Extracting Proofs from Documents

Roland Backhouse and Richard Verhoeven*
Department of Mathematics and Computing Science
Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
Email: {rolandb,river}@win.tue.nl

March 27, 1998

Abstract

Often, theorem checkers like PVS are used to check an existing proof, which is part of some document. Since there is a large difference between the notations used in the documents and the notations used in the theorem checkers, it is usually a laborious task to convert an existing proof into a format which can be checked by a machine. In the system that we propose, the author is assisted in the process of converting an existing proof into the PVS language and having it checked by PVS.

1 Introduction

The now-classic ALGOL 60 report [5] recognized three different levels of language: a reference language, a publication language and several hardware representations, whereby the publication language was intended to admit variations on the reference language and was to be used for stating and communicating processes. The importance of publication language —often referred to nowadays as "pseudo-code"— is difficult to exaggerate since a publication language is the most effective way of communicating among *people*. Unfortunately, the example of ALGOL 60 has not been followed in the ensuing thirty years. Modern programming languages (C++, Java, Haskell etc.) do not admit variations. Worse still they are based on the ASCII standard which itself is based on teletype technology.

The 30-year tyranny of teletype technology also extends to theorem proving systems which, like programming languages, are typically ASCII-based and do not admit notational variation. It is no wonder that theorem provers are hard to use, even for checking theorems or programs that are already known to be correct.

In the recently begun research project entitled "publication-style programming" we are aiming to contribute to the development of systems in which programs and specifications are written in a publication language and then transformed to the hardware representation required by existing implementations of compilers and theorem provers. Such a system will allow programmers to write programs and specifications in application-specific (mathematical) languages, thus facilitating communication with the client and reducing the risk of misunderstandings that can bedevil large-scale projects. A similar project (on a much larger scale) is the Intentional Programming project being conducted by Microsoft under the direction of Charles Simonyi. Ultimately, as argued by Simonyi [13], such systems will turn programmers into language designers.

The basis for our own work in this direction is the Math∫pad editor [3, 2] which we have developed as a tool to assist those writing articles containing a large amount of advanced mathematics. The priorities set for the Math∫pad editor were readability of on-screen documents (thus enabling the use of sophisticated two-dimensional mathematical notation), flexibility (no built-in notational

*Research supported by the Dutch Organisation for Scientific Research (NWO) under contract SION 612-14-001

conventions but, instead, a flexible system for on-the-fly, user-supplied notational definitions) and writability (the ability to mix prose and mathematics freely in the text). Our next priority — the subject of the current project— is to explore architectures that will enable the coupling of Math∫pad, or a similar system, to compilers and theorem provers.

The recent Ph.D. thesis by Matteo Vaccari [14] is illustrative of what we ultimately want to achieve. In his thesis, Vaccari discusses the calculational construction of hardware circuits, where the first 6 chapters contain theoretical discussions of relation algebra, circuits and regular language recognizers, while the later chapters contain simulations of circuits using Ruby [11] and Tangram [10] and a machine verification of the theory using PVS [7]. Vaccari used Math∫pad in the process of developing and documenting the "theoretical" designs in the initial chapters, and then hand-coded these into the forms acceptable to Ruby, Tangram and PVS. Inevitably there is a lot of duplication of information since each system requires its own format. Moreover, the manual transformation from one format to another is a source of errors and adjustments in one format requires the adjustment to the other formats in a similar way. Since each system uses its own format and files, incorporating those files in the main document is often clumsy and also error sensitive. Furthermore, the user has to learn the different formats and systems, which takes a lot of time. It would be better if all interaction could be directed with one interface and one format, which takes care of updating and manipulating all the files needed by the different systems.

In an initial investigation [9] we have looked into linking Math∫pad with Jones' Gofer interpreter with the goal of allowing publication-style development of functional programs. The current paper is a preliminary investigation of the problems that occur in trying to link a Math∫pad-like system to a theorem prover. We use PVS as example theorem prover, simply because Vaccari used PVS to check his thesis and thus both a document containing structured proofs and a PVS version of those proofs are available to us.

