Dependent Types in Practical Programming

Hongwei Xi
hongwei@cse.ogi.edu

Frank Pfenning
fp@cs.cmu.edu

Presented by: Mike Furr
furr@cs.umd.edu
What are Dependent Types

- Extension of traditional types
- Allow the type to depend on an expression’s value
- Check more properties of existing programs
- Contrast work: type a larger class of programs
Authors introduce a new ML-style language called DML(C) which uses dependent types. Support for

- higher-order functions
- general recursion
- let-polymorphism
- mutable refs
- exceptions
Desirable features:

- Can type-check vanilla ML-programs
- Allow *incremental* annotations to add dependent types
- Small number of annotations (only on function boundaries)
- Annotations can be fully trusted since they are checked
Motivation

- Consider the ML append function on lists:

  append: ’a list -> ’a list -> ’a list

- With dependent types, reason about list lengths

  append: ’a list(m) -> ’a list(n) -> ’a list(m+n)

  m,n are index objects
Decidability

- Dependent types fall in the gap between typing and program verification.
- Unfortunately, this makes things undecidable in the general case.
Decidability

- Dependent types fall in the gap between typing and program verification.
- Unfortunately, this makes things undecidable in the general case.

Solution:
Use a Restricted Form of Dependent Types
Constraint Domains

- Traditionally, index objects are language expressions
- Instead, parameterize over a domain of constraints
- Examples include:
  - Linear inequalities over integers
  - Boolean constraints
  - Finite Sets
Index Sorts

- Notice the Constraint Domain (CD) can be typed
- To differentiate, call types of the CD \textit{index sorts}

\[
\begin{align*}
\text{index sorts} & \quad \gamma ::= b | 1 | \gamma_1 \ast \gamma_2 | \{a : \gamma | P\} \\
\text{index propositions} & \quad P ::= \top | \bot | p(i) | P_1 \land P_2 | P_1 \lor P_2
\end{align*}
\]
Example Typing Rule

\[ \phi; \Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2 \]

\[ \phi; \Gamma \vdash e : \tau_2 \]

\( \phi \) - index context
\( \Gamma \) - type context
Existential Dependent Types

What happens when we can’t check list length?

\[
\text{filter: ('a -> bool) 'a list(n) -> 'a list(?)}
\]

Use an existential type: \( \exists m \leq n \) such that the length of the returned list is \( m \).

Allows interfacing dependently-typed code with vanilla ML code.
Existential Type Rules

\[
\phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2
\]

\[
\phi; \Gamma \vdash \text{let}\langle a|x\rangle = e_1\text{in}e_2\text{end} : \tau_2
\]
Constructing DML(C)

\[ \text{ML}_0 \leftarrow \text{ML}_{0}^{\Pi} \leftarrow \text{ML}_{0}^{\Pi,\Sigma} \leftarrow \text{DML}(C) \]

- \text{ML}_0:
  - Explicitly typed
  - Overly verbose
  - Type checking is reduced to constraint satisfaction in C
- \text{ML}^{\Pi}_0 - Add universal dependent types
- \text{ML}^{\Pi,\Sigma}_0 - Add Existential types
List type:

nil $\langle\|\ 'a \text{ list}(0)$

cons $\langle\| \{n:\text{nat}\} 'a \times 'a \text{ list}(n) -> 'a \text{ list}(n+1)$

fun('a)
  | append(nil,ys) = ys
  | append(cons(x,xs), ys) = cons(x, append(xs,ys))

where append $\langle\| \{m:\text{nat}\}{n:\text{nat}\} 'a \text{ list}(m) + 'a \text{ list}(n) -> 'a \text{ list}(m+n)$
Recall Red-Black Trees:

- Every node is either red or black
- All leaves are black
- For every node, the black height of its children is equal
- Both children of any red node are black
Red-Black Trees

type 'a entry = int * 'a

datatype 'a dict =
  | Empty (* considered black *)
  | Black of 'a entry * 'a dict * 'a dict
  | Red of 'a entry * 'a dict * 'a dict

typeref 'a dict of bool * nat with
  | Empty <| 'a dict(true,0)
  | Black <| {cl:bool}{cr:bool}{bh:nat}
    'a entry * 'a dict(cl,bh) * 'a dict(cr,bh)
    -> 'a dict(true,bh+1)
  | Red <| {bh:nat}
    'a entry * 'a dict(true,bh) * 'a dict(true,bh)
    -> 'a dict(false,bh)
exception zipException

fun ('a,'b)
    | zip(nil,nil) = nil
    | zip(cons(x,xs),cons(y,ys)) = cons((x,y),zip(xs,ys))
    | zip(_,_) = raise zipException
Questions?