Constructive Methods in Automatic Analysis of Correctness Proofs *

Alessandro Avellone[1] Marco Benini[1] Dirk Nowotka[2]

[1] Dipartimento di Scienze dell'Informazione
Universitá degli Studi di Milano
via Comelico 39/41 — 20135, Milano, ITALY
[2] Turku Centre for Computer Science
Åbo Akademi University
Lemminkäisenkatu 14 A — 20520 Turku, FINLAND
{avellone,beninim}@dsi.unimi.it, dnowotka@abo.fi

Abstract. In this paper we develop an algorithm which permits to use the information content of a correctness proof Π for a program P to label P with assertions that provide the set of pre- and post-conditions which constitute the information used by the proof Π to establish the correctness of the program P.

1 Introduction

The goal of this contribution is to show that an instrument from Constructive Proof Theory, the *Collection Method* [MO79,MO81], can be used in practice to extract information from a correctness proof. In particular, we use such a method to automatically label the source code of a program with assertions that state the minimal pre- and postconditions which ensure the validity of the specification with respect to a given correctness proof.

Actually a prototypical version of the labelling algorithm has been implemented as a package for the Isabelle generic theorem prover [Pau94]. This package is designed to interface with other provers (the classical reasoner, the simplifier, ...) Isabelle provides; its design purpose was to build a *black box* which analyzes correctness proofs performed using a verification system [BKN98] built on the top of Isabelle .

The choice of a constructive instrument, in our case, the Collection Method, may appear as arbitrary. It is not: constructive proofs contain far more information than classical ones. The study of information extraction methods has a solid tradition in constructive logic [Kre76]; the most

* **Keywords:** Verification, Analysis

well-known instrument is normalization [Sch92]. Computer Science applications are mainly in the field of program synthesis: here, normalization-based techniques are successfully employed [BS95,BBS$^+$98], but it is a fact that more information than needed is produced when transforming a proof into a program. So the main effort is in reducing this extra amount of information [BBS$^+$98].

Our application is in the analysis of correctness proofs, so we have a fixed program we want to *maximize* the amount of information we can get about it. In this respect, as shown in [MO79], the Collection Method is more powerful than normalization-based techniques, since it extracts much more information. Of course, we want to filter this information, since every correctness proof contains parts with are relevant with respect to the program, while other part are accessory, and provide just logical or arithmetical lemmas.

With these premises, the outline of this paper is as follows:

- first, we introduce the relevant properties of Collection Method, that is, the algorithm to extract information from proofs;
- then we use this method to develop a procedure that is able to perform the labeling task mentioned above, and we apply it to a small example;
- in the appendix, we show the detailed construction which constitutes the Collection Method: we want to remark here that this part is not essential to understand the content of this article.

2 Extracting Information from Proofs

There are many ways to extract information from proofs, for example, see [Sch92,BS95,Tro73]. Generally speaking, the *information content* of a formal proof is the set of formulas which are proved to be true, either explicitly or implicitly, inside the proof development.

A natural way to perform this kind of extraction is by using the *Collection Method*. It was developed to synthesize programs from proofs [MO81,MO79,MMO88] and to show that a logic (or theory) is constructive [Fer97,MMO89,Ben98].

The Collection Method takes a set I of proofs as input, and generates a set, $\mathrm{Coll}^*(I)$, as output, which contains the information content of I. In our application I will be the singleton set composed by the correctness proof, and $\mathrm{Coll}^*(I)$ will be the set the labelling algorithm operates on. The definition of the Collection Method is given in Appendix A; the details on how it generates the information content are not relevant for our applications, but some of its properties are important:

1. it is not too difficult to encode the Collection Method as a package for a logical framework, like ISABELLE ;
2. the set $\mathrm{Coll}^*(I)$ can be represented as a lazy list of formulas, so it can be used (in particular, by our labelling algorithm) before it is completely generated; this fact is relevant, since the information content of a proof can be infinite;
3. we have a solid theoretical description for the quality of the extracted information [Fer97]; in particular we are able to show that it enjoys some interesting minimality properties, and that, when the theory where the given proofs are developed, is strongly constructive, the inner structure of the set $\mathrm{Coll}^*(I)$ is constructively closed (i.e., it is a pseudo-truth set).

The first property means that extracting information from a correctness proof is a feasible operation; the second property states that the procedure can be coded so to provide information in stages, making possible to incrementally generating the results of the labelling procedure.