2 The Formatted Proof

The following text is a part of Vaccari's thesis. It defines a law with a condition and gives a proof by induction of that law:

A law about *map* and *fold* is the following: given R and S such that

$$R \circ S{\times}S \quad = \quad S \circ R$$

then

$$\mathit{fold}_n.R \circ \mathit{map}_n.S \quad = \quad S \circ \mathit{fold}_n.R$$

The proof is by induction on n; for $n = 1$ it is trivially true. For $n + 1$ we have

$$\begin{array}{ll}
 & \mathit{fold}_{n+1}.R \circ \mathit{map}_{n+1}.S \\
= & \qquad \{ \ \text{definitions} \ \} \\
 & R \ \circ \ \iota \times \mathit{fold}_n.R \ \circ \ S \times \mathit{map}_n.S \\
= & \qquad \{ \ \text{fusion, induction hypothesis} \ \} \\
 & R \circ S{\times}(S \circ \mathit{fold}_n.R) \\
= & \qquad \{ \ \text{proviso: } R \circ S{\times}S \quad = \quad S \circ R\text{; fusion} \ \} \\
 & S \ \circ \ R \ \circ \ \iota \times \mathit{fold}_n.R \\
= & \qquad \{ \ \text{definition} \ \} \\
 & S \circ \mathit{fold}_{n+1}.R
\end{array}$$

The law is written informally and the proof is formatted with a standard proof notation. That is, each step in the proof is augmented with a comment, explaining why the two expressions in that step relate to each other according to the given operator. We will refer to such a proof as "a formatted proof".

Although the text is concise, there are still some unclear points. The variable n is not specified in the definition of the law and the fact that n has to be positive is part of the proof. Since n is not introduced, there is an ambiguity whether the quantification over n is part of the consequent or also part of the antecedent, which influences both the definition and the use of the law in a theorem prover.

Furthermore, the author assumes that the reader has some basic knowledge on the subject, so trivial rules are not mentioned. The trivial rules in this case are the associativity of composition and the unit of composition ι. In this case, the second "fusion step" can not be applied correctly without first adding the unit of composition to the right hand side of the first S, replacing it with $S \circ \iota$.

The document is written with the Math∫pad system and the structure of the proof and the formulas is known, that is, no additional parsing is needed to construct that structure. Due to the structure, additional operations like formula extraction and matching are available for free.

3 The PVS Proof

The law in the previous section can easily be proven in PVS if the correct environment is available, that is, if all the definitions, theorems and proofs up to this law are already available in PVS. If that is the case, the above law can be entered as the theorem:

```
fold_map:  THEOREM  R o (S*S) = S o R
               IMPLIES fold(n,R) o map(n,S) = S o fold(n,R)
```

This resembles the formatted text closely, because the binary operators `o` and `*` are overloaded and the variables `R`, `S` and `n` are already defined in the PVS context. Otherwise, the operators need to be replaced by functions with two arguments and the variables need to be introduced with a universal quantifier, which would result in the following PVS input:

```
fold_map:  THEOREM  (FORALL (R:rel, S:rel, n:upfrom(1)):
         comp(R,prod(S,S)) = comp(S,R)
            IMPLIES comp(fold(n,R),map(n,S)) = comp(S,fold(n,R)) )
```

Since only a limited number of operators can be overloaded in PVS, it is most likely that, in a general case, a PVS formula will look like this last formula, thereby creating a gap between the familiar syntax used in the document and the unfamiliar syntax used by PVS. The translation from the formatted document to the PVS formula is still straightforward and easy to do automatically once the structure is known.

Following the formatted proof, it is quite easy to construct a standard PVS proof of the theorem:

```
(induct "n" 1)                       %induction
1  (grind)                           %basis: trivial
   (rewrite "id0")
   (rewrite "id1")
2  (skolem!) (ground)                %step
   (skolem!) (ground)
   (expand "fold" :if-simplifies t)  %definition
   (expand "map" :if-simplifies t)   %definition
   (assoc-rewrite "fusion" :dir RL)  %fusion lemma
   (inst?) (ground)                  %induction hypothesis
   (replace*)
   (rewrite "id1")                   %remove unit of composition
   (rewrite "id0" 1 ("R" "S!1") 1 RL) %add unit of composition
```

```
(assoc-rewrite "fusion")          %fusion lemma
(rewrite "id0")                   %remove unit of composition
(replace*)                        %proviso
(rewrite "comp_assoc")            %associative of composition
```

Using PVS all the trivial steps have to be provided. In this case, the trivial lemmas are:

```
id0:        THEOREM R o I = R
id1:        THEOREM I o R = R
comp_assoc: THEOREM (R o S) o T = R o (S o T)
```

Since the last lemma is difficult to instantiate if multiple matches occur, a PVS rewrite rule had been defined that rewrites modulo associativity of composition, called `assoc-rewrite`.

