Proofs as Programs in Classical Logic
Notes

Alexander Pluska

November 16, 2021

1 Plan

Goal:

- Extract program from resolution proofs of $\forall\exists$-sentences over inductive datatypes.

Steps:

- Give extensions of Gödel's System **T** and HAS + EM_1 + SK_1 to arbitrary inductive datatypes (possibly GADTs) and prove (or rather check) properties, i.e. strong normalization, cut-elimination, uniqueness of normal forms.
- Adapt the results of [1] to exhibit realizers in $\mathcal{F} + \Phi$ for cut-free proofs in the extended version of HAS + EM_1 + SK_1 of formulas $\forall x \exists y Pxy$ where P is a predicate in the extended version of **T**.
- Adapt the iterative learning from [1] to extract λ-terms from realizers.
- Give a proof translation from resolution proofs to cut-free sequent calculus proofs (already done?).

Questions:

- When does the translated proof require more than EM_1?
- Using the methods in [1] the extracted term will be in simple λ-calculus. Is is possible to obtain a term in $\mathcal{F}$? (Talk with Federico Aschieri)
- Can the predicates be defined in $\mathcal{F}$ instead of $\mathcal{T}$? (probably yes)
- What happens if we add non-inductive theories?

2 Type theory

As we are interested in a proof-as-programs correspondence for first order logic our underlying type theory has to be of dependant nature, meaning that types can and will depend on terms.

One interesting aspect of choosing a type theory is how to handle equality. There are many different ways to do this but for our purposes the relatively simple model in agda [8] is sufficient (?). Namely the only equality inherent to the type theory is definitional equality, i.e. equality induced by equivalence under β-reductions and η-equivalences whereas propositional equality is itself an inductive type (or rather inductive schema) with a single constructor

$$\mathrm{refl}_x : x =_A x$$

(the type is parametrized by a type A and $x : A$), i.e. for all types A and $x : A$ we have the rule

$$\frac{}{\mathrm{refl}_x : x =_A x}\ \text{=-intro}$$

Many of the expected properties of equality hold. For instance we can derive the term

$$\begin{aligned}&\mathrm{subst} : \Pi_A \Pi_{x,y:A} \Pi_{P:A\to Type} \Pi_{e:x=_A y} \Pi_{p:Px} Py\\ &\mathrm{subst}\ A\ x\ y\ P\ \mathrm{refl}_x\ px = px\end{aligned}$$

by pattern matching, i.e. matching refl the type checker can unify x and y.

However there are some issues with this handling of equality. For example it is impossible to define quotient structures: Let's say we prove prime decomposition for PIDs and want to apply it to some ring. Then the ring will be most likely defined as a setoid-like structure with the equality being a more coarse relation on the set of terms and in particular not equal to the equality above. Our proof-as-programs translation will have to take this into account.

3 Proof translation for saturation-based theorem proving with induction

Most modern theorem provers for first order logic utilize resolution and saturation based techniques. The standard intuitionistic proofs-as-programs correspondence fails as even the very method itself crucially depends on double negation elimination. As a first step we work out a way to convert proofs of $\forall\exists$-proofs based on the calculus presented in [3] into programs. An advantage of the calculus as presented in [3] is that it can (at least in theory) handle general induction for inductive datatype. Many advances have recently been made for incorporating induction in existing theorem provers [5][6] however all of the calculi fail to support induction in the general case, in particular sentences involving existential quantifiers and requiring induction depth greater than 1 are problematic.

The approach in [3] seems promising as, despite being able to handle general induction, there is no explosion in the number of clauses due to a clever use of so-called *constrained clauses*, special symbols $\mathbf{T}_\alpha^{\prec}$ for induction and a special activation mechanism for potential inductive invariants, although how it performs in practice is yet to be seen.

Program translation for classical proofs has been outlined in [4]. Based on this [2] and [**?**] have been developed. Our a approach is a combination of both. At the moment it is quite crude and is mostly based on the exception method from [2]. We aim to later translate this to the CPS approach presented in the same paper to better see how it relates to the normal proofs-as-program correspondence known from intuitionistic logic.