The third property says that our labelling procedure can guarantee an high theoretical quality for the information it produces.

3 Analysis of Correctness Proofs

As we pointed out in the introduction, the goal of this paper is to show how we can extract information from a correctness proof Π for a program P in order to label P with assertions that state what happens inside.

Since our verification environment is for object code programs, we give a specialized version of the labelling algorithm here. It is possible to define a general version, but this is beyond the scope of this work.

We assume that the program P we are analyzing is coded into a proper logical representation Rep_P. We remark that in our verification framework this translation is generated automatically. The format of the representation can be inferred from the example we show.

Definition 1 (Labeling Procedure). *Let* $\prod_{\mathrm{Spec}_P}^{\mathrm{Rep}_P}$ *be a correctness proof for the program* P*, coded in a logical form as* Rep_P*, with respect to the specification* Spec_P*; let* $\mathcal{C} = \mathrm{Coll}^*\left(\{\prod_{\mathrm{Spec}_P}^{\mathrm{Rep}_P}\}\right)$.

We define

$$\begin{aligned}\mathcal{L}_i = \{\exists x.A \mid (\exists x.\mathrm{pc}(x) = i+1 \wedge A) \in \mathcal{C}\} \cup \\ \cup \{\forall x.A \mid (\forall x.\mathrm{pc}(x) = i+1 \rightarrow A) \in \mathcal{C}\}\ ,\end{aligned}$$

```
1: MOVE #−1, d1
2: ADD  #1, d1
3: MOVE d1, d2
4: ADD  d2, d2
5: CMP  d2, d0
6: BGE  2
7: SUB  #1, d1
```

Fig. 1. Our small example program

$$
\begin{aligned}
&\mathrm{Rep} \equiv \mathrm{pc}(0) = 1 \wedge 0 \leq \mathrm{d}_0(0) \wedge (\forall t. I_1(t) \wedge \ldots I_8(t)) \wedge (\forall t. \mathrm{pc}(t) = 1 \vee \ldots \vee \mathrm{pc}(t) = 8)\\
&I_1(t) \equiv \mathrm{pc}(t) = 1 \rightarrow \mathrm{pc}(t+1) = 2 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = -1\\
&I_2(t) \equiv \mathrm{pc}(t) = 2 \rightarrow \mathrm{pc}(t+1) = 3 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = 1 + \mathrm{d}_1(t)\\
&I_3(t) \equiv \mathrm{pc}(t) = 3 \rightarrow \mathrm{pc}(t+1) = 4 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = \mathrm{d}_1(t) \wedge\\
&\qquad\qquad \wedge\, \mathrm{d}_2(t+1) = \mathrm{d}_1(t)\\
&I_4(t) \equiv \mathrm{pc}(t) = 4 \rightarrow \mathrm{pc}(t+1) = 5 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = \mathrm{d}_1(t) \wedge\\
&\qquad\qquad \wedge\, \mathrm{d}_2(t+1) = 2\mathrm{d}_2(t)\\
&I_5(t) \equiv \mathrm{pc}(t) = 5 \rightarrow \mathrm{pc}(t+1) = 6 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = \mathrm{d}_1(t) \wedge\\
&\qquad\qquad \wedge\, (\mathrm{N}(t+1) \leftrightarrow \mathrm{d}_1(t) \leq \mathrm{d}_0(t))\\
&I_6(t) \equiv \mathrm{pc}(t) = 6 \rightarrow (\mathrm{N}(t) \rightarrow \mathrm{pc}(t+1) = 2) \wedge (\neg \mathrm{N}(t) \rightarrow \mathrm{pc}(t+1) = 7) \wedge\\
&\qquad\qquad \wedge\, \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = \mathrm{d}_1(t)\\
&I_7(t) \equiv \mathrm{pc}(t) = 7 \rightarrow \mathrm{pc}(t+1) = 8 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = \mathrm{d}_1(t) - 1\\
&I_8(t) \equiv \mathrm{pc}(t) = 8 \rightarrow \mathrm{pc}(t+1) = 8 \wedge \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \mathrm{d}_1(t+1) = \mathrm{d}_1(t)
\end{aligned}
$$

Fig. 2. The logical representation for the example program

for every instruction (referred to by its position i inside the program code).

We define $\mathcal{G} = \{\forall x.A \mid \forall x.A \in \mathcal{C}\} \setminus \bigcup_i \mathcal{L}_i$.