As this example clearly shows, the PVS version is less readable than the formatted version, partly because the formulas are missing. However, adding the formulas that PVS generates in its output would not really improve the readability, because notational conventions are different and it would increase the length of the proof dramatically, especially if all the premises are added.

The structure of the PVS version resembles the structure of the formatted proof quite closely, except on some small points. In PVS, you prove that the equation is true, while in the formatted text, you prove that the left side is equal to the right side by transitivity of equality. As a result, the last step in the formatted proof (using the definition of *fold*) is already applied at the earlier step in the proof. Furthermore, in the formatted proof, a definition can be used in two directions, while in PVS, a definition can only be expanded. Another small difference is the changed order of the second use of fusion and the proviso, due to the fact that the formatted proof is less strict in giving the hints.

Since PVS provides very powerful strategies, the PVS proof could be made shorter by using strategies like `induct-and-simplify` and installing automatic rewrite rules. These strategies were not used in order to be able to follow the formatted version and to identify problems and solutions.

4 The Scenario

Since the PVS proof resembles the formatted proof, system support for semi-automatically generating the PVS version should be available. After the formatted proof is finished, the user would like to press a "Check" button to see if the proof is correct according to PVS. The system should generate the PVS files containing the theorem and the proof attempt and use PVS to check if it is correct. If errors or ambiguities occur during the generation of those files, the system asks the user for assistance. If PVS is unable to process the files, the user is pointed to the part of the document which is responsible for the error. The user can then add hints to the document to adjust the PVS input that will be generated. Once the PVS input is correct, PVS will check the proof and errors in the proof steps are redirected by the system to the hints in the formatted proof. Again, the user can add PVS specific hints to adjust the behaviour of PVS.

In this feedback loop, the user should see as little PVS syntax as possible and all PVS results should be shown in the syntax preferred by the user. The user should of course be able to add raw PVS commands to the generated PVS files, to make sure that the user has full control over both PVS and the system itself. However, the user should not have to adjust the generated PVS files by hand, as that would introduce synchronisation problems and, at any time, the document itself should be sufficient to regenerate the correct PVS files.

The system would analyse a part of the document and generate as much relevant PVS input as reasonably possible. Since the document is already structured in Math∫pad, much of the structure for the PVS input is already available. The formulas are converted easily as long as the PVS versions use a similar structure. In order to fill in the missing gaps, databases with common keywords, lemmas, definitions and variables are used to collect information and select PVS tactics and strategies. The user and system can extend and adjust these databases to optimise the automatic generation of PVS input or to adjust it to the language native to the user. In the example proof, the known keywords would be 'given', 'such that', 'then', 'proof', 'induction', 'trivially', 'definition', 'induction hypothesis' and 'proviso', the known lemma would be 'fusion', the known definitions would be 'fold' and 'map' and the known variables would be 'R', 'S' and 'n'. With this knowledge, much of the PVS input could be extracted. This form of extracting information can be compared with the methods used in the Eliza system [15] and other natural language processing systems.

5 Problems and Solutions

It is obvious that constructing such a system is not trivial and that a lot of problems will occur while trying. In this section, some of the foreseeable problems are discussed, together with some solutions.

5.1 Extracting the Theorem

As the text is already available, it would be nice if the system can partially extract the theorems and definitions and convert them to the correct PVS syntax. However, a natural language contains a lot of ambiguity, even in a mathematical or technical document, where the author carefully weighs ambiguity with readability. As a result, small ambiguities might be introduced which are not noticed by the reader but are important for a theorem prover. In the example, the variable n is never introduced and as a result, the extracted law might be either

$$\forall(R,S :: \; R \circ S{\times}S \;=\; S \circ R \;\Rightarrow\; \forall(n : \; n \geq 1 : \; \mathit{fold}_n.R \circ \mathit{map}_n.S \;=\; S \circ \mathit{fold}_n.R))$$

or

$$\forall(R,S,n : \; n \geq 1 : \; R \circ S{\times}S \;=\; S \circ R \;\Rightarrow\; \mathit{fold}_n.R \circ \mathit{map}_n.S \;=\; S \circ \mathit{fold}_n.R)$$

In this situation, the meaning of the two expressions is equal, since n does not occur in $R \circ S{\times} S \;=\; S \circ R$ (which might not be true in a more general case). However, the PVS proofs of the two laws are different and, more importantly, applying the laws is different. Therefore, interaction between the user and the system is needed to extract the final unambiguous interpretation of the text. This interaction could be based on a natural language, describing the meaning of the text unambiguously or could be based on the native syntax, possibly extended with standard logical notations, as shown above. The user should be able to select one of the possible interpretations or change the contents of the document to remove the ambiguity.