3.1 The Calculus

As explained above our calculus is based on [3]. We take the core rules, which are not mentioned in [3], from [7].

The first five inferences are standard in resolution based theorem proving. We assume some simplification ordering $\succeq^*$.

Resolution.

$$\frac{A \vee C_1 \qquad \neg B \vee C_2}{(C_1 \vee C_2)\theta}$$

where θ is a mgu of A and $\neg B$.

Factoring.

$$\frac{A \vee \neg B \vee C}{(A \vee C)\theta}$$

where theta is a mgu of A and B.

Superposition.

$$\frac{l = r \vee C_1 \qquad L[s] \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta} \quad \frac{l = r \vee C_1 \qquad t[s] = t' \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta} \quad \frac{l = r \vee C_1 \qquad t[s] \neq t' \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\theta}$$

where θ is a mgu of l and s, s is not a variable, $r\theta \not\succeq^* l\theta$, $L[s]$ is not an equality literal, and $t'\theta \succeq^* t[s]\theta$.

Equality Resolution.

$$\frac{s \neq t \vee C}{C\theta}$$

where θ is a mgu of s and t.

Equality Factoring.

$$\frac{s = t \vee s' = t' \vee C}{(s = t \vee t = t' \vee C)\theta}$$

where θ is an mgu of s and s', $t\theta \not\succeq^* s\theta$, and $t'\theta \succeq^* t\theta$.

In addition to regular clauses and inference rules we introduce c-clauses and special rules for them, i.e. (all of this will be made more precise later)

Definition 3.1. *A constrained clause (c-clause) is an expression of the form* $[\![C \mid \mathcal{X}]\!]$*, where*

- C *is a clause,*
- $\mathcal{X}$ *is a conjunction of the form* $\bigwedge_{i=1}^{n} f_i(\mathbf{u_i}) \simeq v_i \wedge \bigwedge_{i=1}^{m} \mathbf{T}_{\alpha_i}^{\prec}(z_i, \boldsymbol{w}_i)$*, where* $n, m \geq 0$ *and* $\forall i \in \{1 \ldots n\}$*,* f_i *are uninterpreted function symbols (possibly nullary).*

The semantics of $[\![C \mid \mathcal{X}]\!]$ is more or less "if all constraints in $\mathcal{X}$ hold, then C holds" and of $\mathbf{T}_{\alpha_i}^{\prec}(z, \mathbf{w})$ "for all $z' \prec z$ $\alpha(z', \mathbf{w})$ holds" where $\prec$ is the standard wfo of an inductive type.

All of the above inference rules extend naturally to constrained clauses, i.e. if $\frac{H_1, \ldots, H_n}{C\theta}$ is an inference rule for regular clauses, then

$$\frac{[\![H_1 \mid \mathcal{X}_1]\!], \ldots, [\![H_1 \mid \mathcal{X}_1]\!]}{[\![C \mid \mathcal{X}_1 \wedge \cdots \wedge \mathcal{X}_n]\!]\theta}$$

is the corresponding inference rule for c-clauses.

Furthermore we have some special rules for c-clauses:

Constraint Factorization.

$$\frac{[\![C \mid l_1 \wedge l_2 \wedge \mathcal{X}]\!]}{[\![C \mid l_1 \wedge \mathcal{X}]\!]\theta}$$

where θ is an mgu of l_1 and l_2.

Abstraction.

$$\frac{[\![C[f(\mathbf{t})]_p \mid \mathcal{X}]\!]}{[\![C[x]_p \mid f(\mathbf{t}) \simeq x \wedge \mathcal{X}]\!]}$$

where f is a uninterpreted function symbol, x is a fresh variable.

Instantiation.

$$\frac{C \mid \mathcal{X} \wedge t \simeq s]\!]}{[\![C \mid \mathcal{X}]\!]\theta}$$

where θ is a mgu of t and s.

