Extensible Indexed Types

Daniel R. Licata and Robert Harper
Carnegie Mellon University
Indexed Types

Indexed families of types are useful!

- list(t) where t type
- array(n) where n nat
- proof(p) where p prop

Uniform and non-uniform families.

- array(int) = int_array
  array(t*u) = array(t) * array(u)
Indexed Types

Many applications, more every day.

- Bounds checking (Xi, Pfenning)
- Flat data representations (Chak. & Keller)
- Code certification (Sarkar)
- GADT’s (Xi, Hinze, ...)
- Access control (Harper & Kumar)
- Imperative verification (Morrisett)
Some Characteristics

One or more index domains.

• types (qua data)
• numbers, strings
• propositions
• proofs

Typically built-in (and/or abused).
Some Characteristics

Index expressions.

- constants, such as numbers
- variables
- operations, such as arithmetic
- binders, such as propositions or proofs

Varies from one domain to the next.
Some Characteristics

Constraints = predicates on indices.

- definitional equality
- propositional equality
- inequality, entailment

Constraints influence type checking!

- $i = j$ implies $\text{array}(i) = \text{array}(j)$
Some Characteristics

Constrained types.

- \{ a : \text{nat}(n) \mid 0 \leq n \leq 10 \}
- 0 \leq n \leq 10 \Rightarrow \text{nat}(n) \rightarrow \text{array}(n) \rightarrow \text{nat}
- \text{pf(may-access}(p,r)) \Rightarrow \text{file}(r) \rightarrow \text{string}

Impose restrictions on callers.
Some Characteristics

Constraint satisfaction / verification.

- Fragments of arithmetic (Presburger, omega test, integer programs)
- Decision procedures for other domains.

Fundamentally, demand evidence for the validity of a constraint (a proof).
Extensible Index Domains

Would like to have **programmer-defined** index domains and logics.

- Ad hoc logics for reasoning about ADT’s (a little goes a long way).
- Rich language of modeling types for specifications.

Each abstraction comes with a “theory” of why it works.
Extensible Indexing

signature SETS = sig

  fam ind : Type               % elements of sets
  fam set : Type               % finite sets
  obj void : set.
  obj sing : ind → set.
  objs union, diff : set → set → set.

  fam prop : Type               % propositions
  objs eq, neq : set → set → prop.

  fam pf : prop → Type          % proofs
  ...

end
Extensible Indexing

signature QUEUE = sig

  import Sets : SETS

  typ elt : ind ⇒ type
  typ queue : set ⇒ type
  val empty : queue[void]
  val enq :
    ∀ i:ind ∀ s:set
    elt[i] → queue[s] → queue[union(s,sing(i))]
  val deq :
    ∀ s:set ∀ _:pf(neq(s,void)) queue[s] →
    ∃ i:ind elt[i] × queue[diff(s,sing(i))]

end
Extensible Indexing

Goal: integrate an extensible framework for indexing into an ML-like language.

- Run-time language may have effects.
- Type system permits introduction of new families, expressions, constraints, proofs, logics.

Approach: extend ML with a sufficiently expressive logical framework.
Integrating a Logical Framework

Which logical framework?

• Long-term: Full LF.
• Here: Abstract Binding Trees

Enrich programming language with

• a kind of abt’s (inducing a type of abt’s)
• constructors and expressions over abt’s
Abstract Binding Trees

Generalize abstract syntax trees to account for binding and scope.

- variables, $x$
- operators, $o \cdot (a_1, ..., a_n)$
- abstractors, $x.a$

The **valence** of an abt is the # of binders.

The **arity** of an operator is a sequence of valences.
Abstract Binding Trees

For example, the signature of lambda:

- app : (0,0)
- lam : (1)

Thus \( \lambda x.xx \) is represented by \( \text{lam} \cdot (x.\text{app} \cdot (x,x)) \).

Abt’s are identified up to renaming of bound variables!
Abstract Binding Trees

The judgement $\Psi \vdash a \sim I$ means $a$ is an abt of valence $I$ with free variables $\Psi = x_1, \ldots, x_n$.

- Inductively defined by a set of rules.

Sufficient to handle many interesting examples.

- But eventually we need full LF.
Structural Induction
Modulo $\alpha$

To show $P_{\Psi}(a \sim I)$ whenever $\Psi \vdash a \sim I$, show

- for every $x$ st $\Psi = \Psi_1, x, \Psi_2$, show $P_{\Psi}(x \sim 0)$
- if $P_{\Psi}(a_1 \sim I_1), ..., P_{\Psi}(a_n \sim I_n)$, then $P_{\Psi}(o \cdot (a_1, ..., a_n) \sim 0)$, whenever $o \sim (i_1, ..., i_n)$
- if $P_{\Psi, x}(a \sim I)$ then $P_{\Psi}(x.a \sim I + I)$ for “fresh” $x$

Infinitary simultaneous induction!
Structural Induction

For example, to show $P(a)$ for every lambda term $a$ with vars $x_1, \ldots, x_n$,

- show $P(x_i)$ for every variable $x_i$
- if $P(a_1)$ and $P(a_2)$, then $P(app\cdot(a_1,a_2))$
- for “fresh” $x$, if $P(a)$, then $P(lam\cdot(x.a))$

(Context and valence suppressed for clarity.)
The “freshness” condition can always be met by alpha-conversion.

