Proof-Relevant Resolution for Elaboration of Programming Languages

František Farka
University of St Andrews, UK, and
Heriot-Watt University, Edinburgh, UK
ff32@st-andrews.ac.uk
https://orcid.org/0000-0001-8177-1322

Abstract

Proof-relevant resolution is a new variant of resolution in Horn-clause logic and its extensions. We propose proof-relevant resolution as a systematic approach to elaboration in programming languages that is close to formal specification and hence allows for analysis of semantics of the language. We demonstrate the approach on two case studies; we describe a novel, proof-relevant approach to type inference and term synthesis in dependently types languages and we show how proof-relevant resolution allows for analysis of inductive and coinductive soundness of type class resolution. We conclude by a discussion of overall contributions of our current work.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases resolution, elaboration, proof-relevant, dependent types, type classes

Digital Object Identifier 10.4230/OASIcs.ICLP.2018.18

Acknowledgements This thesis abstract is based on published joint work with Ekaterina Komendatskaya and Kevin Hammond.

1 Introduction

First order resolution is a widely utilised technique in type inference. Hindley and Milner [13] were the first to notice that type inference in simply typed lambda calculus can be expressed as a first-order unification problem. This general scheme allows a multitude of extensions. For example, Hindley-Milner type system can be extended to constrained types system. For this extension, a constraint logic programming [15], was suggested, in which constraint solving over a certain domain was added to the existing first-order unification and resolution algorithms. Haskell type classes are another example of the application of logic programming. It is widely understood that type class resolution is in fact implemented as first-order resolution on Horn clauses. However, there is a caveat with respect to the traditional logic programming – a *dictionary* (that is, a proof term) needs to be constructed [14]. The research in the area of type classes is on-going: various extensions to the syntax of type classes are still being investigated [10].

Fu and Komendantskaya analysed type class resolution and proposed *proof-relevant* Horn clause logic [5] as the appropriate formalism. In this logic, Horn clauses are seen as types and proof witnesses as terms inhabiting the types. Given a proposition – a *goal* – and a set of Horn clauses – a *logic program* – the resolution process is captured by an explicit proof term construction. To briefly illustrate the proof-relevant approach, let us state the usual (generalised) modus ponens inference rule in a proof-relevant way:

$$\kappa : A \leftarrow B_1, \ldots, B_n \in \mathcal{P} \quad \frac{\mathcal{P} \vdash \delta_1 : \theta B_1 \qquad \ldots \qquad \mathcal{P} \vdash \delta_n : \theta B_n}{\mathcal{P} \vdash \kappa \delta_1 \ldots \delta_n : \theta A}$$

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 18; pp. 18:1–18:9

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

That is, for a substitution θ and an atom A, an instance θA can deduced in a program $\mathcal{P}$ assuming that there is a Horn clause $A \leftarrow B_1, \ldots, B_n$ in $\mathcal{P}$ and that each θ-instance of the atom B_i in the body of the clause can deduced in $\mathcal{P}$. Moreover, the clause is equipped with an atomic symbol κ (which is unique for each clause in the program). Assuming that deduction of the instance θB_i is witnessed by a proof term δ_i the overall deduction of θA is witnessed by a compound proof-term $\kappa\delta_1 \ldots \delta_n$.

In our work, we investigate proof-relevant resolution and we propose it as a systematic approach to elaboration of programming languages. In order to avoid an excessive technical detail, in this short paper we focus on demonstrating assets of proof-relevant resolution by means of an example. In particular, in Section 2 we show that proof-relevant resolution is a convenient calculus to formulate type inference and term synthesis for dependently typed languages, and in Section 3 we build on such treatment of type-class resolution and show that such approach is convenient for working with the semantics of the language as well; namely we show that proof-relevant treatment of type classes is sound both inductively and coinductively. We refer the reader to our published work [3, 4] for technical details.

