ICOT Technical Report: TR-629

TR-629

Program Synthesis by a Model Generation Theorem Prover

by
M. Fujita & R. Hasegawa

March, 1991

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Program Synthesis by A Model Generation Theorem Prover

Masayuki Fujita
Ruzo Hasegawa
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan
Phone +81-3-456-2514
e-mail mfujita@icot.or.jp, hasegawa@icot.or.jp

Yasuyuki Shirai
Mitsubishi Research Institute, Inc.
2-3-6, Otemachi, Chiyoda-ku, Tokyo 100, Japan
Phone +81-3-536-5813
e-mail shirai@itsoa.co.jp

February 8, 1991

Abstract

The combination of theorem provers and program synthesis, that can obviously provide us of automated programming, has been left out of consideration because of the theoretical darkness how to relate proofs obtained by the resolution based provers to the executable programs.

We here propose a new approach to this goal with using a clear framework of program extraction, which is based on the constructive mathematics. The basic principle of this framework is the *Realizability Interpretation*, which allows us to relate programs with proofs. *Realizability Interpretation* can be a mathematical basis of building a formal program extraction system from proofs obtained by a wide variety of first-order provers.

Besides the theoretical research of this approach, we have built an experimental system with using a model generation theorem prover(MGTP) and a formal logic system(PAPYRUS) and have obtained several programs such as sorting programs.

1 Introduction

More than twenty years have passed after the idea of program synthesis from the formal proof in mathematics was proposed. The methods proposed until now, however, are rather ad hoc and have rather weak methods for theoretical investigation. Moreover, in spite of a great advantage of bug free program synthesis, proof generation has remained as a bottleneck of practical use. On the other hand, research on automated reasoning, independent from program synthesis, has got much progress in the last decade not only in the theory but also in the practice in accordance with the progress in the area of logic programming([Sti88],[MB88]).

The combination of these two technologies. that can obviously provide us of automated programming, has been left out of consideration because of the theoretical darkness of how to relate proofs obtained by the resolution based provers to the executable programs. Our purpose is to find a way to combine these two technologies in order to extract not only sequential programs but also concurrent programs. We used *Realizability Interpretation*(an extension of *Curry-Howard*

Isomorphism) in the area of constructive mathematics in order to give the executable meaning to proofs obtained by efficient theorem provers.

Our approach for combining prover technologies and *Realizability Interpretation* has the following advantages:

- This approach is prover independent and all provers are possibly usable
- *Realizability Interpretation* has a strong theoretical background
- *Realizability Interpretation* is general enough to cover concurrent programs

Two systems **MGTP** and **PAPYRUS**, developed in ICOT, are used for the experiments on sorting algorithms in order to get practical insights of our approach.

A model generation theorem prover (**MGTP**) implemented in KL1 searches for proofs of specification expressed as logical formulae and it runs on a parallel machine:Multi-PSI. **MGTP** is a hyper-resolution based bottom up(infers from premises to goal) prover. Thanks to KL1 programming technology **MGTP** is simple but works very efficiently if problems satisfy *range-restrictedness*. The inference mechanism of **MGTP** is similar to SATCHMO[MB88] in principle. Hyper-resolution has an advantage for program synthesis that inference system is constructive. This means that no further restriction is needed for avoiding useless search.

PAPYRUS(PArallel Program sYnthesis by Reasoning Upon formal Systems) is a cooperative workbench of formal logic. This system handle proof trees of user defined logic in *Edinburgh Logical Framework*(*LF*)[HHP87]. A typed lambda term in LF represents a proof and a program can be extracted from this term by lambda computation. This system treats programs(functions) as the models of a logical formula by user defined *Realizability Interpretation*. **PAPYRUS** is a integrated workbench for logic and provides similar functions of PX[HN88], Nuprl[Con86] and Elf[Pfe88].

We faced two major problems in the research process:

- How to extract a program from a proof in clausal form
- How to incorporate induction and equality

The first problem relates to the fact that programs cannot be extracted from proofs obtained by using excluded middle as used in classical logic. The rules for transforming formulas into clausal form contains such a prohibited process. This problem can be solved if the program specification is given in clausal form because a proof can be obtained from the clause set without using excluded middle by **MGTP**. The second problem is that all induction schemes are expressed as second-order propositions. In order to handle this, second-order unification will be needed, which is impractical to use so far. However, it is possible to transform a second-order proposition to a first-order proposition if the program domain is fixed.