In addition there are some rules that generate candidates for an inductive invariant in [3]. We will leave those out for now an choose our variant "magically" by hand. For every formula $\alpha[z, \mathbf{w}]$ we than have axiom

$$\Gamma_{\mathbf{T}_{\alpha}^{\prec}} := [\![z' \not\prec z \vee \alpha(z', \mathbf{w}) \mid \mathbf{T}_{\alpha}^{\prec}(z, \mathbf{w})]\!]$$

and $\prec$ is axiomatized by

$$\Gamma_{\prec}^{s} := [\![x_i \prec s(x_1, \ldots, x_n) \mid \top]\!] \qquad \Gamma_{\prec} := [\![x \not\prec y \vee y \not\prec z \vee x \prec z \mid \top]\!]$$

for every n-ary inductive constructor s, $i \in \{1 \ldots n\}$.

Finally we have additional rules for inductive types. Since we are (for now) not interested in how to generate inductive invariants it is possible to present

just one combination rule for Domain Decomposition and Induction.
Induction.

$$\frac{[\![C_1 \mid \beta_1(t_1) \wedge \mathcal{X}_1]\!] \dots [\![C_n \mid \beta_n(t_n) \wedge \mathcal{X}_n]\!]}{[\![C_1 \vee \dots \vee C_n \vee \alpha(x, \mathbf{w}_1) \mid \mathcal{X}_1 \setminus \{\mathbf{T}_\alpha^{\prec}(t_1, \mathbf{w}_1)\} \wedge \dots \wedge \mathcal{X}_n \setminus \{\mathbf{T}_\alpha^{\prec}(t_n, \mathbf{w}_n)\} \wedge \beta_1(x)]\!]\theta}$$

where θ is the most general idempotent substitution such that $\beta_j\theta \subseteq \beta_1\theta$ for $n \in \{2 \dots n\}$ and a mgu for $\mathbf{w}_1, \dots, \mathbf{w}_n$, the variables occurring in $t_1, \dots, t_n$ do not occur in $C_i, \beta_i, \mathbf{w}_i$ or $\mathcal{X}_i$. $\{t_1 \dots t_n\}$ is *covering*, i.e. the whole type can be constructed using $\{t_1 \dots t_n\}$, for instance for *nat* both $\{z, s(x)\}$ and $\{z, s(z), s(s(x))\}$ are covering and x is some fresh variable.

Note that setting $\alpha = \perp$ and $\mathbf{w_i} = \emptyset$ we obtain the regular Domain Decomposition Rule from [3].

3.2 Type system

Since we want to apply our method to first-order resolution proofs we will require a dependant type system. For this purpose we shall use the practical type system of agda as described in [8].

3.3 Examples and Translations

Example 3.2.

1.$\forall x_0 : Nat(z + x_0 = x_0)$	*Axiom*
2.$\forall x_0 x_1 : Nat(s(x_0 + x_1) = s(x_0) + x_1)$	*Axiom*
3.$\neg\forall x_0 : Nat\exists x_1 : Nat(x_1 + z = x_0)$	*negated claim*
4.$(\forall x_0 : Nat\exists x_1 : Nat, x_1 + z = x_0) \vee (\forall x_0 : Nat, z \neq x_0 + z) \vee$ $\exists x_0 : Nat(\forall x_1 : Nat, x_1 + z \neq s(x_0) \wedge \exists x_1 : Nat, x_1 + z = x_0)$	*Induction axiom (ENNF)*
5.$\exists x_0 : Nat\forall x_1 : Nat(x_1 + z \neq x_0)$	*ENNF transformation 3*
6.$\forall x_1 : Nat(x_1 + z \neq sk_0)$	*skolemization 5*
7.$(\forall x_0 : Nat, sk_1(x_0) + z = x_0) \vee (\forall x_0 : Nat, z \neq x_0 + z) \vee$ $(\forall x_1 : Nat, x_1 + z \neq s(sk_2) \wedge sk_3 + z = sk_2)$	*skolemization 4*
8.$x_1 + z \neq sk_0$	*cnf 6*
9.$sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee sk_2 = sk_3 + z$	*cnf 7*
10.$sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee x_2 + z \neq s(sk_2)$	*cnf 7*
11.$s(x_0 + x_1) = s(x_0) + x_1$	*cnf 2*
12.$z + x_0 = x_0$	*cnf 1*
13.$sk_1(x_0) + z = x_0 \vee z \neq z \vee sk_2 = sk_3 + z$	*superposition 9, 12*
14.$sk_1(x_0) + z = x_0 \vee sk_2 = sk_3 + z$	*trivial inequality removal 13*
15.$sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee s(x_3 + z) \neq s(sk_2)$	*supserposition 10, 11*
16.$sk_0 \neq x_0 \vee sk_2 = sk_3 + z$	*superposition 8, 14*
17.$sk_2 = sk_3 + z$	*equality resolution 16*
18.$sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee s(sk_2) \neq s(sk_2)$	*superposition 15, 17*
19.$sk_1(x_0) + z = x_0 \vee z \neq x_1 + z$	*trivial inequality removal 18*
20.$sk_1(x_0) + z = x_0 \vee z \neq z$	*superposition 12, 19*
21.$sk_1(x_0) + z = x_0$	*trivial inequality removal 20*
22.$sk_0 \neq x_0$	*superoisition 21, 8*
23.$\perp$	*equality resolution 22*

