

Proof Search in Dependently-typed Functional Programming

or: Dependently-typed Functional Programming *is* Proof
Search

James McKinna
based on joint with o.a.
Conor McBride, Stéphane Lengrand

Radboud Universiteit Nijmegen

MLNL'09 talk, 2009-05-26

a topic in *applied* logic
in the Netherlands
(and elsewhere!)

With apologies to Bart

*we have a . . . algebraic structure (pause)
at least formally: the types match up . . .
. . . and then there are the properties to check*

(opening talk, yesterday)

Dependent type theory à la Martin-Löf *et al.*

As we have already seen, type theory via C-H/deB/M-L" is:

- ▶ a basis for doing Bishop-style mathematics
- ▶ a total language of realisers for constructive logic, AC, ...
- ▶ a very rich syntax for well-orderings
- ▶ a functional language for proofs: *evidence* for typing judgments

hypotheses \vdash *prf* : *conclusion*

harmony between introduction and elimination yields WN

- ▶ a total functional language for programming: evidence for *meeting a specification*

declarations, definitions \vdash *prog* : *specification*

Dependent type theory à la Martin-Löf *et al.*

As we have already seen, type theory via C-H/deB/M-L" is:

- ▶ a basis for doing Bishop-style mathematics
- ▶ a total language of realisers for constructive logic, AC, ...
- ▶ a very rich syntax for well-orderings
- ▶ a functional language for proofs: *evidence* for typing judgments

$$\text{hypotheses} \vdash \text{prf} : \text{conclusion}$$

harmony between introduction and elimination yields WN

- ▶ a total functional language for programming: evidence for *meeting a specification*

$$\text{declarations, definitions} \vdash \text{prog} : \text{specification}$$

Dependent type theory à la Martin-Löf *et al.*

As we have already seen, type theory via C-H/deB/M-L" is:

- ▶ a basis for doing Bishop-style mathematics
- ▶ a total language of realisers for constructive logic, AC, ...
- ▶ a very rich syntax for well-orderings
- ▶ a functional language for proofs: *evidence* for typing judgments

$$\text{hypotheses} \vdash \text{prf} : \text{conclusion}$$

harmony between introduction and elimination yields WN

- ▶ a total functional language for programming: evidence for *meeting a specification*

$$\text{declarations, definitions} \vdash \text{prog} : \text{specification}$$

“We know a proof when we see one” (Kreisel)

Fundamental property:

- ▶ typing judgment $\Gamma \vdash M : A$ is decidable
- ▶ by reduction to *type synthesis* $\Gamma \vdash M \Rightarrow B$
- ▶ and type conversion $\Gamma \vdash B \simeq A$

Idea: to compute B , look at structure of M !

Modern version: *bidirectional* typechecking, mixing synthesis and *checking* $\Gamma \vdash M \Leftarrow A$

programming is interactive, type-directed, problem solving

With apologies to Herman: *not* about program extraction!

Why Proof Search? How?

Under types as propositions

- ▶ type inhabitation $\Gamma \vdash A \ggg M$ corresponds to provability
- ▶ existence of a proof of A is... existence of a program
- ▶ so programming is (the end result of) searching for proofs

Clearly impossible/undecidable in general, but easy heuristics:

- ▶ to inhabit a Π -type, try a λ and recurse; otherwise
- ▶ pick an assumption whose (functional) type suitably matches the goal
- ▶ and recursively search for arguments to supply to yield an eventual application term

For the purely functional fragment: this is *complete* for enumeration of inhabitants (Dowek); for simple enough type structure, even terminating (Dyckhoff/Hudelmaier)

Why Proof Search? How?

Under types as propositions

- ▶ type inhabitation $\Gamma \vdash A \ggg M$ corresponds to provability
- ▶ existence of a proof of A is... existence of a program
- ▶ so programming is (the end result of) searching for proofs

Clearly impossible/undecidable in general, but easy heuristics:

- ▶ to inhabit a Π -type, try a λ and recurse; otherwise
- ▶ pick an assumption whose (functional) type suitably matches the goal
- ▶ and recursively search for arguments to supply to yield an eventual application term

For the purely functional fragment: this is *complete* for enumeration of inhabitants (Dowek); for simple enough type structure, even terminating (Dyckhoff/Hudelmaier)

Why Proof Search? How?

