Program Extraction from Nested Definitions

Kenji Miyamoto[1,⋆], Fredrik Nordvall Forsberg[2,⋆,⋆⋆]
and Helmut Schwichtenberg[1]

[1] Ludwig-Maximilians-Universität München, Germany
[2] Swansea University, UK

Abstract. Minlog is a proof assistant which automatically extracts computational content in an extension of Gödel's T from formalized proofs. We report on extending Minlog to deal with predicates defined using a particular combination of induction and coinduction, via so-called nested definitions. In order to increase the efficiency of the extracted programs, we have also implemented a feature to translate terms into Haskell programs. To illustrate our theory and implementation, a formalisation of a theory of uniformly continuous functions due to Berger is presented.

1 Introduction

Program extraction is a method for obtaining certified algorithms by extracting the computational content hidden in proofs. To get successful algorithms, the formalization of the proof is not a superficial issue but rather an essential one. The *Theory of Computable Functionals* [24], TCF in short, has been developed in order to provide a concrete framework for program extraction. TCF is implemented straightforwardly in the *Minlog* [18] proof assistant. As available in TCF, Minlog supports inductive and coinductive definitions and program extraction from classical proofs as well as from constructive ones. The internal term language of Minlog can be exported to general-purpose programming languages.

This paper reports on new contributions to TCF and Minlog, focusing on two aspects. One is a certain combination of inductive and coinductive definitions, called *nested definitions* [6]. We make use of such definitions in a case study on exact real arithmetic. The other is a feature to translate Minlog algebras and terms into Haskell programs. This makes efficient execution of the extracted programs possible. Translation to a lazy language such as Haskell is especially beneficial when computing with infinite objects, such as in our case study.

We first describe TCF, with an emphasis on nested definitions. Then an application of TCF and program extraction from nested definitions to exact real arithmetic is presented: a translation of the usual type-1 representation of uniformly continuous functions into a type-0 representation. We also extract a program which computes the definite integral of such functions. These case studies are available in the Minlog distribution [18].

⋆ The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement number 238381.
⋆⋆ Supported by EPSRC grant EP/G033374/1.

2 Formal System

We study higher type functionals as well as functions and ground type objects. Functionals in TCF are not necessarily total but partial in general. Based on the understanding that evaluation must be finite, we assume two principles for our notion of computability: the finite support principle and the monotonicity principle. During the evaluation of some functional Φ, only finitely many inputs $\varphi_0, \ldots, \varphi_{n-1}$ are used. Moreover, each of φ_i must be presented to Φ in a finite form. This is the finite support principle. Assume $\Phi(\varphi_0)$ evaluates to a value k and let φ_1 be more informative than φ_0. Then $\Phi(\varphi_1)$ results in k as well. This is the monotonicity principle.

The notion of abstract computability is formulated as follows: an object is computable when its set of finite approximations is primitive recursively enumerable. In this section, we begin by making the notion of such computable objects, called partial continuous functionals, concrete. Then we proceed to our term calculus, its Haskell translation, and inductive/coinductive predicates.

2.1 Algebras and their total and cototal ideals

The formal term language of TCF is an extension of Gödel's T, which is appropriate for higher type computation involving functionals. Types are built from base types by the formation of function types. The base types themselves are formed by free algebras given by their constructors. For instance the list algebra $\mathbf{L}_\alpha$, where α is a type parameter, is defined by the two constructors empty list $[]^{\mathbf{L}_\alpha}$ and the "cons" operator $::^{\alpha\to\mathbf{L}_\alpha\to\mathbf{L}_\alpha}$. Formally, a constructor type is a type expression of the form $\tau_0 \to \ldots \to \tau_{n-1} \to \xi_i$, where each τ_i is a type expression where all $\vec{\xi}$ appear strictly positively (i.e. not to the left of an arrow). For any finite list of constructor types $\vec{\kappa}$, we (simultaneously) define algebras $\mu_{\vec{\xi}}(\vec{\kappa})$, provided there is at least one constructor type such that $\vec{\xi}$ does not occur in $\vec{\tau}$ (this ensures that all algebras are inhabited). For example, the algebra of natural numbers $\mathbf{N}$ is defined by $\mathbf{N} = \mu_\xi(\xi, \xi \to \xi)$. We also adopt the notation $\mathbf{L}_\alpha = \mu_\xi([]^\xi, ::^{\alpha\to\xi\to\xi})$ in order to specify constructor names. Another example is the algebra of branching trees. We simultaneously define $(\mathbf{Ts}, \mathbf{T})$ by $\mu_{\xi,\zeta}(\mathsf{Empty}^\xi, \mathsf{Tcons}^{\zeta\to\xi\to\xi}, \mathsf{Leaf}^\zeta, \mathsf{Branch}^{\xi\to\zeta})$. Using the list algebra, we can define another algebra of branching trees without the simultaneity by defining $\mathbf{NT} = \mu_\xi(\mathsf{Lf}^\xi, \mathsf{Br}^{\mathbf{L}_\xi\to\xi})$. This is an example of a *nested* algebra [6]. Support for such algebras has recently been added to Minlog.

The intended semantics of the term language is based on Scott's information systems [25] (see also Schwichtenberg and Wainer [24]). Algebras are interpreted as sets of *ideals*, i.e. consistent and deductively closed sets of tokens, which are type correct constructor trees possibly involving the special symbol $*$, meaning "no information". Consider a constructor tree $P(*)$ with a distinguished occurrence of $*$. An arbitrary $P(C\vec{*})$, where C is a constructor, is called a *one-step predecessor* of $P(*)$, written $P(C\vec{*}) \succ_1 P(*)$. Here $P(C\vec{*})$ is obtained by substituting $C\vec{*}$ for the distinguished $*$ in $P(*)$. Among ideals, we are especially interested in

total and cototal ideals. A *cototal ideal* x is an ideal whose every constructor tree $P(*) \in x$ has a one-step predecessor $P(C\vec{*}) \in x$. A *total ideal* is a cototal ideal such that the relation $\succ_1$ is well founded. For instance, the cototal ideal $\{*::*, 0::*, *::*::*, 0::*::*, *::1::*, 0::1::*, *::*::*::*, \ldots\}$ denotes the non-well founded list of natural numbers $[0, 1, \ldots]$.

A binary tree with a (possibly) infinite height is informally defined by a term $t := \mathsf{Br}(t::t::[])$ whose denotation is an $\mathbf{NT}$-cototal $\mathbf{L_{NT}}$-total ideal (see also Section 2.5). Total ideals of $(\mathbf{Ts}, \mathbf{T})$ are isomorphic (as information systems) to pairs of $\mathbf{L_{NT}}$-total $\mathbf{NT}$-total ideals and $\mathbf{NT}$-total $\mathbf{L_{NT}}$-total ideals.

2.2 Corecursion

An arbitrary term in Gödel's T is terminating and hence denotes a total ideal. Constructors are used to construct total ideals whereas recursion operators are used to inspect a total ideal from its leaves to the root. In order to accommodate cototal as well as total ideals, we add to Gödel's T two more kinds of constants, namely destructors and corecursion operators, which this section describes. Destructors, the dual of constructors, are used to inspect the structure of cototal ideals, while corecursion operators give a way to construct cototal ideals.