Proof steps of equality have nothing to do with computation, Provers can use efficient algorithms for equality as an attached procedure.

In this paper, we first describe the theorem prover and the program extraction technique. Then technical difficulties and the solutions are outlined. Finally after mentioning the experimentation on sorting algorithms we will discuss concurrent programs extraction and research interests in this new approach of automated programming.

2 MGTP

The MGTP prover adopts model generation as a basic proof procedure. We assume that a theorem to be proven is negated and transformed to a set of clauses, then we try to refute the clause set as in the resolution method. A clause is represented in an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

Type	=	**Proposition**	$\Leftrightarrow$	**Specification**
$\vdots$		$\vdots$		$\vdots$
λ-**term**	=	**Proof**	$\Leftrightarrow$	**Program**

Figure 1: The Curry-Howard Isomorphism

where $A_i (1 \leq i \leq n)$ and $C_j (1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, A_2, \ldots, A_n$; the consequent is a disjunction of $C_1, C_2, \ldots, C_m$.

There are the following two rules in the model generation method.

- Model extension rule: If there is a clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model M and $C\sigma$ is not satisfied in M, then extend the model M by adding $C\sigma$ into the model M.
- Model rejection rule: If there is a negative clause whose antecedent $A\sigma$ is satisfied in a model M, then reject the model M.

The task of model generation is to try to construct a model for a given set of clauses starting with a null set as a model candidate. If the clause set is satisfiable, a model should be found. The method can also be used to prove that the clause set is unsatisfiable, by exploring every possible model candidate to see that no model exists for the clause set.

The model generation method does not need full unification during conjunctive matching of the antecedent literals against model elements if the range-restrictedness[1] condition is imposed on problem clauses. When range-restrictedness is satisfied, it is sufficient to consider one-way unification, or matching, instead of full unification with occurs check because a model candidate constructed by the model generation rules should contain only ground atoms. This property is favorable in implementing a prover in KL1 since matching is easily realized with head unification and the variables in the given clauses can be represented as KL1 variables. Experimental results show that the MGTP prover is efficient in solving range-restricted non-Horn problems. For details of techniques for implementing MGTP, refer to [HFF90], [FH90].

3 Realizability Interpretation

Many attempts to realize program synthesis by theorem proving have been made in recent twenty years. For instance, Manna and Waldinger[Man80] proposed a method embedded in the Tableau Method and extracted a program from a proof by hand. Traugott[Tra89] extended this method and applied to a variety of sort algorithms. But the proposed methods are rather ad hoc and have insufficient justification of the correctness of the extracted program.

A more strict approach to extract programs from proofs is based on the *Curry-Howard Isomorphism*(CHI) or proposition as type[Mar82]. The CHI, first proposed at the end of the 1960's, is based on the correspondence between a typed lambda term and its type and between a proposition and its proof. CHI is shown in Fig. 1.

The major advantage of this approach is the correspondence between the soundness of the extracted program and the correctness of the proof. Correctness of a program is assured by the correctness of a proof. *Realizability Interpretation* is a natural extension of CHI with enriched recursive functions and recursive data types. Realizability Interpretation is one of the hot

[1]To ensure range-restrictedness, a *dom/1* predicate is added to the antecedents of problem clauses and extra clauses for the predicate are added to the original set of clauses, if necessary. This transformation does not change the satisfiability of the original set of clauses.

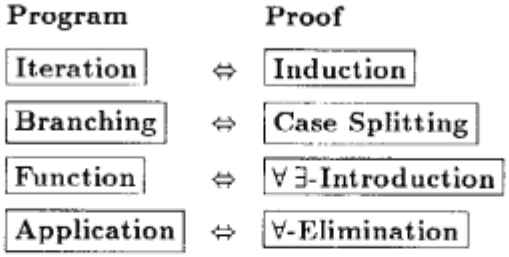


Figure 2: The Correspondence between Proofs and Programs

$$\forall x \exists y C(x) \supset R(x,y) \quad \Leftrightarrow \quad \neg C(X) \vee R(X, f(X))$$

Spec :
input x;
output y *such that if* x *satisfies condition* C(x)
then R(x,y)

$X \leftarrow$input
$f(X) \leftarrow$output

Figure 3: Specification in Clausal Form

research areas of the constructive mathematics. Our program extraction method is based on [Tak87] and is extended to handle some kind of recursive domains.

