Towards Dependent Types over Programmer-Defined Index Domains

Daniel R. Licata   Robert Harper

Carnegie Mellon University

Dependent Types

- `int`

Dependent Types

- `int`

  `int` (2)

Dependent Types

- $\texttt{int}$

  $\texttt{int}\,(2)$

- $2 : \texttt{int}$

Dependent Types

- $\texttt{int}$

  $\texttt{int}\,(2)$

- $2 : \texttt{int}$

  $2 : \texttt{int}\,(2)$

Dependent Types

- $\texttt{int}$

  $\texttt{int}\,(2)$

- $2 : \texttt{int}$

  $2 : \texttt{int}\,(2)$

- $\texttt{list}\,(\texttt{string})$

Dependent Types

- `int`

  `int`(2)
- 2 : `int`

  2 : `int`(2)
- `list`(`string`)

  `list`(`string`)(10)

Dependent Types

- $\texttt{int}$

  $\texttt{int}\,(2)$

- $2 : \texttt{int}$

  $2 : \texttt{int}\,(2)$

- $\texttt{list}\,(\texttt{string})$

  $\texttt{list}(\texttt{string})(10)$

- $\texttt{cons} : \tau \longrightarrow \texttt{list}\,(\tau) \longrightarrow \texttt{list}\,(\tau)$

Dependent Types

- $\texttt{int}$

  $\texttt{int}\,(2)$

- $2 : \texttt{int}$

  $2 : \texttt{int}\,(2)$

- $\texttt{list}\,(\texttt{string})$

  $\texttt{list}(\texttt{string})(10)$

- $\texttt{cons} : \tau \longrightarrow \texttt{list}\,(\tau) \longrightarrow \texttt{list}\,(\tau)$

  $\texttt{cons} : \Pi\, \texttt{i} : \texttt{int}.\, \tau \longrightarrow \texttt{list}(\tau)(\texttt{i}) \longrightarrow \texttt{list}(\tau)(\texttt{i} + 1)$

Dependent Types are Useful

- Express interesting properties
- Bake reasoning into the code
- Serve as machine-checked documentation
- Enable richer interfaces at module boundaries
- Obviate some dynamic checks

Dependent Types are Tricky

- No phase distinction

Dependent Types are Tricky

- No phase distinction
- $\mathtt{int}\,(\mathtt{fix}\ \lambda\,\mathtt{i}:\mathtt{int}.\ \mathtt{i})$

Dependent Types are Tricky

- No phase distinction
- $\texttt{int}\,(\texttt{fix}\ \lambda\,\texttt{i} : \texttt{int}.\ \texttt{i})$

  $\texttt{int}\,(\texttt{print}\ \text{“}\texttt{hello}\text{”}; 4)$

Dependent Types are Tricky

- No phase distinction
- $\mathtt{int}\,(\mathtt{fix}\ \lambda\,\mathtt{i} : \mathtt{int}.\ \mathtt{i})$

  $\mathtt{int}\,(\mathtt{print}\ \text{“}\mathtt{hello}\text{”}; 4)$
- Type checking depends on term equivalence: undecidable for a sufficiently powerful language

Dependent Types are Tricky

- No phase distinction
- $\texttt{int}\,(\texttt{fix}\ \lambda\,\texttt{i} : \texttt{int}.\ \texttt{i})$

  $\texttt{int}\,(\texttt{print}\ \text{“}\texttt{hello}\text{”}; 4)$
- Type checking depends on term equivalence: undecidable for a sufficiently powerful language

Some languages address these issues

[Augustsson; Ou, Tan, Mandelbaum, Walker]

Dependent Types are Tricky

- No phase distinction
- $\mathtt{int}\,(\mathtt{fix}\;\lambda\,\mathtt{i}:\mathtt{int}.\,\mathtt{i})$

  $\mathtt{int}\,(\mathtt{print}\;\text{“}\mathtt{hello}\text{”};\,4)$
- Type checking depends on term equivalence: undecidable for a sufficiently powerful language

Some languages address these issues
[Augustsson; Ou, Tan, Mandelbaum, Walker]

Is there another way out?

Index Domains Solve these Problems

Xi and Pfenning's realization:

instead of $2 : \texttt{int}\,(2)$,

$2 : \texttt{int}\,(\texttt{s}\ (\texttt{s}\ \texttt{z}))$

Index Domains Solve these Problems

Xi and Pfenning's realization:

$$\text{instead of } 2 : \texttt{int}\,(2),$$
$$2 : \texttt{int}\,(\texttt{s}\;(\texttt{s}\;\texttt{z}))$$

- Types depend on static proxies for run-time data (proxies are drawn from *index domains*)

Index Domains Solve these Problems

Xi and Pfenning's realization:

instead of $2 : \texttt{int}\,(2)$,

$$2 : \texttt{int}\,(\texttt{s}\;(\texttt{s}\;\texttt{z}))$$

- Types depend on static proxies for run-time data (proxies are drawn from *index domains*)
- Indices are pure

Index Domains Solve these Problems

Xi and Pfenning's realization:

instead of $2 : \mathtt{int}\,(2)$,

$$2 : \mathtt{int}\,(\mathtt{s}\ (\mathtt{s}\ \mathtt{z}))$$

- Types depend on static proxies for run-time data (proxies are drawn from *index domains*)
- Indices are pure
- Constraint solver decides relationships between indices

DML Example

$$\texttt{append} : \Pi\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{list}(\tau)(\texttt{i}) \times \texttt{list}(\tau)(\texttt{j}) \longrightarrow \texttt{list}(\tau)(\texttt{plus i j})$$

DML Example

$$\texttt{append} : \Pi\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{list}(\tau)(\texttt{i}) \times \texttt{list}(\tau)(\texttt{j}) \longrightarrow \texttt{list}(\tau)(\texttt{plus i j})$$
$$\texttt{zip} : \Pi\, \texttt{i} :: \texttt{I}.\, \texttt{list}(\tau_1)(\texttt{i}) \times \texttt{list}(\tau_2)(\texttt{i}) \longrightarrow \texttt{list}(\tau_1 \times \tau_2)(\texttt{i})$$