As an example, we consider the algebra $\mathbf{NT}$ of nested trees. Define the disjoint sum of α and β by $\alpha + \beta = \mu_\xi(\mathsf{inl}^{\alpha\to\xi}, \mathsf{inr}^{\beta\to\xi})$, and the unit type $\mathbf{U} = \mu_\xi(\mathsf{u}^\xi)$. The destructor $\mathcal{D}_{\mathbf{NT}}$ has the following type and conversion relation:

$$\begin{aligned} &\mathcal{D}_{\mathbf{NT}} : \mathbf{NT} \to \mathbf{U} + \mathbf{L_{NT}} \\ &\mathcal{D}_{\mathbf{NT}}\,\mathsf{Lf} \mapsto \mathsf{inl}\ \mathsf{u}, \qquad \mathcal{D}_{\mathbf{NT}}\,(\mathsf{Br}\ as) \mapsto \mathsf{inr}\ as. \end{aligned} \tag{1}$$

Corecursion operators give a way to construct cototal ideals. The corecursion operator ${}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}}$ has the following type and conversion relation:

$$\begin{aligned} &{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}} : \tau \to (\tau \to \mathbf{U} + \mathbf{L}_{\mathbf{NT}+\tau}) \to \mathbf{NT} \\ &{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}}\,N\,M \mapsto \text{case } MN \text{ of} \\ &\qquad\qquad \mathsf{inl}\ \mathsf{u} \to \mathsf{Lf} \\ &\qquad\qquad \mathsf{inr}\ qs \to \mathsf{Br}\ (\mathcal{M}^{\mathbf{NT}+\tau\to\mathbf{NT}}_{\lambda_\alpha \mathbf{L}_\alpha}\ qs\ [\mathrm{id}, \lambda_x({}^{\mathrm{co}}\mathcal{R}xM)]). \end{aligned}$$

where $[f, g]^{\rho+\sigma\to\tau}$ is defined for $f^{\rho\to\tau}$ and $g^{\sigma\to\tau}$ by

$$[f, g](\mathsf{inl}\ x^\rho) \mapsto fx, \qquad [f, g](\mathsf{inr}\ y^\sigma) \mapsto gy,$$

and the map constant $\mathcal{M}^{\sigma\to\rho}_{\lambda_\alpha \mathbf{L}_\alpha}$ witnesses the functoriality of $\mathbf{L}_\alpha$. It has the following type and conversion relation:

$$\begin{aligned} &\mathcal{M} : \mathbf{L}_\sigma \to (\sigma \to \rho) \to \mathbf{L}_\rho \\ &\mathcal{M}\ []\ f \mapsto [], \qquad \mathcal{M}\ (x :: xs)\ f \mapsto fx :: (\mathcal{M}\ xs\ f). \end{aligned}$$

In the conversion rule for ${}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}}$, the first argument of the corecursion operator is passed to the second functional argument. The result of this application

determines what the construction of the cototal ideal is. In the case of nested algebras, the value algebra of corecursion operators occurs inside of other algebras as a parameter. Map operators play a crucial role in reaching the value algebra so that the corecursion operator can do its work.

2.3 Realizability

We now address the issue of extracting computational content from proofs. The method of program extraction is based on *modified realizability* as introduced by Kreisel [16] and described in detail in Schwichtenberg and Wainer [24]. In short, from every constructive proof M of a non-Harrop formula A (in natural deduction) one extracts a program $\mathrm{et}(M)$ "realizing" A, essentially by removing computationally irrelevant parts from the proof (proofs of Harrop formulas have no computational content). The extracted program has some simple type $\tau(A)$ which depends solely on the logical shape of the proven formula A. In its original form the extraction process is fairly straightforward, but often leads to unnecessarily complex programs. In order to obtain better programs, proof assistants (for instance Coq [9], Isabelle/HOL [13], Agda [1], Nuprl [21], Minlog [18]) offer various optimizations of program extraction. Below we describe optimizations implemented in Minlog [22], which are relevant for our present case study.

Quantifiers without computational content Besides the usual quantifiers, $\forall$ and $\exists$, Minlog has so-called *non-computational quantifiers*, $\forall^{\mathrm{nc}}$ and $\exists^{\mathrm{nc}}$, which allow for the extraction of simpler programs. These quantifiers, which were first introduced by Berger [2], can be viewed as a refinement of the Set/Prop distinction in constructive type systems like Coq. Intuitively, a proof of $\forall^{\mathrm{nc}}_x A(x)$ ($A(x)$ non-Harrop) represents a procedure that assigns to any x a proof $M(x)$ of $A(x)$ where $M(x)$ does not make "computational use" of x, i.e., the extracted program $\mathrm{et}(M(x))$ does not depend on x. Dually, a proof of $\exists^{\mathrm{nc}}_x A(x)$ is a proof of $M(x)$ for some x where the witness x is "hidden", that is, not available for computational use; in fact, $\exists^{\mathrm{nc}}$ can be seen as inductively defined by the clause $\forall^{\mathrm{nc}}_x(A \to \exists^{\mathrm{nc}}_x A)$. The types of extracted programs for non-computational quantifiers are $\tau(\forall^{\mathrm{nc}}_{x^\rho} A) = \tau(\exists^{\mathrm{nc}}_{x^\rho} A) = \tau(A)$ as opposed to $\tau(\forall_{x^\rho} A) = \rho \to \tau(A)$ and $\tau(\exists_{x^\rho} A) = \rho \times \tau(A)$. The extraction rules are, for example in the case of $\forall^{\mathrm{nc}}$-introduction and -elimination, $\mathrm{et}((\lambda_x M^{A(x)})^{\forall^{\mathrm{nc}}_x A(x)}) = \mathrm{et}(M)$ and $\mathrm{et}((M^{\forall^{\mathrm{nc}}_x A(x)} t)^{A(t)}) = \mathrm{et}(M)$ as opposed to $\mathrm{et}((\lambda_x M^{A(x)})^{\forall_x A(x)}) = \mathrm{et}(\lambda_x M)$ and $\mathrm{et}((M^{\forall_x A(x)} t)^{A(t)}) = \mathrm{et}(Mt)$. For the extracted programs to be correct the variable condition for $\forall^{\mathrm{nc}}$-introduction must be strengthened by additionally requiring that the abstracted variable x does not occur in the extracted program $\mathrm{et}(M)$, and similarly for $\exists^{\mathrm{nc}}$. Note that for a Harrop formula A the formulas $\forall^{\mathrm{nc}}_x A$, $\forall_x A$ are equivalent.

2.4 Translation to a general-purpose programming language

The programs extracted from proofs in Minlog are once again represented as terms in the internal term language. This has the advantage that a general soundness

theorem can be stated (and automatically proven) in the system. However, for efficiency and interoperability reasons, it is sometimes beneficial to translate the extracted terms into programs in a general-purpose programming language. We now describe our new translation from Minlog terms into Haskell programs (there is also limited support for translating into Scheme). Coq [17], Isabelle/HOL [5] and Agda [26] provide similar features.

