Minlog - A Tool for Program Extraction Supporting Algebras and Coalgebras

Ulrich Berger[1], Kenji Miyamoto[⋆2], Helmut Schwichtenberg[2], and Monika Seisenberger[1]

[1] Swansea University, Wales
[2] LMU University, Munich

Abstract. Minlog is an interactive system which implements proof-theoretic methods and applies them to verification and program extraction. We give an overview of Minlog and demonstrate how it can be used to exploit the computational content in (co)algebraic proofs and to develop correct and efficient programs. We illustrate this by means of two examples: one about parsing, the other about exact real numbers in signed digit representation.

1 Introduction

In this paper we give an overview of the interactive proof system Minlog and describe a proof-theoretic method based on realizability for developing correct programs. We particularly address inductive and coinductive proofs and the associated computation principles: iteration for initial algebras and coiteration for terminal coalgebras. Minlog is not a type-theoretic system, such as Coq or Isabelle, but based on first-order logic and has a simple mathematical (i.e. denotational) semantics. This makes it accessible to a wide range of researchers including those outside the type-theoretically minded community. Minlog is implemented in Scheme. It is an "open" system giving users full access to the code, thus inviting them to contribute to its development. Although designed as a general purpose system, most of the recent developments in Minlog are concerned with program extraction from proofs. It seems fair to say that, regarding program extraction, Minlog is the most advanced proof system. Minlog implements various methods of program extraction (realizability, Dialectica Interpretation) and extends them to classical proofs via the Friedman/Dragalin A-translation. All these techniques are refined and optimized in order improve usability and to obtain simpler programs. In addition to extracting a program from a proof, Minlog also automatically extracts a proof that the program meets its specification. In Sect. 2 we give a more detailed and technical description of Minlog and its program extraction facilities. A number of substantial case studies on program extraction have been carried out in Minlog reaching from the extraction of a normalisation-by-evaluation algorithm to the extraction of programs in

⋆ Supported by the Marie Curie Initial Training Network in Mathematical Logic – MALOA – From MAthematical LOgic to Applications, PITN-GA-2009-238381 constructive analysis. In this paper we present two case studies demonstrating the use of inductive and coinductive definitions in program extraction (Sect. 3) and show that they work well with Minlog's optimized extraction mechanism.

2 Program extraction in Minlog

2.1 The Interactive Proof System Minlog

Minlog [Min,BBS^{+}98] is an interactive proof system based on first order natural deduction calculus. It is intended to reason about higher type computable functionals, using minimal rather than classical or intuitionistic logic. Minlog implements a *theory of computable functionals*, as described in [SW11]. The underlying semantics is the Scott-Ershov model of partial continuous functionals, with *free algebras* as base types. These algebras are viewed as domains represented by Scott's information systems, whose tokens are constructor trees possibly involving the symbol $*$ ("no information"). The ideals (points, objects) of base type are consistent and deductively closed sets of tokens, possibly infinite. *Initial algebras* and *final coalgebras* are modelled by notions of totality and cototality: An ideal x is "cototal" if every constructor tree $P(*) \in x$ has a "one-step extension" $P(\mathrm{C}\boldsymbol{a}^{\boldsymbol{*}}) \in x$, and "total" if it is cototal and this extension relation is well-founded. Totality and cototality are instances of strictly positive inductive and coinductive definitions which are supported in general in Minlog. With every initial algebra and final coalgebra are associated operators for (co)iteration and (co)recursion. Computation is implemented efficiently via *normalization by evaluation* [BS91,BES03]. Computation is extended to proofs via the Curry-Howard correspondence. *Intuitionistic* and *classical logic* are represented by the axiom schemes $\bot \to A$ and $\neg\neg A \to A$. *Interactive proofs* are organized in a goal-directed backwards-reasoning fashion. Forward reasoning is modelled by a form of cut rule. Minlog also contains an *automated prover* for a certain fragment of (simply typed) minimal logic. Its theory (based on [Mil91]) is developed in [Sch04].