DML Example

$$\mathtt{append} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{list}(\tau)(\mathtt{plus\ i\ j})$$

$$\mathtt{zip} : \Pi\, \mathtt{i} :: \mathtt{I}.\, \mathtt{list}(\tau_1)(\mathtt{i}) \times \mathtt{list}(\tau_2)(\mathtt{i}) \rightarrow \mathtt{list}(\tau_1 \times \tau_2)(\mathtt{i})$$

$$\mathtt{zipApp} :$$
$$\Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{list}(\tau \times \tau)(\mathtt{plus\ i\ j})$$

$$\mathtt{fun\ zipApp}\ (\mathtt{lst1}, \mathtt{lst2}) =$$
$$\mathtt{zip}\ (\mathtt{append}\ (\mathtt{lst1}, \mathtt{lst2}),\ \mathtt{append}\ (\mathtt{lst2}, \mathtt{lst1}))$$

DML Example

$$\texttt{append} : \Pi\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{list}(\tau)(\texttt{i}) \times \texttt{list}(\tau)(\texttt{j}) \rightarrow \texttt{list}(\tau)(\texttt{plus i j})$$

$$\texttt{zip} : \Pi\, \texttt{i} :: \texttt{I}.\, \texttt{list}(\tau_1)(\texttt{i}) \times \texttt{list}(\tau_2)(\texttt{i}) \rightarrow \texttt{list}(\tau_1 \times \tau_2)(\texttt{i})$$

$$\texttt{zipApp} :$$

$$\Pi\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{list}(\tau)(\texttt{i}) \times \texttt{list}(\tau)(\texttt{j}) \rightarrow \texttt{list}(\tau \times \tau)(\texttt{plus i j})$$

$$\texttt{fun zipApp}\ (\texttt{lst1}, \texttt{lst2}) =$$

$$\texttt{zip}\ (\texttt{append}\ (\texttt{lst1}, \texttt{lst2}),\ \texttt{append}\ (\texttt{lst2}, \texttt{lst1}))$$

Why does this type check?

Type Checking in DML

$$\Pi\,\texttt{i},\texttt{j} :: \texttt{I}.\, \texttt{list}(\tau)(\texttt{i}) \times \texttt{list}(\tau)(\texttt{j}) \rightarrow \texttt{list}(\tau \times \tau)(\texttt{plus i j})$$

$$\texttt{fun zipApp}\ (\texttt{lst1}, \texttt{lst2}) =$$
$$\texttt{zip}\ (\texttt{append}\ (\texttt{lst1}, \texttt{lst2}), \texttt{append}\ (\texttt{lst2}, \texttt{lst1}))$$

- Synthesize obvious type $\texttt{list}(\tau)(\texttt{plus j i})$

Type Checking in DML

$$\Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{list}(\tau \times \tau)(\mathtt{plus\ i\ j})$$

$$\mathtt{fun\ zipApp}\ (\mathtt{lst1}, \mathtt{lst2}) =$$
$$\mathtt{zip}\ (\mathtt{append}\ (\mathtt{lst1}, \mathtt{lst2}), \mathtt{append}\ (\mathtt{lst2}, \mathtt{lst1}))$$

- Synthesize obvious type $\mathtt{list}(\tau)(\mathtt{plus\ j\ i})$
- Observe that it must have type $\mathtt{list}(\tau)(\mathtt{plus\ i\ j})$

Type Checking in DML

$$\Pi\,\mathtt{i},\mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \to \mathtt{list}(\tau \times \tau)(\mathtt{plus\ i\ j})$$

$$\mathtt{fun\ zipApp}\ (\mathtt{lst1}, \mathtt{lst2}) =$$
$$\mathtt{zip}\ (\mathtt{append}\ (\mathtt{lst1}, \mathtt{lst2}), \mathtt{append}\ (\mathtt{lst2}, \mathtt{lst1}))$$

- Synthesize obvious type $\mathtt{list}(\tau)(\mathtt{plus\ j\ i})$
- Observe that it must have type $\mathtt{list}(\tau)(\mathtt{plus\ i\ j})$
- Generate constraint $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{plus\ i\ j} = \mathtt{plus\ j\ i}$

Type Checking in DML

$$\Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{list}(\tau \times \tau)(\mathtt{plus\ i\ j})$$

$$\mathtt{fun\ zipApp}\ (\mathtt{lst1}, \mathtt{lst2}) =$$
$$\mathtt{zip}\ (\mathtt{append}\ (\mathtt{lst1}, \mathtt{lst2}), \mathtt{append}\ (\mathtt{lst2}, \mathtt{lst1}))$$

- Synthesize obvious type $\mathtt{list}(\tau)(\mathtt{plus\ j\ i})$
- Observe that it must have type $\mathtt{list}(\tau)(\mathtt{plus\ i\ j})$
- Generate constraint $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{plus\ i\ j} = \mathtt{plus\ j\ i}$
- Constraint solver (presumably) OKs

Type Checking in DML

$$\Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{list}(\tau \times \tau)(\mathtt{plus\ i\ j})$$

$$\mathtt{fun}\ \mathtt{zipApp}\ (\mathtt{lst1}, \mathtt{lst2}) =$$

$$\mathtt{zip}\ (\mathtt{append}\ (\mathtt{lst1}, \mathtt{lst2}), \mathtt{append}\ (\mathtt{lst2}, \mathtt{lst1}))$$

- Synthesize obvious type $\mathtt{list}(\tau)(\mathtt{plus\ j\ i})$
- Observe that it must have type $\mathtt{list}(\tau)(\mathtt{plus\ i\ j})$
- Generate constraint $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{plus\ i\ j} = \mathtt{plus\ j\ i}$
- Constraint solver (presumably) OKs
- Replace equal indices

