A Proof-Theoretic Perspective on Dependently-typed Functional Programming

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

James McKinna
based on joint work with a.o.
Conor McBride, Stéphane Lengrand, Roy Dyckhoff
especial debt to Rod Burstall

Unbound, but affiliated, to Radboud Universiteit Nijmegen

DTP talk 2011-09-16 NII/Shonan Village Center

introduction

- some old (JFP 2004; EPIGRAM 1) and more recent (CSL 2006; LMCS 2011) work
- some extensions/generalisations:
  - modest extensions to EPIGRAM 1-style intended to reduce *premature commitment*
  - re-designing type theory in *sequent calculus* style to support postponed decisions
- some (open?) questions; stimulus for discussion

Morrisett: pragmatics as the ‘undiscovered country’ for PL researchers

introduction

- some old (JFP 2004; EPIGRAM 1) and more recent (CSL 2006; LMCS 2011) work
- some extensions/generalisations:
  - modest extensions to EPIGRAM 1-style intended to reduce *premature commitment*
  - re-designing type theory in *sequent calculus* style to support postponed decisions
- some (open?) questions; stimulus for discussion

Morrisett: pragmatics as the 'undiscovered country' for PL researchers

introduction

- some old (JFP 2004; EPIGRAM 1) and more recent (CSL 2006; LMCS 2011) work
- some extensions/generalisations:
  - modest extensions to EPIGRAM 1-style intended to reduce *premature commitment*
  - re-designing type theory in *sequent calculus* style to support postponed decisions
- some (open?) questions; stimulus for discussion

Morrisett: pragmatics as the ‘undiscovered country’ for PL researchers

introduction

- some old (JFP 2004; EPIGRAM 1) and more recent (CSL 2006; LMCS 2011) work
- some extensions/generalisations:
  - modest extensions to EPIGRAM 1-style intended to reduce *premature commitment*
  - re-designing type theory in *sequent calculus* style to support postponed decisions
- some (open?) questions; stimulus for discussion

Morrisett: pragmatics as the ‘undiscovered country’ for PL researchers

introduction

- some old (JFP 2004; EPIGRAM 1) and more recent (CSL 2006; LMCS 2011) work
- some extensions/generalisations:
  - modest extensions to EPIGRAM 1-style intended to reduce *premature commitment*
  - re-designing type theory in *sequent calculus* style to support postponed decisions
- some (open?) questions; stimulus for discussion

Morrisett: pragmatics as the ‘undiscovered country’ for PL researchers

blossom gathered at NII/Shonan

dialogue systems are for the interactive construction of a mathematical object with a dependent type (Bengt)

smart case: you don't want to work with the 'with' rule (Thorsten)

extend recursion beyond the non-structural case (Tim)

parametricity: the interpretation of a type is a relation (Patrik)

functional induction: induction on the graph relation is partial correctness for a function definition; 'Below' is bad (Matthieu)

you want to turn off the termination checker in Agda (Stephanie)

perspectives

(intuitionistic) dependent type theory via C-H/de B/M-L" is:

- . . . lots of interesting things. . . (*deleted*)
- a very rich *syntax* for well-orderings
- a functional language for proofs: *evidence* for typing judgments

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

  harmony between introduction and elimination yields WN
- a total functional language for programming: evidence for *meeting a specification*

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

perspectives

(intuitionistic) dependent type theory via C-H/de B/M-L" is:

- . . . lots of interesting things. . . (*deleted*)
- a very rich *syntax* for well-orderings
- a functional language for proofs: *evidence* for typing judgments

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

  harmony between introduction and elimination yields WN
- a total functional language for programming: evidence for *meeting a specification*

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

perspectives

(intuitionistic) dependent type theory via C-H/de B/M-L" is:

- …lots of interesting things…(*deleted*)
- a very rich *syntax* for well-orderings
- a functional language for proofs: *evidence* for typing judgments

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

  harmony between introduction and elimination yields WN
- a total functional language for programming: evidence for *meeting a specification*

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

“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* $\Gamma \vdash M \Leftarrow A$

Trellys, F^* take an alternative view (or do they?)

“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* $\Gamma \vdash M \Leftarrow A$

Trellys, F^* take an alternative view (or do they?)

programming is interactive,
type-directed,
problem solving

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

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 Π, try λ and recur; otherwise
- pick an assumption whose type *suitably matches* the goal
- recursively search for arguments to supply to yield an application term

For the purely functional fragment:

- Dowek: *complete* for enumeration of inhabitants
- Dyckhoff/Hudelmaier: *terminating*, for simple enough types

So: seek presentations aligned towards proof search

interaction/implementation styles: identify your favourite!