To reduce the number of the errors that PVS reports, the system has to make sure that all dependencies of the formulas are resolved by including the definitions of all referred functions, lemmas and variables. However, PVS remembers which lemmas are already proven and which theory contains them. Copying a lemma to a working document will result in error messages from PVS about multiple occurrences of the same lemma, with the result that the copied version is not proven. Therefore, the proven lemmas have to be collected in theory files, which are included by the temporary PVS file containing the unproven theorem. After this theorem is proven, it can be added to the appropriate theory file, where it has to be proven once again to update the PVS state and its databases.

5.2 Proving the Formatted Proof

In order to prove that the formatted proof is correct, a proof has to be given for each step in the formatted proof, that is, each consecutive pair of formulas is indeed related to each other according to the given relator (and this proof should be based on the hints that are given). For example, the proof step

$$R \;\circ\; \iota \times \mathit{fold}_n.R \;\circ\; S \times \mathit{map}_n.S$$

$$
\begin{array}{ll}
= & \{ \text{ fusion, induction hypothesis } \} \\
& R \circ S \times (S \circ \mathit{fold}_n . R)
\end{array}
$$

is proven in PVS by showing that the two formulas are equal, by using fusion and the induction hypothesis. This would result in a small lemma stating that this equality is a valid consequent of the induction hypothesis, as in:

```
pstep2:  LEMMA fold(n,R) o map(n,S) = S o fold(n,R)
           IMPLIES R o (I*fold(n,R)) o (S*map(n,S)) = R o (S*(S o fold(n,R)))
```

Again, the translation of the formulas is straightforward and can be applied to each step of the formatted proof and, since the document is already structured, determining the steps and formulas in a proof is trivial. Such a small lemma can be used to check if the corresponding step is correct and together these small lemmas are used to verify that the proof of the law is correct. The advantage of using these small lemmas is that unclear hints and problems in PVS are directly related to specific steps in the formatted proof. A disadvantage would be that a large collection of small lemmas results in large PVS files, what would effect the performance of PVS. This problem might be resolved by combining the proofs of the several small lemmas into one larger proof, thereby removing the need to define those lemmas.

A different approach to proving that the formatted proof is correct would be to prove that the first formula is related to the last formula and check the PVS output to verify that the intermediate formulas do occur at the correct positions and that the PVS tactics have some effect on the formulas. However, this approach does not work in a general case. Each of the following proofs is in fact a combination of two proofs.

$$
\begin{array}{ll}
& P \\
\Rightarrow & \{ \ \vee \text{ introduction } \} \\
& P \vee (Q \wedge P) \\
\equiv & \{ \text{ DeMorgan } \} \\
& (P \vee Q) \wedge P \\
\Rightarrow & \{ \ \wedge \text{ elimination } \} \\
& P
\end{array}
\qquad
\begin{array}{ll}
& P \\
\equiv & \{ \Rightarrow\!: \vee \text{ introduction} \\
& \quad \Leftarrow\!: \text{DeMorgan}, \wedge \text{ elimination } \} \\
& P \vee (Q \wedge P)
\end{array}
$$

In the proof on the left, the purpose is not to show that $P \Rightarrow P$, but to show that all the intermediate formulas are equal. In the proof on the right, the hint contains an explanation why the equality holds, by reasoning about both directions. In both cases the proof of $P \Rightarrow P \vee (Q \wedge P)$ and $P \vee (Q \wedge P) \Rightarrow P$ is combined into a single proof, which is difficult to extract in a more complicated case.

5.3 Hiding Information

After the PVS files are generated, PVS can be used to check if the definition is correct. In case of errors, the system should start an interaction with the user to resolve the problems. During this interaction, the user should not be exposed to the PVS syntax, as it is often awkward, unfamiliar and unreadable in comparison with the standard notational conventions used in mathematics and other technical fields. Instead, the system should translate all the PVS communication to the notational conventions native to the user. This requires origin tracing and, since PVS can not be adjusted freely, a parser for the PVS messages and formulas.