Intuitively $\mathcal{L}_i$ is a set of assertions which holds on the i^{th} position (line) of the source code, while $\mathcal{G}$ contains a set of facts which are true everywhere in the program.

Consider the assembly code program in Figure 1: given a natural number in register d_0, it is divided by 2 and the result is returned in register d_1. Formally:

$$\mathrm{Spec} \equiv \exists t. \mathrm{pc}(t) = 8 \wedge (\mathrm{d}_0(0) = 2\mathrm{d}_1(t) \vee \mathrm{d}_0(0) = 2\mathrm{d}_1(t) + 1)$$

The logical representation for this program is shown in Figure 2.

The complete correctness proof can be easily performed in first-order classical arithmetic. The essential schema for the correctness proof in natural deduction can be found in Figure 3.

$$
\begin{aligned}
A \equiv\ & \mathrm{pc}(t) = 2 \rightarrow (2(1 + \mathrm{d}_1(t)) \leq \mathrm{d}_0(t) \rightarrow \mathrm{pc}(t+5) = 2) \wedge \\
& \wedge (\mathrm{d}_0(t) < 2(1 + \mathrm{d}_1(t)) \rightarrow \mathrm{pc}(t+5) = 7) \wedge \\
& \wedge \mathrm{d}_1(t+5) = \mathrm{d}_1(t) \\
B \equiv\ & \forall t.\mathrm{d}_0(t) = \mathrm{d}_0(0) \\
C \equiv\ & \exists t.\mathrm{pc}(t) = 2 \wedge \mathrm{d}_0(t) < 2(1 + \mathrm{d}_1(t)) \\
D \equiv\ & \forall t.\mathrm{pc}(t) = 2 \rightarrow 2\mathrm{d}_1(t) \leq \mathrm{d}_0(t)
\end{aligned}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A}\,\mathrm{Rep} \quad \dfrac{}{B}\,\mathrm{Rep} \quad \dfrac{\dfrac{}{A}\,\mathrm{Rep}}{C} \quad \dfrac{\dfrac{}{B}\,\mathrm{Rep}}{D}}{\exists t.2(\mathrm{d}_1(t) - 1) \leq \mathrm{d}_0(0) \leq 2\mathrm{d}_1(t) - 1 \wedge \mathrm{pc}(t) = 7}}{\mathrm{Rep}}}{\mathrm{Spec}}\,\exists\mathrm{E}
$$

Fig. 3. The schema of the correctness proof for our example

If we apply the Collection Method to this correctness proof (for the sake of brevity, we restrict our extraction to the schema) and we calculate the sets for our labeling procedure, we get $\mathcal{G} = \{\forall t.\mathrm{d}_0(t) = \mathrm{d}_0(0)\}$ as the set of facts which are true in the whole program, and we can label our program as in Figure 4, using the sets $\mathcal{L}_i$ whose values are:

$$
\begin{aligned}
& \mathcal{L}_0 = \{0 \leq \mathrm{d}_0(0)\} \\
& \mathcal{L}_1 = \{\exists t.\mathrm{d}_0(t) < 2(\mathrm{d}_1(t) + 1), \forall t.2\mathrm{d}_1(t) < \mathrm{d}_0(t)\} \\
\mathcal{L}_2 = \mathcal{L}_3 = \mathcal{L}_4 = & \mathcal{L}_5 = \emptyset \\
& \mathcal{L}_6 = \{\exists t.2(\mathrm{d}_1(t) - 1) \leq \mathrm{d}_0(0) \leq 2\mathrm{d}_1(t) - 1\} \\
& \mathcal{L}_7 = \{\mathrm{Spec}\}
\end{aligned}
$$

Since every correctness proof must establish results on the structure of the code, an algorithm which is able to extract information from a proof must collect these results, as well. In the case of object code it is easy to understand where in the program this information is true: we use the program counter register.