- programming = program construction
- program construction = proof term construction (really?)
- proof term construction = ...
- ...but: we tend to think of this as λ-*term* construction
  - direct-style (term editing): ALF, Agda, ...
  - indirect-style (tactic scripts): NuPRL, Coq, ...
  - semi-indirect (elaboration): Epigram, Agda (?), Equations...
  - 'Joe Programmer' (writes it all, machine maybe typechecks it): Idris (Brady), $F*$ (?), Trellys (?)...

Question is this last what people really want?

More serious: how much does the user write? what does the machine supply?

interaction/implementation styles: identify your favourite!

- programming = program construction
- program construction = proof term construction (really?)
- proof term construction = . . .
- . . . but: we tend to think of this as λ-*term* construction
  - direct-style (term editing): ALF, Agda, . . .
  - indirect-style (tactic scripts): NuPRL, Coq, . . .
  - semi-indirect (elaboration): Epigram, Agda (?), Equations. . .
  - ‘Joe Programmer’ (writes it all, machine maybe typechecks it): Idris (Brady), $F*$ (?), Trellys (?). . .

Question is this last what people really want?

More serious: how much does the user write? what does the machine supply?

interaction/implementation styles: identify your favourite!

- programming = program construction
- program construction = proof term construction (really?)
- proof term construction = . . .
- . . . but: we tend to think of this as λ-*term* construction
  - direct-style (term editing): ALF, Agda, . . .
  - indirect-style (tactic scripts): NuPRL, Coq, . . .
  - semi-indirect (elaboration): Epigram, Agda (?), Equations. . .
  - ‘Joe Programmer’ (writes it all, machine maybe typechecks it): Idris (Brady), $F*$ (?), Trellys (?). . .

Question is this last what people really want?

More serious: how much does the user write? what does the machine supply?

interaction/implementation styles: identify your favourite!

- programming = program construction
- program construction = proof term construction (really?)
- proof term construction = . . .
- . . . but: we tend to think of this as λ-*term* construction
  - direct-style (term editing): ALF, Agda, . . .
  - indirect-style (tactic scripts): NuPRL, Coq, . . .
  - semi-indirect (elaboration): Epigram, Agda (?), Equations. . .
  - ‘Joe Programmer’ (writes it all, machine maybe typechecks it): Idris (Brady), F^* (?), Trellys (?). . .

Question is this last what people really want?

More serious: how much does the user write? what does the machine supply?

programming pragmatics/psychology of programming

obstacles to fully-fledged DTP from opposite directions:

- theoretical: desire for an evolutionary path from Hindley-Milner languages
- cognitive: lack of evolutionary path from Hindley-Milner languages

each an entirely understandable cultural conservatism

HCI/PPoP perspective: Green/Blackwell framework of *cognitive dimensions of notations*

programming pragmatics/psychology of programming

obstacles to fully-fledged DTP from opposite directions:

- theoretical: desire for an evolutionary path from Hindley-Milner languages
- cognitive: lack of evolutionary path from Hindley-Milner languages

each an entirely understandable cultural conservatism

HCI/PPoP perspective: Green/Blackwell framework of *cognitive dimensions of notations*

programming pragmatics/psychology of programming

obstacles to fully-fledged DTP from opposite directions:

- theoretical: desire for an evolutionary path from Hindley-Milner languages
- cognitive: lack of evolutionary path from Hindley-Milner languages

each an entirely understandable cultural conservatism

HCI/PPoP perspective: Green/Blackwell framework of *cognitive dimensions of notations*

programming pragmatics/psychology of programming

obstacles to fully-fledged DTP from opposite directions:

- theoretical: desire for an evolutionary path from Hindley-Milner languages
- cognitive: lack of evolutionary path from Hindley-Milner languages

each an entirely understandable cultural conservatism

HCI/PPoP perspective: Green/Blackwell framework of *cognitive dimensions of notations*

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/case analysis
- 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 1

Inductive families, with declarations

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

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.

