Proof-Relevant Resolution for Constructive Automation

František Farka

April 29, 2019

Hi thank you for having me, it is my pleasure to be here. I hope you will enjoy the talk.

The talk I am going to deliver today is about PRR, a framework for constructive automation. Let me start with what is the context of the talk; or what are we going to automate. Automation is just a tool, a means. It is not the end in itself. The context of the talk is programming languages. And for me, that means programming languages with strong typing discipline. A lot of people is interested in safety and security of software, in verification. There are different ways how to approach verification of software and one of the ways is by types. An expressive type system allows you to encode specifications as types and a type checker then ensures that the code adheres to specification.

Let us have a look at an example.

1 Automation for Programming Languages

Type inference, term synthesis, and type classes

```
-- type inference, term synthesis
data maybe_A : Bool → Set where
  nothing : maybe_A false
  just : A → maybe_A true

fromJust : maybe_A true → A
fromJust (just x) = x
fromJust = λ (m : maybe_A true) →
        elim_maybe_A  true  m
          -- nothing
          (λ (w :  true ≡ false )
            →  elim≡  w )
          -- just x
          (λ (w :  true ≡ true ) (x : A)
            → x)
```

```
-- resolution trace:

term (?_M ?_N) A [m : maybe_A, w : tt ≡_bool ff]

  ⇝_κelim

  term ?_M (Πx : ?_A. A) [...] ∧ term ?_N ?_A
    [..., w : tt ≡_bool ff] ∧ A[?_N/x] ≡ ?_B

  ⇝_κelim≡bool

  term ?_N tt ≡_bool ff [..., w : tt ≡_bool ff] ∧
    A[?_N/x] ≡ ?_B

  ⇝_κproj_w

  A[?_N/x] ≡ ?_B

  ⇝_κsubst_A ⊥

-- proof-term:

    κelim κelim≡bool κproj_w κsubst_A
```

First, consider this simple example (on the left). If you are familiar with Haskell, you will know the maybe data type that allows you to store a value or indicate its absence. You might recognise it from other languages as option. This is its dependent version, it carries a boolean index in the type that indicates whether the value is present or not.

In Haskell, the data type comes with function fromJust with signature maybe → A that allows you to extract the value. Such function is partial, when the value of the data type is nothing, there is nothing to extract. Calling the function with value nothing will raise an error in runtime, which is something we would like to avoid. Here come dependent types to rescue. Our version of maybe is indexed and we can specify that fromJust can be called only with arguments that actually contain a value of the underlying type.

However, guaranteeing such properties does not come for free. Most often, a type-checker, or a tool in general, translates the surface representation into an internal calculus of a type theory. The internal representation of the function fromJust needs to satisfy a range of obligations raised by the technical nature of type theory. For example, here the definiton by pattern matching is internally represented using an induction principle. Note that some information that is re-

quired in the internal representation, like the term for absurd case of constructor nothing, does not occure in the above definition. Similarly, certain types need to be provided, like the types in positions of $?_A$ and $?_B$. We will understand these as metavariables.

This is the task for automation – to provide this missing information in such terms. Obviously, this is a very simplistic example and you can imaging the code geting arbitrarily complex. Also, such automation is not necessarly limited to translation of a surface language to an internal representation. A similar problem occurs in the interactive style of depenent type programming, for example in Agda or Idris, where you expect the development environment to inhabit holes in your programs upon request

Consider one more example. Perhaps you are familiar with type class mechanism in Haskell. Type class mechanism is a mechanism that provides ad-hoc polymorpishm. You can declare a class of types, in this example a type of class equipped with equality. A class provides certain methods that are defined for members of the class. Types are identified as members of the class via instance mechanism. In our example, instance of the typeclass for Int is carried out by some primitive notion of equality for integers. In the case of pairs, we can see the compositionality of the mechanism, equality for pairs is defined assuming there are instances for types of each component. The implementation is then the obvious, pairwise.

In a use site, like the function test in our example, compiler automatically composes in a process called *type class resolution* the available instances to obtain instance of the desired type. Again, the compiler translates Haskell program to an internal representation. There, type classes are elaborated away. In technical terms, the type class method is given an extra argument, a dictionary, that witnesses that a type is member of the type class. In use site, type class resolution provides a dictionary of the required type. Again, this is a clear use-case for automation.

The framework I am going to present addresses exactly these use-cases for automation. First, let me stress out what requirements are laid upon the framework by these use cases.

