic, provided an elegant unified framework for logic program synthesis, verification, transformation, and execution. His team showed how to extract logic programs from constructive proofs performed within this calculus [47], and synthesised a unification algorithm [29], among others.

The INRIA (France) group uses Coquand & Huet's calculus of inductive constructions (COQ), and the Chalmers (Sweden) group exploits Martin-Löf's type theory, both towards the synthesis of functional programs. Their results are compiled in [71, 51], for instance.

5 Achievements of Mixed-Inference Synthesis

Considering that human programmers rarely resort to only safe reasoning — such as deductive inference — it would be unwise to focus all synthesis research on only deduction-based mechanisms. Indeed, a growing importance needs to be given to so-called unsafe reasoning — such as inductive, abductive, or analogical inference — if we want synthesis to cope with the full range of human software development activities.

I here discuss one representative mixed-inference synthesiser, namely Multi-tac (see Section 5.1), which performs both deductive and inductive inference. In Section 5.2, I outline the efforts of the other main research centres in mixed-inference synthesis.

5.1 Multi-tac

Multi-tac, the *Multi-Tactic Analytic Compiler* [69] of Minton, who was then at NASA Ames (California, USA), automatically synthesises efficient problem-specific solvers for *constraint satisfaction problems* (CSPs), such that they perform on par with solvers hand-written by competent programmers. While the ability of human experts remains elusive, the results are very encouraging, and popular general-purpose solvers are almost systematically outperformed.

This is so because there is no universally best solver for all CSPs, and, worse, that there is not even a best solver for all instances of a given CSP. Today, the programming of an efficient solver for any instance of some CSP is still considered a black art. Indeed, a CSP *solver* essentially consists of three components, namely a *search algorithm* (such as backtracking search, with or without forward checking), *constraint propagation and pruning rules* (based on consistency techniques, such as node and arc consistency), as well as *variable and value ordering heuristics* (such as most-constrained-variable-first or least-constraining-value-first), with each of these components having a lot of recognised problem-independent incarnations, each of which usually has many problem-specific instantiations. The right combination of components for a given instance of a CSP lies thus in a huge solver space, often at an unintuitive place, and human programmers rarely have the inclination or patience to experiment with many alternatives. On the premise that synthesis time does not matter, say because the synthesised program will be run many times for different instances, Multi-tac undertakes a more systematic exploration of this solver space.

The inputs to synthesis are a formal first-order sorted logic specification of a CSP, assumed to be consistent and complete wrt the requirements, as well as a set of training instances (or an instance generator) reflecting the distribution — in terms of the number of domain variables and the number of constraints between them — of instances on which the resulting solver will normally be run. In

```
procedure $solve(FreeVars)$ :
begin
  if $FreeVars = \emptyset$ then return the solution;
  $Var \leftarrow bestVar(FreeVars, \mathbf{VarOrdRules})$;
  $FreeVars \leftarrow FreeVars - \{Var\}$;
  $PossVals \leftarrow possVals(Var, \mathbf{PruneRules})$;
  while $PossVals \neq \emptyset$ do begin
    $Val \leftarrow bestVal(Var, PossVals, \mathbf{ValOrdRules})$;
    $PossVals \leftarrow PossVals - \{Val\}$;
    if **fwdChecking** $= true$ or **Constraints** on $Var$ are satisfied by $Val$
    then begin
      $assign(Var, Val)$;
      if **fwdChecking** $= true$ then $updatePossVals(FreeVars, \mathbf{Constraints})$;
      if $solve(FreeVars)$ then return the solution;
      if **fwdChecking** $= true$ then $restorePossVals(FreeVars)$;
      $prune(Var, PossVals, \mathbf{PruneRules})$
    end;
  end;
  $unassign(Var, Val)$;
  fail
end
```

Fig. 7. Schema for backtracking search

the following, I only mention training instances, abstracting thus whether they are given by the user or generated by the given instance generator. The synthesis mechanism is mixed-inference, performing both inductive and deductive inference, and is fully automatic. Algorithm design and data structure knowledge are exploited. Synthesis is non-deterministic and sound. The output of synthesis is a solver in LISP that is finetuned not only for the problem at hand, but also for the given instance distribution.