EPIGRAM 1: use the programmer to control search

programmer chooses:

- left-hand sides: ‘case analysis’ ($\Leftarrow$)
- recursion schemes: identify allowable recursive calls (also $\Leftarrow$!)
- right-hand sides: solutions to ‘leaf’ problems ($\Rightarrow$)
- intermediate computation ($\|$, not ‘let’ as such)

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

Informal justification by appeal to left-/right-rules in *sequent calculus*; ‘with’ is *cut*)

Problem every program begins with commitment to some **rec**!
Question what is the right syntax for ‘sufficient evidence’?
Question what evidence is (run-time) erasable?

Eliminator types: what are allowable recursive calls?

Standard case analysis for family D always available:

$$\mathsf{D}\textbf{-case} \ : \ \forall_{\vec{t}}\, x : \mathsf{D}\,\vec{t} \to \ \forall P : (\forall_{\vec{t}}\, x : \mathsf{D}\,\vec{t} \to \star) \to \\ \forall m_1 : (\forall \Delta_1 .\, P\,(\mathsf{c}_1\,\vec{s}_1)) \to \ \ldots \ \forall m_n : (\forall \Delta_n .\, P\,(\mathsf{c}_n\,\vec{s}_n)) \to \ P\,x$$

while a general form of recursion principle:

$$\mathsf{D}\textbf{-Frec} \ : \ \forall_{\vec{t}}\, x : \mathsf{D}\,\vec{t} \to \ \forall P : (\forall_{\vec{t}}\, x : \mathsf{D}\,\vec{t} \to \star) \to \\ (\forall_{\vec{t}}\, y : \mathsf{D}\,\vec{t} \to \ \mathbf{F}(P)\,y \ \to \ P\,y) \ \to \ P\,x$$

may be admissible according to the form of **F**. *Always* have:

primitive recursion recursive calls on the immediate subterms

$$\mathbf{F_{pr}}(P)(\mathsf{c}_i\,\vec{s}_i) \simeq \times_j (P\,s_{ij})$$

structurally smaller recursive calls on all subterms (‘Below’):

$$\mathbf{F_{ss}}(P)(\mathsf{c}_i\,\vec{s}_i) \simeq (\times_j ((\mathbf{F_{ss}}(P))\,s_{ij})) \ \times \ (\times_j (P\,s_{ij}))$$

well-founded for provably well-founded relations R

$$\mathbf{F_{wf}}(P)(y) \simeq \forall z \to (R\,z\,y) \to P z$$

Containers, algebras, coalgebras

- The predicate transformer (functional) **F** describes a *container* of possible recursive calls $\mathbf{F}(P)\,y$ available for a given argument y, obtained by *lookup*. (Bad!)
- Allowable recursion/co-recursion given by identifying suitable algebras/co-algebras for such functors (Uustalu, Capretta, Vene); modern treatments of data/codata systematically go via (indexed) containers (Thorsten *et al.*)

Question: is there a compositional account of such functors?
Question: is there a convenient syntax for such things?
Question: what about size-change termination (SCT)?
Extension: require outermost appeal to **Frec**, but delay choice of **F**

Other sources of premature commitment

- functional induction: graph of a higher-order function is a predicate transformer (cf. parametricity), and the proof that the function inhabits the graph is a proof transformer
- elimination with a motive: not necessarily with respect to *equality*, but with resect to an arbitrary *reflexive* R which reflects congruences for appropriate constructors (e.g. permutation on lists)
- equality elimination/substitutivity in a type $P\ x$: not necessarily with respect to *equality*, but with respect to some R for which given P is 'good' (cf. setoid rewriting)
- identify your favourites!
- failure of syntax-directedness leading to smart case?
- computational behaviour of programs defined by Equations; corresponding choices in EPIGRAM 1, Idris

However, deferral imposes different heavy cognitive burden

Other sources of premature commitment

- functional induction: graph of a higher-order function is a predicate transformer (cf. parametricity), and the proof that the function inhabits the graph is a proof transformer
- elimination with a motive: not necessarily with respect to *equality*, but with resect to an arbitrary *reflexive* R which reflects congruences for appropriate constructors (e.g. permutation on lists)
- equality elimination/substitutivity in a type $P\ x$: not necessarily with respect to *equality*, but with respect to some R for which given P is 'good' (cf. setoid rewriting)
- identify your favourites!
- failure of syntax-directedness leading to smart case?
- computational behaviour of programs defined by Equations; corresponding choices in EPIGRAM 1, Idris

However, deferral imposes different heavy cognitive burden

Part II: proof search in type theory

Classical approach to premature commitment in proof search in natural deduction (NJ): use sequent calculus (LJ)