Example 3.3.

1. $\vdash \lambda f.f(\lambda x.id_x) : Cont\ \Pi_{x_0:Nat}(z + x_0 = x_0)$ *Axiom*
2. $\vdash \lambda f.f(\lambda x_0 x_1.id_{s(x_0+x_1)}) : Cont\ \Pi_{x_0:Nat}\Pi_{x_1:Nat}(s(x_0 + x_1) = s(x_0) + x_1)$ *Axiom*
3. $\vdash f : (\Pi_{x_0:Nat}\Sigma_{x_1:Nat}x_1 + z = x_0) \to \bot$ *negated claim*
4. $\vdash \lambda f. : Cont\ (\Pi_{x_0:Nat}\Sigma_{x_1:Nat}x_1 + z = x_0) \vee (\Pi_{x_0:Nat}z \neq x_0 + z) \vee$ $\Sigma_{x_0:Nat}(\Pi_{x_1:Nat}x_1 + z \neq s(x_0) \wedge \Sigma_{x_1:Nat}x_1 + z = x_0)$ *Induction axiom (ENNF)*
5. $\exists x_0 : Nat \forall x_1 : Nat(x_1 + z \neq x_0)$ *ENNF transformation 3*
6. $\forall x_1 : Nat(x_1 + z \neq sk_0)$ *skolemization 5*
7. $(\forall x_0 : Nat, sk_1(x_0) + z = x_0) \vee (\forall x_0 : Nat, z \neq x_0 + z) \vee$ $(\forall x_1 : Nat, x_1 + z \neq s(sk_2) \wedge sk_3 + z = sk_2)$ *skolemization 4*
8. $x_1 + z \neq sk_0$ *cnf 6*
9. $sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee sk_2 = sk_3 + z$ *cnf 7*
10. $sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee x_2 + z \neq s(sk_2)$ *cnf 7*
11. $s(x_0 + x_1) = s(x_0) + x_1$ *cnf 2*
12. $z + x_0 = x_0$ *cnf 1*
13. $sk_1(x_0) + z = x_0 \vee z \neq z \vee sk_2 = sk_3 + z$ *superposition 9, 12*
14. $sk_1(x_0) + z = x_0 \vee sk_2 = sk_3 + z$ *trivial inequality removal 13*
15. $sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee s(x_3 + z) \neq s(sk_2)$ *supserposition 10, 11*
16. $sk_0 \neq x_0 \vee sk_2 = sk_3 + z$ *superposition 8, 14*
17. $sk_2 = sk_3 + z$ *equality resolution 16*
18. $sk_1(x_0) + z = x_0 \vee z \neq x_1 + z \vee s(sk_2) \neq s(sk_2)$ *superposition 15, 17*
19. $sk_1(x_0) + z = x_0 \vee z \neq x_1 + z$ *trivial inequality removal 18*
20. $sk_1(x_0) + z = x_0 \vee z \neq z$ *superposition 12, 19*
21. $sk_1(x_0) + z = x_0$ *trivial inequality removal 20*
22. $sk_0 \neq x_0$ *superoisition 21, 8*
23. $\bot$ *equality resolution 22*

For now we will consider two minimal examples. The first doesn't use induction. There are two constant symbols a, b and a binary proposition variable $p(-,-)$. The only axiom is $\forall x\, p(x,a) \vee p(x,b)$. We seek to prove $\forall x \exists y p(x,y)$. It is immediately clear that unless the axiom has computational content, our extracted program cannot yield a value. On the other hand we will see that we can still create a well-typed program. And if the axiom contains computational content it will indeed yield a result.