The Operationalisation System. MULTI-TAC is a schema-guided synthesiser, with a *schema* being a syntactic program template showing how some search algorithm can be parameterised by the other components of a CSP solver. For instance, the *backtracking schema* for backtracking search is approximately as in Figure 7, with the place-holders typeset in boldface. A full discussion of this schema is beyond the scope of this paper, the important issues being as follows. At each iteration, a chosen "best" value is assigned to a chosen "best" variable, with backtracking occurring when this is impossible without violating some constraint. Also, the template is generic in the constraints, the variable and value ordering rules, the pruning rules, and a flag controlling the use of forward checking. Many well-known variations of backtracking search fit this schema. Branch-and-bound and iterative-repair schemas are also available.

The cornerstone of synthesis is the problem-specific instantiation of the rules of the chosen schema. This is done by *operationalisation* of generic heuristics into rules, as described next. For instance, in problems where a subset of the edges of a given graph is sought, the *most-constrained-variable-first* variable-ordering heuristic — stating that the variable with the fewest possible values left should be chosen next — could be operationalised into at least the following rules:

– Choose the edge with the most adjacent edges.
– Choose the edge with the most adjacent edges whose presence in or absence from the sought subset has already been decided.
– Choose the edge with the most adjacent edges whose absence from the sought subset has already been decided.

Operationalisation is thus non-deterministic. The obtained candidate rules have different application costs in terms of evaluation time and different effectiveness in terms of how much the search is reduced, so a trade-off analysis is needed (see *configuration search* below).

Multi-tac features two methods for operationalisation of generic heuristics, as described next.

Analytic operationalisation is based only on the problem constraints and ignores the training instances. Each heuristic is described by a meta-level theory that enables the system to reason about the problem constraints. For instance, the meta-theory of the most-constrained-variable-first heuristic describes circumstances where some variable is likely to be more constrained than another one. A good example thereof is that the tightness of the generic constraint $\forall X : S \,.\, P(X) \rightarrow Q(X)$ is directly related to the cardinality of the set $\{X : S \mid P(X)\}$. From such algorithm design knowledge, candidate search control rules can be inferred.

Inductive operationalisation is based mainly on the training instances, though also uses the problem constraints. Brute-force simplest-first inductive inference is achieved through a generate-and-test algorithm. First, all rules expressible within a given grammar — based on the vocabulary of the problem constraints — are generated, starting with the shortest, that is simplest, rules, until a predetermined upper bound on the number of atoms in the rule is reached, or until a predetermined time bound is reached. The number of rules generated grows exponentially with the size bound, but fortunately the most useful rules tend to be relatively short. The testing step weeds out all the generated rules that do not well approximate the desired effects of the generic heuristics. Towards this, positive and negative examples are inferred from the training instances, and all rules that are more often correct than incorrect on these examples are retained. This is a surprisingly effective criterion.

The analytic method may fail to generate useful short rules, but can infer longer rules. The inductive method often finds excellent short rules, but cannot infer longer rules or may accidentally eliminate a good rule due to the statistical nature of its testing process. The two methods are thus complementary and should be used together to increase the robustness of the system.

The Synthesis Process. Once the generic heuristics have been somehow operationalised into candidate rules, a process called *configuration search* looks for a suitable selection of these rules and for suitable flag values, such that, if plugged into the schema with the problem-specific constraints, they interact nearly optimally in solving instances of the given CSP that fit the given distribution.

Since the space of such possible configurations of rules and flags is exponential in the number of rules and flags, a beam search (a form of parallel hill-climbing) is performed over only a small portion of that space. Given a beam width b, a time bound t, and the training instances, one starts from the single parent configuration that has no rules and where all flags are turned off. At each iteration, child configurations are generated from all parent configurations, by adding one rule from the candidate rules or by activating one flag. Several candidate rules may be retained for a given place-holder in the schema, if this is found to be advantageous; they are then sequenced, so that each rule acts as a tie-breaker for its predecessors. The b configurations that solve the most instances within t seconds enter the next iteration as parent configurations, provided they solve a superset of their own parents' instances. This process continues until no parent configuration can be improved or until the user interrupts it.