An interesting proof contains a complification step, that is, a step where a simplification is used in the opposite direction or where a smart case analysis is applied. Such steps can not be determined automatically, since there are usually many possible steps, but only few of them will have a positive effect. Since the formatted proof already exists, the user knows where each complification step occurs and how each simplification lemma is applied. Somehow, the user has to specify these steps to the system and add them to the generated PVS input, without being exposed to the PVS syntax. This can be achieved by showing the formula and the tactic that failed on the formula. The user can modify the formula with a restricted set of actions, all corresponding to actions in PVS, until the failed tactic can be applied successfully. Alternatively, the user can adjust the proof, if it seems to be incorrect. The actions needed to adjust the formula will be stored in the hints in the proof, such that a regeneration of the PVS proof will be correct the next time.

All formulas are given a number in PVS, to allow the user to refer to them when a tactic is applied. In the formatted proof these numbers are not available and the user is usually not interested in these numbers. In fact, the PVS user is advised to apply the tactics without specifying these numbers, as they might change if the proof or the theorem is edited afterwards. Instead, the user should only specify whether to apply the tactic to the premises or the consequents. However, such general tactics might result in changes to formulas which should not change. In the example, the general tactic `(replace*)` will use the premise to replace matching parts of the consequent.

While proving something in PVS, PVS will introduce variables with ugly names. These names are usually not needed in the tactics, except when a rewrite rule is applied and the specific substitution is needed. Since the user should not be exposed to the PVS syntax, the user should be able to construct such substitutions with the familiar syntax, where the system will translate it back to PVS syntax. In the example, the complification step with lemma `id0` contains such a substitution.

Due to tactics in PVS, the goal might get split into several subgoals. Some of these subgoals could be trivial if a powerful strategy is applied, such as `grind`. If such a strategy can successfully prove a subgoal, the system should not mention such a subgoal to the user, who could get disappointed by a large collection of unproven but trivial subgoals. Furthermore, the system should warn the user if the number of non-trivial subgoals exceeds a certain threshold, as it could indicate that an additional lemma or case analysis is needed to make the proof more intuitive.

5.4 Defining Tactics

When the formatted proof is used to construct a PVS proof, the PVS proof will start with a formula of the form $A \trianglelefteq B$ and should end with a formula of the form $B \trianglelefteq B$. However, while applying PVS tactics, it is very likely that certain actions will change formula B, which could result in a failed proof. To prevent this, the right hand side of the formula, B in this case, can be replaced using a name definition before applying the tactics and hide it from PVS. After all the tactics are applied, the name definition can be used to check if the proof has succeeded or not. This approach has one disadvantage: a definition can only be expanded in PVS, which, in the example, prevents the last step of the proof. To solve this, a trivial lemma can be defined to reverse the expansion of a definition and an additional tactic can be used to apply the rule.

In the formatted proof, trivial steps are often not mentioned, since it would make the proof less readable. In the PVS proof, these steps are needed before the next tactic can be applied. An obvious solution would be to install automatic rewrite rules and allow the user to specify which rules can be regarded as trivial. However, if a trivial rule is applied as a complification step, the automatic rewrite rules should not undo such a step, such as the complex rewrite step in the example. Therefore, the trivial rewrite rules should only be applied if the next tactic would fail without them. This can be achieved by defining new PVS tactics.

A lemma or rewrite rule is often used in two directions. In PVS, the direction of the rewrite rule has to be specified in order to apply it successfully. In the formatted proof, the hints do not contain this information, as it is obvious to the reader. The solution would be to define a new PVS strategy to apply the rewrite rule in the default direction and, if that fails, apply it in the opposite direction. The drawback of such a strategy would be that the result is unpredictable, as the rule might be applicable in both directions. In the example proof, the `fusion` lemma is applied twice in different directions and the PVS strategy would remove the requirement to specify the direction.

5.5 The PVS Interface

The PVS system itself also causes some problems. The PVS core system is written in ilisp and uses the Emacs editor as a user interface. The communication between this core system and Emacs is hidden from the normal user and is not well documented. Since the Emacs system is too large to be used as a communication layer and would introduce more problems than that it would solve, the PVS core system has to be extract and documented. Such documentation is of course also helpful for other systems that try to interact with PVS.