Under types as propositions

- ▶ type inhabitation $\Gamma \vdash A \ggg M$ corresponds to provability
- ▶ existence of a proof of A is... existence of a program
- ▶ so programming is (the end result of) searching for proofs

Clearly impossible/undecidable in general, but easy heuristics:

- ▶ to inhabit a Π -type, try a λ and recurse; otherwise
- ▶ pick an assumption whose (functional) type suitably matches the goal
- ▶ and recursively search for arguments to supply to yield an eventual application term

For the purely functional fragment: this is *complete* for enumeration of inhabitants (Dowek); for simple enough type structure, even terminating (Dyckhoff/Hudelmaier)

Why Proof Search? How?

Under types as propositions

- ▶ type inhabitation $\Gamma \vdash A \ggg M$ corresponds to provability
- ▶ existence of a proof of A is... existence of a program
- ▶ so programming is (the end result of) searching for proofs

Clearly impossible/undecidable in general, but easy heuristics:

- ▶ to inhabit a Π -type, try a λ and recurse; otherwise
- ▶ pick an assumption whose (functional) type suitably matches the goal
- ▶ and recursively search for arguments to supply to yield an eventual application term

For the purely functional fragment: this is *complete* for enumeration of inhabitants (Dowek); for simple enough type structure, even terminating (Dyckhoff/Hudelmaier)

Why Proof Search? How?

Under types as propositions

- ▶ type inhabitation $\Gamma \vdash A \ggg M$ corresponds to provability
- ▶ existence of a proof of A is... existence of a program
- ▶ so programming is (the end result of) searching for proofs

Clearly impossible/undecidable in general, but easy heuristics:

- ▶ to inhabit a Π -type, try a λ and recurse; otherwise
- ▶ pick an assumption whose (functional) type suitably matches the goal
- ▶ and recursively search for arguments to supply to yield an eventual application term

For the purely functional fragment: this is *complete* for enumeration of inhabitants (Dowek); for simple enough type structure, even terminating (Dyckhoff/Hudelmaier)

Why Proof Search? How?

Under types as propositions

- ▶ type inhabitation $\Gamma \vdash A \ggg M$ corresponds to provability
- ▶ existence of a proof of A is... existence of a program
- ▶ so programming is (the end result of) searching for proofs

Clearly impossible/undecidable in general, but easy heuristics:

- ▶ to inhabit a Π -type, try a λ and recurse; otherwise
- ▶ pick an assumption whose (functional) type suitably matches the goal
- ▶ and recursively search for arguments to supply to yield an eventual application term

For the purely functional fragment: this is *complete* for enumeration of inhabitants (Dowek); for simple enough type structure, even terminating (Dyckhoff/Hudelmaier)

Type Theory in Sequent Calculus style

For NJ/NK, type inhabitation characterised by corresponding sequent calculus LJ/LK.

For arbitrary PTSs, can develop a term calculus based on Herbelin's $\bar{\lambda}$ (Lengrand, Dyckhoff, JHM: CSL'06 and in preparation) with two judgment forms:

- ▶ $\Gamma \Vdash M : A$ corresponding to $\Gamma \vdash A \ggg M$
- ▶ $\Gamma | A \Vdash I : B$ corresponding to computing argument lists to “match” A

Can see this as a rational reconstruction of `intros/Refine` in LEGO, `intros/apply` in COQ.

cf. Gödel, Curry-Howard, ... already knew this (propositional clausal resolution)

what about data(types)?

Two views of data and control in programming

Classical view:

- ▶ data structures consist of structures containing data
- ▶ general recursion/iteration as universal traversal over such structure, exposing the data by repeated computation
- ▶ termination and even correctness (!), analysed *post hoc*

“Easier” view:

- ▶ data structures consist of data exposing visible (inductive) structure
- ▶ primitive (structural) recursion traverses over such structure; no need to expose substructure by computation
- ▶ termination “for free”; correctness easier if you choose datatypes carefully

Datatype families and programming with DTs

data:

- ▶ usual (strictly positive) algebraic datatypes from FP
- ▶ non-context-free syntax, e.g. well-typed terms
- ▶ inductively-defined relations (incl. partial functions)
- ▶ sos definitions of your favourite operational semantics
- ▶ set(oid) theoretic definitions of algebraic structure, so denotational semantics too

programs:

- ▶ λ -calculus for contextual/higher-order functional plumbing
- ▶ case analysis/primitive recursion for well-founded (inductive) data
- ▶ ... for productive infinite computation on co-inductive data

Programming with real data in this style: EPIGRAM

Inductive families, with declarations

$$\underline{\text{data}} \quad \frac{\vec{t} : \vec{T}}{\mathbf{D} \vec{t} : \star} \quad \underline{\text{where}} \quad \frac{\Delta_1}{\mathbf{c}_1 \Delta_1 : \mathbf{D} \vec{s}_1} \quad \cdots \quad \frac{\Delta_n}{\mathbf{c}_n \Delta_n : \mathbf{D} \vec{s}_n}$$

giving rise to standard Martin-Löf elimination constants **D-elim** and corresponding ι -reductions.

Programs are top-level definitions of typed terms in the underlying type theory, but syntax is “high-level”: typechecker fills in many details.

use the programmer to control search

- ▶ choose left-hand sides: case analysis
- ▶ identify allowable recursive calls: choose recursion schemes
- ▶ choose right-hand sides: solutions to “leaf” problems
- ▶ choose cut formulae (for intermediate computation/let-binding)

Each amounts to supplying (sufficient) evidence to solve the corresponding problem.

Two pieces of a kernel syntax

Programmer programs incrementally, building a tree: either

- ▶ supplying a *solution* to an open leaf problem (type); or
- ▶ decomposing the current open problem according to a term of *eliminator type*

Two possible applications of proof search:

- ▶ “typed Herbrand universe”: solution step builds a term out of “what we know so far”: variables introduced locally; vars and constants in context; allowable recursive calls on the functions being defined (by *lookup*)
- ▶ “type matching modulo propositional equality”: decomposition steps give rise to (first-order) unification problems in the type theory

McBride and McKinna, *The View from the Left*, JFP 2004.

Eliminator types: what are allowable recursive calls?

The standard case analysis principle for an inductive family D ,

$$\begin{aligned} \mathbf{D\text{-case}} : & \forall \vec{t}; x : D \vec{t}. \forall P : \forall \vec{t}; x : D \vec{t}. \star. \\ & \forall m_1 : \forall \Delta_1. P(\mathbf{c}_1 \vec{s}_1). \dots \forall m_n : \forall \Delta_n. P(\mathbf{c}_n \vec{s}_n). P x \end{aligned}$$

is always available, while a general form of recursion principle,

$$\begin{aligned} \mathbf{D\text{-Frec}} : & \forall \vec{t}; x : D \vec{t}. \forall P : \forall \vec{t}; x : D \vec{t}. \star. \\ & (\forall \vec{t}; y : D \vec{t}. \mathbf{F}(P) y \rightarrow P y) \rightarrow P x \end{aligned}$$

may be admissible according to the form of \mathbf{F} . *Always* have:

primitive recursion recursive calls on the immediate subterms

$$\mathbf{F}_0(P)(\mathbf{c}_i \vec{s}_i) \simeq \times_j (P s_{ij})$$

structurally smaller recursive calls on subterms:

$$\mathbf{F}_1(P)(\mathbf{c}_i \vec{s}_i) \simeq \times_j ((\mathbf{F}_1(P)) s_{ij}) \times \times_j (P s_{ij})$$

Containers, algebras, coalgebras

- ▶ The predicate transformer (functional) F describes a *container* of the possible recursive calls $F(P)$ available for a given argument y . For a recursive call to be allowable, it suffices to do lookup in the container structure.
- ▶ What about co-recursion?
- ▶ Modern treatment (McBride, Altenkirch, Ghani, Pattinson, Hancock. . .) of datatypes and co-datatypes via general theory of indexed containers.
- ▶ Allowable recursion/co-recursion given by identifying suitable algebras/co-algebras for such functors.

Open problems

- ▶ How to enumerate the Herbrand universe in the dependently-typed setting?
- ▶ What is a good syntax for constructing algebras/coalgebras?
- ▶ What is a good “sequent calculus” for inductive families?

Conclusions

- ▶ dependent type theory as a nice place to study correct-by-construction programming
- ▶ ... which is type-directed, interactive, proof search
- ▶ machinery for type-checking/type synthesis/conversion testing modulo unknowns
- ▶ unification as a pervasive technology from classical proof search
- ▶ outstanding problem: high-level syntax for sufficient evidence to yield well-typed terms in the underlying theory

Questions?