Terms of Gödel's T – lambda abstraction, application, variables etc – are translated to corresponding Haskell terms. For recursion and corecursion operators, polymorphic functions are generated. For example, the translation of the corecursion operator ${}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{NT}}$ above is implemented as

```
ntCoRec :: a -> (a -> Maybe [Either NT a]) -> NT
ntCoRec e f = case (f e) of
               Nothing -> Lf
               Just z  ->
                 Br (fmap (\ w -> case w of
                                  Left x -> x
                                  Right y -> ntCoRec y f) z)
```

Here Haskell's lazy evaluation means that we do not need to worry about guarding the recursive call. The occurrence of the map operator $\mathcal{M}^{\mathbf{NT}+\tau\to\mathbf{NT}}_{\lambda_\alpha \mathbf{L}_\alpha}$ gets translated to the `fmap` function from the `Functor` type class. In this case, the list algebra from Minlog gets translated to the list data type in Haskell, which already has a `Functor` instance. For custom data types, the instance is derived automatically by GHC using the `DeriveFunctor` flag.

Lists, integers, rational numbers, sum types, product types and the unit type are translated to their standard implementation in the Haskell prelude. For efficiency reasons, natural numbers are translated to integers. Other algebras are translated into algebraic data types.

Program constants and their computation rules are translated to functions defined by pattern matching. Here some care must be taken for e.g. natural numbers, since they are translated to integers, for which no pattern matching is available. Instead *guard conditions* are used, as in the translation of the following parity function for natural numbers:

```
parity :: Integer {-Nat-} -> Bool
parity 0 = True
parity 1 = False
parity n | n > 1 = parity (n - 2)
```

The realizer for *ex-falso-quodlibet* $\mathsf{ff} \to A$ makes use of a *canonical inhabitant* $\mathsf{inhab}_{\tau(A)}$ of type $\tau(A)$. This is justified since all types are inhabited in the intended semantics, but not so in Haskell. Hence we define a type class

```
class Inhabited a where
  inhab :: a
```

and ensure we generate instances and track inhabitedness constraints in the types of the generated functions.

2.5 Inductive and coinductive definitions

We are particularly interested in dealing with a combination of induction and coinduction in TCF. Starting from simultaneous inductive definitions, we describe nested inductive definitions and furthermore nested inductive/coinductive definitions. See e.g. Jacobs and Rutten [14] for a gentle introduction to coinduction.

As an example, consider the simultaneously defined algebras $(\mathbf{Ts}, \mathbf{T}) = \mu_{\xi,\zeta}(\mathsf{Empty}^{\xi}, \mathsf{Tcons}^{\zeta\to\xi\to\xi}, \mathsf{Leaf}^{\zeta}, \mathsf{Branch}^{\xi\to\zeta})$ of finitely branching trees again. The totality predicates $(T_{\mathbf{Ts}}, T_{\mathbf{T}})$ of $(\mathbf{Ts}, \mathbf{T})$, of arity $(\mathbf{Ts})$ and $(\mathbf{T})$ respectively, are simultaneously inductively defined by the following four clauses:

$$T_{\mathbf{Ts}}\ \mathsf{Empty}, \qquad \forall^{\mathrm{nc}}_{a,as}(T_{\mathbf{T}}\ a \to T_{\mathbf{Ts}}\ as \to T_{\mathbf{Ts}}\ (\mathsf{Tcons}\ a\ as)),$$
$$T_{\mathbf{T}}\ \mathsf{Leaf}, \qquad \forall^{\mathrm{nc}}_{as}(T_{\mathbf{Ts}}\ as \to T_{\mathbf{T}}\ (\mathsf{Branch}\ as)).$$

From the above, a nested definition of branching trees is derived by removing the simultaneity. This leads to the definition of the algebras $\mathbf{L}_\alpha = \mu_\xi([]^{\xi}, ::^{\alpha\to\xi\to\xi})$ and $\mathbf{NT} = \mu_\xi(\mathsf{Lf}^{\xi}, \mathsf{Br}^{\mathbf{L}_\xi\to\xi})$. To define the totality predicate for $\mathbf{NT}$, we first define the relativised totality predicate RT_X for lists, with arity $(\mathbf{L}_\alpha)$ for X of arity (α). Relativised totality means the totality relative to the parameter predicate X. It is given by the following clauses:

$$RT_X\ [], \qquad \forall^{\mathrm{nc}}_{x,xs}(Xx \to RT_X\ xs \to RT_X\ (x::xs)).$$

We can now define the totality predicate of nested trees using the relativised totality predicate RT_X of lists, with X instantiated to $T_{\mathbf{NT}}$:

$$T_{\mathbf{NT}}\ \mathsf{Lf}, \qquad \forall_{as}(RT_{T_{\mathbf{NT}}}\ as \to T_{\mathbf{NT}}\ (\mathsf{Br}\ as)).$$

We call a predicate definition *nested* if the predicate to be defined occurs strictly positively as a parameter of an already defined predicate in a clause formula. Witnesses of nested predicates have nested algebras as their types.

Coinductive predicates arise as "duals" of inductive ones. For example, for the totality predicate $T_{\mathbf{NT}}$ we can define its companion predicate ${}^{\mathrm{co}}T_{\mathbf{NT}}$ by the single clause

$$\forall^{\mathrm{nc}}_{a}({}^{\mathrm{co}}T_{\mathbf{NT}}a \to a = \mathsf{Lf} \vee \exists^{\mathrm{nc}}_{as}(RT_{{}^{\mathrm{co}}T_{\mathbf{NT}}}as \wedge a = \mathsf{Br}as)). \tag{2}$$

We call such a companion predicate definition derived from an inductive one a *coinductive definition*. A witness for a proposition ${}^{\mathrm{co}}T_{\mathbf{NT}}\ a$ is an $\mathbf{NT}$-cototal $\mathbf{L}_{\mathbf{NT}}$-total ideal, which is a finitely branching tree of (possibly) infinite height. The computational content of (2) is the destructor $\mathcal{D}_{\mathbf{NT}}$. We still need to express that RT_X is the least predicate satisfying the clauses, and that ${}^{\mathrm{co}}T_{\mathbf{NT}}$ is the greatest predicate satisfying the clause. The former is done by means of the least-fixed-point axiom

$$\begin{aligned}\forall^{\mathrm{nc}}_{xs}(RT_X\ xs \to & P\ [] \to \\ & \forall^{\mathrm{nc}}_{x,xs}(Xx \to RT_X\ xs \to P\ xs \to P\ x::xs) \to \\ & P\ xs).\end{aligned} \tag{3}$$

The latter is done by means of the greatest-fixed-point axiom.

$$\forall_a^{\mathrm{nc}}(Q\ a \to \forall_a^{\mathrm{nc}}(Q\ a \to a = \mathsf{Lf} \vee \exists_{as}^{\mathrm{nc}}(RT^{\mathrm{co}}{}_{T_{\mathbf{NT}} \vee Q}\ as \wedge a = \mathsf{Br}\ as)) \to {}^{\mathrm{co}}T_{\mathbf{NT}}\ a). \tag{4}$$

The predicates P and Q are called competitor predicates which satisfy the same clause(s) as RT_X and $T_{\mathbf{NT}}$, respectively. From (3) and (4), we see that P is a superset of RT_X and Q is a subset of ${}^{\mathrm{co}}T_{\mathbf{NT}}$.