Fig. 2 will be of good help for intuitionistic understanding of the CHI.

4 Difficulties and solutions

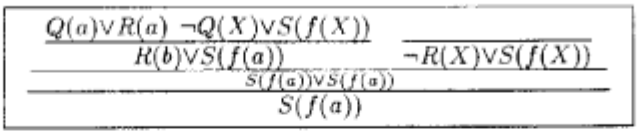
4.1 Clausal forms and resolution

The most provers, including MGTP, treat propositions in the clausal form. On the other hand, CHI means the correspondence between logical connectives and quantifiers and type constructors. But both resolution based systems and natural deduction have the completeness for the full first-order logic, so there is a natural correspondence between proofs of them. We can see skolem functions as the existentially quantified objects and free variables as universally quantified objects. Fig. 3 shows an example of this correspondence.

Transformation of formulae into clausal form uses inference rules out of constructive logic. For example, you cannot deduce $\neg$ A $\vee$ B from A$\supset$B. However the resolution principle is not out of intuitionistic logic. Hence, the problems in using clausal form based provers are combined to the transformation to the clausal form. If the specification can be initially expressed in a clausal form, all the prover based on the resolution principle are valid for program synthesis. The resolution principle itself is safe because it is constructive but structure rules such as contraction are harmful for program extraction.

But MGTP does not need to use the contraction rule and has very good property for program synthesis. The intuition is that MGTP uses disjunctive rules only for dividing the world into each case. But the information about cases should be vanished by contraction. So some structure rules in provers will affect the realizability of proofs[2]. Fig. 4 shows a typical example of contraction.

[2]Precise argument of structure rules is in [Gir89]



In this proof contraction made the origin
(left or right tree) of S(f(a)) in the
bottom be anonymous.

Figure 4: Contraction problem

4.2 The recursion and the induction

Many complicated controls of iteration can be expressed in a simple recursion. The recursion corresponds to the induction in proofs. All induction schemas are based on the well-founded induction on the ordinal number. Each schema corresponds to each data domain such as list or natural number.

Induction can be expressed as a second-order proposition. For example, the induction schema for list is as follows, where ℓ represents the type of lists.

$$\forall\ \mathbf{P}{:}\ell \to o\ (\forall\ \mathbf{L}{:}\ell\ (\forall\ \mathbf{L'}{:}\ell\ \mathbf{L'} \prec \mathbf{L} \supset \mathbf{P(L')}) \supset \mathbf{P(L)}) \supset \forall\ \mathbf{L}{:}\ell\ \mathbf{P(L)}$$

Second-order propositions cannot be utilized by the first-order provers because of their second-order variables with which provers need the second-order unification. In addition second-order unification is not practical because of its inefficiency.

If the second-order variables are assigned by a proposition, the induction schema becomes a first-order proposition. For example, the above schema for the sort algorithm becomes as follows.

$$[\forall\ \mathbf{Z}{:}\ell\ (\forall\ \mathbf{Y}{:}\ell\ \mathbf{Y} \prec \mathbf{Z} \supset \exists\ \mathbf{U1}{:}\ell\ \pi\mathbf{(U1,\ Y)} \wedge \chi\mathbf{(U1)}) \supset \exists\ \mathbf{U2}{:}\ell\ \pi\mathbf{(U2,\ Z)} \wedge \chi\mathbf{(U2)}]$$
$$\supset \forall\ \mathbf{Z}{:}\ell\ \exists\ \mathbf{S}{:}\ell\ \pi\mathbf{(S,\ Z)} \wedge \chi\mathbf{(S)}$$

Where Y $\prec$ Z means a partial-order relation that Y is a sublist of Z
(there is a one to one mapping from Y to Z). π(a, b) represents that a is a sorted permutation of b. This is a first-order proposition, which first-order provers can handle.

Induction rule must be expressed in the clausal form in MGTP. We used the following formulae to prove a quick sort problem.