2 Type Inference and Term Synthesis

In the last decade, dependent types [17, 1] have gained popularity in the programming language community. They allow reasoning about program values within the types, and thus give more general, powerful and flexible mechanisms to enable verification of code. However, such verification comes for a price. There are many proof obligations in form of computationally irrelevant terms that manifest that the code has properties that are stated in types, that themselves beige a specification become very complicated. Extensive automation of type inference and term (proof obligation) synthesis is a necessity for any system that aims to be usable in practice. We propose a novel approach for such automation. We use the notion *refinement* to refer to the combined problem of type inference and term synthesis.

Using an abstract syntax that closely resembles existing functional programming language we define $\texttt{maybe}_{\texttt{A}}$, an option type over a fixed type `A`, indexed by a Boolean:

▶ **Example 1.**

```
data maybeA (a : A)   : bool → type where
     nothing          : maybeA ff
     just             : A → maybeA tt
```

Here, `nothing` and `just` are the two *constructors* of the `maybe` type. The type is indexed by `ff` when the `nothing` constructor is used, and by `tt` when the `just` constructor is used (`ff` and `tt` are constructors of `bool`). A function `fromJust` extracts the value from the `just` constructor:

```
fromJust : maybeA tt → A
fromJust (just x) = x
```

Note that the value `tt` appears within the type $\texttt{maybe}_{\texttt{A}}\ \texttt{tt} \rightarrow \texttt{A}$ of this function (the type *depends* on the value), allowing for a more precise function definition that omits the redundant case when the constructor of type $\texttt{maybe}_{\texttt{A}}$ is `nothing`. The challenge for the type checker is to determine that the missing case `fromJust nothing` is contradictory (rather than being omitted by mistake). Indeed, the type of `nothing` is $\texttt{maybe}_{\texttt{A}}\ \texttt{ff}$. However, the function specifies its argument to be of type $\texttt{maybe}_{\texttt{A}}\ \texttt{tt}$.

```
A : type
bool : type
ff tt : bool

(≡bool) : bool → bool → type
refl : Π(b:bool).b ≡bool b
elim≡bool : tt ≡bool ff → A

maybeA : bool → type
nothing : maybeA ff
just : A → maybeA tt
elim maybeA : Π(b:bool).maybeA b
→ (b ≡bool ff → A)
→ (b ≡bool tt → A → A)
→ A
```

Figure 1 Signature of t_{fromJust}.

To type check such functions, the compiler translates them into terms in an *internal*, type-theoretic *calculus*. We rely on the calculus LF [9], a standard and well-understood first-order dependent type theory as a choice of such internal calculus. A signature that we use to encode our example in LF is give in Figure 1. We employ $A \to B$ as an abbreviation for $\Pi(a : A).B$ where a does not occur free in B. The final goal of type checking of the function `fromJust` in the programming language is to obtain the following encoding in the internal calculus:

▶ **Example 2.**

```
tfromJust := λ (m:maybeA tt).elim maybeA tt m
(λ (w:tt≡boolff).elim≡bool w)
(λ (w:tt≡booltt).λ (x:A).x)
```

The missing case for `nothing` must be accounted for (*cf.* the line `(λ (w:tt≡boolff).elim≡bool w)` above). In this example (as is generally the case), only partial information is given in the programming language. To address this problem, we extend the internal language with term level metavariables, denoted by $?_a$, and type level metavariables, denoted by $?_A$. These stand for the parts of a term in the internal language that are not yet known. Using metavariables, the term that directly corresponds to `fromJust` is:

▶ **Example 3.**

```
tfromJust := λ (m:maybeA tt).elim maybeA ?a m
(λ ( w: ?A ). ?b )
(λ ( w: ?B ).λ ( x:A).x)
```

The missing information comprises the two types $?_A$ and $?_B$ and the term $?_b$ for the constructor `nothing`. Obtaining types $?_A$, $?_B$ amounts to type inference whereas obtaining the term $?_b$ amounts to term synthesis. We translate *refinement problems* into the syntax of logic programs. The *refinement* algorithm that we propose takes a signature and a term with metavariables in the extended internal calculus to a logic program and a goal in proof-relevant Horn clause logic. The unifiers that are computed by resolution give an assignment of types

to type-level metavariables. At the same time, the computed proof terms are interpreted as an assignment of terms to term-level metavariables. We illustrate the process in the following paragraphs.