Operationalisation and configuration search are able to discover rules for many well-known heuristics from the literature, for each search algorithm.

Once the rules and flags of the chosen schema are instantiated — in a problem-specific and instance-distribution-specific way thus — through operationalisation and configuration search, synthesis proceeds by automatically optimising the winning configuration through refinements (including the choice of adequate data structures), formula simplifications, partial evaluation, and code simplifications (including finite differencing).

A Sample Synthesis. Consider the *Minimum-Maximum-Matching* (MMM) problem: given an integer K and a graph with vertex set V and edge set E, determine whether there is a subset $E' \subseteq E$ with $|E'| \leq K$ such that no two edges in E' share a vertex and every edge in $E - E'$ shares a vertex with some edge in E'. This is an NP-complete problem and can be modelled for Multi-tac as follows, representing E' as a set of $m(I, B)$ atoms, where Boolean B is t when edge I of E is in E', and f otherwise:

$$\begin{array}{l}
\forall V, E : \mathit{set}(\mathit{term}) \,.\, \forall K : \mathit{int} \,.\;\; \mathit{mmm}(\langle V, E\rangle, K) \leftrightarrow \\
\forall I : E \,.\, m(I,t) \rightarrow (\forall W : V \,.\, \forall J : E \,.\, I \neq J \wedge e(I,W) \wedge e(J,W) \rightarrow m(J,f)) \\
\wedge\ \forall I : E \,.\, m(I,f) \rightarrow (\exists W : V \,.\, \exists J : E \,.\, I \neq J \wedge e(I,W) \wedge e(J,W) \wedge m(J,t)) \\
\wedge\ \mathit{cardinality}(\{I : E \mid m(I,t)\}) \leq K
\end{array}$$

where problem instances are assumed given through a set of $e(I, W)$ atoms, stating that edge I has vertex W as one of its two endpoints.

In the first constraint, there are two sub-expressions matching the generic expression $\forall X : S \,.\, P(X) \rightarrow Q(X)$ mentioned for analytic operationalisation, namely the two formulas starting with the universal quantifications on W and J, respectively. From the former, the variable-ordering rule 'Choose the edge with the most endpoints' is inferred, though it is useless, as *every* edge has exactly two endpoints; from the latter, the already mentioned rule 'Choose the edge with the most adjacent edges' is inferred. All variable-ordering rules mentioned above can also be generated by inductive operationalisation.

In three well-documented experiments [69] with different instance distributions for the MMM problem, the solvers synthesised by Multi-tac outperformed at least one of two written by competent human programmers, while totally outclassing general-purpose Boolean satisfiability algorithms and CSP solvers, under their default heuristics. Interesting rules were discovered, and Multi-tac won by the largest margin on the toughest instance distribution, confirming that massive automated search does often better than human intuition.

5.2 Other Schools

The exclusive use of *inductive* and *abductive* inference in program synthesis, from incomplete specifications, has been studied under two angles, for three decades.

First, in *programming-by-example* (PBE), also and more adequately known as *programming-by-demonstration* (PBD), the specifier provides sample execution traces of the task to be programmed, and the synthesiser generalises them into a program that can re-enact at least these traces. The user thus has to know how to perform the specified task, but there are interesting applications for this, such as the synthesis of macro operations for word processors or operating systems. See [58] for a collection of state-of-the-art papers, especially geared at enabling children and other novices to program. Consult Biermann's surveys [12, 13] and edited collections [14, 15] for details on underlying mechanisms.

Second, in what should be known as PBE, the specifier provides positive and possibly negative input/output examples of the desired program, and the synthesiser generalises them into a program that covers at least these positive examples, but none of the negative examples. The user need thus not know how to perform the specified task, nor even how to completely specify it, and there are useful applications for this, say for novice programmers. The Machine Learning community is looking extensively into such synthesis, especially its Inductive Logic Programming (ILP) branch. Some surveys and edited collections include [14, 15, 12, 13, 27, 34] or are dedicated to [79, 37] the underlying mechanisms.