2.2 Program extraction

One of the main motivations behind Minlog is to use it as a tool for program verification, and to exploit the proofs-as-programs paradigm for program development. Minlog's program extraction is based on Kreisel's modified realizability [Kre59]: to each formula A a type $\tau(A)$ (the type of realizers of A) and a formula $x^{\tau(A)}\ \mathbf{r}\ A$ are assigned. The formula $x\ \mathbf{r}\ A$ is to be read as "x realizes A" and intuitively means "x solves the computational problem expressed by A". Program extraction computes from a derivation d of A a term $\mathrm{et}(d)$ (the extracted program) and a proof that $\mathrm{et}(d)$ realizes A. Of particular interest are the realization of induction and coinduction by algebras and coalgebras, as well as the computational and non-computational versions of logical operators. The latter are crucial for obtaining practically useful results as they lead to drastically simplified proofs and extracted programs. The case studies on parsing (3.1) and exact real numbers (3.2) below will highlight these points. From a type-theoretic point of view, realizability collapses a dependently typed lambda-calculus (which Minlog's proof calculus is an instance of) to a simply typed lambda-calculus. The collapse happens on the level of atomic formulas, since $\tau(P\boldsymbol{t}) = \tau(P)$ where $\tau(P)$ is a simple type assigned to the predicate P, discarding the first-order terms $\boldsymbol{t}$. Minlog extends program extraction to classical proofs via a refined A-translation [BBS02] or Dialectica Interpretation (see eg. [SW11]). The theoretical foundations of program extraction from formal proofs, as implemented in Minlog, are presented in detail in [SW11]. Realizability for induction and coinduction, including applications to exact real numbers, are developed in [Ber09,BS10].

2.3 Related Work

Program extraction from proofs is also implemented in Isabelle [Isa] (for algebras) and in Coq [Coq] (cf [BBLS06] for a joint case study). The implementation in Isabelle has been modelled after Minlog's extraction. The correctness of program extraction in Coq is not based on realizability and a Soundness Theorem (as in Minlog), but on the fact that reduction of proofs is correctly simulated by reduction of extracted programs [Let03]. There exists also an experimental implementation of program extraction in Agda [Agd] (cf [Chu11]). We are not aware of substantial case studies on program extraction in these systems. Also, Minlog seems to be the only system implementing the Dialectica Interpretation and program extraction from classical proofs. We also mention RZ [BS07], a tool that computes the realizability interpretation of a mathematical statement (but does not extract programs from proofs).

3 Case studies

3.1 Algebras for parsing

Consider strings x, y of left and right parentheses L, R. We define inductively the predicate (grammar) S of balanced strings of parentheses by the clauses

$$\text{InitS}\colon S(\text{nil}), \qquad \text{ApS}\colon Sx \to Sy \to S(xy), \qquad \text{ParS}\colon Sx \to S(LxR).$$

The type $\tau(S)$ of realizers of S is the algebra `algS` of generation trees for S. It has one nullary constructor, cInitS, one binary, cApS, and one unary, cParS. Our goal is to prove decidability of S, i.e. $\forall_x(Sx \vee \neg Sx)$, and extract from the proof a program computing for each string x a boolean value p and a tree t: `algS` such that p decides whether Sx holds and, in the positive case, t is a generation tree for x. Note that negation, $\neg A$, is expressed in Minlog as $A \to \bot$ and disjunction, $A \vee B$, as $\exists_p((p \to A) \wedge (\neg p \to B))$. Hence the goal reads $\forall_x \exists_p((p \to Sx) \wedge ((p \to \bot) \to Sx \to \bot))$. We also consider an alternative grammar U for the same set of strings (which, in contrast to S, is deterministic)

$$\text{InitU}\colon U(\text{nil}), \qquad \text{ApU}\colon Ux \to Uy \to U(xLyR),$$