Consider the inference rule Π-T-ELIM in LF for application of a dependent function to an argument:

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}\ \Pi\text{-T-ELIM}$$

When type checking the term $t_{\texttt{fromJust}}$ an application of $\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m$ to the term $\lambda(w : ?_A).?_b$ in the context $m : \texttt{maybe}_\texttt{A}\ \texttt{tt}$ needs to be type checked. This amounts to providing a derivation of the typing judgement that contains the following instance of the rule Π-T-ELIM:

$$\frac{\begin{array}{l} m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m \\ \qquad : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}) \to \cdots \to \texttt{A} \end{array} \qquad m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \lambda(w : ?_A).?_b : ?_A \to ?_B}{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash (\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)\ (\lambda(w : ?_A).?_b) : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{tt} \to \texttt{A} \to \texttt{A}) \to \texttt{A}}$$

For the above inference step to be a valid instance of the inference rule Π-T-ELIM, it is necessary that $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) = ?_A$ and $\texttt{A} = ?_B$. This is reflected in the following goal:

$$((\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) = ?_A) \wedge (\texttt{A} = ?_B) \wedge G_{(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)} \wedge G_{\lambda(w:?_A).?_b} \tag{1}$$

The additional goals $G_{(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)}$ and $G_{\lambda(w:?_A).?_b}$ are recursively generated by the algorithm for the terms $\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m$ and $\lambda(w : ?_A).?_b$, respectively. Similarly, assuming the term $\lambda(w : ?_A).?_b$ is of type $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) \to \texttt{A}$, type checking places restrictions on the term $?_b$:

$$\frac{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} : \texttt{type} \qquad m : \texttt{maybe}_\texttt{A}\ \texttt{tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \vdash ?_b : \texttt{A}}{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \lambda(w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}).?_b : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}}$$

That is, $?_b$ needs to be a well-typed term of type $\texttt{A}$ in a context consisting of m and w. Recall that in the signature there is a constant $\texttt{elim}_{\equiv_{\texttt{bool}}}$ of type $\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}$. Our translation will turn this constant into a clause in the generated logic program. There will be a clause that corresponds to the inference rule for elimination of a Π type as well:

$$\begin{array}{rl} \kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}} : & term(\texttt{elim}_{\equiv_{\texttt{bool}}}, \Pi x : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}\,.\,\texttt{A}, ?_\Gamma) \leftarrow \\ \kappa_{\texttt{elim}} : & term(?_M ?_N, ?_B, ?_\Gamma) \leftarrow term(?_M, \Pi x : ?_A.?_{B'}, ?_\Gamma) \wedge term(?_N, ?_A, ?_\Gamma) \wedge ?_{B'}[?_N/x] \equiv ?_B \end{array}$$

The above clauses are written in the proof-relevant Horn clause logic, and thus $\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}$ and $\kappa_{\texttt{elim}}$ now play the role of proof-term symbols ("witnesses" for the clauses). In this clause, $?_M$, $?_N$, $?_A$, $?_B$, $?_{B'}$ and $?_\Gamma$ are *logic variables*, i.e. variables of the first-order logic. By an abuse of notation, we use the same symbols for metavariables of the internal calculus and logic variables in the logic programs generated by the refinement algorithm. We also use the same notation for objects of the internal language and terms of the logic programs. This is possible since we represent variables using *de Bruijn indices*.

The presence of $w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}$ in the context allows us to use the clause $\texttt{elim}_{\equiv_{\texttt{bool}}}$ to resolve the goal $term(?_M ?_N, \texttt{A}, [m : \texttt{maybe}_\texttt{A}\ \texttt{tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}])$:

$$\begin{array}{l} term(?_M ?_N, \texttt{A}, [m : \texttt{maybe}_\texttt{A}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \rightsquigarrow_{\kappa_{\texttt{elim}}} \\ \quad term(?_M, \Pi x : ?_A.\,\texttt{A}, [\ldots]) \wedge term(?_N, ?_A, [\ldots, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \wedge \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}} \\ \quad term(?_N, \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}, [\ldots, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \wedge \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{proj}_w}} \\ \quad \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{subst}_\texttt{A}}} \bot \end{array} \tag{2}$$