$\pi(\mathbf{U},\ ind) \rightarrow \pi(\sigma(\mathbf{X}),\ \mathbf{X})$
(if the proposition is satisfied for the introduced constant,
it is satisfied for any list)
$\mathbf{Y} \prec ind \rightarrow \pi(\sigma(\mathbf{Y}),\ \mathbf{Y})$
(by induction hypothesis all sublists of the constant can be sorted)

These rules for MGTP are sufficient for obtaining various sort algorithms. Realizability Interpretation of list induction is as follows:

$$\pi(\sigma(\mathbf{a}),\ \mathbf{a}) \Rightarrow \sigma(\mathbf{a})$$
$$\pi(\mathbf{T},\ ind) \Rightarrow \mu\sigma.\lambda ind\ \mathbf{T}$$

Where T is a proper term obtained by the proof and μ is a fixpoint operator of lambda calculus.

4.3 Equality

Reasoning upon equality is essential for program synthesis(ex. $A = B \rightarrow P(A) \supset P(B)$) . For example, by using the fact that cons(car(X), cdr(X)) = X if X is not a null list we can divide a list. The equality axiom is a higher-order proposition and is beyond the area of first-order provers. The same approach to this as to induction is not practical because it is not so general that we need many rules for each problem.

As there is no program information in the proof of equality nor in the proof with equality axiom, this part can be checked by any problem solver. Efficient algorithms for problems with equality such as *paramodulation* and *demodulation* are suitable.

5 An example

A small example of sort algorithm would be helpful to understand how program can be extracted from the proof tree of MGTP. Insert sort is the target program. Programs will be extracted by the following three steps.

1. Proof tree generation from trace of MGTP
   From the trace of an MGTP proof procedure PAPYRUS generates a proof tree in Natural Deduction form [3]. The difference in notation from the previous section has no meaning. This proof tree corresponds to the case splitting. 'sort' is a predicate which means the second argument is an ordered permutation of the first argument. 'ins' is a function introduced by the insert lemma.

$$\dfrac{\begin{array}{ccc} & [\mathrm{nil(L)}] & [\mathrm{nonnil(L)}] \\ & \vdots & \vdots \\ \mathrm{nil(L)} \vee \mathrm{nonnil(L)} & \mathrm{sort(L,nil)} & \mathrm{sort(L,ins(car(L),z(cdr(L))))} \end{array}}{\mathrm{sort(L,or(nil,ins(car(L),z(cdr(L)))))}} \vee\text{-}elim$$

2. Getting typed lambda term in LF
   The corresponding LF term can be obtained by simple transformation. $\circ$ means application of a term to a function. 'prf' represents the corresponding LF term of the proof of each goal. ':' is an infix oprerator of typed lambda calculus which means that the lefthandside of this inhabits the righthand.

```
∨-elim∘prf(nil(L) ∨ nonnil(L))
      ∘prf(nil(L) → prf(sort(L,nil))
      ∘prf(nonnil(L) →
          prf(sort(L,ins(car(L),z(cdr(L))))))
  : sort(L,or(nil,ins(car(L),z(cdr(L)))))
```

3. Program Extraction
   A program can be extracted by substitution of corresponding lambda terms for inference rules and simplification by lambda computation.

[3] In order to making the program extraction simple and mechanical process, we designed a logic for MGTP

$\vee$-elim(:= λ cond. λ left. λ right.condoleftoright)
o(λ left. λ right. if L = nil then leftonop else rightonop)
o (λ p. nil)
o (λ p. ins(car(L),z(cdr(L))))
$\Rightarrow$ if L=nil then nil else insert(car(L),z(cdr(L))

6 Discussion

The following two topics are of the interest of this section.

- The ability of our approach in extracting concurrent programs
- Relations between proofs and programs with multiple I/O such as logic programs.

6.1 Concurrent programs and program extraction

Parallel reduction of redexes is the basic computation mechanism of the lambda calculus. We can see lambda terms as concurrent programs because of this. And this corresponds to the independent part of a proof. Although this is not sufficient to express communication processes such as co-process, a feasible extension of the realizability interpretation can cover concurrent processes that communicate by streams in concurrent logic programming languages. That is to introduce a list data type with infinite length and induction on the infinite list as an axiom. Concurrent programs such as humming sequence or prime number generation are in the scope of this extension. But the most of important programs in concurrent programming are not able to be extracted by this extension. In order to handle non-deterministic programs, non-trivial extension of logic will be needed.