In a more general setting the same labeling could be performed, by using an implicational representation (like ours) which has in the antecedent enough information in order to identify in an unique way a position inside the program code.

$$
\begin{array}{ll}
: & \mathcal{L}_0 = \{0 \leq \mathrm{d}_0(0)\} \\
1: & \text{MOVE } \#-1, \mathrm{d}_1 \\
: & \mathcal{L}_1 = \{\exists t.\mathrm{d}_0(t) < 2(\mathrm{d}_1(t)+1), \forall t.2\mathrm{d}_1(t) < \mathrm{d}_0(t)\} \\
2: & \text{ADD } \quad \#1, \mathrm{d}_1 \\
3: & \text{MOVE } \mathrm{d}_1, \mathrm{d}_2 \\
4: & \text{ADD } \quad \mathrm{d}_2, \mathrm{d}_2 \\
5: & \text{CMP } \quad \mathrm{d}_2, \mathrm{d}_0 \\
6: & \text{BGE } \quad 2 \\
: & \mathcal{L}_6 = \{\exists t.2(\mathrm{d}_1(t)-1) \leq \mathrm{d}_0(0) \leq 2\mathrm{d}_1(t)-1\} \\
7: & \text{SUB } \quad \#1, \mathrm{d}_1 \\
: & \mathcal{L}_7 = \{\exists t.\mathrm{d}_0(0) = 2\mathrm{d}_1(t) \vee \mathrm{d}_0(0) = 2\mathrm{d}_1(t)+1\}
\end{array}
$$

Fig. 4. The result of the labeling algorithm on our example program

4 Conclusions

In this paper we have shown how it is possible to automatically extract useful information from a correctness proof of an object code program.

There are many other possible applications of the Collection Method to the automatic analysis of programs, and we think that the labeling algorithm we discussed in this work is just one of the results which could be drawn from this line of research. Anyway, even this single result is very useful for the following applications:

- Reusing software requires a precise description of the behavior of the code and our labeling algorithm provides it;
- Tracking changes in a formally verified program is difficult because one does not know the impact they have on the correctness proof, but our labeling algorithm creates a link between the proof and the code, so we are able to guess what should be changed in the correctness proof, when we modify the code;
- Convincing a non-expert about the validity of a correctness proof is hard since it is huge and difficult to understand, but our labeling algorithm, because of the way it works, provides information a programmer can check, and this information is present in the correctness proof, giving empirical evidence of the validity of the proof.

References

[BBS+98] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolgang Zuber. Proof theory at work: Program development in the

MINLOG system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications, Volume II: Systems and Implementation Techniques*, pages 41–71. Kluwer Academic Publishers, 1998.

[Ben98] Marco Benini. The collection method in second-order intuitionistic logic. Presented during the Conference *Logic Colloquium '98*. Available at `ftp://dotto.usr.dsi.unimi.it/~benini/coll2.ps.gz`, 1998.

[BKN98] Marco Benini, Sara Kalvala, and Dirk Nowotka. Program abstraction in a temporal logic framework. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, 1998.

[BS95] Ulrich Berger and Helmut Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC'94, Indianapolis, IN, USA, October 1994*, volume 960 of *Lecture Notes in Computer Science*, pages 77–97. Springer-Verlag, 1995.

[Fer97] Mauro Ferrari. *Strongly Constructive Formal Systems*. PhD thesis, Departement of Computer Science — University of Milano, 1997. Available at `http://dotto.usr.dsi.unimi.it/~ferram/thesis.ps.gz`.

[Kre76] Georg Kreisel. Some uses of proof theory in finding computer programs. Notes for a talk in the Logical Symposium of Clermond Ferrand, 1976.

[MMO88] Pierangelo Miglioli, Ugo Moscato, and Mario Ornaghi. Constructive theories with abstract data types for program synthesis. In D. Skordev, editor, *Mathematical Logic and its Applications*, pages 293–302. Plenum Press, New York, 1988.

[MMO89] Pierangelo Miglioli, Ugo Moscato, and Mario Ornaghi. Semi-constructive formal systems and axiomatization of abstract data types. In J. Diaz and F. Orejas, editors, *TAPSOFT '89*, Lecture Notes in Computer Science, pages 337–351. Springer-Verlag, 1989.

[MO79] Pierangelo Miglioli and Mario Ornaghi. A purely logical computing model: the open proofs as programs. Technical Report MIG-7, Istituto di Cibernetica – University of Milano, 1979.

[MO81] Pierangelo Miglioli and Mario Ornaghi. A logically justified model of computation. *Fundamenta Informaticae*, IV((1,2)):151–172, 277–341, 1981.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Sch92] Helmut Schwichtenberg. Proof as programs. In *Proof Theory: A Selection of Papers from the Leeds Proof Theory Programme 1990*. Cambridge University Press, 1992.

[Tro73] Anne S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.