The resolution steps are denoted by $\rightsquigarrow$. Each step is indexed by the name of the clause that was used. First, the goal is resolved in one step using the clause $\kappa_{\texttt{elim}}$. A clause $\kappa_{\texttt{proj}_w}$ is used to project the variable w from the context. We omit a discussion of the exact shape of the clauses since it depends on the representation we use. In this presentation, we are just interested in composing the proof terms occurring in these resolution steps into one composite proof term: $\kappa_{\texttt{elim}}\ \kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}\ \kappa_{\texttt{proj}_w} \kappa_{\texttt{subst}_{\texttt{A}}}$. Note that, by resolving the goal (1), we obtain a substitution θ that assigns the type A to the logic variable $?_B$, *i.e.* $\theta(?_B) = A$. At the same time, the proof term computed by the the derivation (2) is interpreted as a solution $(\texttt{elim}_{\equiv_{\texttt{bool}}}\ w)$ for the term-level metavariable $?_b$. However, the proof term can be used to reconstruct the derivation of well-typedness of the judgement $m : \texttt{maybe}_{\texttt{A}}\ \texttt{tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \vdash \texttt{elim}_{\equiv_{\texttt{bool}}}\ w : \texttt{A}$ as well. In general, a substitution is interpreted as a solution to a type-level metavariable and a proof term as a solution to a term-level metavariable. The remaining solution for $?_A$ is computed using similar methodology, and we omit the details here. Thus, we have computed values for all metavariables in Example 3, *i.e.* we inferred all types and synthesised all terms.

3 Type Class Resolution

Type classes are a versatile language construct for implementing ad-hoc polymorphism and overloading in functional languages. The approach originated in Haskell [16, 8] and has been further developed in dependently typed languages [7, 2]. For example, it is convenient to define equality for all data structures in a uniform way. In Haskell, this is achieved by introducing the equality class `Eq`:

▶ **Example 4.**

```
class Eq x where
 eq :: Eq x ⇒ x → x → Bool
```

and then declaring any necessary instances of the class, e.g. for pairs and integers:

```
instance (Eq x, Eq y) ⇒ Eq (x, y) where
 eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2
instance Eq Int where
 eq x y = primtiveIntEq x y
```

Type class resolution is performed by the Haskell compiler and involves checking whether all the instance declarations are valid. For example, the following function triggers a check that `Eq (Int, Int)` is a valid instance of type class `Eq`:

```
test :: Eq (Int, Int) ⇒ Bool
test = eq (1,2) (1,2)
```

It is folklore that type class instance resolution resembles SLD-resolution from logic programming. The type class instance declarations above could, for example, be viewed as the following two Horn clauses:

▶ **Example 5** (Logic program P_{Pair}).

$$\kappa_{\text{pair}} : \texttt{eq}(x),\ \texttt{eq}(y) \Rightarrow \texttt{eq}(\texttt{pair}(x, y))$$
$$\kappa_{\text{int}} : \Rightarrow \texttt{eq}(\texttt{int})$$

Then, given the query $\texttt{eq}(\texttt{pair}(\texttt{int}, \texttt{int}))$, resolution terminates successfully with the following sequence of inference steps:

$$\texttt{eq}(\texttt{pair}(\texttt{int}, \texttt{int})) \rightarrow_{\kappa_{\text{pair}}} \texttt{eq}(\texttt{int}), \texttt{eq}(\texttt{int}) \rightarrow_{\kappa_{\text{int}}} \texttt{eq}(\texttt{int}) \rightarrow_{\kappa_{\text{int}}} \emptyset$$

The proof witness $\kappa_{\text{pair}}\kappa_{\text{int}}\kappa_{\text{int}}$ (called a "dictionary") is constructed by the compiler. This is treated internally as an executable function.

Despite the apparent similarity of type class syntax and type class resolution to Horn clause resolution they are not, however, identical. At a syntactic level, type class instance declarations correspond to a restricted form of Horn clauses, namely ones that:

- **(i)** do not *overlap* (*i.e.* whose heads do not unify); and that
- **(ii)** do not contain existential variables (*i.e.* variables that occur in the bodies but not in the heads of the clauses). At an algorithmic level,
- **(iii)** type class resolution corresponds to Horn-clause resolution in which unification is restricted to term-matching.