Example 3.4.

$$
\begin{array}{llr}
p(x_1,a) \vee p(x_1,b) & \textit{Axiom} & (1)\\
[\![\neg p(x_2,y_2) | x \simeq x_2]\!] & \textit{negated Hypothesis} & (2)\\
[\![p(x_1,b) | x \simeq x_1]\!] & \textit{res.}\ (1)(2)[x_1/x_2, a/y_2] & (3)\\
[\![\Box | x \simeq x_1]\!] & \textit{res.}\ (3)(2)[x_1/x_2, b/y_2] & (4)\\
\Box & \textit{Instantiation}\ (4) & (5)
\end{array}
$$

Now for our computational translation we introduce a special exception terms $e_\alpha(t_1,\ldots,t_n)$ and $\overline{e}_\alpha(t_1,\ldots,t_n)$ for every atom $\alpha[t_1,\ldots,t_n]$ with n term-variables, which intuitively indicate that $\alpha[t_1,\ldots,t_n]$ is not false. They are usually introduced in the manner of the last derivation of definition 3.4 in [2]. If one were to assign types we would have $e_\alpha(t_1.\ldots,t_n) : (\alpha(t_1,\ldots,t_n) \to \perp) \to \perp$ and $\overline{e}_\alpha(t_1.\ldots,t_n) : ((\alpha(t_1,\ldots,t_n) \to \perp) \to \perp) \to \perp$. However as we have the (classical) identification, we can simplify $\overline{e}_\alpha(t_1.\ldots,t_n) : \alpha(t_1,\ldots,t_n) \to \perp$. Having this in mind both $e_\alpha(t_1.\ldots,t_n)\overline{e}_\alpha(t_1.\ldots,t_n) : \perp$ and $\overline{e}_\alpha(t_1.\ldots,t_n)e_\alpha(t_1.\ldots,t_n) : \perp$ make "type-sense" in the exception sense of [2]. We will use this in our program. Let us now translate the above proof. Note that the terms for axioms are given. For clarity we will use braces for function application. For convenience we will write α_i for the i-the generated term.

Example 3.5.

$$
\begin{array}{llr}
\alpha_1 & (p(x_1,a) \to \perp) \to (p(x_1,b) \to \perp) \to \perp & (1)\\
\overline{e}_p(x_2,y_2) & p(x_2,y_2) \to \perp & (2)\\
\lambda t.\mathbf{let}\ v : \neg p(x_1,a) & (p(x_1,b) \to \perp) \to \perp & (3)\\
\quad \mathbf{in}\ \alpha_1(v,t) & & \\
\quad \mathbf{handle}\ v(w) \Rightarrow \overline{e}_p(x_1,a)(w) \quad [= \alpha_2[x_1/x_2, a/y_2](w)] & & \\
\mathbf{let}\ v : \neg p(x_1,b) & \perp & (4)\\
\mathbf{in}\ \alpha_3(v) & & \\
\mathbf{handle}\ v(w) \Rightarrow \overline{e}_p(x_1,b)(w) \quad [= \alpha_2[x_1/x_2, b/y_2](w)] & & \\
\lambda x_1.\alpha_4 & (x : X) \to \perp & (5)
\end{array}
$$

Now to retrieve a value we just need to handle the final exception, i.e.

$$\lambda x_1.\alpha_4\ \mathbf{handle}\ \overline{e}_p(x_2,y_2)(w) \Rightarrow y_2$$

Of course this will not yield an actual value unless α_1 yields.
We will now consider a minimal inductive example.