- source of premature commitment: choice of antecedent formula in $\rightarrow$-elim
- solution: left-/right rules (LJ), rather than intro-/elim- rules (NJ)
- a calculus for inhabitation of corresponding NJ formulas-as-types
- unification/meta-variables delay choice of term witnesses to $\forall$-left instances

Lots of literature, esp. now on extensions to dependent types

Almost none on using this for programming (Wadler, 1990s, unpublished)

Part II: proof search in type theory

Classical approach to premature commitment in proof search in natural deduction (NJ): use sequent calculus (LJ)

- source of premature commitment: choice of antecedent formula in $\rightarrow$-elim
- solution: left-/right rules (LJ), rather than intro-/elim- rules (NJ)
- a calculus for inhabitation of corresponding NJ formulas-as-types
- unification/meta-variables delay choice of term witnesses to $\forall$-left instances

Lots of literature, esp. now on extensions to dependent types

Almost none on using this for programming (Wadler, 1990s, unpublished)

Part II: proof search in type theory

Classical approach to premature commitment in proof search in natural deduction (NJ): use sequent calculus (LJ)

- source of premature commitment: choice of antecedent formula in $\rightarrow$-elim
- solution: left-/right rules (LJ), rather than intro-/elim- rules (NJ)
- a calculus for inhabitation of corresponding NJ formulas-as-types
- unification/meta-variables delay choice of term witnesses to $\forall$-left instances

Lots of literature, esp. now on extensions to dependent types

Almost none on using this for programming (Wadler, 1990s, unpublished)

Type Theory in Sequent Calculus style (CSL 2006)

For arbitrary PTSs, can develop a term calculus with two judgment forms:

- $\Gamma \vdash M : A$ corresponding to $\Gamma \vdash A \ggg M$
- $\Gamma ; A \vdash l : B$ corresponding to computing argument lists to "match" A against B

Key idea: LJ is too permissive, so tighten up to remove inessential variation (permutation of rules)

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

Type Theory in Sequent Calculus style (CSL 2006)

For arbitrary PTSs, can develop a term calculus with two judgment forms:

- $\Gamma \vdash M : A$ corresponding to $\Gamma \vdash A \ggg M$
- $\Gamma; A \vdash l : B$ corresponding to computing argument lists to "match" A against B

Key idea: LJ is too permissive, so tighten up to remove inessential variation (permutation of rules)

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

Adding meta-variables (LMCS 2011)

leads to a calculus in which

- Dowek's complete semi-recursive type inhabitation procedure can be recovered, hence higher-order unification
- Dyckhoff/Hudelmaier complete search for propositional sub-languages

Challenge extend analysis to *datatypes*, thereby

- making solid the EPIGRAM 1/sequent calculus informal connection
- modernising, to deal with e.g. bidirectional type checking, ...

Adding meta-variables (LMCS 2011)

leads to a calculus in which

- Dowek's complete semi-recursive type inhabitation procedure can be recovered, hence higher-order unification
- Dyckhoff/Hudelmaier complete search for propositional sub-languages

Challenge extend analysis to *datatypes*, thereby

- making solid the EPIGRAM 1/sequent calculus informal connection
- modernising, to deal with e.g. bidirectional type checking, …

Advantages for the implementor?

Such calculi combine

- explicit substitutions
- spine representations

so hopefully better adapted towards

- abstract machines for evaluation
- ‘internal’ (inferential mode) and ‘external’ (checking mode) categories of abstract syntax in recent presentations of EPIGRAM 2 (Chapman, Alti, McBride . . .)

Metavariables and unification/conversion are baked in from the start, so there is no *separate* ‘program construction’ layer distinct from that of eventually elaborated programs: these are just terms containing no open meta-variables.