[TvD88] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics: An Introduction I and II*, volume 121, 122 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1988.

A The Collection Method

Now we show the formal presentation of this method, by working within first-order intuitionistic arithmetic [TvD88,Tro73].

Definition 2 (Closed Proof). *A proof is closed if and only if all its free variables are parameters (eigenvariables) of inference rules.*

Definition 3 ($\prec$ Relation). *We write* $\prod_{A}^{\Gamma} \prec \mathcal{I}$ *to indicate that* $\prod_{A}^{\Gamma}$ *is a closed subproof of a proof in the set* $\mathcal{I}$.

The goal of the Collection Method is to provide a set of formulas, derived by analyzing the proofs we give as an input. This set will be named $\mathrm{Coll}^{*}(\mathcal{I})$ where $\mathcal{I}$ is the set containing our input proofs.

Definition 4 (Coll Operator). *Let* $\mathcal{I}$ *be a finite set of closed proofs;* $\mathrm{Coll}(\mathcal{I})$ *is the least set such that, if* $\prod_{A}^{\Gamma} \prec \mathcal{I}$ *and* $\Gamma \subseteq \mathrm{Coll}(\mathcal{I})$, *then* $A \in \mathrm{Coll}(\mathcal{I})$.

We remark that, by an easy argument one can show that there is an inductive definition for this set, and that one can use such a definition to generate in a lazy way $\mathrm{Coll}(\mathcal{I})$.

Definition 5 ($\forall\mathrm{I}-\mathrm{Subst}$ Operator). *Let* $\mathcal{I}$ *be any finite set of closed proofs. We define* $\forall\mathrm{I}{-}\mathrm{Subst}(\mathcal{I})$ *as the smallest set such that, if*

$$\frac{\prod_{A(p)}^{\Gamma(p)}}{\forall x.A(x)}\,\forall\mathrm{I} \prec \mathcal{I} \ ,$$

where $\Gamma \subseteq \mathrm{Coll}(\mathcal{I})$, *and there is a term* t *such that* $A(t) \in \mathrm{Coll}(\mathcal{I})$, *then*

$$\prod_{A(p)}^{\Gamma}{}_{(p:=t)} \in \forall\mathrm{I}{-}\mathrm{Subst}(\mathcal{I}) \ .$$

Definition 6 ($\exists\mathrm{E}-\mathrm{Subst}$ Operator). *Let* $\mathcal{I}$ *be any finite set of closed proofs. We define* $\exists\mathrm{E}{-}\mathrm{Subst}(\mathcal{I})$ *as the smallest set such that, if*

$$\frac{\prod_{\exists x.D(x)}^{\Gamma}{}_{1} \quad \prod_{A}^{\Gamma,D(p)}{}_{2}(p)}{A}\,\exists\mathrm{E} \prec \mathcal{I} \ ,$$

where $\Gamma \subseteq \mathrm{Coll}(\mathcal{I})$, A *is a closed formula, and there is a term* t *such that* $D(t) \in \mathrm{Coll}(\mathcal{I})$, *then*

$$\prod_{A}^{\Gamma,D(p)}{}_{2}{}_{(p:=t)} \in \exists\mathrm{E}{-}\mathrm{Subst}(\mathcal{I}) \ .$$

Definition 7 ($\mathrm{Ind}-\mathrm{Subst}$ **Operator).** *Let* $\mathcal{I}$ *be a finite set of closed proofs. We define* $\mathrm{Ind}-\mathrm{Subst}\left(\mathcal{I}\right)$ *as the smallest set such that, if*

$$\frac{\prod\limits_{A(0)}^{\Gamma}{}_1 \quad \prod\limits_{A(\mathrm{Suc}(p))}^{\Gamma,A(p)}{}_2\left(p\right)}{A\left(t\right)}\,\mathrm{Ind} \prec \mathcal{I} \ ,$$

where $\Gamma \subseteq \mathrm{Coll}\left(\mathcal{I}\right)$*, and* $A\left(t\right)$ *is a closed formula, then, there is a value* i *such that* $t = \mathrm{Suc}^{i}\left(0\right)$ *so we put*

$$\prod_{\mathrm{Suc}^{i}(0)=t} \ , \ \textit{and} \quad \prod_{A(t)}^{\mathrm{Suc}^{i}(0)=t,A\left(\mathrm{Suc}^{i}(0)\right)}$$

in $\mathrm{Ind}-\mathrm{Subst}\left(\mathcal{I}\right)$*, and, for all* j*,* $0 \leq j < i$*,*

$$\prod_{A(\mathrm{Suc}(p))}^{\Gamma,A(p)}{}_2\left(p:=\mathrm{Suc}^{j}\left(0\right)\right) \in \mathrm{Ind}-\mathrm{Subst}\left(\mathcal{I}\right) \ .$$