Assuming there is a clause $B_1, \ldots B_n \Rightarrow A'$, then a query $?\ A'$ can be resolved with this clause only if A can be matched against A', *i.e.* if a substitution σ exists such that $A = \sigma A'$. In comparison, Horn-clause resolution incorporates *unifiers*, as well as *matchers*, *i.e.* it also proceeds to resolve the above query and clause in all the cases where $\sigma A = \sigma A'$ holds.

These restrictions guarantee that type class inference computes the *principal* (most general) type. Restrictions (i) and (ii) amount to deterministic inference by resolution, in which only one derivation is possible for every query. Restriction (iii) means that no substitution is applied to a query during inference, *i.e.* we prove the query in an implicitly universally quantified form. It is common knowledge that (as with Horn-clause resolution) type class resolution is *inductively sound*, *i.e.* that it is sound relative to the least Herbrand models of logic programs [12]. Moreover we established [3], for the first time, that it is also *universally inductively sound*, *i.e.* that if a formula A is proved by type class resolution, every ground instance of A is in the least Herbrand model of the given program. In contrast to Horn-clause resolution, however, type class resolution is *inductively incomplete*, *i.e.* it is incomplete relative to least Herbrand models, even for the class of Horn clauses that is restricted by conditions i and ii. For example, given a clause $\Rightarrow \texttt{q}(\texttt{f}(x))$ and a query $?\ \texttt{q}(x)$, Horn-clause resolution is able to find a proof (by instantiating x with $\texttt{f}(x)$), but type class resolution fails. Lämmel and Peyton Jones have suggested [11] an extension to type class resolution that accounts for some non-terminating cases of type class resolution. Consider, for example, the following mutually defined data structures:

▶ **Example 6.**

```
data OddList a = OCons a (EvenList a)
data EvenList a = Nil | ECons a (OddList a)
```

which give rise to the following instance declarations for the `Eq` class:

```
instance (Eq a, Eq (EvenList a)) ⇒ Eq (OddList a) where
 eq (OCons x xs) (OCons y ys) = eq x y && eq xs ys

instance (Eq a, Eq (OddList a)) ⇒ Eq (EvenList a) where
 eq Nil Nil = True
 eq (ECons x xs) (ECons y ys) = eq x y && eq xs ys
 eq _ _ = False
```

The `test` function below triggers type class resolution in the Haskell compiler:

```
test :: Eq (EvenList Int) ⇒ Bool
test = eq Nil Nil
```

However, inference by resolution does not terminate in this case. Consider the Horn clause representation of the type class instance declarations:

▶ **Example 7** (Logic program $P_{EvenOdd}$).

$$\begin{aligned} \kappa_{\text{odd}} &: \texttt{eq}(x),\ \texttt{eq}(\texttt{evenList}(x)) &&\Rightarrow \texttt{eq}(\texttt{oddList}(x)) \\ \kappa_{\text{even}} &: \texttt{eq}(x),\ \texttt{eq}(\texttt{oddList}(x)) &&\Rightarrow \texttt{eq}(\texttt{evenList}(x)) \\ \kappa_{\text{int}} &: &&\Rightarrow \texttt{eq}(\texttt{int}) \end{aligned}$$

The non-terminating resolution trace is given by:

$$\begin{aligned} &\underline{\texttt{eq}(\texttt{evenList}(\texttt{int}))} \to_{\kappa_{\text{even}}} \texttt{eq}(\texttt{int}),\ \texttt{eq}(\texttt{oddList}(\texttt{int})) \to_{\kappa_{\text{int}}} \texttt{eq}(\texttt{oddList}(\texttt{int})) \\ &\qquad \to_{\kappa_{\text{int}}} \texttt{eq}(\texttt{int}),\ \texttt{eq}(\texttt{evenList}(\texttt{int})) \to_{\kappa_{\text{int}}} \underline{\texttt{eq}(\texttt{evenList}(\texttt{int}))} \to_{\kappa_{\text{even}}} \dots \end{aligned}$$