with an algebra of realizers `algU` with constructors `cInitU` (nullary) and `cApU` (binary). Equality of U and S is expressed by $\forall_x^{\mathrm{nc}}(Ux \to Sx)$ and $\forall_x^{\mathrm{nc}}(Sx \to Ux)$. These formulas can be easily proven by induction on Ux and Sx. The non-computational universal quantifier $\forall_x^{\mathrm{nc}}$ has the same logical meaning as the usual $\forall_x$, but it indicates that the extracted programs only operate on the generation trees for x and not on the string x itself. The variables x, y in the defining clauses for S and U are implicitly quantified by $\forall^{\mathrm{nc}}$ as well. The extracted program for the proof of $\forall_x^{\mathrm{nc}}(Sx \to Ux)$ is ([`b0`] and [`b1`, ...] denote lambda-abstractions)

```
[b0](Rec algS=>algU)b0 cInitU
([b1,b2,a3,a4](Rec algU=>algU)a4 a3([a5,a6,a7,a8]cApU a7 a6))
([b1]cApU cInitU)
```

The fact that the proof is by induction on Sx is witnessed by the occurrence of the recursion operator `(Rec algS=>algU)` implementing (an instance of) structural recursion on `algS`. There is also a side induction witnessed by `(Rec algU=>algU)`. The term above is equivalent to a program `SU` defined by the recursive equations

```
SU cInitS       = cInitU
SU (cApS b1 b2) = UU (SU b2) where
    UU cInitU       = SU b1
    UU (cApU a5 a6) = cApU (UU a5) a6
SU (CParS b)    = cApU cInitU (SU b)
```

The boolean value deciding whether or not Sx holds is computed as `Test 0 x`, the function `Test` being defined as a constant `Test` (`py` means parse-type):

```
(add-program-constant "Test" (py "nat=>list par=>boole"))
(add-computation-rules
 "Test 0(Nil par)"        "True"
 "Test 0(R::x)"           "False"
 "Test(Succ n)(Nil par)"  "False"
 "Test n(L::x)"           "Test(Succ n)x"
 "Test(Succ n)(R::x)"     "Test n x")
```

Note that (`Nil par`) denotes the empty list of parentheses and :: is the cons operation for lists. Soundness, $\forall_x^{\mathrm{nc}}(Ux \to \mathrm{Test}(0,x))$, is easy and the proof has no computational content. Completeness, $\forall_x(\mathrm{Test}(0,x) \to Sx)$, needs an inductively defined predicate State with clauses

$$\mathrm{InitState}\colon \mathrm{State}(0,\mathrm{nil}), \quad \mathrm{ApState}\colon Ss \to \mathrm{State}(n,x) \to \mathrm{State}(\mathrm{S}n, xsL)$$

and a lemma $\forall_y \forall_{n,x}^{\mathrm{nc}}(\mathrm{State}(n,x) \to \forall_s^{\mathrm{nc}}(Ss \to \mathrm{Test}(n,y) \to S(xsy)))$, proved by induction on y. Now $\forall_x \exists_p((p \to Sx) \wedge ((p \to \bot) \to Sx \to \bot))$ is proved easily[3]. The `parser-term` extracted from this proof is (`@` is a pairing operator in infix notation, [`if` ...] is a generalization of the usual if-then-else allowing pattern matching on the constructors of an algebra)

[3] The third author is grateful to Makoto Takeyama for explaining his Agda code `http://code.haskell.org/Agda/examples/ParenDepTac.agda` to him.

```
[x0]Test 0 x0 @
 (Rec list par=>algState=>algS=>algS)x0
 ([st1,b2][if st1 b2 ([b3,st4]cInitS)])
 ([par1,x2,f3,st4,b5]
   [if par1
     (f3(cApState b5 st4)cInitS)
     [if st4 cInitS ([b6,st7]f3 st7(cApS b6(cParS b5)))]])
 cInitState cInitS
```

which corresponds to the following recursive program:

```
P x0 = Test 0 x0 @ P0 x0 cInitState cInitS   where
   P0 Nil cInitState b2          = b2
   P0 Nil(cApState b3 st4)b2     = cInitS
   P0 (L::x2)st4 b5              = P0 x2(cApState b5 st4)cInitS
   P0 (R::x2)cInitState b5       = cInitS
   P0 (R::x2)(cApState b6 st7)b5 = P0 x2 st7(cApS b6(cParS b5))
```