The term extracted from (3) is Gödel's (structural) recursion operator $\mathcal{R}_{\mathbf{L}_\alpha}$, and the term extracted from (4) is the corecursion operator ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{NT}}$ defined in Section 2.2.

3 Case study: uniformly continuous functions

To illustrate program extraction in TCF, we formalize the theory of uniformly continuous functions from constructive analysis [7,23]. Our first case study provides an alternative view of uniformly continuous functions of type-1 as a cototal object of type-0; i.e. of ground type. We then extract a program which computes the definite integral of a uniformly continuous function of type-0. This was first studied by Berger [3] in the setting of program extraction. We now offer machine extraction from formalized proofs of these results in Minlog. Before continuing, we review representations of real numbers of type-1 and type-0. In this section, we only consider real numbers and uniformly continuous functions in the interval $[-1, 1]$ in order to work with stream represented real numbers [8] and uniformly continuous functions on them.

A real number of type-1 is a Cauchy real with a modulus, namely a pair $\langle x, M\rangle$, where x is a bounded function of type $\mathbf{N} \to \mathbf{Q}$ and $M : \mathbf{N} \to \mathbf{N}$ satisfies the following Cauchy condition:

$$\forall_{k^{\mathbf{N}}} \forall_{n,m \geq Mk}(|x\,n - x\,m| \leq 2^{-k}).$$

Define the type of signed digits by $\mathbf{SD} = \mu_\xi(-1^\xi, 0^\xi, 1^\xi)$. A real number of type-0 is a signed digit stream $d_0{::}d_1{::}\ldots$, where d_i is of type $\mathbf{SD}$. Informally, the stream $d_0{::}d_1{::}\ldots$ denotes the real number $\sum_{i=0} \frac{d_i}{2^{i+1}}$. We represent such an object by a cototal ideal of $\mathbf{L}_{\mathbf{SD}}$, which is a possibly infinite list of signed digits. An arbitrary real number can be represented by a type-0 object, for example by a stream of integers in $\{-9, -8, \ldots, 8, 9\}$ with a decimal point [27,28].

3.1 Data types of uniformly continuous functions

Consider a triple $\langle h, \alpha, \omega\rangle$, where $h : \mathbf{Q} \to \mathbf{N} \to \mathbf{Q}$ is a bounded function and α and ω are of type $\mathbf{N} \to \mathbf{N}$. Suppose that it satisfies

$$\forall_{a^{\mathbf{Q}}, k^{\mathbf{N}}, n \geq \alpha(k), m \geq \alpha(k)}(|h\,a\,n - h\,a\,m| \leq 2^{-k}),$$
$$\forall_{a^{\mathbf{Q}}, b^{\mathbf{Q}}, k^{\mathbf{N}}, n \geq \alpha(k)}(|a - b| \leq 2^{-\omega(k)+1} \to |h\,a\,n - h\,b\,n| \leq 2^{-k}).$$

The first formula states the Cauchyness of $\langle h\, a, \alpha\rangle$. Classically this can be stated by the formula $\forall_{a,k}\exists_l\forall_{n,m\geq l}(|h\, a\, n - h\, a\, m| \leq 2^{-k})$, while constructively the way to determine l has to be given, for instance by a Cauchy modulus α. The second formula states the uniform continuity of $\langle h, \alpha\rangle$, once again with explicit modulus of uniform continuity ω for constructive reasons. Finally, we assume that h is bounded between -1 to 1. We adopt objects of this kind as our uniformly continuous functions of type-1, namely of first order function type. Application of a type-1 uniformly continuous function $\langle h, \alpha, \omega\rangle$ to a Cauchy real $\langle x, M\rangle$ is defined to be

$$\langle \lambda_n(h\,(x\, n)\, n), \lambda_k \max(\alpha(k+2), M(\omega(k+1)-1))\rangle.$$

For our type-0 representation of uniformly continuous functions we adopt so-called read-write machines [3] or stream processors [11,12]. These are $\mathbf{W}$-cototal $\mathbf{R}_\mathbf{W}$-total ideals where

$$\begin{aligned}\mathbf{R}_\alpha &:= \mu_\xi(\mathsf{Put}^{\mathbf{SD}\to\alpha\to\xi}, \mathsf{Get}^{\xi\to\xi\to\xi\to\xi}),\\ \mathbf{W} &:= \mu_\xi(\mathsf{Stop}^\xi, \mathsf{Cont}^{\mathbf{R}_\xi\to\xi}).\end{aligned}$$

A read-write machine is a potentially non-well founded tree with internal Put nodes and branching at Get nodes. It intuitively represents a function from signed digit streams to signed digit streams as follows: start at the root of the tree. If we are at the node $(\mathsf{Put}\ d\ t)$, output the digit d and carry on with the tree t. If we are at the node $(\mathsf{Get}\ t_{-1}\ t_0\ t_1)$, read a digit d from the input stream and continue with the tree t_d. If we reach a Stop node, we return the rest of the input unprocessed as output. Because a read-write machine is a $\mathbf{W}$-cototal $\mathbf{R}_\mathbf{W}$-total ideal, the output might be infinite, but $\mathbf{R}_\mathbf{W}$-totality ensures that the machine can only read finitely many input digits before producing another output digit; the machine represents a continuous function.

3.2 Formalization

We work with the abstract theory of uniformly continuous functions. Suppose that φ is a type variable representing abstract uniformly continuous functions. Due to the use of non-computational connectives, any object of type φ appearing in the proofs will disappear when a program is extracted. This theory is axiomatized in Appendix A. Note that all axioms are non-computational.

Let f range over the type variable φ, and also p, q range over $\mathbf{Q}$ and k, l range over $\mathbf{N}$. We define a comprehension term C (for "continuous") of abstract uniformly continuous functions as follows.

$$\mathrm{C} := \{f|\forall_k\exists_l \mathrm{B}_{l,k} f\},\ \text{where}\ \mathrm{B}_{l,k} := \{f|\forall_p\exists_q(f[I_{p,l}] \subseteq I_{q,k})\}.$$