A goal $\texttt{eq}(\texttt{evenList}(\texttt{int}))$ is simplified using the clause κ_{even} to two new goals $\texttt{eq}(\texttt{int})$ and $\texttt{eq}(\texttt{oddList}(\texttt{int}))$. The first of these is discarded using the clause κ_{int}. Resolution continues using κ_{odd} and κ_{int}, resulting in the original goal $\texttt{eq}(\texttt{evenList}(\texttt{int}))$. It is easy to see that such a process could continue infinitely and that this goal constitutes a *cycle* (underlined above). As suggested by Lämmel and Peyton Jones [11], the compiler can terminate the infinite inference process as soon as it detects the underlined cycle. Moreover, it can also construct the corresponding proof witness in a form of a recursive function. For the example above, such a function is given by the fixed point term $\nu\alpha.\kappa_{\text{even}}\kappa_{\text{int}}(\kappa_{\text{odd}}\kappa_{\text{int}}\alpha)$, where ν is a fixed point operator. The intuitive reading of such a proof is that an infinite proof of the query $\texttt{eq}\ (\texttt{evenList}(\texttt{int}))$ exists, and that its shape is fully specified by the recursive proof witness function above. We say that the proof is given by *corecursive type class resolution*.

Corecursive type class resolution is not inductively sound. For example, the formula $\texttt{eq}(\texttt{evenList}(\texttt{int}))$ is not in the least Herbrand model of the corresponding logic program. However, we proved [3] that it is *(universally) coinductively sound*, *i.e.* it is sound relative to the greatest Herbrand models. For example, $\texttt{eq}(\texttt{evenList}(\texttt{int}))$ is in the greatest Herbrand model of the program $P_{EvenOdd}$. Similarly to the inductive case, corecursive type class resolution is coinductively incomplete. Consider the clause $\kappa_{\text{inf}} : \texttt{p}(x) \Rightarrow \texttt{p}(\texttt{f}(x))$. This clause may be given an interpretation by the greatest (complete) Herbrand models. However, corecursive type class resolution does not yield infinite proofs.

Unfortunately, this simple method of cycle detection does not work for all non-terminating programs. Consider the following example, which defines a data type `Bush` (for bush trees), and its corresponding instance for `Eq`:

```
data Bush a = Nil | Cons a (Bush (Bush a))
instance Eq a, Eq (Bush (Bush a)) ⇒ Eq (Bush a) where { ... }
```

Here, type class resolution does not terminate. However, it does not exhibit cycles either. Consider the Horn clause translation of the problem:

▶ **Example 8** (Logic program P_{Bush}).

$$\begin{aligned} \kappa_{\text{int}} &: &&\Rightarrow \texttt{eq}(\texttt{int}) \\ \kappa_{\text{bush}} &: \texttt{eq}(x),\ \texttt{eq}(\texttt{bush}(\texttt{bush}(x))) &&\Rightarrow \texttt{eq}(\texttt{bush}(x)) \end{aligned}$$

The derivation below shows that no cycles arise when we resolve the query ? $\texttt{eq}(\texttt{bush}(\texttt{int}))$ against the program P_{Bush}:

$$\texttt{eq}(\texttt{bush}(\texttt{int})) \to_{\kappa_{\text{bush}}} \texttt{eq}(\texttt{int}), \texttt{eq}(\texttt{bush}(\texttt{bush}(\texttt{int})) \to_{\kappa_{\text{int}}} \ldots \to_{\kappa_{\text{bush}}} \texttt{eq}(\texttt{bush}(\texttt{int})), \texttt{eq}(\texttt{bush}(\texttt{bush}(\texttt{bush}(\texttt{int})))) \to_{\kappa_{\text{int}}} \ldots$$