Consider the inductive type nat with constructors $z : nat$ and $s : nat \to nat$ and some predicate $p(x,y) \subseteq nat^2$. We are going to prove $\forall x \exists y\, p(x,y)$ from $p(z,z)$ and $\forall n,m : nat(p(n,m) \Rightarrow p(s(n), s(s(m))))$.

Example 3.6.

$p(z, z)$	*Axiom*	(6)
$\neg p(n_2, m_2) \vee p(s(n_2), s(s(m_2)))$	*Axiom*	(7)
$[\![\neg p(n_3, m_3) \| x \simeq n_3]\!]$	*negated Hypothesis*	(8)
$[\![\Box \| x \simeq z]\!]$	$res.$ $(1)(3)[z/n_3, z/m_3]$	(9)
$[\![\neg p(n_2, m_2) \| x \simeq s(n_2)]\!]$	$res.(2)(3)[s(n_2)/n_3], s(s(m_2))/m_3]$	(10)
Activate $\alpha := p(a, b)[a, b]$	*Trigger (4)*	(11)
$[\![n_7 \not\prec n_7' \vee p(n_7, m_7) \| \mathbf{T}_\alpha^\prec(n_7', m_7)]\!]$	$\Gamma_{\mathbf{T}_\alpha^\prec}$	(12)
$[\![n_7 \not\prec n_7' \| x \simeq s(n_7) \wedge \mathbf{T}_\alpha^\prec(n_7', m_7)]\!]$	$res.(5)(7)[n_7/n_2, m_7/m_2]$	(13)
$[\![n_7 \prec s(n_7) \| \top]\!]$	$\Gamma_\prec^s$	(14)
$[\![\Box \| x \simeq s(n_7) \wedge \mathbf{T}_\alpha^\prec(s(n_7), m_7)]\!]$	$res.(8)(9)[s(n_7)/n_7']$	(15)
$[\![\Box \| x \simeq n_{10}]\!]$	*Induction* (4)(9)	(16)
$\Box$	*Instantiation* (10)	(17)

3.4 Intuitionistic Example

For simplicity we will use a single sort (0-level type).
Function symbols: $f(-)$
Predicate symbols: $P(-), Q(-,-)$
Axioms: $\forall x P(x)$, $\forall x \exists y x = f(y)$, $\forall x(P(x) \supset Q(f(x), x))$
To Show: $\forall x \exists y Q(x, y)$

Below is a ND-style proof and translation based on the calculus and correspondence presented in [9].

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{Axiom}
\UnaryInfC{$\vdash \forall x \exists y f(y) = x$}
\RightLabel{$\forall E$}
\UnaryInfC{$\vdash \exists y f(y) = x$}
\AxiomC{}
\RightLabel{Axiom}
\UnaryInfC{$\vdash \forall x(P(x) \supset Q(f(x), x))$}
\RightLabel{$\forall E$}
\UnaryInfC{$\vdash P(y) \supset Q(f(y), y)$}
\AxiomC{}
\RightLabel{Axiom}
\UnaryInfC{$\vdash \forall x P(x)$}
\RightLabel{$\forall E$}
\UnaryInfC{$\vdash P(y)$}
\RightLabel{$\supset E$}
\BinaryInfC{$\vdash Q(f(y), y)$}
\AxiomC{}
\RightLabel{u}
\UnaryInfC{$[\vdash f(y) = x]$}
\RightLabel{subst}
\BinaryInfC{$\vdash Q(x, y)$}
\RightLabel{$\supset I^u$}
\UnaryInfC{$f(y) = x \vdash Q(x, y)$}
\RightLabel{$\exists I$}
\UnaryInfC{$f(y) = x \vdash \exists y Q(x, y)$}
\RightLabel{$\exists E$}
\BinaryInfC{$\vdash \exists Q(x, y)$}
\RightLabel{$\forall I$}
\UnaryInfC{$\vdash \forall x \exists y Q(x, y)$}
\end{prooftree}
$$