DML Subset Sorts

Subset sorts require/assert the truth of a proposition:

$$\texttt{nth} : \Pi\, \texttt{i}, \texttt{j} :: \texttt{I} \mid \texttt{i} < \texttt{j}.\, \texttt{list}(\tau)(\texttt{j}) \to \texttt{int}\,(\texttt{i}) \to \tau$$

$$\texttt{filter} : \Pi\, \texttt{i} :: \texttt{I}.\, (\tau \to 2) \to \texttt{list}(\tau)(\texttt{i}) \to \Sigma\, \texttt{j} :: \texttt{I} \mid \texttt{j} < \texttt{i}.\, \texttt{list}(\tau)(\texttt{j})$$

These propositions about indices are checked/assumed by the constraint solver

DML(C) Language Schema

Different implementations use different index domains:

- Xi’s DML has integer indices with linear integer constraints
- Another of Xi’s uses finite sets with a constraint solver based on model checking
- Sarkar’s language has LF terms as indices with a constraint solver based on Twelf

Problems with DML(C)

- *Language designer* chooses the constraint domain
- Particular constraint solver is part of the language specification

Our Goal Language

- *Programmer* specifies the index domains appropriate to her program
- Constraint solver is just library code that helps her prove properties

Our Goal Language

- *Programmer* specifies the index domains appropriate to her program
- Constraint solver is just library code that helps her prove properties

Verifying interesting properties must be practical

Key Design Issues

1. Indices as static data
2. Notions of equality
3. Proofs and propositions
4. Using proofs in run-time terms

Key Design Issues

1. Indices as static data
2. Notions of equality
3. Proofs and propositions
4. Using proofs in run-time terms

Two Levels

- Types (τ) classify terms ($\mathtt{e}$)
- Kinds (κ) classify constructors (σ)

  Constructors of kind $\mathtt{T}$ are types

Basic Expressions

$$
\begin{aligned}
\kappa \quad &::= \quad \mathtt{T} \\
\sigma, \tau \quad &::= \quad \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathtt{unit} \mid \mathtt{void} \\
\mathsf{e} \quad &::= \quad \mathtt{x} \mid \lambda\, \mathtt{x} : \tau.\, \mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \mathtt{fix}\ \mathsf{e} \\
&\quad\ \mid (\mathsf{e}_1, \mathsf{e}_2) \mid \mathtt{fst}\ \mathsf{e} \mid \mathtt{snd}\ \mathsf{e} \\
&\quad\ \mid \mathtt{inl}^{\tau_2}\ \mathsf{e} \mid \mathtt{inr}^{\tau_1}\ \mathsf{e} \\
&\quad\ \mid \mathtt{case}\ \mathsf{e}\ \mathtt{of}\ (\mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathsf{e}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathsf{e}_2) \\
&\quad\ \mid () \mid \mathtt{abort}^{\tau}\ \mathsf{e}
\end{aligned}
$$

Static Semantics

Separate contexts so phase distinction is as clear as in ML:

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, \mathtt{x} : \tau \\ \Delta & ::= & \cdot \mid \Delta, \mathtt{u} :: \kappa \end{array}$$

Basic judgements:

- $\Delta \vdash \kappa\ \mathtt{kind}$
- $\Delta \vdash \sigma :: \kappa$
- $\Delta ; \Gamma \vdash \mathtt{e} : \tau$

Index Domains are Kinds

Indices are *static* proxies for run-time data:

- Indices are constructors
- An index domain is a kind

Index Domains are Kinds

$$
\begin{array}{rcl}
\kappa & ::= & \mathtt{T} \mid \mathtt{I} \\
\\
\sigma, \tau, \iota & ::= & \ldots \\
 & & \mid \mathtt{int}\,(\iota) \mid \mathtt{list}(\tau)(\iota) \\
 & & \mid \mathtt{z} \mid \mathtt{s}\ \iota \\
\\
\mathtt{e} & ::= & \ldots \mid \mathtt{n} \mid \mathtt{e}_1\ +\ \mathtt{e}_2 \mid \mathtt{cons}\ \mathtt{e}_1\ \mathtt{e}_2 \mid \ldots
\end{array}
$$

Kinding of Indices and Types

$$\frac{}{\Delta \vdash \mathtt{z} :: \mathtt{I}} \qquad \frac{\Delta \vdash \iota :: \mathtt{I}}{\Delta \vdash \mathtt{s}\ \iota :: \mathtt{I}}$$

$$\frac{\Delta \vdash \iota :: \mathtt{I}}{\Delta \vdash \mathtt{int}\,(\iota) :: \mathtt{T}} \qquad \frac{\Delta \vdash \tau :: \mathtt{T} \quad \Delta \vdash \iota :: \mathtt{I}}{\Delta \vdash \mathtt{list}(\tau)(\iota) :: \mathtt{T}}$$

Primitives have Index-Aware Types

$$\frac{}{\Delta\,;\Gamma \vdash \mathtt{n} : \mathtt{int}\,(\mathtt{s}^{\mathtt{n}}\ \mathtt{z})} \qquad \frac{\Delta\,;\Gamma \vdash \mathtt{e}_1 : \mathtt{int}\,(\iota_1) \quad \Delta\,;\Gamma \vdash \mathtt{e}_2 : \mathtt{int}\,(\iota_2)}{\Delta\,;\Gamma \vdash \mathtt{e}_1\ +\ \mathtt{e}_2 : \mathtt{int}\,(\mathtt{plus}\ \iota_1\ \iota_2)}$$

$$\frac{\Delta\,;\Gamma \vdash \mathtt{e}_1 : \tau \quad \Delta\,;\Gamma \vdash \mathtt{e}_2 : \mathtt{list}(\tau)(\iota)}{\Delta\,;\Gamma \vdash \mathtt{cons}\ \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{list}(\tau)(\mathtt{s}\ \iota)}$$