First, the framework needs to be fairly general and principled, and that is in several ways. First, in is easy to imagine a language of your choice combines both these two features and many others that require automation during elaboration into an internal, explicit representation.

It has been argued that such a general framework is provided by Horn clauses, both for verification of imperative programs [1] and for functional programs [2, 4]. If you think about the example of type classes, instances exactly correspond to Horn clauses. The problem of finding the dictionary is then Horn-clause resolution of the desired goal.

Secondly, we want the framework to be proof-relevant. In the example of type

classes proof-relevance is essential. I already said that we need the dictionary as it has operational interpretation. First, a pair of intergers is compared for equality pairwise, which corresponds to the second instance, then, each of integers is compared using some primitive notion, which corresponds to the first instance. Assume we equip the instances, now Horn clauses, with atomic symbols κ_{Pair} and κ_{Int}, names that identify them. Then the resolution can by captured by the term $\kappa_{\text{Pair}}\ \kappa_{\text{Int}}\ \kappa_{\text{Int}}$. And this dictionary is exactly the proof term that witnesses instance resolution.

We can work with the other example in a similar way. It is more complicated and I am not going to do it now.

How are we going to design such a framework? We are going to work in extension of HC, HH. It is more convenient as it will allow us Curry-Howard interpretation, but it is also neccessary as I will argue later; I will show examples of type class problem that extends HC fragment.

We are going to look at automation in our framework as goal-directed search in HH. The function with metavariables, or the desired instance are going to be represented as a goal. Via Curry-Howard interpretation, we are going to instrument formulae with proof terms and the search trace will be captured by this proof term.

2 Proof-Relevant Resolution

Big-step operational semantics

$$
\begin{aligned}
D &:= A \mid G \Rightarrow D \mid \forall x : A.D \\
G &:= A \mid D \Rightarrow G \mid \forall x : A.G \mid \exists x : A.G \\
e &:= \kappa \mid e\ e \mid \lambda\kappa.e \mid \langle M, e\rangle
\end{aligned}
$$

$\boxed{\mathcal{S};\mathcal{P} \xrightarrow{e':D} e : A}$

$$\frac{}{\mathcal{S};\mathcal{P} \xrightarrow{e:A} e : A}\ \text{init}$$

$$\frac{\mathcal{S};\mathcal{P} \longrightarrow e_1 : A_1 \qquad \mathcal{S};\mathcal{P} \xrightarrow{e\,e_1:D} e_2 : A_2}{\mathcal{S};\mathcal{P} \xrightarrow{e:A_1 \Rightarrow D} e_2 : A_2}\ \Rightarrow\text{L}$$

$$\frac{\mathcal{S};\mathcal{P} \xrightarrow{e:D[M/x]} e_2 : A_2 \qquad \mathcal{S};\cdot \vdash M : A_1}{\mathcal{S};\mathcal{P} \xrightarrow{e:\forall x:A_1.D} e_2 : A_2}\ \forall\text{L}$$

$\boxed{\mathcal{S}; \mathcal{P} \longrightarrow e : G}$

$$\frac{\mathcal{S}; \mathcal{P} \xrightarrow{\kappa : D} e : A \qquad \kappa : D \in \mathcal{P}}{\mathcal{S}; \mathcal{P} \longrightarrow e : A}\ \text{decide}$$

$$\frac{\mathcal{S}; \mathcal{P} \longrightarrow e : G[M/x] \qquad \mathcal{S}; \cdot \vdash M : A}{\mathcal{S}; \mathcal{P} \longrightarrow \langle M, e \rangle : \exists x : A.G}\ \exists\text{R}$$

$$\frac{\mathcal{S}; \mathcal{P}, \kappa : D \longrightarrow e : G}{\mathcal{S}; \mathcal{P}, \kappa : D \longrightarrow \lambda\kappa.e : D \Rightarrow G}\ \Rightarrow\text{R}$$

$$\frac{\mathcal{S}, c : A; \mathcal{P} \longrightarrow e : G[c/x]}{\mathcal{S}; \mathcal{P} \longrightarrow e : \forall x : A.G}\ \forall\text{R}$$

The language contains definite clauses D that specify the search space, goals G for the automation, and proof terms e. For the sake of simplicity, we are going to keep the language minimal and consider only implication and existential and universal quantification. Definite clauses consist of atoms, implication that has a goal as a premise and a definite clause as a conclusion, and universal quantification over a definite clause. The language of atoms I consider is LF, a first-order dependent type theory to allow for application like those I showed on the previous slide. Atoms then are types of certain proper kind. Though the particular shape of atoms I consider is not important for the rest of this talk. The syntax of goals consists of the same constructs with roles of goals and clauses reversed, and existential quantification over goals. Proof-terms consists of atomic symbols, application, abstraction and existential witness.