Here, $I_{q,k}$ represents the interval $[q - 2^{-k}, q + 2^{-k}]$ of length 2^{1-k} centered at q, while $f[I_{p,l}]$ represents the image of $I_{p,l}$ under f; the exact behavior is axiomatized in Appendix A. We write I for the interval $I_{0,0} = [-1, 1]$. Witnesses

for $\mathrm{B}_{l,k}f$ and $\mathrm{C}f$ are total ideals of type (an isomorphic copy of) $\mathbf{Q} \to \mathbf{Q}$ and $\mathbf{N} \to \mathbf{N} \times (\mathbf{Q} \to \mathbf{Q})$, respectively. The latter represents $\langle h, \alpha, \omega \rangle$ by a term $\lambda_n \langle \omega\, n, \lambda_a\, h\, a\, (\alpha\, n) \rangle$. Let $(\mathrm{Out}_d \circ f)(x) = 2f(x) - d$, $(f \circ \mathrm{In}_d)(x) = f(\frac{x+d}{2})$ and $I_d = [\frac{d-1}{2}, \frac{d+1}{2}]$ for each $d \in \{-1, 0, 1\}$. We call I_d a basic interval. We inductively define a predicate Read_X of arity (φ) by the following clauses

$$\forall_f^{\mathrm{nc}} \forall_d (f[I] \subseteq I_d \to X(\mathrm{Out}_d \circ f) \to \mathrm{Read}_X f), \qquad (\mathrm{Read}_X)_0^+$$

$$\forall_f^{\mathrm{nc}} (\mathrm{Read}_X(f \circ \mathrm{In}_{-1}) \to \mathrm{Read}_X(f \circ \mathrm{In}_0) \to \mathrm{Read}_X(f \circ \mathrm{In}_1) \to \mathrm{Read}_X f). \qquad (\mathrm{Read}_X)_1^+$$

The least-fixed-point axiom $(\mathrm{Read}_X)^-$ is defined to be

$$\begin{aligned} \forall_f^{\mathrm{nc}} (\mathrm{Read}_X f \to \forall_f^{\mathrm{nc}} \forall_d (f[I] \subseteq I_d \to X(\mathrm{Out}_d \circ f) \to Pf) \to \\ \forall_f^{\mathrm{nc}} (\mathrm{Read}_X(f \circ \mathrm{In}_{-1}) \to P(f \circ \mathrm{In}_{-1}) \to \\ \mathrm{Read}_X(f \circ \mathrm{In}_0) \to P(f \circ \mathrm{In}_0) \to \\ \mathrm{Read}_X(f \circ \mathrm{In}_1) \to P(f \circ \mathrm{In}_1) \to Pf) \to \\ Pf). \end{aligned} \qquad (\mathrm{Read}_X)^-$$

Furthermore, we give a nested inductive definition of a predicate Write of abstract uniformly continuous functions by the following clauses

$$\mathrm{Write}(\mathrm{Id}), \qquad \forall_f^{\mathrm{nc}} (\mathrm{Read}_{\mathrm{Write}} f \to \mathrm{Write} f),$$

where Id is the identity function. Witnesses for $\mathrm{Read}_X f$ and $\mathrm{Write}f$ are total ideals of $\mathbf{R}_\alpha$ and $\mathbf{W}$, respectively. We define ${}^{\mathrm{co}}\mathrm{Write}$, a companion predicate of Write, by the following clause

$$\forall_f^{\mathrm{nc}} ({}^{\mathrm{co}}\mathrm{Write} f \to f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}} f). \qquad ({}^{\mathrm{co}}\mathrm{Write})^-$$

The greatest-fixed-point axiom $({}^{\mathrm{co}}\mathrm{Write})^+$ of ${}^{\mathrm{co}}\mathrm{Write}$ is

$$\forall_f^{\mathrm{nc}} (Q\, f \to \forall_f^{\mathrm{nc}} (Q\, f \to f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write} \vee Q} f) \to {}^{\mathrm{co}}\mathrm{Write}\, f). \qquad ({}^{\mathrm{co}}\mathrm{Write})^+$$

A witness for ${}^{\mathrm{co}}\mathrm{Write}f$ is a $\mathbf{W}$-cototal $\mathbf{R}_{\mathbf{W}}$-total ideal. Intuitively, ${}^{\mathrm{co}}\mathrm{Write}f$ says that f is productive as a function on signed digit representations. If we can use axiom $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_0^+$, we know that the image of f is contained in an interval of radius $\frac{1}{2}$ centered at the digit d, so that the first output digit must be d independently of the input. By using the function $\mathrm{Out}_d \circ f$, we remove the leading digit and shift the input sequence one digit to the left. We continue to prove that f is productive on the rest of the input sequence. If the image of f is not contained in a basic interval, we can split the interval in three subintervals and check that f is productive on all of them by using axiom $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_1^+$. This corresponds to reading another input digit. Since Read_X is inductively defined, we can only use $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_1^+$ finitely many times before we are forced to use $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_0^+$ and another output digit is determined.

In our Minlog formalization, φ is given as a type variable, and In_d, Out_d and Id are defined as constants without computational meaning with value type φ. This is not a problem, since all such constants will disappear in the program extraction process due to careful use of non-computational connectives.

3.3 Informal proofs

We present informal proofs from which programs on uniformly continuous functions are extracted. Formalized proofs can be found in the Minlog distribution in the file `examples/analysis/readwrite.scm`.

For the first case study, Axiom 1 in Appendix A is used.

Theorem 1 (Type-1 u.c.f. into type-0 u.c.f.).

$$\forall_f^{\mathrm{nc}}(\mathrm{C}f \to {}^{\mathrm{co}}\mathrm{Write}f).$$

Proof. Let f be given and assume $\mathrm{C}f$. We prove ${}^{\mathrm{co}}\mathrm{Write}f$ by the greatest fixed point axiom ${}^{\mathrm{co}}\mathrm{Write}^+$ with C for the competitor. It suffices to prove $\forall_f^{\mathrm{nc}}(\mathrm{C}f \to f = \mathrm{Id} \vee \mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}}f)$. Again let f be given and assume $\mathrm{C}f$, i.e. in particular $\mathrm{B}_{l,2}f$ for some l. By Lemma 2, the right disjunct of the goal holds. □

The above proof considerably depends on the following lemma, which in turn depends on the next ones.

Lemma 2. $\forall_l\forall_f^{\mathrm{nc}}(\mathrm{B}_{l,2}f \to \mathrm{C}f \to \mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}}f)$.

Proof. By induction on l. *Base:* $l = 0$. Let f be given, and assume $\mathrm{B}_{0,2}f$ and $\mathrm{C}f$. Applying Lemma 3, there is a d such that $f[I] \subseteq I_d$. By Lemma 4, $\mathrm{C}f$ implies $\mathrm{C}(\mathrm{Out}_d \circ f)$, hence $({}^{\mathrm{co}}\mathrm{Write} \vee \mathrm{C})(\mathrm{Out}_d \circ f)$. Now use the introduction axiom $(\mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}})^+_0$. *Step:* $l \mapsto l+1$. Suppose the following induction hypothesis

$$\forall_f^{\mathrm{nc}}(\mathrm{B}_{l,2}f \to \mathrm{C}f \to \mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}}f), \tag{5}$$

and prove $\forall_f^{\mathrm{nc}}(\mathrm{B}_{l+1,2}f \to \mathrm{C}f \to \mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}}f)$. Assume $\mathrm{B}_{l+1,2}f$ and $\mathrm{C}f$ for a given f. Our goal is $\mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}}f$. By Lemma 4, we have $\mathrm{B}_{l,2}(f \circ \mathrm{In}_d)$ and $\mathrm{C}(f \circ \mathrm{In}_d)$ for each d. The induction hypothesis (5) yields $\mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}}(f \circ \mathrm{In}_d)$ for each d, hence we can apply the introduction axiom $(\mathrm{Read}^{\mathrm{co}}_{\mathrm{Write}\vee\mathrm{C}})^+_1$ to finish the proof. □

Lemma 3. $\forall_f^{\mathrm{nc}}(\mathrm{B}_{0,2}f \to \exists_d(f[I] \subseteq I_d))$.