Considering the difficulty of correctly extrapolating the desired behaviour from such declared-to-be-incomplete specifications, it is not surprising that purely inductive and abductive synthesis has not been shown yet to scale beyond toy problems. The ensuing uncertainty for the specifier cannot be held against inductive and abductive synthesis, because there also is uncertainty in deductive synthesis, due to the difficulty of formalisation of assumed-to-be-complete specifications. Appropriate combinations of inductive, abductive, and deductive inference do however give leverage in synthesis from incomplete specifications [34].

Even when starting from complete specifications, the use of examples and a combination of deductive and inductive inference can still be interesting, if not necessary, as shown for Multi-tac (see Section 5.1). Other successful such combinations are reported by Ellman *et al.* [28], with applications to jet engine nozzle and racing yacht design, as well as by Gratch & Chien [43], towards scheduling ground-based radio antennas for maintaining communication with research satellites and deep space probes.

Program synthesis by *analogical* inference was tackled by Dershowitz [26].

6 Prospects of Synthesis

Program synthesis research is as old as the first computer, and a lot of theoretical research and practical development have gone into its various incarnations. Today, we stand at the dawn of a new era in programming, with languages moving away from the von Neumann model, with powerful tools generating significant amounts of tedious low-level code from higher-level descriptions, and with end-users becoming enabled to program by themselves. It is clear that program synthesis, in its traditional Artificial Intelligence understanding, can provide great leaps forward in this arena, in addition to the simpler advances offered by conventional code generation, such as through visual programming, spreadsheets, etc. The challenge is thus to scale up from techniques demonstrated in research labs on toy problems to the development of real-life software and to enable a technology transfer to commercial software development. I here propose challenges and directions for future research, as far as the inputs (Section 6.1), mechanisms (Section 6.2), and outputs (Section 6.3) of synthesis are concerned.

6.1 Synthesis Inputs

Formalisation Assistance. The acceptance bottleneck for synthesisers will always be the input language, in which the specification and domain theory have to be formalised. Most professional programmers and IT students who became somehow used to low-level languages are clearly reluctant to be re-trained in the more advanced mathematics and logic necessary to interact with synthesisers, despite the appeals of working at a higher level. They may well eventually be bypassed and made obsolete by a synthesis-induced revolution in commercial software development under web-speed market pressures, but that is yet an uncertain outcome. At the same time, end-users — from engineers in other disciplines to computer novices — hope to be enabled to program by themselves, and they will also resist the learning curve. Hence *a significant challenge is to assist users in the formalisation of the specification and domain theory.*

PlanWare and amphion can acquire and formalise them automatically from information provided by the specifiers, due to adequate human-computer-interface engineering. The current trend is thus towards *domain-specific languages* that are intuitive to qualified users, if not identical to the notations they already use anyway, thus masking the underlying mathematics and logic. Turing completeness often needs to be sacrificed, so that highly — if not fully — automated synthesisers can be developed. Research in domain analysis is needed, because the acquisition of a suitable domain theory will always be a bottleneck for synthesisers. Domains have to be identified where the payoff threshold is suitable, in terms of the size and importance of the covered problem class, the existence of a language and interface in which it is easy to describe these problems, and the difficulty of manually writing correct and efficient programs for these problems. This does not mean that the previous trends on general-purpose specification languages and semi-automatic synthesisers must decline.

6.2 Synthesis Mechanisms

Reuse. Most synthesisers are demonstrated on toy problems with little bearing to real-world problems. A main cause is that the granularity of their building blocks is too small. *The challenge is to make synthesis bottom out in reusable, assumed-correct components rather than in the primitives of the target language.*

We have seen that some existing synthesis mechanisms were designed so that libraries of formally-specified reusable components can be used during synthesis.

In KIDS/DESIGNWARE, reuse is attempted before synthesis for each specification, whether it is the initial one or one constructed during synthesis. The number of reuse queries can be significantly reduced by applying heuristics detecting that an *ad hoc* component can be trivially built from the specification. This has the further advantage of keeping the index of the component-base lean and thus accelerating reuse queries. It should be noted that the definition schemas used in algorithm design refinements also represent reused code.