The operational semantics of resolution we give to the language is uniform proof semantics, which is well-understood semantics of logic programming. The novelty is that we equip uniform proofs with proof terms. If you are not familiar with uniform proofs; it is a semantics that can be traced back to Gentzen's sequent calculus, which is a nice feature here if you think we are giving a Curry-Howard interpretation and connects our framework nicely with foundations. If you recall, in Gentzen's work there are sequent calculi for systems LK for classical logic and LJ for intuitionistic logic. Since we work in constructive settings, the appropriate calculus is LJ. The characterising feature of LJ is that there is exactly one formula on right hand side of the sequent. This formula is our goal formula. The formulae on left-hand side of the sequent are collections of definite clauses, which are by convention called programs. The goal-directed search means that goals are decomposed using right-introduction rules of the sequent calculus. This is given by the judgement on the right-hand side of the slide. When a goal cannot be decomposed further using right-introduction rules, a program clause is selected and this clause is decomposed using left-introduction rules. This is the difference between uniform proofs and sequent calculus. One clause is selected and it is this clause only being decomposed in the process. This is given by the judgement on the left-hand side of the slide. Note the symmetry in definition of resolution that corresponds to the symmetry

in definition of goals and clauses. Finally, we collect proof-terms alongside the decomposition of goals and clauses.

However, if you think about resolution and logic programming, computation is carried out by unification. Here, there is no unification involved. This is a big step semantics and does not provide a computational device. The resolution proceeds by decomposition of the goal and a well-formed term needs to be provided right away is in the case of $\exists$R rule. In order to provide a computational device we will develop a small-step operational semantics.

Small-step operational semantics

$\boxed{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}'' : D}{\rightsquigarrow} \Gamma' \mid \hat{e}'}$

$$\frac{\mathcal{S}; \Gamma \vdash \sigma : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \sigma A \equiv \sigma A' : o}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e} : A'}{\rightsquigarrow} \Gamma' \mid (\sigma C)\{\hat{e}\}}$$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}_1\, A_1 : D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}_1 : A_1 \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid C\{A_2\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A_2\} \overset{\hat{e}_1 : \forall x : A_1 . D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$$

$$\hat{e} := \kappa \mid G \mid \hat{e}\,\hat{e} \mid \langle M, \hat{e} \rangle \mid \lambda\kappa.\hat{e}$$
$$C := \bullet \mid e\ C \mid \langle M, C \rangle \mid \lambda\kappa.C$$

$\boxed{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'}$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\kappa : D}{\rightsquigarrow} \Gamma' \mid \hat{e} \qquad \kappa : D \in \mathcal{P}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \rightsquigarrow \Gamma' \mid \hat{e}}$$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid C\{\langle Y, G[Y/x] \rangle\} \rightsquigarrow \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\exists x : A.G\} \rightsquigarrow \Gamma' : A \mid \hat{e}'}$$

$$\frac{\mathcal{S}; \mathcal{P}, \kappa : D \vdash \Gamma \mid C\{\lambda\kappa.G\} \rightsquigarrow \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{D \Rightarrow G\} \rightsquigarrow \Gamma' \mid \hat{e}}$$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma, x : A \mid C\{G\} \rightsquigarrow \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\forall x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}}$$

I am not going to go into a detail here and I will just highlight the essential features. The key characteristics is the use of what we call *mixed terms*. Mixed terms combine proof-terms with not-yet-resolved goals and describe an intermediate state of the resolution. Resolution proceeds by rewriting goals in a mixed term into mixed terms. As there might be several goals in a mixed term, we use *rewriting contexts* to identify the goal that is being rewritten. Rewriting context consists of a hole, that identifies the mixed term that is subject to a rewriting step and other syntactic constructs that follow structure of mixed terms. The structure of judgements of small-step operational semantics corresponds to the big step semantics. The difference is that terms that need to be provided straight away in the big step operational semantics are delayed as unification variables. The initial sequent is replaced by unification. Note that the substitution needs to be well-formed in the type theory of the language of atoms.