- cf Pierce/Weirich, Pitts, Pollack/McKinna, ...

Can be avoided using *globally nameless* representations.

- access the context positionally
- (more below)
Integrating ABT’s

Structure of the ambient PL:

- **static part**: constructors classified by kinds
  - includes types qua data and indices
  - restricted to be pure, decidable equiv.

- **dynamic part**: terms classified by types
  - no restrictions on purity
Integrating ABT’s

Type families are indexed by constructors.

- uniform and non-uniform type operators
- indexed families such as array(n::nat)
- constraints and proofs (ensures adequacy)
- “modeling types” for specifications.

Decidedly not “true” dependent types!
Integrating ABT’s’s

Add a kind of abt’s of valence l.

- $K ::= \ldots | \text{abt}[l]$

Treat abt’s as constructors (of this kind).

- $C ::= \ldots | a$

Define a $a :: \text{abt}[l]$ to hold iff $a \sim l$.

- ABT’s provide a general form of static data
Computing With ABT’s

Internalize structural induction at the constructor and expression levels.

- Permits non-uniform families of types.
- Permits non-uniform recursion over such families.

(Also need propositional equality for GADT-like examples. See paper.)
Computing With ABT’s

Example: the size of a lambda term.
\[
\lambda u::\text{abt}[0].\text{abtrec}
\{
\text{var} \Rightarrow 1
\|
\text{ops} \Rightarrow \{ \text{lam} \Rightarrow \lambda m.m+1, \\
\text{app} \Rightarrow \lambda (m,n).m+n+1 \}
\|
\text{abs} \Rightarrow \lambda m.m \}
\] (u)

Deceptively simple!
Computing With ABT’s

Example: id :: abt[0] → abt[0].
λx.abtrec
 { var ⇒ ... the variable ... 
  | abs ⇒ λa. ...abstract free var of a...
  | ops ⇒ { lam ⇒ λa. ... lam(a) ..., 
          app ⇒ λ(a_1,a_2). ... app(a_1,a_2) ... 
     }
 } (x)
Computing With ABT’s

Several issues arise:

- must consider variable valences
- must “compute” with abt’s
- what to do about free variables?

The first is easily handled, but variables create some complications.
Computing With ABT’s

How do we compute ABT’s?

- Create \( o \cdot (a_1, ..., a_n) \) from \( a_i : \text{abt} \).
- Create \( x.a \) from ??? and \( a : \text{abt} \).

Central issue: handling variables and scope.

- Ensure respect for \( \alpha \)-conversion.
- Avoid bureaucracy of names.
Managing Variables

**Nominal** approach (tried and abandoned):

- make names “first-class values”
- explicitly manage binding
- apartness conditions permeate

We use *contextual modal type theory*.

- cf Sarkar, Nanevski/Pientka
Managing Variables

Generalize kind of abt’s to abt[I][L]

• **valence** I (as before)

• **arity** L = context of free variables

Kind abt[0][x:0 * y:0] represents ground abt’s with free variables (parameters) x and y.

• eg, app·(x,y) :: abt[0][x:0 * y:0]
Managing Variables

Formally, arities are (chosen) products of (computed and fixed) valences.

- (Some technical complications arise here.)

Free variables are accessed by projection from the context.

- $\pi_1(\pi_2(...(\pi_2(it))...))$
- globally nameless, locally nameful form!
Managing Variables

General instantiation of parameters:

- if \( P :: abt[l][L] \) and \( L' \vdash S :: L \), then
  \( P \cdot S :: abt[l][L'] \)

Example:

- \( u :: abt[0][x:0] \vdash \text{lam}\cdot(y. u\cdot(y)) :: abt[0][] \)
Copying Identity

\[ \text{id} : \forall w::\text{ctx} \ \forall i::\text{val} \ \text{abt}[i][w] \rightarrow \text{abt}[i][w] = \]

\[ \lambda w. \lambda i. \lambda u. \text{abtrec} \]

\[ \begin{cases} \text{var}(x) \Rightarrow x & \text{return the parameter itself} \\ \text{abs} \Rightarrow \lambda a. (x. a \cdot (x, \text{it})) & \text{rebind after copy} \\ \text{ops} \Rightarrow \{ \text{lam} \Rightarrow \lambda a. \text{lam} \cdot (a \cdot (\text{it})), \\ \text{app} \Rightarrow \\ \quad \lambda (a_1, a_2). \text{app} \cdot (a_1 \cdot (\text{it}), a_2 \cdot (\text{it})) \} \} \\ (u) \end{cases} \]
Copying Identity

What’s really happening with parameters:

- type check $\pi_1(\pi_2(\text{it}))$ relative to the context $w * 0 * w'$, for arbitrary $w, w' :: \text{ctx}$

- i.e., for each variable in the context

The globally name-free form avoids freshness conditions.

- in examples we “label” the variable
Further Examples

In the paper we present examples such as

• substitution and normalization
• Hinze’s tries, with “let”’s over types
• GADT of terms of a specified type

No further machinery required, except propositional equality for GADT’s.
Summary

A first step towards an integration of LF with ML to support extensible indexing.

- parameterization by a signature
- structural induction modulo $\alpha$
- handling of free names during recursion

Please see the paper for many more details.