Experiments (pp, nt, pt mean pretty-print, normalize-term, parse-term - Four inputs to Minlog, followed by Minlog's response):

```
(pp (nt (mk-term-in-app-form parser-term (pt "L::R:"))))
"True@cApS cInitS(cParS cInitS)"
(pp (nt (mk-term-in-app-form parser-term (pt "R::L:"))))
"False@cInitS"
(pp (nt (mk-term-in-app-form parser-term (pt "L::R::L::R:"))))
"True@cApS(cApS cInitS(cParS cInitS))(cParS cInitS)"
(pp (nt (mk-term-in-app-form parser-term (pt "L::L::R::R:"))))
"True@cApS cInitS(cParS(cApS cInitS(cParS cInitS)))"
```

3.2 Coalgebras for exact real numbers

Our second case study concerns algorithms in exact real arithmetic. Whilst such algorithms have been *verified* before (see eg. [CDG06,MRE07,GNSW07,BH08]), in the present paper we show by means of an example how to *extract* them. We extract a program which for every rational number $a \in [-1, 1]$ computes a signed binary digit representation, that is, a (finite or infinite) stream of digits $d_0, d_1, \ldots \in \{-1, 0, 1\}$ such that

$$a = \sum_i \frac{d_i}{2^{i+1}} \tag{1}$$

We let a range over abstract real numbers (we only use the properties of an ordered field) and let Qa mean that a is a rational number with absolute value ≤ 1. Our program will be extracted from a proof of the formula

$$\forall_a^{\mathrm{nc}}(Qa \to Ja) \tag{2}$$

where the predicate J is defined coinductively by the clause

$$\forall_a^{\mathrm{nc}}(Ja \to a = 0 \lor \exists_b^{\mathrm{r}}(a = \frac{b-1}{2} \land Jb) \lor \exists_b^{\mathrm{r}}(a = \frac{b}{2} \land Jb) \lor \exists_b^{\mathrm{r}}(a = \frac{b+1}{2} \land Jb)) \quad (3)$$

that is, J is the largest predicate satisfying (3). The proof of (2) proceeds by coinduction, that is, by showing that (3) holds when J is replaced by Q. The superscript `r` attached to the quantifier $\exists^{\mathrm{r}}$ stands for "right" and means that from a proof of a formula $\exists_b^{\mathrm{r}} A$ only the realizer of A is kept while the witness b contained in the proof is discarded. The type of realizers for J is the coalgebra of finite and infinite streams of signed digits. In our setting it is modelled as the set of cototal ideals (see Sect. 2.1) of the algebra $\mathbf{I}$ of "standard rational intervals", whose constructors are $\mathbb{I}$ (for the initial interval $[-1, 1]$) and $\mathrm{C}_{-1}, \mathrm{C}_0, \mathrm{C}_1$ (for the left, middle, right part of the argument interval, of half its length). For example, $\mathrm{C}_{-1}\mathbb{I}$, $\mathrm{C}_0\mathbb{I}$ and $\mathrm{C}_1\mathbb{I}$ should be viewed as the intervals $[-1, 0]$, $[-\frac{1}{2}, \frac{1}{2}]$ and $[0, 1]$. The cototal ideals include, for example, $\{\mathrm{C}_{-1}^n *\}_{n\geq 0}$, a "stream" representation of the real -1, and also $\{\mathrm{C}_{-1}\mathrm{C}_1^n *\}_{n\geq 0}$ and $\{\mathrm{C}_1\mathrm{C}_{-1}^n *\}_{n\geq 0}$, which both represent the real 0. Generally, the cototal ideals give us all reals in $[-1, 1]$, in the representation (1). We have formalized in Minlog a proof of (2) and extracted from it a term `neterm` of type $\iota \to \mathbf{I}$ involving the corecursion operator ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{I}}^{\iota}$ associated with (3). The value of the term obtained by applying `neterm` to, say, 1/2 (in Minlog: `cGenQ(1#2)`) is an infinite ideal starting with $\mathrm{C}_1, \mathrm{C}_0, \mathrm{C}_0, \mathrm{C}_0 \ldots$ (`CIntP`, `CintZ`, ...). To compute it (again via normalization-by-evaluation, i.e., `nt`) we delay unfolding ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{I}}^{\iota}$ at a fixed depth, say 5:

```
(pp (nt (undelay-delayed-corec
 (mk-term-in-app-form neterm (pt "cGenQ(1#2)")) 5)))
"CIntP (CIntZ (CIntZ (CIntZ (CIntZ (CoRec algQ=>intv)..."
```