As a certain sanity check, we can recover notion familiar from logic programming, which we used as a motivation for the design of the framework. Namely, if you compose substitutions computed in the unification steps, you obtain an answer substitution. The judgement of small-step semantics can then be read as giving an answer substitution from domain Γ to codomain Γ'. Similarly, if you restrict to a first order language of atoms you can recover implicit quantification, for an untyped language the notions of predicate and function symbols, and arity.

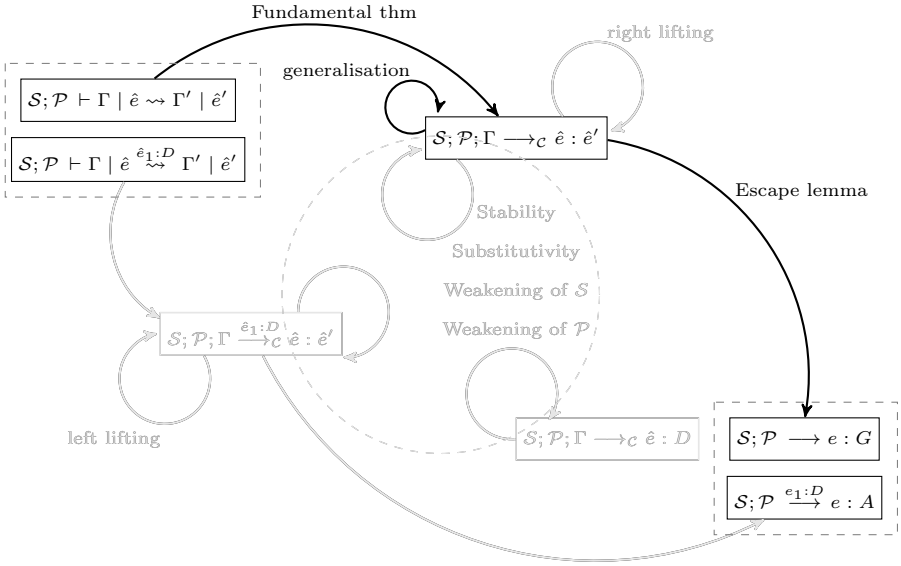
What we need to show is, that this semantics is sound w.r.t the big-step semantics.

Soundness

Theorem 1 (Soundness)**.** *If* $\mathcal{S}; \mathcal{P} \vdash \cdot \mid G \rightsquigarrow \cdot \mid e$ *then* $\mathcal{S}; \mathcal{P} \longrightarrow e : G$.

Theorem 2 (Generalised soundness)**.** *If* $\mathcal{S}; \mathcal{P} \vdash \cdot \mid G \rightsquigarrow \Gamma' \mid e$ *then* $\mathcal{S}; \mathcal{P} \longrightarrow e : \forall \Gamma'.G$.

Theorem of soundness looks like above, for a fully quantified goal the resolution is sound as long as we can read a proper proof term, that is not a mixed term, in an empty context. We can allows ourselves a slightly more general general result. We can allow logical variables in the codomain of the answer substitution and then these have universal interpretation. The syntax $\forall \Gamma'.G$ stand for quantification of goal with each variable in the context.



The question is how to prove these theorems. Again, I am not going to go into a detail. I will only outline the structure of the prove and mention the general ingredients involved.

The goal of the prove is; we have a derivation of small step operational semantics. These are the judgements in the upper left corner. We want to obtain a derivation of the big step operational semantics. These are the judgements in the lower right corner. By the empty space in between you might guess that there is something missing. The issue is that the big step semantics is given on proper proof terms and there are no logical variables subject to unification whereas the small step semantics is given using mixed terms and with variables. It is not immediately clear how the proof should proceed.

A possible approach is to introduce *logical relation in between* with the idea that we can show an embedding of SSS to the LR and a projection of LR to BSS. The logical relation is formulated with variables and relates mixed terms for each of the left and right judgements. A caveat here is, that we need to introduce an auxiliary judgement that relates mixed terms with definite clauses. If you recall both BSS and SSS, the decide step requires the annotating clause to be in the program. The implicit assumption here is that the clause is well-formed, and the assumption is maintained in the course of resolution by virtue of subformula property. The auxiliary judgement explicates this assumption. If you are not familiar with logical relations, it is a device that was originally introduced to study strong normalisation in type theory. And that makes a perfect sense here. You can see BSS as a degenerate typing judgement for proper proof terms in empty context. SSS is then evaluation of mixed terms. Proper proof terms are

the normal forms.