6.2 Logic programs and program extraction

Programs with multiple I/O such as append in prolog are in logic programming. However, I/O is fixed in Curry-Howard Isomorphism as the interpretation of quantifiers determines I/O of data. Thus there is no proof corresponding to programs with multiple I/O. One of the ways to find a solution of this question is to change skolem functions of premises in clausal form in order to alter the I/O of premises. In the clausal form free variables correspond to inputs and skolem functions so do to outputs. So this change means the change of I/O of the premises. If the proof is valid after the changes and the position of output term moves in accordance with the changes in premises, we can prove I/O altered multiple goals without changing the structure of proof tree but only by changing the specifications of premises. Hence the corresponding program has multiple I/O.

7 Conclusion

The realizability interpretation is an elegant and formal method to relate proofs with programs. This principle together with prover technology make automated programming possible. Through the experiments, on sort problem, of the theorem proving and the program extraction, we have shown that this method is practical. The followings are what problems we have faced and how we have solved them.

- Clausal forms and structural rules
  Not all proofs generated by theorem provers have the corresponding programs because of structural rules. But MGTP does not use any structural rules and is suitable for program synthesis. The transformation process destroies the procedural meaning of formulas, but the correspondence between variables and skolem functions and quantified variables gives a simple extension of CHI.

- Induction
  Though induction is a hyper-order scheme, we can use first-order instances for each problem domain.

- Equality
  Handling equality is inevitable for program synthesis. And it is also a higher-order problem. However inference with equality or proof of equality has nothing to do with computation thus we can use any equality problem solver.

Acknowledgements

We would like to thank Kazuhiro Fuchi for giving us the opportunity of doing this research. We also wish to thank Koichi Furukawa for introducing us to other related works, and Mark E. Stickel for his helpful comments. Thanks are also due to Miyuki Koshimura at JBA Co. Ltd., Hiroshi Fujita at Mitsubishi Electric Corp. and Jun Iimura and Fumihiro Kumeno at MRI Inc.

References

[Bib86] Bibel, W., *Automated Theorem Proving,* Vieweg, 1986.

[Con86] Constable, R.L. et al, *Implementing Mathematics with the Nuprl Proof Development System,* Prentice-Hall, NJ, 1986.

[FH90] Fujita, H. and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using Ramified-Stack Algorithm, ICOT TR-606, 1990.

[Fuc90] Fuchi, K., Impression on KL1 programming – from my experience with writing parallel provers –, in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990 (in Japanese).

[Gir89] Girard, J.Y., et al *Proofs and Types,* Cambridge Tracts in Theoretical Computer Science, Vol 7, 1989.

[HFF90] Hasegawa, R., Fujita, H., Fujita M., A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis, in *Italy-Japan-Sweden Workshop '90,* ICOT TR-588, 1990.

[HN88] Hayashi, S., Nakano, H., *PX: A Computational Logic,* MIT Press, Cambridge, 1988.

[HHP87] Harper, R., Honsell, F., Plotkin, G., A Framework for Defining Logics, in *Symposium on Logic in Computer Science,* IEEE, 1987, pp194-204.

[Lov78] Lovelend, D.W., *Automated Theorem Proving: A Logical Basis,* North-Holland, 1978.

[Man80] Manna, Z. and Waldinger R., A deductive approach to program synthesis, in *ACM Trans. Programming Languages and Systems***2**(1), pp.91-121.

[Mar82] Martin-Löf P., Constructive mathematics and computer programming in *Proc. International Congress for Logic, Methodology and Philosophy of Science,* pp. 153-175, 1982.

[MB88] Manthey R. and Bry, F., SATCHMO: a theorem prover implemented in Prolog, in *Proc. of CADE 88, Argonne, illinois,* 1988.

[Ove90] Overbeek, R., private communication, 1990.

[Pfe88] Pfenning, F., Elf: A Language for Logic Definition and Verified Meta-Programming, in *Fourth Annual Symposium on Logic in ComputerScience,* IEEE, 1989 pp.313-322.

[Sch89] Schumann, J., SETHEO: User's Manual, Technical report, ATP-Report, Technische Universitat Munchen, 1989.

[Sti88] Stickel, M.E., A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, in *Journal of Automated Reasoning* 4 pp.353-380, 1988.

[Tak87] Takayama, Y., Writing Programs as QJ Proof and Compiling into Prolog Programs, in *Proc. of IEEE The Symposium on Logic Programming '87*, pp.278-287, 1987.

[Tra89] Traugott, J., Deductive System of Sorting Programs, in

[Wos88] Wos, L., *Automated Reasoning 33 Basic Research Problems* –, Prentice-Hall, 1988.