Definition 8 (Exp **Operator).** *Given* $\mathcal{I}$*, a finite set of closed proofs, its expansion is defined as*

$$\mathrm{Exp}\left(\mathcal{I}\right) = \mathcal{I} \cup \forall\mathrm{I}{-}\mathrm{Subst}\left(\mathcal{I}\right) \cup \exists\mathrm{E}{-}\mathrm{Subst}\left(\mathcal{I}\right) \cup \mathrm{Ind}-\mathrm{Subst}\left(\mathcal{I}\right) \ .$$

Definition 9 (Exp^{*} **and** Coll^{*} **Operators).** *Let* $\mathcal{I}$ *be any finite set of closed proofs. We define*

$$\begin{aligned} \mathrm{Exp}^{0}\left(\mathcal{I}\right) &= \mathcal{I} \\ \mathrm{Exp}^{i+1}\left(\mathcal{I}\right) &= \mathrm{Exp}\left(\mathrm{Exp}^{i}\left(\mathcal{I}\right)\right) \ . \end{aligned}$$

We call $\mathrm{Exp}^{*}\left(\mathcal{I}\right)$ *the closure on the expansion operation over* $\mathcal{I}$*; formally*

$$\mathrm{Exp}^{*}\left(\mathcal{I}\right) = \bigcup_{i\in\omega} \mathrm{Exp}^{i}\left(\mathcal{I}\right) \ .$$

The collection operator applied to $\mathrm{Exp}^{*}\left(\mathcal{I}\right)$ *gives us* $\mathrm{Coll}^{*}\left(\mathcal{I}\right)$*; formally*

$$\mathrm{Coll}^{*}\left(\mathcal{I}\right) = \bigcup_{i\in\omega} \mathrm{Coll}\left(\mathrm{Exp}^{i}\left(\mathcal{I}\right)\right) \ .$$

We note that both $\mathrm{Exp}^{*}\left(\mathcal{I}\right)$ *and* $\mathrm{Coll}^{*}\left(\mathcal{I}\right)$ *are recursively enumerable, but, in general, they are not finite nor recursive.*

We remark that both the sets $\mathrm{Exp}^*(\mathcal{I})$ and $\mathrm{Coll}^*(\mathcal{I})$ can be generated in a lazy way, since they are monotone with respect to set inclusion, and inductively defined.

We do not show all the properties which this method enjoys, since their proofs are long and beyond the scope of this work. A more comprehensive discussion of the method along with many derived results can be found in [Fer97,MMO89,Ben98].

The only property which is relevant to our application is stated in Theorem 1. This result defines what we mean when we say that "the information content as generated by the collection procedure, is constructively complete".

Definition 10 (Pseudo-Truth Set). *A set $\mathcal{F}$ of formulas is a pseudo-truth set if and only if*

- $A \in \mathcal{F}$ *implies* $\mathrm{Arith} \vdash A$.
- $A \in \mathcal{F}$ *implies* A *is closed.*
- $A \vee B \in \mathcal{F}$ *implies* $A \in \mathcal{F}$ *or* $B \in \mathcal{F}$.
- $A \wedge B \in \mathcal{F}$ *implies* $A \in \mathcal{F}$ *and* $B \in \mathcal{F}$.
- $A \rightarrow B \in \mathcal{F}$ *and* $A \in \mathcal{F}$ *implies* $B \in \mathcal{F}$.
- $\exists x.A(x) \in \mathcal{F}$ *implies that there is a term* t, *such that* $A(t) \in \mathcal{F}$.

Theorem 1. *Let $\mathcal{I}$ be a finite set of closed proofs, then* $\mathrm{Coll}^*(\mathcal{I})$ *is a pseudo-truth set.*

If we use classical arithmetic, Theorem 1 does not hold anymore, since this theory is not constructive. But still the Collection Method can be used to extract information from proofs, but one does not know in advance to what extent, i.e., no theoretical measure for the quality of the extracted information is available.