For LR, we can show some structural properties, namely; it is stable under substitution over programs, it is preserved under substitution of variables in context, and both signatures and programs can be weakened. Using these properties, we can show transformations that correspond to inference rules of BSS can be propagated through LR. We call this lifting since it allows delaying these up in the derivation tree towards initial sequents hence simulating SSS. It also allows us to state generalisation for the purpose of the second theorem. Finally, using this machinery, we can show the embedding and projection, or the Fundamental theorem and an Escape lemma. These come in pairs corresponding to the mutually defined judgements of SSS and BSS.

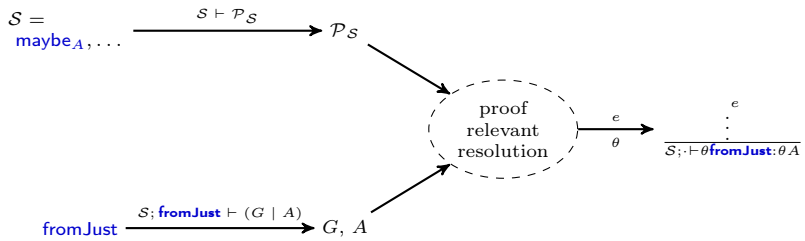
When we leave the technical machinery out of the picture, the theorem follow by composing the Fundamental theorem and the Escape lemma, respectively interleaving it with generalisation.

3 Application to the Examples

Type inference and term synthesis[1]

$\mathsf{fromJust} = \lambda\ (m : \mathsf{maybe}_A\ \mathsf{true}) \to (\lambda\ (w : \mathsf{true} \equiv \mathsf{false}) \to \mathsf{elim}_{\equiv}\ w)$

$\mathsf{elim}_{\mathbf{maybe}_A}\ \mathsf{true}\ m\ (\lambda\ (w : \mathsf{true} \equiv \mathsf{true})\ (x : A) \to x)$



Let us get to the first example. How does proof-relevant resolution help us?

I was talking about internal representation and the need for automation, for type inference and term synthesis. Type checking here is complex exactly due to the need to provide this information that is missing in the surface code. Arguable, we want a compiler of a dependently typed language to be verified, implemented in the language itself, with strong guarantees. We say that the language is intended for writing verified software and then a bug in the compiler may compromise this claim. Yet, it is not case that dependent languages would be implemented using dependent types. My conjecture is that is is due to this complexity of type inference and the involved automation. A possible approach

[1] `https://github.com/frantisekfarka/slepice`

to overcome this issue is a proof-carrying architecture. The automation is off loaded to an external tool that takes care of issues like limiting the search space or search strategy and provides only a proof-term as result. This proof-term witnesses well-formedness for the purpose of the type theory of the internal calculus and it is a nice a simple object to work with. And this is exactly the approach that can benefit from proof-relevant resolution.

In my previous work, I have provided a formalised implementation of such type inference engine:

- specification in Ott
- formalisation in Coq
- proof-relevant resolution in Elpi
- all embeded in OCaml (parser, . . .)

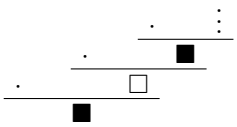
Conclusions:

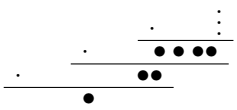
- type inference and term synthesis in first-order dependent type theory as first-order resolution
- soundness; details in the paper (ICLP'18)
- proof-relevant resolution provides certificates of well-formedness

Type class resolution[2]

```
data OddList a = OCons (EvenList a)
instance (Eq a, Eq (OddList b)) ⇒ Eq (EvenList b)
```

```
data EvenList a = Nil | ECons (OddList a)
instance (Eq a, Eq (EvenList b)) ⇒ Eq (OddList b)
```





```
data Bush a = Nil | Cons (Bush (Bush a))
```

```
instance (Eq a, Eq (Bush (Bush b))) ⇒ Eq (Bush b)
```

[2] https://github.com/frantisekfarka/cotcr

We mentioned that there are also semantical considerations regarding the automation. Let's return to the example of type classes. It was the common understanding that type class resolution semantically corresponds to SLD resolution. At the same time it is not entirely true. The composition of instances represented by the dictionary has an operation interpretation, it dictates how the method eq work in the particular use site. This means that the dictionary must have a unique interpretation. More importantly, Haskell is lazy evaluated language and allows infinite data structures, for example lists. An infinite list of zeros is a perfectly valid value in Haskell. Then it is more challenging to understand the semantics of type class resolution as the composition of instances might give rise to cycles. Such an example is the example of lists of odd and even length that are mutually defined. Than the resolution gives rise to infinite cycles. The is existing work that studies such coninductive type class resolution [3]. In my previous work, I carried out semantical analysis and showed coinductive resolution can be accurately modeled by using greatest Herbrand models.