Sources. The Minlog system is available at `www.minlog-system.de`. To run the examples download the latest version `minlog-latest.tar.gz` (SVN snapshot) and follow the installation instructions. Note that, as prerequisites, Scheme and Emacs are required. The reader new to Minlog is referred to the tutorial [CSS11] for introductory examples. The two case studies of this paper can be found in `examples/parsing/parens.scm` and `examples/analysis/ratsds.scm` together with readme files `readme-parens.txt and readme-ratsds.txt` explaining the background and how to run the case studies.

Acknowledgements. The authors would like to thank the referees for their constructive comments and criticism.

References

[Agd] Agda. `http://wiki.portal.chalmers.se/agda/`.

[BBLS06] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82:27–51, 2006.

[BBS$^+$98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction*, volume II of *Applied Logic Series*, pages 41–71. Kluwer, 1998.

[BBS02] U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *APAL*, 114:3–25, 2002.

[Ber09] U. Berger. From coinductive proofs to exact real arithmetic. In E. Grädel and R. Kahle, editors, *Computer Science Logic*, volume 5771 of *LNCS*, pages 132–146. Springer, 2009.

[BES03] U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183:19–42, 2003.

[BH08] U. Berger and T. Hou. Coinduction for exact real number computation. *Theory of Computing Systems*, 43:394–409, 2008.

[BS91] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proceedings 6'th Symposium on Logic in Computer Science, LICS'91*, pages 203–211. IEEE Computer Society Press, 1991.

[BS07] A. Bauer and C. A. Stone. RZ: A tool for bringing constructive and computable mathematics closer to programming practice. In *Computation and Logic in the Real World, CiE 2007*, volume 4497 of *LNCS*, pages 28–42, 2007.

[BS10] U. Berger and M. Seisenberger. Proofs, programs, processes. In F. Ferreira B. Löwe, E. Mayordomo, and L. Mendes Gomes, editors, *Programs, Proofs, Processes, CiE 2010*, volume 6158 of *LNCS*, pages 39–48, 2010.

[CDG06] A. Ciaffaglione and P. Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theor. Comp. Sci.*, 351:39–51, 2006.

[Chu11] C. M. Chuang. *Extraction of Programs for Exact Real Number Computation Using Agda.* PhD thesis, Swansea University, Wales, 2011.

[Coq] The Coq Proof Assistant. `http://coq.inria.fr/`.

[CSS11] L. Crosilla, M. Seisenberger, and H. Schwichtenberg. A Tutorial for Minlog, Version 5.0, 2011.

[GNSW07] H. Geuvers, M. Niqui, B. Spitters, and F. Wiedijk. Constructive analysis, types and exact real numbers. *Math. Struct. Comp. Sci*, 17(1):3–36, 2007.

[Isa] Isabelle. `http://isabelle.in.tum.de/`.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pages 101–128, 1959.

[Let03] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *LNCS*, 2003.

[Mil91] D. Miller. A logic programming language with lambda–abstraction, function variables and simple unification. *Jour. Logic Comput.*, 2(4):497–536, 1991.

[Min] The Minlog System. `http://www.minlog-system.de`.

[MRE07] J. R. Marcial-Romero and M. H. Escardo. Semantics of a sequential language for exact real-number computation. *Theor. Comp. Sci*, 379(1-2):120–141, 2007.

[Sch04] H. Schwichtenberg. Proof search in minimal logic. In B. Buchberger and J.A. Campbell, editors, *Artificial Intelligence and Symbolic Computation AISC 2004, Proceedings*, volume 3249 of *LNAI*, pages 15–25. Springer, 2004.

[SW11] H. Schwichtenberg and S. S. Wainer. *Proofs and Computations.* Perspectives in Logic. Assoc. Symb. Logic and Cambridge Univ. Press, to appear, 2011.