Proof. Assume f and $\mathrm{B}_{0,2}f$. From the definition of $\mathrm{B}_{l,k}$, $f[I_{0,0}] \subseteq I_{q,2}$ for some q holds. Because q is a rational number, either $q \le -\frac{1}{4}$, $-\frac{1}{4} \le q \le \frac{1}{4}$ or $\frac{1}{4} \le q$. Recall that our uniformly continuous function is bounded in $[-1,1]$. It is possible to determine either of $I_{q,2} \subseteq I_{-1}$, $I_{q,2} \subseteq I_0$ or $I_{q,2} \subseteq I_1$, hence $\exists_d(f[I] \subseteq I_d)$. □

Lemma 4. *(i)* $\forall^{\mathrm{nc}}_{f,k,l}\forall_d(f[I] \subseteq I_d \to \mathrm{B}_{l,k+1}f \to \mathrm{B}_{l,k}(\mathrm{Out}_d \circ f))$.
(ii) $\forall^{\mathrm{nc}}_f\forall_d(f[I] \subseteq I_d \to \mathrm{C}f \to \mathrm{C}(\mathrm{Out}_d \circ f))$.
(iii) $\forall^{\mathrm{nc}}_{f,k,l}\forall_d(\mathrm{B}_{l+1,k}f \to \mathrm{B}_{l,k}(f \circ \mathrm{In}_d))$.
(iv) $\forall^{\mathrm{nc}}_f\forall_d(\mathrm{C}f \to \mathrm{C}(f \circ \mathrm{In}_d))$. □

We now turn to calculating the definite integral of uniformly continuous functions to an arbitrary precision. In order to stay in the interval $[-1,1]$, we compute the definite integral from -1 to 1 divided by two. We abbreviate $\frac{1}{2}\int_{-1}^{1} f$ by $\int^{\mathrm{H}} f$ (H for "half"). The properties we need of the integral and the real numbers are axiomatized in Axiom 2 and Axiom 3 in Appendix A.

Theorem 5 (Definite integral from -1 to 1).

$$\forall_f^{\mathrm{nc}}({}^{\mathrm{co}}\mathrm{Write}f \to \forall_n \exists_p(\textstyle\int^{\mathrm{H}} f \in I_{p,n})).$$

Proof. Let f be given and assume ${}^{\mathrm{co}}\mathrm{Write}f$. We finish the proof by induction on n. *Case* $n = 0$. Choose p to be 0; then $\int^{\mathrm{H}} f \in I_{0,0}$ by the axiom. *Case* $n \mapsto n+1$. We prove $\exists_p(\int^{\mathrm{H}} f \in I_{p,n+1}))$. By $({}^{\mathrm{co}}\mathrm{Write}f)^-$, we can do case distinction on $f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}}f$. *Left case.* Suppose $f = \mathrm{Id}$. Let p be 0, then our goal is $\int^{\mathrm{H}} \mathrm{Id} \in I_{0,n+1}$, which is clear by the axioms. *Right case.* Suppose $\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}}f$ and use $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})^-$. *Side base case.* Let f and d be given and assume $f[I] \subseteq I_d$ and ${}^{\mathrm{co}}\mathrm{Write}(\mathrm{Out}_d \circ f)$. We prove $\exists_p(\int^{\mathrm{H}} f \in I_{p,n+1})$. By i.h. there is a p' such that $\int^{\mathrm{H}}(\mathrm{Out}_d \circ f) \in I_{p',n}$, which implies $\int^{\mathrm{H}} f \in I_{\frac{p'+d}{2},n+1}$ as desired. *Side step case.* Let f be given and assume side i.h. We prove $\exists_p(\int^{\mathrm{H}} f \in I_{p,n+1})$. By the side i.h., there are p_d such that $\int^{\mathrm{H}}(f \circ \mathrm{In}_d) \in I_{p_d,n+1}$ for each d, thus $\frac{1}{2}(\int^{\mathrm{H}}(f \circ \mathrm{In}_{-1}) + \int^{\mathrm{H}}(f \circ \mathrm{In}_1)) \in I_{\frac{p_{-1}+p_1}{2},n+1}$ holds. This implies $\int^{\mathrm{H}} f \in I_{\frac{p_{-1}+p_1}{2},n+1}$ as desired. □

3.4 Extraction

From a proof, Minlog extracts a term in an extension of Gödel's T. In the next stage, these, together with relevant algebras and program constants, can be translated into a Haskell program using the `term-to-haskell-program` function of Minlog. We present the Haskell programs obtained from our formalized proofs. For aesthetic reasons, we present slightly formatted versions of the programs as suggested by e.g. HLint [19].

The algebras involved get translated to the following Haskell data types:

```
data AlgB = CInitB (Rational -> Rational)

data AlgRead a = Put Sd a
               | Get (AlgRead a) (AlgRead a) (AlgRead a)
  deriving (Show, Read, Eq, Ord, Functor)

data AlgWrite = Stop | Cont (AlgRead AlgWrite)
  deriving (Show, Read, Eq, Ord)

data Sd = L | M | R
  deriving (Show, Read, Eq, Ord)
```

We see how `AlgB` is just an isomorphic copy of `Rational -> Rational`, and how `AlgRead` has a type parameter `a` which gets instantiated to `AlgWrite` in the constructor `Cont`.

The extracted program from Lemma 3 is

```
cLemmaThree :: AlgB -> Sd
cLemmaThree (CInitB g) =
  if (numerator ((g 0) + (1/4)) > 0) then
    if (numerator ((g 0) - (1/4)) > 0) then R else M
  else L
```

This program computes a signed digit d such that the image of f – an abstract function which does not appear in the extracted term – is contained in I_d. It takes a rational function `g` which realizes $\mathrm{B}_{l,2}\, f$ as input; hence the image is contained in an interval of length $\frac{1}{2}$, centered at `g 0`. The calculation of the output is reduced to a simple decision of rational inequalities.