Primitives have Index-Aware Types

$$\frac{}{\Delta\,;\Gamma \vdash \mathtt{n} : \mathtt{int}\,(\mathtt{s}^{\mathtt{n}}\ \mathtt{z})} \qquad \frac{\Delta\,;\Gamma \vdash \mathtt{e}_1 : \mathtt{int}\,(\iota_1) \quad \Delta\,;\Gamma \vdash \mathtt{e}_2 : \mathtt{int}\,(\iota_2)}{\Delta\,;\Gamma \vdash \mathtt{e}_1\ +\ \mathtt{e}_2 : \mathtt{int}\,(\mathtt{plus}\ \iota_1\ \iota_2)}$$

$$\frac{\Delta\,;\Gamma \vdash \mathtt{e}_1 : \tau \quad \Delta\,;\Gamma \vdash \mathtt{e}_2 : \mathtt{list}(\tau)(\iota)}{\Delta\,;\Gamma \vdash \mathtt{cons}\ \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{list}(\tau)(\mathtt{s}\ \iota)}$$

What's `plus`*?*

Recursion and Functions

$$\kappa \quad ::= \quad \mathtt{T} \mid \mathtt{I} \mid \kappa_1 \to \kappa_2$$

$$\begin{array}{rcl} \sigma, \tau, \iota & ::= & \ldots \\ & & \mid \mathtt{NATrec_c}\ \iota\ \mathtt{of}\ (\mathtt{z} \Rightarrow \sigma_1 \mid \mathtt{s}\ \mathtt{i}'\ \mathtt{with}\ \mathtt{res} \Rightarrow \sigma_2) \\ & & \mid \mathtt{u} \mid \lambda_{\mathtt{c}}\, \mathtt{u} :: \kappa.\, \sigma \mid \sigma_1\ \sigma_2 \end{array}$$

Kind formation and kinding rules are standard

$\texttt{plus}$ is Definable

$$\texttt{plus} ::= \lambda_{\mathsf{c}}\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{NATrec}_{\mathsf{c}}\ \texttt{i}\ \texttt{of}\ \left(\texttt{z} \Rightarrow \texttt{j} \mid \texttt{s}\ \texttt{i}'\,\texttt{with}\,\texttt{res} \Rightarrow \texttt{s}\ \texttt{res}\right)$$

Dependent Types are Polymorphism

$$\texttt{append} : \Pi\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{list}(\tau)(\texttt{i}) \times \texttt{list}(\tau)(\texttt{j}) \rightarrow \texttt{list}(\tau)(\texttt{plus i j})$$

Some terms require/produce indices

Dependent Types are Polymorphism

$$\mathtt{append} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \to \mathtt{list}(\tau)(\mathtt{plus\ i\ j})$$

Some terms require/produce indices

$$\sigma, \tau, \iota \quad ::= \quad \ldots \mid \Pi\, \mathtt{u} :: \kappa.\, \tau \mid \Sigma\, \mathtt{u} :: \kappa.\, \tau$$

$$\begin{aligned} e \quad ::= \quad & \ldots \mid \Lambda\, \mathtt{u} :: \kappa.\, \mathtt{e} \mid \mathtt{e}[\sigma] \\ & \mid \mathtt{pack}\ (\sigma, \mathtt{e})\ \mathtt{as}\ (\Sigma\, \mathtt{u} :: \kappa.\, \tau) \\ & \mid \mathtt{unpack}\ (\mathtt{u}, \mathtt{x}) = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \end{aligned}$$

Dependent Functions

$$\frac{\Gamma\,;\,\Delta, \mathtt{u} :: \kappa \vdash \mathtt{e} : \tau}{\Delta\,;\,\Gamma \vdash \Lambda\,\mathtt{u} :: \kappa.\,\mathtt{e} : \Pi\,\mathtt{u} :: \kappa.\,\tau}$$

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \Pi\,\mathtt{u} :: \kappa.\,\tau \qquad \Delta \vdash \sigma :: \kappa}{\Delta\,;\,\Gamma \vdash \mathtt{e}[\sigma] : [\sigma/\mathtt{u}]\tau}$$

Dependent Pairs

$$\frac{\Delta \vdash \sigma :: \kappa \quad \Delta ; \Gamma \vdash \mathtt{e} : [\sigma/\mathtt{u}]\tau}{\Delta ; \Gamma \vdash \mathtt{pack}\ (\sigma, \mathtt{e})\ \mathtt{as}\ (\Sigma\, \mathtt{u} :: \kappa.\, \tau) : \Sigma\, \mathtt{u} :: \kappa.\, \tau}$$

$$\frac{\Delta ; \Gamma \vdash \mathtt{e}_1 : \Sigma\, \mathtt{u} :: \kappa_1.\, \tau_1 \quad \Gamma, \mathtt{x} : \tau_1 ; \Delta, \mathtt{u} :: \kappa_1 \vdash \mathtt{e}_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathtt{type}}{\Delta ; \Gamma \vdash \mathtt{unpack}\ (\mathtt{u}, \mathtt{x}) = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \tau_2}$$

Key Design Issues

1. Indices as static data
2. Notions of equality
3. Proofs and propositions
4. Using proofs in run-time terms

Definitional Equality

- Given by some terminating decision procedure (often reduction to normal form)
- Type system always allows the silent replacement of definitional equals; e.g.,

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta \vdash \tau \equiv \tau' :: \mathtt{T}}{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau'}$$

Definitional Equality Judgements

- $\Delta \vdash \kappa_1 \equiv \kappa_2\,\texttt{kind}$
  congruent equivalence relation
- $\Delta \vdash \sigma_1 \equiv \sigma_2 :: \kappa$
  congruent equivalence relation with β, rules for primitive recursion, etc.
- None for terms

`zipApp` with Definitional Equality

Key constraint: $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\ \mathtt{plus\ i\ j} = \mathtt{plus\ j\ i}$

Does $=$ mean $\equiv$?
Is commutativity of addition part of definitional equality?