There are more evolved examples, like Bush, where the resolution does not leas to cycles. This is an example that escapes the Horn-clause fragment and where implicative goals, hence hereditary Harrop formulae, are necessary.

4 Where to Next

Future work

Applications

- coinductive proof-search for parallel and distibuted computation
- constrained Horn clauses for resource-aware computation
- automation for e.g. dependently type-based probabilistic programming

Theory of proof search

different classes of sequents for efficient search space

Try to focus on applications relevant to Aleks and to CLIP

Type inference, term synthesis, and type classes

```
-- type inference, term synthesis
data maybe_A : Bool → Set where
  nothing : maybe_A false
  just    : A → maybe_A true

fromJust : maybe_A true → A
fromJust (just x) = x

fromJust = λ (m : maybe_A true) →
           elim_{maybe_A} ?_0 m
             (λ (w : ?_1) → ?_2)
             (λ (w : ?_3) (x : A) → x)
```

```
-- type class resolution
class Eq a where
  eq : a → a → Bool

instance Eq Int where
  eq x y = ...
instance (Eq a, Eq b) ⇒ Eq (a,b)
  where
  eq (x1, x2) (y1, y2) =
    eq x1 y1 ∧ eq x2 y2

test : Eq ( Int , Int ) ⇒ Bool
test = eq (1, 2) (1, 3)

test : Bool
test = eq {?_EqD} (1, 2) (1, 3)
```


Big-step operational semantics

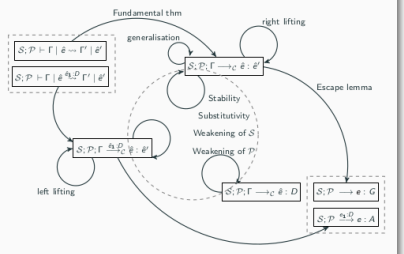
$$D ::= A \mid G \Rightarrow D \mid \forall x : A.D$$
$$G ::= A \mid D \Rightarrow G \mid \forall x : A.G \mid \exists x : A.G$$
$$e ::= \kappa \mid e\ e \mid \lambda \kappa.e \mid \langle M, e \rangle$$

$$\frac{}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa:A} e : A}\ \text{init} \qquad \frac{\mathcal{S};\mathcal{P} \xrightarrow{\kappa:D} e : A \qquad \kappa : D \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow e : A}\ \text{decide}$$

$$\frac{\mathcal{S};\mathcal{P} \longrightarrow e : G[M/x] \qquad \mathcal{S} \vdash M : A}{\mathcal{S};\mathcal{P} \longrightarrow \langle M, e \rangle : \exists x : A.G}\ \exists\text{R} \qquad \frac{\mathcal{S};\mathcal{P},\kappa : D \longrightarrow e : G}{\mathcal{S};\mathcal{P} \longrightarrow \lambda \kappa.e : D \Rightarrow G}\ \Rightarrow\text{R}$$

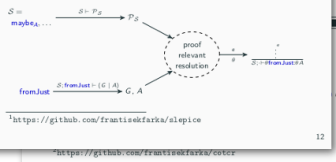
$$\frac{\mathcal{S};\mathcal{P} \longrightarrow e_1 : A_1 \qquad \mathcal{S};\mathcal{P} \xrightarrow{\kappa:A_1 \Rightarrow D} e_2 : A_2}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa:A_1 \Rightarrow D} e_2 : A_2}\ \Rightarrow\text{L} \qquad \frac{\mathcal{S}, c : A;\mathcal{P} \longrightarrow e : G[c/x]}{\mathcal{S};\mathcal{P} \longrightarrow e : \forall x : A.G}\ \forall\text{R}$$

$$\frac{\mathcal{S};\mathcal{P} \xrightarrow{\kappa:D[M/x]} e_2 : A_2 \qquad \mathcal{S} \vdash M : A_1}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa:\forall x:A_1.D} e_2 : A_2}\ \forall\text{L}$$



Type inference and term synthesis[1]

$$\text{fromJust} = \lambda\,(m : \text{maybe}_A\ \text{true}) \rightarrow \text{elim}_{\text{maybe}_A}\ \text{true}\ m\ (\lambda\,(w : \text{true} \equiv \text{false}) \rightarrow \text{elim}_{\equiv}\ w)\ (\lambda\,(w : \text{true} \equiv \text{true})\,(x : A) \rightarrow x)$$



[1] https://github.com/frantisekfarka/slepice

[2] https://github.com/frantisekfarka/coterc

A Appendix

References

References

[1] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[2] Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. Higher-order constrained horn clauses for verification. *PACMPL*, 2(POPL):11:1–11:28, 2018.