The extracted program from Lemma 2 is

```
cLemmaTwo :: Integer -> AlgB -> (Integer -> (Integer, AlgB)) ->
             AlgRead (Either AlgWrite
                             (Integer -> (Integer, AlgB)))
cLemmaTwo n =
  natRec n
         (\ w h ->
             Put (cLemmaThree w)
                 (Right (cLemmaFour_ii (cLemmaThree w) h)))
         (\ n3 g w h ->
             Get (g (cLemmaFour_iii L w) (cLemmaFour_iv L h))
                 (g (cLemmaFour_iii M w) (cLemmaFour_iv M h))
                 (g (cLemmaFour_iii R w) (cLemmaFour_iv R h)))
```

The extracted term `cLemmaTwo` takes as input a natural number `n`, a rational function `w` and a function $\mathtt{h} : \mathbf{N} \to \mathbf{N} \times \mathtt{AlgB}$ (in our application, we only call `cLemmaTwo` with $\langle \mathtt{n}, \mathtt{w} \rangle = \mathtt{h}\, 2$). Using recursion over `n`, it computes an approximation of `h` by a complete tree of height `n` with $3^{\mathtt{n}}$ leaves – a $\mathbf{R}_{\mathbf{W}+(\mathbf{N}\to\mathbf{N}\times\mathtt{AlgB})}$-total ideal. At the leaves, a signed digit d – computed from `w` using `cLemmaThree` – and the remainder of the approximation of `h` – computed by `cLemmaFour_ii` below, using d – is stored. At internal branching nodes, we split the domain of `h` into three subdomains – left, middle and right – modify `w` and `h` accordingly (using `cLemmaFour_iii` and `cLemmaFour_iv` below), and recurse.

The above term involves terms extracted from Lemma 4. They work in the following ways.

```
cLemmaFour_i :: Sd -> AlgB -> AlgB
cLemmaFour_i sd (CInitB h) =
  CInitB (\ a -> (2 * h a) - (sDToInt sd % 1))

cLemmaFour_ii :: Sd -> (Integer -> (Integer, AlgB)) ->
                 Integer -> (Integer, AlgB)
cLemmaFour_ii sd g n = case g (n + 1) of
  (n1, w1) -> (n1 , cLemmaFour_i sd w1)
```

```
cLemmaFour_iii :: Sd -> AlgB -> AlgB
cLemmaFour_iii sd (CInitB h) =
  CInitB (\ a -> h ((a + (sDToInt sd % 1)) / 2))

cLemmaFour_iv :: Sd -> (Integer -> (Integer, AlgB)) ->
                 Integer -> (Integer, AlgB)
cLemmaFour_iv sd g n = case g n of
  (n1, w) -> if n1 == 0 then (0, cLemmaFour_iii sd w)
             else (n1 - 1, cLemmaFour_iii sd w)
```

The extracted term from Theorem 1 is

```
type1to0 :: (Integer -> (Integer, AlgB)) -> AlgWrite
type1to0 r = algWriteCoRec r
  (\ h -> (Just (case h 2 of (n, w) -> cLemmaTwo n w h)))
```

This program corecursively constructs a $\mathbf{W}$-cototal $\mathbf{R_W}$-total ideal by stacking $\mathbf{R_W}$-total ideals computed by `cLemmaTwo`. It uses the modulus of continuity `n` at precision 2 to calculate an approximation of `s` as an $\mathbf{R_W}$-total ideal, as in Lemma 2. In fact, `n` is the number of input signed digits to be read to determine one output signed digit. The extracted term from Theorem 5 is

```
integration :: AlgWrite -> Integer -> Rational
integration h n = natRec n (const 0)
  (\ n1 t h1 ->
    (case algWriteDestr h1 of
      Nothing -> 0
      Just s  -> algReadRec s
                 (\ sd h2 -> (t h2 + (sDToInt sd % 1)) / 2)
                 (\ s1 a1 s2 a2 s3 a3 -> (a1 + a3) / 2)))
  h
```

This program reads the given type-0 function to accumulate the possible output digits to compute the definite integral. The second argument is a number `n` to specify the bound of the computation in such a way that the program processes the read-write machine from its root up to the `n`th $\mathbf{R_W}$-total ideals. At a branch, the recursively computed integral on the middle interval, i.e. `a2`, is ignored because it suffices to see value on the left and the right subintervals, i.e. $[-1, 0]$ and $[0, 1]$. At a leaf, the output digit is counted to contribute to the output with its height in the tree.

3.5 Experiment

As a first example, we instantiate the theorems to the function $f(x) \coloneqq -x$. From a type-1 representation of f, we compute a type-0 representation by means of our extracted program. We define f by $\langle h, \alpha, \omega \rangle$ where $h\, a\, n \coloneqq -a$, $\alpha\, n \coloneqq 0$ and $\omega\, n \coloneqq n + 1$. The input of `type1to0` is $\lambda_n \langle \omega n, \lambda_a(ha(\alpha n)) \rangle$ which turns into the Haskell expression `fIn = \ n -> (n+1, CInitB (\ x -> -x))`. The

output `type1to0 fIn` is graphically presented in Figure 1, where Cont is omitted, Get is a branching node and Put d is denoted by $-$, 0 or $+$ respectively for $d = -1, 0$ or 1.

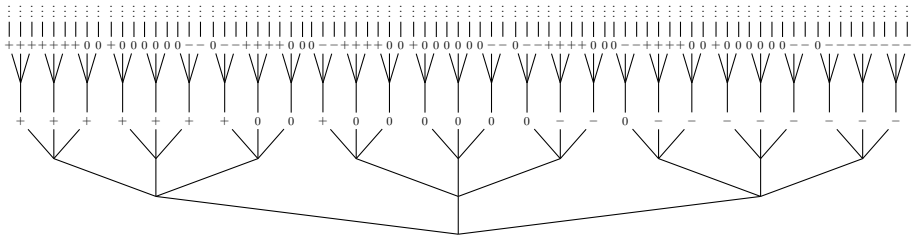

Fig. 1. Type-0 representation of $f(x) = -x$.

In our next example, we compute half of the definite integral for the function $f(x) := \sqrt{x+2} - 1$. This integrand is defined to be $\langle h, \alpha, \omega \rangle$, where $h\, a\, n :=$ $(\mathcal{R}_{\mathbf{N}}\, n\, 1\, \lambda_{\_,b}\, (\frac{b+\frac{a+2}{b}}{2})) - 1$, $\alpha\, n := n+1$, and $\omega\, n :=$ Pred n where Pred : $\mathbf{N} \to \mathbf{N}$ is the predecessor function. Converting this to the Haskell function it represents, we end up with `\ n -> (Main.pred n, CInitB (\ x -> h x (n+1)))`. We give two arguments to `integration`, a type-0 function and a natural number, the accuracy. The first input to `integration` is computed by `type1to0` from the above type-1 function. Specifying 8 as the second argument, the output is `1633 % 4096` whose decimal expansion is $0.398681640625\cdots$. Comparing our result with the manually calculated definite integral $\frac{1}{2}\int_{-1}^{1} f(x)dx = \sqrt{3} - \frac{4}{3}$, the error is $0.00003583\cdots$, which indeed is smaller than $2^{-8} = 0.00390625$.

4 Conclusion

We presented the formal theory TCF and its implementation Minlog which support nested inductive/coinductive definitions. Minlog extracts programs in an extension of Gödel's T from proofs involving nested definitions. Moreover, terms in the extension of Gödel's T can be translated into programs in programming languages such as Haskell. We gave an application to the theory of uniformly continuous functions as an illustration.