The PVS core system is rather large and calculations might be time consuming, especially if advanced and powerful strategies are used. Therefore, the system has to protect the user from trying to proceed with the document preparation while PVS is busy, but it should allow the user to perform other tasks. The best solution would be to block editing actions on certain parts of the document, such that PVS is always computing results which are still valid and needed.

6 Related work and conclusions

There are already some projects to improve the interface of PVS. TAME [1] is a layer on top of PVS for reasoning about timed automata and consists of a number of strategies to reduce the number of steps made in a typical PVS proof to the number of steps made in a hand-made proof. With these additional strategies, the user of TAME will not be exposed to the low-level steps and commands needed in PVS, thereby making the commands field specific. However, since the PVS interface is used, there is still a gap between the notational conventions used by PVS and those used in the documentation.

The system PAMELA [4] is designed to check partial correctness of VDM-like specifications in the area of code generators. By providing a connection with PVS, the system supports a larger class of specifications, using PVS to discharge proof obligations. Since PAMELA provides its own interface, communication between PVS and another system seems to be feasible and some experience is available.

Simons [12] has been working on a system to combine proofs in Isabelle [8] with documentation. The system uses the structured documentation technique introduced by Knuth [6] to allow one file to contain both the proofs and the documentation and uses programs to separate those. This solves the problem of combining several files into one document, at the expense of using different languages in a single file, namely, LaTeX for formatting the document, Isabelle for specifying the proof and the meta language to instruct the programs. For a user, this might be confusing.

Constructing a tool which extracts a PVS proof from a document seems to be feasible, although there are a lot of problems to overcome. We already have a collection of electronic documents which are structured and a tool to edit and manipulate those documents. For some documents, the PVS definitions and proofs are already available, which is very helpful to get started.

Since PVS is not the only available theorem prover, the connection with PVS should be made general enough, such that a connection with a different theorem prover would not be too difficult. The fact that PVS is a closed system could in this case be helpful, as modifications to PVS would indicate that modifications to other theorem provers are needed if such a connection is required.

References

[1] Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 33–48, Murray Hill, NJ, August 1997. Springer-Verlag.

[2] R.C. Backhouse, R. Verhoeven, and O. Weber. Mathʃpad: A system for on-line preparation of mathematical documents. *Software – Concepts and Tools*, 18:80–89, 1997.

[3] Roland Backhouse and Richard Verhoeven. Mathʃpad *Ergonomic Document Preparation*, version 0.60 edition, February 1996. Manual of the Mathʃpad system. See also: `http://www.win.tue.nl/cs/wp/mathspad/`.

[4] Bettina Buth. *Operation Refinement Proofs for VDM-like Specifications*. PhD thesis, Institute of Computer Science and Practical Mathematics of the Christian-Albrechts-University Kiel, February 1995. See also: `http://www.informatik.uni-bremen.de/ bb`.

[5] P. Naur (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6:1–17, January 1963. Also in *The Computer Journal*, 5: 349–67 (1963); *Numerische Mathematik*, 4: 420–52 (1963).

[6] D.E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[7] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. See also: `http://www.csl.sri.com/pvs.html`.

[8] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[9] Daniël Peeters. Towards a publication style of functional programming. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, June 1997.

[10] Frits D. Schalij. Tangram manual. Technical Report UR 008/93, Philips Electronics N.V., 1996.

[11] Robin Sharp and Ole Rasmussen. An introduction to Ruby. 2nd edition. Technical report, Dept. of Computer Science, Technical University of Denmark, 1995. Available at `ftp://ftp.it.dtu.dk/pub/Ruby/intro.ps.Z`.

[12] Martin Simons. Proof presentation for Isabelle. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 259–274, Murray Hill, NJ, August 1997. Springer-Verlag.

[13] Charles Simonyi. The death of computer languages, the birth of intentional programming. Proceedings of the 28th Annual International Seminar on the Teaching of Computing Science at University Level, Sponsored by ICL and University of Newcastle upon Tyne, Department of Computing Science, September 1995.

[14] Matteo Vaccari. *Calculational Derivation of Circuits*. PhD thesis, Department of Computing Science, Università degli Studi di Milan, December 1997. Draft version, to be published.

[15] Joseph Weizenbaum. Contextual understanding by computers. *CACM*, 10(8):474–480, 1967.