[3] Peng Fu and Ekaterina Komendantskaya. Operational semantics of resolution and productivity in horn clause logic. *Formal Asp. Comput.*, 29(3):453–474, 2017.

[4] C.-H. Luke Ong and Dominik Wagner. Hochc: a refutationally-complete and semantically-invariant system of higher-order logic modulo theories. *CoRR*, abs/1902.10396, 2019.

Big-step operational semantics

Example 3.

$$\begin{aligned}\mathcal{P} = \kappa_z &: odd(z),\\ \kappa_e &: \forall x : a.odd\, a \Rightarrow even\,(s\,x)\\ \kappa_o &: \forall x : a.even\, a \Rightarrow odd\,(s\,x)\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\, c \longrightarrow \kappa_x : even\, c}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\, c \longrightarrow \kappa_o\, \kappa_x : odd\,(s\, c)}}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\, c \longrightarrow \kappa_e\,(\kappa_o\, \kappa_x) : even\,(s\,(s\, c))}}{\mathcal{S}, c : a; \mathcal{P} \longrightarrow \lambda \kappa_x.\kappa_e\,(\kappa_o\, \kappa_x) : even\, c \Rightarrow even\,(s\,(s\, c))}}{\mathcal{S}; \mathcal{P} \longrightarrow \lambda \kappa_x.\kappa_e(\kappa_o \kappa_x) : \forall x : a.even\, x \Rightarrow even\,(s\,(s\, x))}$$

Small-step operational semantics

Example 4.

$$\cdot \mid \forall x : a.even\, x \Rightarrow even\,(s\,(s\, x)) \rightsquigarrow \cdot \mid even\, c \Rightarrow even\,(s\,(s\, c)) \rightsquigarrow$$

$$\cdot \mid \lambda\kappa_x.even\,(s\,(s\,c)) \rightsquigarrow \cdot \mid \lambda\kappa_x.even\,(s\,(s\,c))^{\kappa_e:\forall x:a.odd\,x \Rightarrow even\,(s\,x)} \rightsquigarrow$$
$$X : a \mid \lambda\kappa_x.even\,(s\,(s\,c))^{\kappa_e:odd\,X \Rightarrow even\,(s\,X)} \rightsquigarrow$$
$$X : a \mid \lambda\kappa_x.even\,(s\,(s\,c))^{\kappa_e(odd\,X):even\,(s\,X)} \rightsquigarrow$$
$$\cdot \mid \lambda\kappa_x.\kappa_e\,(odd\,(s\,c)) \rightsquigarrow \cdot \mid \lambda\kappa_x.\kappa_e\,(odd\,(s\,c))^{\kappa_o:\forall x:a.even\,x \Rightarrow odd\,(s\,x)} \rightsquigarrow$$
$$Y : a \mid \lambda\kappa_x.\kappa_e(odd\,(s\,c))^{\kappa_o:even\,Y \Rightarrow odd\,(s\,Y)} \rightsquigarrow \cdot \mid \lambda\kappa_x.\kappa_e\,(\kappa_o\,(even\,c)) \rightsquigarrow$$
$$\cdot \mid \lambda\kappa_x.\kappa_e\,(\kappa_o\,(even\,c)^{\kappa_x:even\,c}) \rightsquigarrow \cdot \mid \lambda\kappa_x.\kappa_e\,(\kappa_o\,\kappa_x)$$

Logical relation

$\boxed{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'}$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}':D}_{\mathcal{C}} \hat{e} : A \qquad \mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}' : D}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : A}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G[M/x] \qquad \mathcal{S};\Gamma \vdash M : A}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \langle M, \hat{e}\rangle : \exists x : A.G}$$

$$\frac{\mathcal{S};\mathcal{P},\kappa : D;\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \lambda\kappa.\hat{e} : D \Rightarrow G}$$

$$\frac{\mathcal{S},c : A;\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}[c/x] : G[c/x]}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \forall x : A.G}$$

$$\frac{\mathcal{S} \vdash \mathcal{P} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} A : A}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : \hat{e}_2}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} (\theta\hat{e})\,\hat{e}_1 : \hat{e}\,\hat{e}_2}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : \hat{e}_2}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \langle\theta M, \hat{e}_1\rangle : \langle M, \hat{e}_2\rangle}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : \hat{e}_2}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \lambda\kappa.\hat{e}_1 : \lambda\kappa.\hat{e}_2}$$