In DEDALUS, reuse is possible, but not especially catered for through heuristics. Fischer & Whittle [33] propose a better integration of reuse into DEDALUS-like constructive synthesisers.

In AMPHION, reuse is the leading principle: as there is no induction rule, the mechanism is *forced* to reuse components that embody iterative or recursive calculations, in its synthesis of straight-line code.

Other than for AMPHION-like approaches, the payoff of reuse versus brute-force synthesis is however still unclear. Much research needs thus to be done towards full-scale synthesis in the style of component-based software development, i.e., bottom-up incremental programming. The synthesis of software architectures, for instance, is still a rather unexplored topic.

Schemas. I believe that *an important challenge is to make formalised algorithm design schemas [36, 80, 81], design patterns [39], plans [31], or clichés [76] continue to play a major role in scaling synthesis up.* Indeed, they allow the reuse of recognised successful product or process skeletons, which have been somehow, and not necessarily formally, proved off-line, once and for all.

Furthermore, they provide a nice division of concerns by focusing, at any given moment, the user's attention and the available options to just one well-delimited part of the current description, as opposed to, say, having to decide which transform to apply to which expression of the *entire* current description. This also enables users to understand intermediate descriptions and the synthesis process at a suitable level of abstraction.

Inference. As MULTI-TAC shows, inductive inference is sometimes necessary to achieve synthesis of efficient programs, but virtually all research — except PBE and PBD — so far has been on purely-deductive synthesis. Just like human programmers perform all kinds of inference, *the challenge is to further explore mixed-inference synthesis, in order to exploit complementary forms of reasoning.*

Similarly, even within deductive inference, there is no single mechanism that can handle all the proof obligations occurring during synthesis, hence *another* *challenge is to investigate suitable combinations of deductive proof mechanisms*, thereby achieving multi-level synthesis [11].

Finally, it seems that transformational and constructive synthesis are just two facets of a same deductive approach,[3] so that their reconciliation should be worth investigating.

6.3 Synthesis Outputs

Target Language. In order to facilitate the integration of synthesised programs with otherwise developed code modules, it is important that target languages other than the clean-semantics logic languages, that is the functional and relational ones, are supported. This is not a major research challenge, except if efficiency of the code is an issue, but rather a development issue, but it is often neglected in favour of the more attractive research challenges, thereby missing technology transfer and feedback opportunities.

Efficiency. For some problem classes, such as constraint satisfaction problems (CSPs), the efficiency of programs is crucial, such as those solving NP-complete CSPs with high constrainedness. *The challenge is that effective code optimisation must be somehow integrated with a program synthesiser towards its application in real-world circumstances.*

For instance, in constraint programming, a lot of research has been made about how to craft new variable-and-value-ordering heuristics. However, little is said about the application domain of these heuristics, so programmers find it hard to decide when to apply a particular heuristic, especially that there is no universally best heuristic for all CSPs, and not even for all instances of a given CSP (as we saw in Section 5.1). Adequate heuristics are thus problem-and-instance-specific, and must therefore be dynamically chosen at run-time rather than at programming time. It has also been noted that suitable implied constraints and symmetry-breaking constraints may considerably reduce the search space, but few results are available on how to systematise their inference. Overall, effective constraint programming remains a black art thus. *When targeting constraint programming languages, the challenge is to infer implied constraints and symmetry-breaking constraints and to synthesise problem-specific heuristics, if not solvers, that perform well on all problem instances.*

7 Conclusion

After introducing the topic and proposing a classification scheme for program synthesis, I have overviewed past and current achievements in synthesis, across three main research directions, with special focus on some of the most promising systems. I have also laid out a set of directions for future research, believing that

[3] At least the developers of DEDALUS, LOPS, and PERIWINKLE reported difficulties in classifying their systems.

they will make the technology go beyond the already-reached break-even point, compared to conventional programming and maintenance.

Program synthesis thus promises to revolutionise accepted practice in software development. Ultimately, acceptance problems due to the necessity for rigorous formalisation are bound to disappear, because programming itself is obviously a formalisation process and synthesis just provides other programming languages or different ways of programming. Similarly, the steps of any followed software lifecycle will not really change, because validation and verification will not disappear, but rather become higher-level activities, at the level of what we today call formal specifications.