Rules, I

$$\boxed{\Gamma \vdash_{\mathrm{PE}} M : A \mid \Sigma}$$

$$\frac{C \longrightarrow^{*}_{\mathrm{Bx}} s \quad (s', s) \in \mathcal{A}}{\Gamma \vdash_{\mathrm{PE}} s' : C \mid []}\ \text{sorted}$$

$$\frac{C \longrightarrow^{*}_{\mathrm{Bx}} s \quad (s_1, s_2, s) \in \mathcal{R} \quad \Gamma \vdash_{\mathrm{PE}} A : s_1 \mid \Sigma_1 \quad \Gamma, x : A \vdash_{\mathrm{PE}} B : s_2 \mid \Sigma_2}{\Gamma \vdash_{\mathrm{PE}} \Pi x^{A}.B : C \mid \Sigma_1, \Sigma_2}$$

$$\frac{(x : A) \in \Gamma \quad \Gamma; A \vdash_{\mathrm{PE}} l : C \mid \Sigma}{\Gamma \vdash_{\mathrm{PE}} x\ l : C \mid \Sigma}\ \text{Select}_x$$

$$\frac{C \longrightarrow^{*}_{\mathrm{Bx}} \Pi x^{A}.B \quad \Gamma, x : A \vdash_{\mathrm{PE}} M : B \mid \Sigma}{\Gamma \vdash_{\mathrm{PE}} \lambda x^{A}.M : C \mid \Sigma}\ \Pi\text{r}$$

Rules, II

$$\boxed{\Gamma; B \vdash_{\mathsf{PE}} l : C \mid \Sigma}$$

$$\frac{\Gamma = x_1 : A_1, \ldots, x_n : A_n}{\Gamma \vdash_{\mathsf{PE}} \alpha(x_1\ [], \ldots, x_n\ []) : C \mid (\Gamma \vdash \alpha : C)}\ \text{Claim}_\alpha$$

$$\frac{\Gamma = x_1 : A_1, \ldots, x_n : A_n}{\Gamma; D \vdash_{\mathsf{PE}} \beta(x_1\ [], \ldots, x_n\ []) : C \mid (\Gamma; D \vdash \beta : C)}\ \text{Claim}_\beta$$

$$\frac{}{\Gamma; D \vdash_{\mathsf{PE}} [] : C \mid D \stackrel{\Gamma}{=} C}\ \text{axiom}$$

$$\frac{D \longrightarrow^*_{\mathsf{Bx}} \Pi x^A.B \quad \Gamma \vdash_{\mathsf{PE}} M : A \mid \Sigma_1 \quad \Gamma; \langle M/x \rangle B \vdash_{\mathsf{PE}} l : C \mid \Sigma_2}{\Gamma; D \vdash_{\mathsf{PE}} M \cdot l : C \mid \Sigma_1, \Sigma_2}\ \Pi\mathsf{l}$$

Rules, III

$$\boxed{\Sigma \Longrightarrow_{\mathsf{PE}} \sigma}$$

$$\frac{\Gamma; B \vdash_{\mathsf{PE}} I : C \mid \Sigma'' \qquad \Sigma, \Sigma'', (\beta \mapsto \mathsf{Dom}(\Gamma).I)(\Sigma') \Longrightarrow_{\mathsf{PE}} \sigma_{\Sigma}, \sigma_{\Sigma''}, \sigma_{\Sigma'}}{\Sigma, (\Gamma; B \vdash \beta : C), \Sigma' \Longrightarrow_{\mathsf{PE}} \sigma_{\Sigma}, (\beta \mapsto \mathsf{Dom}(\Gamma).(\sigma_{\Sigma}, \sigma_{\Sigma''})(I)), \sigma_{\Sigma'}} \;\mathsf{Solve}_{\beta}$$

$$\frac{\Gamma \vdash_{\mathsf{PE}} M : A \mid \Sigma'' \qquad \Sigma, \Sigma'', (\alpha \mapsto \mathsf{Dom}(\Gamma).M)(\Sigma') \Longrightarrow_{\mathsf{PE}} \sigma_{\Sigma}, \sigma_{\Sigma''}, \sigma_{\Sigma'}}{\Sigma, (\Gamma \vdash \alpha : A), \Sigma' \Longrightarrow_{\mathsf{PE}} \sigma_{\Sigma}, (\alpha \mapsto \mathsf{Dom}(\Gamma).(\sigma_{\Sigma}, \sigma_{\Sigma''})(M)), \sigma_{\Sigma'}} \;\mathsf{Solve}_{\alpha}$$

$$\frac{\Sigma \text{ is solved}}{\Sigma \Longrightarrow_{\mathsf{PE}} \emptyset} \;\mathsf{Solved}$$

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 traditional proof search
- many (?) more places during construction when unknowns allow progress without over-committing the programmer
- outstanding problem: high-level syntax for sufficient evidence to yield well-typed terms in the underlying theory
- outstanding disadvantage: we make the programmer supply (nearly) everything
- no treatment yet of **undo**

Questions?