`zipApp` with Definitional Equality

Key constraint: $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{plus\ i\ j} = \mathtt{plus\ j\ i}$

Does $=$ mean $\equiv$?
Is commutativity of addition part of definitional equality?

Problems:

- What if we forget commutativity of multiplication?
- What about equalities at programmer-defined kinds?

`zipApp` with Definitional Equality

Key constraint: $\forall\, \texttt{i}, \texttt{j} :: \texttt{I}.\ \texttt{plus i j} = \texttt{plus j i}$

Does $=$ mean $\equiv$?
Is commutativity of addition part of definitional equality?

Problems:

- What if we forget commutativity of multiplication?
- What about equalities at programmer-defined kinds?

Programmer must be allowed to add new equalities!

Propositional Equality

Add separate notion of *propositional equality* ($\texttt{EQ}_\kappa(\sigma_1, \sigma_2)$) introduced by explicit proofs

Propositional Equality

Add separate notion of *propositional equality* ($\mathtt{EQ}_\kappa(\sigma_1, \sigma_2)$) introduced by explicit proofs

We might make $\mathtt{PF}(\mathtt{EQ}_\kappa(\sigma_1, \sigma_2))$ a type with inhabitants

- $\mathtt{refl\ s\ z} : \mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{s\ z}, \mathtt{s\ z}))$
- $\mathtt{Eq\_ss} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{i}, \mathtt{j})) \longrightarrow \mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{s\ i}, \mathtt{s\ j}))$

Propositional Equality

Add separate notion of *propositional equality* ($\mathtt{EQ}_\kappa(\sigma_1, \sigma_2)$) introduced by explicit proofs

We might make $\mathtt{PF}(\mathtt{EQ}_\kappa(\sigma_1, \sigma_2))$ a type with inhabitants

- $\mathtt{refl}\ \mathtt{s}\ \mathtt{z} : \mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{s}\ \mathtt{z}, \mathtt{s}\ \mathtt{z}))$
- $\mathtt{Eq\_ss} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{i}, \mathtt{j})) \to \mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{s}\ \mathtt{i}, \mathtt{s}\ \mathtt{j}))$

How can you use a $\mathtt{PF}(\mathtt{EQ}_\mathtt{I}(\mathtt{i}, \mathtt{j}))$*?*

Extensional Equality Elim Rule

Propositional equality induces definitional equality:

$$\frac{\pi : \mathtt{PF}(\mathtt{EQ}_\kappa(\sigma_1, \sigma_2))}{\sigma_1 \equiv \sigma_2 :: \kappa}$$

Extensional Equality Elim Rule

Propositional equality induces definitional equality:

$$\frac{\pi : \mathtt{PF}(\mathtt{EQ}_\kappa(\sigma_1, \sigma_2))}{\sigma_1 \equiv \sigma_2 :: \kappa}$$

- Called the *equality reflection* or *extensionality* rule
- Studied in Martin-Löf's extensional type theory

  [Martin-Löf; Constable et al.; Hofmann]
- Makes type checking undecidable

Intensional Equality Elim Rule

Explicitly use an equality proof to change the type of a particular term:

$$\frac{\Delta\,;\,\Gamma \vdash \texttt{e} : \texttt{int}\,(\iota_1) \quad \Delta\,;\,\Gamma \vdash \pi : \texttt{PF}(\texttt{EQ}_{\texttt{I}}(\iota_1, \iota_2))}{\Delta\,;\,\Gamma \vdash \texttt{e because}\ \pi : \texttt{int}\,(\iota_2)}$$

Intensional Equality Elim Rule

Explicitly use an equality proof to change the type of a particular term:

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta\,;\,\Gamma \vdash \pi : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau'))}{\Delta\,;\,\Gamma \vdash \mathtt{e}\ \mathtt{because}\ \pi : \tau'}$$

Intensional Equality Elim Rule

Explicitly use an equality proof to change the type of a particular term:

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta\,;\,\Gamma \vdash \pi : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau'))}{\Delta\,;\,\Gamma \vdash \mathtt{e}\ \mathtt{because}\ \pi : \tau'}$$

- Studied in intensional Martin-Löf type theory
- Preserves decidability of type checking
- Some “extensional concepts” can be added

[Hofmann; Altenkirch]

Quiz

In DML, the type checker uses a constraint solver to prove indices equal. Is this extensional or intensional?

Quiz

In DML, the type checker uses a constraint solver to prove indices equal. Is this extensional or intensional?

- Extensional: the constraint solver comes up with a proof; this proof induces a definitional equality
- Intensional: definitional equality is given (in part) by the constraint solver

Quiz

In DML, the type checker uses a constraint solver to prove indices equal. Is this extensional or intensional?

- Extensional: the constraint solver comes up with a proof; this proof induces a definitional equality
- Intensional: definitional equality is given (in part) by the constraint solver

In both views, definitional equality is more complicated than simple expansion of definitions

Key Design Issues

1. Indices as static data
2. Notions of equality
3. Proofs and propositions
4. Using proofs in run-time terms

Proofs of Type Equality in Haskell

Recently, proofs of *type* equality in Haskell have been studied with applications to:

- type `dynamic`

  [Baars, Swierstra; Cheney, Hinze; Weirich]

- polytypic programming

  [Cheney, Hinze]

- tagless interpreters and metaprogramming

  [Sheard, Pasalic; Peyton Jones]

Proofs of Type Equality in Haskell

$$\mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \;\; := \;\; \Pi\, \mathtt{f} :: \mathtt{T} \longrightarrow \mathtt{T}.\, (\mathtt{f}\ \tau_1) \longrightarrow (\mathtt{f}\ \tau_2)$$

Proofs of Type Equality in Haskell

$$\mathtt{PF}\big(\mathtt{EQ}_{\mathtt{T}}\big(\tau_1, \tau_2\big)\big) \;:=\; \Pi\, \mathtt{f} :: \mathtt{T} \longrightarrow \mathtt{T}.\, \big(\mathtt{f}\ \tau_1\big) \longrightarrow \big(\mathtt{f}\ \tau_2\big)$$