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{Axiom}
\UnaryInfC{$\vdash \alpha : \Pi_x \Sigma_y f(y) = x$}
\RightLabel{$\forall E$}
\UnaryInfC{$\vdash \alpha(x) : \Sigma_y f(y) = x$}
\AxiomC{}
\RightLabel{Axiom}
\UnaryInfC{$\vdash \beta : \Pi_x(P(x) \supset Q(f(x), x))$}
\RightLabel{$\forall E$}
\UnaryInfC{$\vdash \beta(y) : P(y) \supset Q(f(y), y)$}
\AxiomC{}
\RightLabel{Axiom}
\UnaryInfC{$\vdash \gamma : \Pi_x P(x)$}
\RightLabel{$\forall E$}
\UnaryInfC{$\vdash \gamma(y) : P(y)$}
\RightLabel{$\supset E$}
\BinaryInfC{$\vdash \beta(y)(\gamma(y)) : Q(f(y), y)$}
\AxiomC{}
\RightLabel{u}
\UnaryInfC{$[\vdash \delta : f(y) = x]$}
\RightLabel{subst}
\BinaryInfC{$\vdash \mathrm{subst}(f(y), x, \Pi_t Q(t, y), \delta, \beta(y)(\gamma(y))) : Q(x, y)$}
\RightLabel{$\supset I^u$}
\UnaryInfC{$\delta : f(y) = x \vdash \mathrm{subst}(f(y), x, \Pi_t Q(t, y), \delta, \beta(y)(\gamma(y))) : Q(x, y)$}
\RightLabel{$\exists I$}
\UnaryInfC{$\delta : f(y) = x \vdash \langle y, \mathrm{subst}(f(y), x, \Pi_t Q(t, y), \delta, \beta(y)(\gamma(y)))\rangle : \Sigma_y Q(x, y)$}
\RightLabel{$\exists E$}
\BinaryInfC{$\vdash \mathbf{let}\ \langle y, \delta\rangle = \alpha(x)\ \mathbf{in}\ \langle y, \mathrm{subst}(f(y), x, \Pi_t Q(t, y), \delta, \beta(y)(\gamma(y)))\rangle : \Sigma_y Q(x, y)$}
\RightLabel{$\forall I$}
\UnaryInfC{$\lambda x.\mathbf{let}\ \langle y, \delta\rangle = \alpha(x)\ \mathbf{in}\ \langle y, \mathrm{subst}(f(y), x, \Pi_t Q(t, y), \delta, \beta(y)(\gamma(y)))\rangle : \Pi_x \Sigma_y Q(x, y)$}
\end{prooftree}
$$

We can reduce the last term to $\lambda x.\langle \alpha(x)_1, \mathrm{subst}(f(\alpha(x)_1), x, \Pi_t Q(t, \alpha(x)_1), \alpha(x)_2, \beta(\alpha(x)_1)(\gamma(\alpha(x)_1)))\rangle$. Finally replacing the proof terms by units we obtain $\lambda x.\langle \alpha(x)_1, ()\rangle : \Pi_x \Sigma_y \top$ from which we may further extract $\lambda x.\alpha(x)_1$.

References

[1] ASCHIERI, F. Interactive realizability for second-order heyting arithmetic with em1 and sk1. *Mathematical Structures in Computer Science 24*, 6 (2014).

[2] DE GROOTE, P. A simple calculus of exception handling. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995, pp. 201–215.

[3] ECHENIM, M., AND PELTIER, N. Combining induction and saturation-based theorem proving. *Journal of Automated Reasoning 64*, 2 (mar 2019), 253–294.

[4] GRIFFIN, T. G. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '90* (1990), ACM Press.

[5] HAJDU, M. Automating inductive reasoning with recursive functions, 2021.

[6] HAJDÚ, M., HOZZOVÁ, P., KOVÁCS, L., SCHOISSWOHL, J., AND VORONKOV, A. Induction with generalization in superposition reasoning. In *Intelligent Computer Mathematics* (Cham, 2020), C. Benzmüller and B. Miller, Eds., Springer International Publishing, pp. 123–137.

[7] KOVÁCS, L., AND VORONKOV, A. First-order theorem proving and vampire. In *Computer Aided Verification*. Springer Berlin Heidelberg, 2013, pp. 1–35.

[8] NORELL, U. *Towards a practical programming language based on dependent type theory*, vol. 32. Citeseer, 2007.

[9] PFENNING, F. Constructive logic, 2000.