Acknowledgements

I wish to thank the anonymous referees for their constructive comments on the previous versions of this paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. R. Balzer. A 15 year perspective on automatic programming. *IEEE TSE* 11(11):1257–1268, 1985.
3. A. Barr and E.A. Feigenbaum. *The Handbook of Artificial Intelligence, Chapter X: Automatic Programming*, pp. 297–379. Morgan Kaufmann, 1982.
4. D.R. Barstow. A perspective on automatic programming. *AI Magazine*, Spring 1984:5–27. Also in [74], pp. 537–559.
5. J.L. Bates and R.L. Constable. Proofs as programs. *ACM TOPLAS* 7(1):113–136, 1985.
6. F.L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations: Computer-aided, intuition-guided programming. *IEEE TSE* 15(2):165–180, 1989. Details in LNCS 183/292, Springer-Verlag, 1985/87.
7. M.J. Beeson. *Foundations of Constructive Mathematics.* Modern Surveys in Mathematics, Volume 6. Springer-Verlag, 1985.
8. W. Bibel. Syntax-directed, semantics-supported program synthesis. *AI* 14(3):243–261, 1980.
9. W. Bibel. Concurrent software production. In [61], pp. 243–261. Toward predicative programming. In [61], pp. 405–424.
10. W. Bibel and K.M. Hörnig. LOPS: A system based on a strategic approach to program synthesis. In [15], pp. 69–89.
11. W. Bibel *et al.* A multi-level approach to program synthesis. In N.E. Fuchs (ed), *Proc. of LOPSTR'97*, pp. 1–28. LNCS 1463. Springer-Verlag, 1998.
12. A.W. Biermann. Automatic programming: A tutorial on formal methodologies. *J. of Symbolic Computation* 1(2):119–142, 1985.
13. A.W. Biermann. Automatic programming. In S.C. Shapiro (ed), *Encyclopedia of Artificial Intelligence*, pp. 59–83. John Wiley, 1992.
14. A.W. Biermann and G. Guiho (eds). *Computer Program Synthesis Methodologies.* Volume ASI-C95. D. Reidel, 1983.

15. A.W. Biermann, G. Guiho, and Y. Kodratoff (eds). *Automatic Program Construction Techniques*. Macmillan, 1984.
16. A.W. Biermann and W. Bibel (guest eds), Special Issue on Automatic Programming. *J. of Symbolic Computation* 15(5–6), 1993.
17. C.B. Jones. *Systematic Software Development using* VDM. Prentice-Hall, 1990.
18. L. Blaine, L. Gilham, J. Liu, D.R. Smith, and S. Westfold. PLANWARE: Domain-specific synthesis of high-performance schedulers. In *Proc. of ASE'98*, pp. 270–279. IEEE Computer Society Press, 1998.
19. A. Bundy. A broader interpretation of logic in logic programming. In R.A. Kowalski and K.A. Bowen (eds), *Proc. of ICLP'88*, pp. 1624–1648. The MIT Press, 1988.
20. A. Bundy, A. Smaill, and G. Wiggins. The synthesis of logic programs from inductive proofs. In J.W. Lloyd (ed), *Proc. of the ESPRIT Symp. on Computational Logic*, pp. 135–149. Springer-Verlag, 1990.
21. A. Bundy, F. van Harmelen, C. Horn, A. Smaill. The OYSTER/CLAM system. In M.E. Stickel (ed), *Proc. CADE'90*, pp. 647–648. LNCS 449. Springer-Verlag, 1990.
22. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. of the ACM* 24(1):44–67, 1977.
23. K.L. Clark and S. Sickel. Predicate logic: A calculus for deriving programs. In *Proc. of IJCAI'77*, pp. 410–411.
24. R.L. Constable, S.F. Allen, H.M. Bromley, *et al. Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice-Hall, 1986.
25. J. Darlington. An experimental program transformation and synthesis system. *AI* 16(1):1–46, 1981. Also in [74], pp. 99–121.
26. N. Dershowitz. *The Evolution of Programs*. Birkhäuser, 1983.
27. Y. Deville and K.-K. Lau. Logic program synthesis. *J. of Logic Programming* 19–20:321–350, 1994.
28. T. Ellman, J. Keane, A. Banerjee, and G. Armhold. A transformation system for interactive reformulation of design optimization strategies. *Research in Engineering Design* 10(1):30–61, 1998.
29. L.-H. Eriksson. Synthesis of a unification algorithm in a logic programming calculus. *J. of Logic Programming* 1(1):3–33, 1984.
30. M.S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens (ed), *Program Specification and Transformation*, pp. 165–195. Elsevier, 1987.
31. S.F. Fickas. Automating the transformational development of software. *IEEE TSE* 11(11):1268–1277, 1985.
32. B. Fischer, J. Schumann, and G. Snelting. Deduction-based software component retrieval. In W. Bibel and P.H. Schmidt (eds), *Automated Deduction: A Basis for Applications*, vol. III, chap. 11. Kluwer, 1998.
33. B. Fischer and J. Whittle. An integration of deductive retrieval into deductive synthesis. In *Proc. of ASE'99*, pp. 52–61. IEEE Computer Society, 1999.
34. P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.
35. P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In *Proc. of ASE'97*, pp. 153–160. IEEE Computer Society, 1997.
36. P. Flener, K.-K. Lau, M. Ornaghi, and J.D.C. Richardson. An abstract formalisation of correct schemas for program synthesis. *J. of Symbolic Computation* 30(1):93–127, July 2000.
37. P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. of Logic Programming* 41(2–3):141–195, November/December 1999.