Reasonable intro rules definable:

$$\mathtt{refl} : \mathtt{PF}\big(\mathtt{EQ}_{\mathtt{T}}\big(\tau, \tau\big)\big) := \Lambda\, \mathtt{f} :: \mathtt{T} \longrightarrow \mathtt{T}.\, \lambda\, \mathtt{x} : \big(\mathtt{f}\ \tau\big).\, \mathtt{x}$$

Proofs of Type Equality in Haskell

$$\mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \;\; := \;\; \Pi\, \mathtt{f} :: \mathtt{T} \longrightarrow \mathtt{T}.\, (\mathtt{f}\ \tau_1) \longrightarrow (\mathtt{f}\ \tau_2)$$

Reasonable intro rules definable:

$$\mathtt{refl} : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau)) := \Lambda\, \mathtt{f} :: \mathtt{T} \longrightarrow \mathtt{T}.\, \lambda\, \mathtt{x} : (\mathtt{f}\ \tau).\, \mathtt{x}$$

$$\mathtt{trans} : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \longrightarrow \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_2, \tau_3)) \longrightarrow \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_3)) :=$$

Proofs of Type Equality in Haskell

$$\mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \;:=\; \Pi\, \mathtt{f} :: \mathtt{T} \rightarrow \mathtt{T}.\, (\mathtt{f}\ \tau_1) \rightarrow (\mathtt{f}\ \tau_2)$$

Reasonable intro rules definable:

$$\mathtt{refl} : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau)) := \Lambda\, \mathtt{f} :: \mathtt{T} \rightarrow \mathtt{T}.\, \lambda\, \mathtt{x} : (\mathtt{f}\ \tau).\, \mathtt{x}$$

$$\begin{aligned}\mathtt{trans} : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \rightarrow \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_2, \tau_3)) \rightarrow \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_3)) :=\\ \lambda\, \mathtt{p}_1 : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)).\, \lambda\, \mathtt{p}_2 : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_2, \tau_3)).\\ \Lambda\, \mathtt{f} :: \mathtt{T} \rightarrow \mathtt{T}.\, \lambda\, \mathtt{x} : (\mathtt{f}\ \tau_1).\, \mathtt{p}_2[\mathtt{f}]\ (\mathtt{p}_1[\mathtt{f}]\ \mathtt{x})\end{aligned}$$

Proofs of Type Equality in Haskell

$$\mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \;:=\; \Pi\, \mathtt{f} :: \mathtt{T} \rightarrow \mathtt{T}.\, (\mathtt{f}\ \tau_1) \rightarrow (\mathtt{f}\ \tau_2)$$

Casting elim definable, too:

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta\,;\,\Gamma \vdash \pi : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau'))}{\Delta\,;\,\Gamma \vdash \mathtt{e}\ \mathtt{because}\ \pi : \tau'}$$

$$\mathtt{e}\ \mathtt{because}\ \mathtt{p} :=$$

Proofs of Type Equality in Haskell

$$\mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau_1, \tau_2)) \; := \; \Pi\, \mathtt{f} :: \mathtt{T} \rightarrow \mathtt{T}.\, (\mathtt{f}\ \tau_1) \rightarrow (\mathtt{f}\ \tau_2)$$

Casting elim definable, too:

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta\,;\,\Gamma \vdash \pi : \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau'))}{\Delta\,;\,\Gamma \vdash \mathtt{e}\ \mathtt{because}\ \pi : \tau'}$$

$$\mathtt{e}\ \mathtt{because}\ \mathtt{p} := \mathtt{p}[\lambda_{\mathtt{c}}\, \mathtt{u} :: \mathtt{T}.\, \mathtt{u}]\ \mathtt{e}$$

Proofs *Terms* are Problematic

- Many applications of λ `x. x` at run-time (unless you do something clever with coercions)

Proofs *Terms* are Problematic

- Many applications of $\lambda\,\mathtt{x}.\,\mathtt{x}$ at run-time (unless you do something clever with coercions)
- Proofs can be non-terminating or have other effects

Proofs *Terms* are Problematic

- Many applications of $\lambda\,\mathtt{x.\,x}$ at run-time (unless you do something clever with coercions)
- Proofs can be non-terminating or have other effects
- Conceptually, the proof's purpose is to convince the type checker of some fact; why should it exist at run-time?

Proofs *Terms* are Problematic

- Many applications of λ `x.x` at run-time (unless you do something clever with coercions)
- Proofs can be non-terminating or have other effects
- Conceptually, the proof's purpose is to convince the type checker of some fact; why should it exist at run-time?

Make the proof terms static

Static Proofs

$$\kappa \quad ::= \quad \ldots \mid \texttt{PROP} \mid \texttt{PF}(\phi)$$

$$\begin{array}{rcl} \sigma, \iota, \phi, \pi & ::= & \ldots \\ & & \mid \texttt{EQ}_{\kappa}(\sigma_1, \sigma_2) \\ & & \mid \texttt{refl}\ \sigma \mid \texttt{sym}\ \pi \mid \texttt{trans}\ \pi_{12}\pi_{23} \\ & & \mid \texttt{Eq\_zz} \mid \texttt{Eq\_ss} \mid \ldots \end{array}$$

Are These Propositions Enough?

- Key `zipApp` constraint:
  $\forall\, \texttt{i}, \texttt{j} :: \texttt{I}.\, \texttt{plus i j} = \texttt{plus j i}$

Are These Propositions Enough?

- Key `zipApp` constraint:
  $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\ \mathtt{EQ}_{\mathtt{I}}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i})$

Are These Propositions Enough?

- Key `zipApp` constraint:
  $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\ \mathtt{EQ}_{\mathtt{I}}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i})$

  What about the $\forall$?

Are These Propositions Enough?

- Key `zipApp` constraint:
  $\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_{\mathtt{I}}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i})$

  What about the $\forall$?