$\boxed{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:D}_{\mathcal{C}} \hat{e} : \hat{e}'}$

$$\frac{}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:A}_{\mathcal{C}} \hat{e} : A}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : A_1 \qquad \mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}\,\hat{e}_1:D}_{\mathcal{C}} \hat{e}_2 : A_2}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:A_1 \Rightarrow D}_{\mathcal{C}} \hat{e}_2 : A_2}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:D[M/x]}_{\mathcal{C}} \hat{e}_2 : A_2 \qquad \mathcal{S};\Gamma \vdash M : A_1}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:\forall x:A_1.D}_{\mathcal{C}} \hat{e}_2 : A_2}$$

$\boxed{\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D}$

$$\frac{\mathcal{S} \vdash \mathcal{P} \qquad \kappa : D \in \mathcal{P} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \kappa : D}$$

$$\frac{\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : A \Rightarrow D \qquad \mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e}' : A}{\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e}\ \hat{e}' : D}$$

$$\frac{\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \forall x : A.D \qquad \mathcal{S}; \Gamma \vdash M : A}{\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D[M/x]}$$

Type class resolution - Pair

Example 5. $\mathcal{P}_{Pair} =$

$$\begin{array}{ll} \kappa_1 : & \mathtt{eq}(x),\ \mathtt{eq}(y) \Rightarrow \mathtt{eq}(\mathtt{pair}(x, y)) \\ \kappa_2 : & \Rightarrow \mathtt{eq}(\mathtt{int}) \end{array}$$

$$\dfrac{\dfrac{}{\mathcal{P}_{Pair} \longrightarrow \kappa_2 : \mathtt{eq}(\mathtt{int})}\ \textsc{Lp-m} \qquad \dfrac{}{\mathcal{P}_{Pair} \longrightarrow \kappa_2 : \mathtt{eq}(\mathtt{int})}\ \textsc{Lp-m}}{\mathcal{P}_{Pair} \longrightarrow \kappa_1\kappa_2\kappa_2 : \mathtt{eq}(\mathtt{pair}(\mathtt{int}, \mathtt{int}))}\ \textsc{Lp-m}$$

Type class resolution - Bush

Example 6. $\mathcal{P}_{Bush} =$

$$\begin{array}{ll} \kappa_1 : & \Rightarrow \mathtt{eq}(\mathtt{int}) \\ \kappa_2 : & \mathtt{eq}(x),\ \mathtt{eq}(\mathtt{bush}(\mathtt{bush}(x))) \Rightarrow \mathtt{eq}(\mathtt{bush}(x)) \end{array}$$

$$\dfrac{\dfrac{}{\mathcal{P}_{Bush} \longrightarrow \kappa_1 : \mathtt{eq}(\mathtt{int})} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\mathcal{P}_{Bush}, (\alpha : \mathtt{eq}(x) \Rightarrow \mathtt{eq}(\mathtt{bush}(x))), (\beta : \ \Rightarrow \mathtt{eq}(x)) \longrightarrow \kappa_2\beta(\alpha(\alpha\beta)) : \mathtt{eq}(\mathtt{bush}(x))}}{\mathcal{P}_{Bush}, (\alpha : \_) \longrightarrow \lambda\beta.\kappa_2\beta(\alpha(\alpha\beta)) : \mathtt{eq}(x) \Rightarrow \mathtt{eq}(\mathtt{bush}(x))}\ \textsc{Lam}}{\mathcal{P}_{Bush} \longrightarrow \nu\alpha.\lambda\beta.\kappa_2\beta(\alpha(\alpha\beta)) : \mathtt{eq}(x) \Rightarrow \mathtt{eq}(\mathtt{bush}(x))}\ \textsc{Nu}}{\mathcal{P}_{Bush} \longrightarrow (\nu\alpha.\lambda\beta.\kappa_2\beta(\alpha(\alpha\beta)))\kappa_1 : \mathtt{eq}(\mathtt{bush}(\mathtt{int}))}$$