38. P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. In *Proc. of ASE'98*, pp. 168–176. IEEE Computer Society, 1998.
39. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
40. J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. The MIT Press, 1997.
41. A.T. Goldberg. Knowledge-based programming: A survey of program design and construction techniques. *IEEE TSE* 12(7):752–768, 1986.
42. M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh* LCF – *A Mechanised Logic of Computation*. LNCS 78. Springer-Verlag, 1979.
43. J.M. Gratch and S.A. Chien. Adaptive problem-solving for large scale scheduling problems: A case study. *J. of Artificial Intelligence Research* 4:365–396, 1996.
44. C. Green. Application of theorem proving to problem solving. *Proc. of IJCAI'69*, pp. 219–239. Also in B.L. Webber and N.J. Nilsson (eds), *Readings in Artificial Intelligence*, pp. 202–222. Morgan Kaufmann, 1981.
45. C. Green and D.R. Barstow. On program synthesis knowledge. *AI* 10(3):241–270, 1978. Also in [74], pp. 455–474.
46. C. Green and S. Westfold. Knowledge-based programming self applied. *Machine Intelligence* 10, 1982. Also in [74], pp. 259–284.
47. Å. Hansson. *A Formal Development of Programs*. Ph.D. Thesis, Univ. of Stockholm (Sweden), 1980.
48. J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In D. Kapur (ed), *Proc. of CADE'92*. LNCS 606. Springer-Verlag, 1992.
49. C.J. Hogger. Derivation of logic programs. *J. of the ACM* 28(2):372–392, 1981.
50. W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490. Academic Press, 1980.
51. G. Huet and G.D. Plotkin (eds). *Logical Frameworks*. Cambridge Univ. Press, 1991.
52. E. Kant. On the efficient synthesis of efficient programs. *AI* 20(3):253–305, 1983. Also in [74], pp. 157–183.
53. R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.
54. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *J. of Automated Reasoning* 16(1–2):113–145, 1996.
55. K.-K. Lau and S.D. Prestwich. Synthesis of a family of recursive sorting procedures. In V. Saraswat and K. Ueda (eds), *Proc. ILPS'91*, pp. 641–658. MIT Press, 1991.
56. K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini (eds), *Proc. of LOPSTR'94 and META'94*, pp. 104–121. LNCS 883. Springer-Verlag, 1994.
57. B. Le Charlier and P. Flener. Specifications are necessarily informal, or: Some more myths of formal methods. *J. of Systems and Software* 40(3):275–296, 1998.
58. H. Liebermann (guest ed), Special Section on Programming by Example. *Comm. of the ACM* 43(3):72–114, 2000.
59. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
60. M.R. Lowry and R. Duran. Knowledge-based software engineering. In A. Barr, P.R. Cohen, and E.A. Feigenbaum (eds), *The Handbook of Artificial Intelligence*. Volume IV, pp. 241–322. Addison-Wesley, 1989.
61. M.R. Lowry and R.D. McCartney (eds). *Automating Software Design*. The MIT Press, 1991.
62. M.R. Lowry, J. Van Baalen. METAAMPHION: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engineering* 4:199–241, 1997.