- Binary search constraints $\Rightarrow$ need hypothetical reasoning

Are These Propositions Enough?

- Key `zipApp` constraint:
  $\forall\, \texttt{i}, \texttt{j} :: \texttt{I}.\ \texttt{EQ}_{\texttt{I}}(\texttt{plus i j}, \texttt{plus j i})$

  What about the $\forall$?

- Binary search constraints $\Rightarrow$ need hypothetical reasoning

Need a more expressive logic

Intuitionistic Logic is a Good Option

- Economy of constructs
- Proving is nothing new

We could pick something else, though
(continuation-based classical logic)

Intuitionistic Logic is a Good Option

- Economy of constructs
- Proving is nothing new

We could pick something else, though
(continuation-based classical logic)

How do we set it up?

Propositions

Introduce richer set of propositions:

$$\kappa \quad ::= \quad \ldots \mid \mathtt{PROP} \mid \ldots$$

$$\begin{array}{rcl} \sigma, \iota, \phi, \pi & ::= & \ldots \mid \forall\, \mathbf{u} :: \kappa.\, \phi \mid \exists\, \mathbf{u} :: \kappa.\, \phi \mid \phi_1 \supset \phi_2 \\ & & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \top \mid \bot \end{array}$$

Restrict to FOL in formation rules

Proofs are Constructor-level Programs

$$
\begin{array}{rcl}
\kappa & ::= & \ldots \mid \Pi_{\mathtt{k}}\,\mathtt{u}_1 :: \kappa_1.\,\kappa_2 \mid \Sigma_{\mathtt{k}}\,\mathtt{u}_1 :: \kappa_1.\,\kappa_2 \mid \kappa_1 +_{\mathtt{k}} \kappa_2 \\
 & & \mid \mathtt{UNIT} \mid \mathtt{VOID} \\
 & & \\
\sigma, \pi, \phi, \iota & ::= & \ldots \mid \mathtt{u} \mid \lambda_{\mathtt{c}}\,\mathtt{u} :: \kappa.\,\sigma \mid \sigma_1\ \sigma_2 \\
 & & \mid \mathtt{pack}_{\mathtt{c}}\ (\sigma_1, \sigma_2)\ \mathtt{as}\ \Sigma_{\mathtt{k}}\,\mathtt{u} :: \kappa_1.\,\kappa_2 \mid \mathtt{fst}_{\mathtt{c}}\ \sigma \mid \mathtt{snd}_{\mathtt{c}}\ \sigma \\
 & & \mid \mathtt{inl}_{\mathtt{c}}^{\kappa_2}\ \sigma \mid \mathtt{inr}_{\mathtt{c}}^{\kappa_1}\ \sigma \\
 & & \mid \mathtt{case}_{\mathtt{c}}\ \sigma\ \mathtt{of}\ (\mathtt{inl}\ \mathtt{u}_1 \Rightarrow \sigma_1 \mid \mathtt{inr}\ \mathtt{u}_2 \Rightarrow \sigma_2) \\
 & & \mid \mathtt{unit}_{\mathtt{c}} \mid \mathtt{abort}_{\mathtt{c}}^{\kappa}\ \sigma
\end{array}
$$

Proofs are Constructor-level Programs

$$\overline{\Delta \vdash \mathtt{PF}(\forall\, \mathtt{u} :: \kappa.\, \phi) \equiv \Pi_{\mathtt{k}}\, \mathtt{u} :: \kappa.\, \mathtt{PF}(\phi)\ \mathtt{kind}}$$

$$\overline{\Delta \vdash \mathtt{PF}(\exists\, \mathtt{u} :: \kappa.\, \phi) \equiv \Sigma_{\mathtt{k}}\, \mathtt{u} :: \kappa.\, \mathtt{PF}(\phi)\ \mathtt{kind}}$$

$$\overline{\Delta \vdash \mathtt{PF}(\phi_1 \supset \phi_2) \equiv \Pi_{\mathtt{k}}\, \_ :: \mathtt{PF}(\phi_1).\, \mathtt{PF}(\phi_2)\ \mathtt{kind}}$$

$\mathtt{plus}$ is Commutative

Recall $\mathtt{plus}$ ::=
$\lambda_c\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{NATrec}_c\ \mathtt{i}\ \mathtt{of}\ (\mathtt{z} \Rightarrow \mathtt{j} \mid \mathtt{s}\ \mathtt{i}'\,\mathtt{with}\,\mathtt{res} \Rightarrow \mathtt{s}\ \mathtt{res})$

We can give a $\mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_\mathtt{I}(\mathtt{plus}\ \mathtt{i}\ \mathtt{j}, \mathtt{plus}\ \mathtt{j}\ \mathtt{i}))$

- by induction (primitive recursion) on $\mathtt{i}$
- uses lemmas

$$\mathtt{plus\_rhz} :: \mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_\mathtt{I}(\mathtt{plus}\ \mathtt{i}\ \mathtt{z}, \mathtt{i}))$$

$$\mathtt{plus\_rhs} :: \mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_\mathtt{I}(\mathtt{plus}\ \mathtt{i}\ (\mathtt{s}\ \mathtt{j}), \mathtt{s}\ (\mathtt{plus}\ \mathtt{i}\ \mathtt{j})))$$

Key Design Issues

1. Indices as static data
2. Notions of equality
3. Proofs and propositions
4. Using proofs in run-time terms

Can We Finish Off `zipApp`?

Given the $\mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ_I}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i}))$, can we use `because` rule to finish off `zipApp`?

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta \vdash \pi :: \mathtt{PF}(\mathtt{EQ_T}(\tau, \tau'))}{\Delta\,;\,\Gamma \vdash \mathtt{e\ because}\ \pi : \tau'}$$

Can We Finish Off `zipApp`?

Given the $\mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_{\mathtt{I}}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i}))$, can we use `because` rule to finish off `zipApp`?