Related work Nested definitions are used by Ghani, Hancock and Pattinson [11,12] to define uniformly continuous functions. They are also studied by Bird and Meertens [6] from a purely programming perspective. Krebbers and Spitters [15] give effective certified programs for exact real number computation. Berger and Seisenberger [4] considers "pen and paper" program extraction for a system with induction and coinduction. Berger [3] studies program extraction and its application to exact real arithmetic. He manually extracts programs from proofs

dealing with uniformly continuous functions. Our case study is heavily based on his results. More case studies based on Berger's work, e.g. function application and composition, are available in the Minlog distribution. Also other researchers have studied the combination of induction and coinduction. Nakata and Uustalu [20] study the semantics of interactive programs by means of induction nested into coinduction, and give a formalization in Coq. Danielsson and Altenkirch [10] study so-called mixed induction and coinduction, using Agda.

References

1. Agda. `http://wiki.portal.chalmers.se/agda/`.
2. U. Berger. Program extraction from normalization proofs. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer, 1993.
3. U. Berger. From coinductive proofs to exact real arithmetic: theory and applications. *Logical Methods in Computer Science*, 7(1):1–24, 2011.
4. U. Berger and M. Seisenberger. Proofs, programs, processes. *Theory of Computing Systems*, 51:313–329, 2012.
5. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.
6. R. Bird and L. Meertens. Nested datatypes. In *Mathematics of program construction*, pages 52–67. Springer, 1998.
7. E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.
8. A. Ciaffaglione and P. D. Gianantonio. A co-inductive approach to real numbers. In *Types 1999*, volume 1956 of *LNCS*, pages 114–130. Springer, 1999.
9. Coq. `http://coq.inria.fr/`.
10. N. A. Danielsson and T. Altenkirch. Mixing Induction and Coinduction. Draft, 2009.
11. N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. In J. Power, editor, *CMCS 2006*, Electr. Notes in Theoret. Computer Science, 2006.
12. P. Hancock, D. Pattinson, and N. Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3), 2009.
13. Isabelle. `http://isabelle.in.tum.de/`.
14. B. Jacobs and J. Rutten. An introduction to (co)algebras and (co)induction. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52, pages 38–99. Cambridge University Press, 2011.
15. R. Krebbers and B. Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2013.
16. G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, Amsterdam, 1959.
17. P. Letouzey. Coq extraction, an overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *CiE 2008*, volume 5028 of *LNCS*. Springer, 2008.
18. The Minlog System. `http://www.minlog-system.de`.
19. N. Mitchell. *HLint*. `http://community.haskell.org/~ndm/hlint/`.
20. K. Nakata and T. Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: An exercise in mixed induction-coinduction. In L. Aceto and P. Sobocinski, editors, *SOS*, volume 32 of *EPTCS*, pages 57–75, 2010.

21. Nuprl. `http://www.nuprl.org/`.
22. H. Schwichtenberg. Minlog. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *LNAI*, pages 151–157. Springer, 2006.
23. H. Schwichtenberg. Constructive analysis with witnesses. Manuscript, April 2012. `http://www.math.lmu.de/~schwicht/seminars/semws11/constr11.pdf`.
24. H. Schwichtenberg and S. S. Wainer. *Proofs and Computations*. Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press, 2012.
25. D. Scott. Domains for denotational semantics. In M. Nielsen and E. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *LNCS*, pages 577–610. Springer, 1982.
26. M. Takeyama. A new compiler MAlonzo. `https://lists.chalmers.se/pipermail/agda/2008/000219.html`.
27. E. Wiedmer. *Exaktes Rechnen mit reellen Zahlen und anderen unendlichen Objekten*. PhD thesis, ETH Zürich, 1977.
28. E. Wiedmer. Computing with infinite objects. *Theoretical Comput. Sci.*, 10:133–155, 1980.

A Axioms

Axiom 1 (Abstract Theory of Uniformly Continuous Functions)

$$\forall_{f,d,p,l,q,k}(f[I] \subseteq I_d \to \mathrm{Out}_d \circ f[I_{p,l}] \subseteq I_{q,k} \to f[I_{p,l}] \subseteq I_{\frac{q+d}{2},k+1}), \tag{OutElim}$$

$$\forall_{f,d,p,l,q,k}(f[I] \subseteq I_d \to f[I_{p,l}] \subseteq I_{\frac{q+d}{2},k+1} \to \mathrm{Out}_d \circ f[I_{p,l}] \subseteq I_{q,k}), \tag{OutIntro}$$

$$\forall_{f,d,p,l,q,k}(f \circ \mathrm{In}_d[I_{p,l}] \subseteq I_{q,k} \to f[I_{\frac{p+d}{2},l+1}] \subseteq I_{q,k}), \tag{InElim}$$

$$\forall_{f,d,p,l,q,k}(f[I_{\frac{p+d}{2},l+1}] \subseteq I_{q,k} \to f \circ \mathrm{In}_d[I_{p,l}] \subseteq I_{q,k}), \tag{InIntro}$$

$$\forall_{f,p}(f[I_{p,0}] \subseteq I), \tag{UcfBound}$$

$$\forall_{p,l}(\mathrm{Id}[I_{p,l}] \subseteq I_{p,l}), \tag{UcfId}$$

$$\forall_{f,p,l,q,k}(f[I_{p,l}] \subseteq I_{q,k} \to f[I_{p,l+1}] \subseteq I_{q,k}), \tag{UcfInputSucc}$$

$$\forall_{f,q}(q \le -\tfrac{1}{4} \to f[I] \subseteq I_{q,2} \to f[I] \subseteq I_{-1}), \tag{UcfLeft}$$

$$\forall_{f,q}(-\tfrac{1}{4} \le q \le \tfrac{1}{4} \to f[I] \subseteq I_{q,2} \to f[I] \subseteq I_0), \tag{UcfMiddle}$$

$$\forall_{f,q}(\tfrac{1}{4} \le q \to f[I] \subseteq I_{q,2} \to f[I] \subseteq I_1). \tag{UcfRight}$$

Axiom 2 (Abstract Theory of Real Numbers)

$$\forall_n(0 \in I_{0,n}), \tag{RealZero}$$

$$\forall_{x,p,n,d}(x \in I_{p,n} \to \tfrac{x+d}{2} \in I_{\frac{p+d}{2},n+1}), \tag{AvIntro}$$

$$\forall_{x,y,p,q,n}(x \in I_{p,n} \to y \in I_{q,n} \to \tfrac{x+y}{2} \in I_{\frac{p+q}{2},n}). \tag{RealAvrg}$$

Axiom 3 (Abstract Theory of Integration)

$$\forall_{f,d}(\textstyle\int^{\mathrm{H}} f = \frac{1}{2}(\int^{\mathrm{H}}(\mathrm{Out}_d \circ f) + d)), \tag{HIntOut}$$

$$\forall_f(\textstyle\int^{\mathrm{H}} f = \frac{1}{2}(\int^{\mathrm{H}}(f \circ \mathrm{In}_{-1}) + \int^{\mathrm{H}}(f \circ \mathrm{In}_1))), \tag{HIntIn}$$

$$\textstyle\int^{\mathrm{H}} \mathrm{Id} = 0, \tag{HIntId}$$

$$\forall_f(\textstyle\int^{\mathrm{H}} f \in I_{0,0}). \tag{HIntBound}$$