Fu *et al.* [6] have introduced an extension to corecursive type class resolution that allows implicative queries to be proved by corecursion and uses the recursive proof witness construction. Implicative queries require the language of proof terms to be extended with λ-abstraction. For example, in the above program the Horn formula $\texttt{eq}(x) \Rightarrow \texttt{eq}(\texttt{bush}(x))$ can be (coinductively) proven with the recursive proof witness $\kappa_{\text{new}} = \nu\alpha.\lambda\beta.\kappa_{\text{bush}}\beta(\alpha(\alpha\beta))$. If we add this Horn clause as a third clause to our program, we obtain a proof of $\texttt{eq}(\texttt{bush}(\texttt{int}))$ by applying κ_{new} to κ_{int}. In this case, it is even more challenging to understand whether the proof $\kappa_{\text{new}}\kappa_{\text{int}}$ of $\texttt{eq}(\texttt{bush}(\texttt{int}))$ is indeed sound: whether inductively, coinductively or in any other sense. We established [3], for the first time, *coinductive soundness* for proofs of such implicative queries, relative to the greatest Herbrand models of logic programs. Namely, we determined that proofs that are obtained by extending the proof context with coinductively proven Horn clauses (such as κ_{new} above) are coinductively sound but inductively unsound. This result demonstrates feasibility of proof-relevant approach for study of the semantic properties of elaboration of programming language constructs.

4 Contributions

In our work, we study proof-relevant resolution as a systematic approach to elaboration in programming languages. We argue that proof-relevant resolution is an appropriate technique for elaboration while it stays very close to formal specification and allows for analysis of semantics. In this short paper, we demonstrate this claim on two examples. We discuss refinement if dependently typed languages and soundness of type class resolution.

The main contributions of our work are:

1. We present a novel approach to refinement for a first-order type theory with dependent types;
2. we prove that our approach (*i.e.* generation of goals and logic programs) is decidable and hence can serve as a basis for a verified implementation;
3. we show that proof-relevant first-order Horn clause resolution gives an appropriate inference mechanism for dependently typed languages: firstly, it is sound with respect to type checking in LF; secondly, the proof term construction alongside the resolution trace allows to reconstruct the derivation of well-typedness judgement.
4. We show that proof-relevant approach to type class resolution and its two recent corecursive extensions [6, 11] are sound relative to the standard (Herbrand model) semantics of logic programming; and
5. we show that these new extensions are indeed "corecursive", *i.e.* that they are modelled by the greatest Herbrand model semantics rather than by the least Herbrand model semantics.

References

1 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

2 Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *Proc. of ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143–155, 2011. doi:10.1145/2034773.2034796.

3 Frantisek Farka, Ekaterina Komendantskaya, and Kevin Hammond. Coinductive Soundness of Corecursive Type Class Resolution. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 2016. doi:10.1007/978-3-319-63139-4_18.

4 František Farka, Ekaterina Komendantskaya, and Kevin Hammond. Proof-relevant Horn Clauses for Dependent Type Inference and Term Synthesis. In *accepted to ICLP 2018*, 2018.

5 Peng Fu and Ekaterina Komendantskaya. Operational semantics of resolution and productivity in Horn clause logic. *Formal Asp. Comput.*, 29(3):453–474, 2017. doi:10.1007/s00165-016-0403-1.

6 Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. Proof Relevant Corecursive Resolution. In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, volume 9613 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2016. doi:10.1007/978-3-319-29604-3_9.

7 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *Proc. of ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 163–175, 2011. doi:10.1145/2034773.2034798.

8 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. doi:10.1145/227699.227700.

9 Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.*, 6(1):61–101, 2005. doi:10.1145/1042038.1042041.

10 Georgios Karachalias and Tom Schrijvers. Elaboration on functional dependencies: functional dependencies are dead, long live functional dependencies! In Iavor S. Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 133–147. ACM, 2017. doi:10.1145/3122955.3122966.

11 Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. of ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 204–215, 2005. doi:10.1145/1086365.1086391.

12 John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

13 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

14 Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, Amsterdam, January 1997. URL: https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/.

15 Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *J. Funct. Program.*, 18(2):251–283, 2008. doi:10.1017/S0956796807006569.

16 P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proc. of POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM. doi:10.1145/75277.75283.

17 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *PACMPL*, 1(ICFP):31:1–31:29, 2017. doi:10.1145/3110275.