63. Z. Manna and R.J. Waldinger. Synthesis: Dreams $\rightarrow$ Programs. *IEEE TSE* 5(4):294–328, 1979.
64. Z. Manna and R.J. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS* 2(1):90–121, 1980. Also in [15], pp. 33–68. Also in [74], pp. 3–34.
65. Z. Manna and R.J. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming* 1:5–48, 1981. Also in [14], pp. 251–307.
66. Z. Manna and R.J. Waldinger. The origin of a binary-search paradigm. *Science of Computer Programming* 9:37–83, 1987.
67. Z. Manna and R.J. Waldinger. Fundamentals of deductive program synthesis. *IEEE TSE* 18(8):674–704, 1992.
68. P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of the 1979 Int'l Congress for Logic, Methodology, and Philosophy of Science*, pp. 153–175. North-Holland, 1982.
69. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.
70. J. Mostow (guest ed), Special Issue on AI and Software Engineering. *IEEE TSE* 11(11), 1985.
71. B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, 1990.
72. H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
73. H.A. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys* 15(3):199–236, 1983.
74. C. Rich and R.C. Waters (eds). *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann, 1986.
75. C. Rich and R.C. Waters. Automatic programming: Myths and prospects. *IEEE Computer* 21(8):40–51, 1988.
76. C. Rich and R.C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer* 21(11):10–25, 1988.
77. T. Sato and H. Tamaki. First-order compiler: A deterministic logic program synthesis algorithm. *J. of Symbolic Computation* 8(6):605–627, 1989.
78. D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9:229–269, 1997.
79. D.R. Smith. The synthesis of LISP programs from examples: A survey. In [15], pp. 307–324.
80. D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *AI* 27(1):43–96, 1985.
81. D.R. Smith. KIDS: A semiautomatic program development system. *IEEE TSE* 16(9):1024–1043, 1990.
82. D.R. Smith. Towards the synthesis of constraint propagation algorithms. In Y. Deville (ed), *Proc. of LOPSTR'93*, pp. 1–9, Springer-Verlag, 1994.
83. D.R. Smith. Constructing specification morphisms. *J. of Symbolic Computation* 15(5–6):571–606, 1993.
84. D.R. Smith. Toward a classification approach to design. *Proc. of AMAST'96*, pp. 62–84. LNCS 1101. Springer-Verlag, 1996.
85. J.M. Spivey. *The Z Notation: A reference manual*. Prentice-Hall, 1992.
86. Y.V. Srinivas and R. Jüllig. SPECWARE: Formal support for composing software. In B. Möller (ed), *Proc. of MPC'95*, pp. 399–422. LNCS 947. Springer-Verlag, 1995.
87. D.M. Steier and A.P. Anderson. *Algorithm Synthesis: A Comparative Study*. Springer-Verlag, 1989.

88. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy (ed), *Proc. of CADE'94*, pp. 341–355. LNCS 814. Springer-Verlag, 1994.
89. S.-Å. Tärnlund. An axiomatic data base theory. In H. Gallaire and J. Minker (eds), *Logic and Databases*, pp. 259–289. Plenum Press, 1978.
90. R.J. Waldinger and R.C.T. Lee. ProW: A step toward automatic program writing. *Proc. of IJCAI'69*, pp. 241–252.
91. G. Wiggins. Synthesis and transformation of logic programs in the WHELK proof development system. In K. Apt (ed), *Proc. of the JICSLP'92*, pp. 351–365. The MIT Press, 1992.