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : \tau \qquad \Delta \vdash \pi :: \mathtt{PF}(\mathtt{EQ}_{\mathtt{T}}(\tau, \tau'))}{\Delta\,;\,\Gamma \vdash \mathtt{e}\ \mathtt{because}\ \pi : \tau'}$$

- Need a $\mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_{\mathtt{T}}(\mathtt{list}(\tau)(\mathtt{plus\ i\ j}), \mathtt{list}(\tau)(\mathtt{plus\ j\ i})))$

Can We Finish Off `zipApp`?

Given the $\mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ_I}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i}))$, can we use `because` rule to finish off `zipApp`?

$$\frac{\Delta\, ;\, \Gamma \vdash \mathtt{e} : \tau \qquad \Delta \vdash \pi :: \mathtt{PF}(\mathtt{EQ_T}(\tau, \tau'))}{\Delta\, ;\, \Gamma \vdash \mathtt{e\ because}\ \pi : \tau'}$$

- Need a $\mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ_T}(\mathtt{list}(\tau)(\mathtt{plus\ i\ j}), \mathtt{list}(\tau)(\mathtt{plus\ j\ i})))$

- Seems like we need congruence constants

Congruence Constants are Avoidable

The `because` rule can reach inside a type and substitute:

$$\frac{\Delta\,;\,\Gamma \vdash \mathtt{e} : [\sigma_1/\mathtt{u}]\tau \quad \Delta \vdash \pi :: \mathtt{PF}(\mathtt{EQ}_\kappa(\sigma_1, \sigma_2))}{\Delta\,;\,\Gamma \vdash \mathtt{e}\ \mathtt{because}\ \pi\mathtt{u}\kappa\tau : [\sigma_2/\mathtt{u}]\tau}$$

Finishing Off `zipApp`

$\mathtt{p} :: \mathtt{PF}(\forall\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{EQ}_{\mathtt{I}}(\mathtt{plus\ i\ j}, \mathtt{plus\ j\ i}))$

```
FN i,j :: I =>
fn (lst1, lst2) =>
      zip (append (lst1, lst2),
           (append (lst2, lst1)
             because (sym (p i j))
             as u::I. (list t u))
```

$: \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \mathtt{list}(\tau)(\mathtt{i}) \times \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{list}(\tau \times \tau)(\mathtt{plus\ i\ j})$

Subset Sorts are Proof Quantification

Xi's subset sorts restrict indices to those that satisfy certain propositions:

$$\mathtt{nth} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I} \mid \mathtt{Lt}_{\mathtt{I}}(\mathtt{i}, \mathtt{j}).\, \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{int}\,(\mathtt{i}) \rightarrow \tau$$

Subset Sorts are Proof Quantification

Xi's subset sorts restrict indices to those that satisfy certain propositions:

$$\mathtt{nth} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I} \,|\, \mathtt{Lt}_{\mathtt{I}}(\mathtt{i}, \mathtt{j}).\, \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{int}\,(\mathtt{i}) \rightarrow \tau$$

We handle this by quantification over *proofs*:

$$\mathtt{nth} : \Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\, \Pi\, \mathtt{p} :: \mathtt{PF}(\mathtt{Lt}_{\mathtt{I}}(\mathtt{i}, \mathtt{j})).\, \mathtt{list}(\tau)(\mathtt{j}) \rightarrow \mathtt{int}\,(\mathtt{i}) \rightarrow \tau$$

Subset Sorts are Proof Quantification

$$\mathtt{filter} : \Pi\, \mathtt{i} :: \mathtt{I}.\, (\tau \to 2) \to \mathtt{list}(\tau)(\mathtt{i}) \to$$
$$\Sigma\, \mathtt{j} :: \mathtt{I} \mid \mathtt{Lt}_{\mathtt{I}}(\mathtt{j}, \mathtt{i}).\, \mathtt{list}(\tau)(\mathtt{j})$$

$$\mathtt{filter} : \Pi\, \mathtt{i} :: \mathtt{I}.\, (\tau \to 2) \to \mathtt{list}(\tau)(\mathtt{i}) \to$$
$$\Sigma\, \mathtt{j} :: \mathtt{I}.\, \Sigma\, \mathtt{p} :: \mathtt{PF}(\mathtt{Lt}_{\mathtt{I}}(\mathtt{j}, \mathtt{i})).\, \mathtt{list}(\tau)(\mathtt{j})$$

Run-Time Checks are Proof Quantification

$<:$

$$\Pi\, \mathtt{i}, \mathtt{j} :: \mathtt{I}.\ \mathtt{int}\,(\mathtt{i}) \times \mathtt{int}\,(\mathtt{j}) \rightarrow \Sigma\, \mathtt{p} :: \mathtt{PF}(\mathtt{Lt}_{\mathtt{I}}(\mathtt{i}, \mathtt{j})).\ \mathtt{unit}$$
$$+ \Sigma\, \mathtt{p} :: \mathtt{PF}(\mathtt{Gte}_{\mathtt{I}}(\mathtt{i}, \mathtt{j})).\ \mathtt{unit}$$

Key Design Issues

1. Indices as static data

2. Notions of equality

3. Proofs and propositions

4. Using proofs in run-time terms

Interesting Questions

Phase 1: Redo DML(Int) with explicit proofs

- Operational semantics: type-passing?
- Safety proof and `because`
- Types are *not* parametric in indices

- Fancier recursion
- Programmer-specified *logic*

[Crary, Vanderwaart]

Interesting Questions

Phase 2: Add constructs for declaring new kinds and constructors

- For the kind $\mathtt{I}$, we needed:
  - ▷ constructors $\mathtt{s}$ and $\mathtt{z}$
  - ▷ primitive recursion
  - ▷ inductive equality proof constructors $\mathtt{Eq\_ss}\ldots$
- We also declared new propositions such as $\mathtt{Lt}_{\mathtt{I}}(\iota_2, \iota_2)$

How does this generalize?

Interesting Questions

Phase 3: Reintroduce the constraint solvers as proof search tools

Programmer-Defined Index Domains

Thanks for listening!