A Dependent Language with Type-Safe Extraction

Work in Progress

Greg Brown
greg.brown01@ed.ac.uk

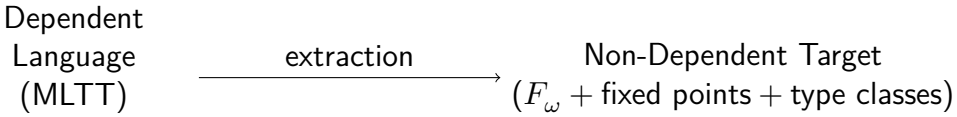
University of Edinburgh

2026-02-11

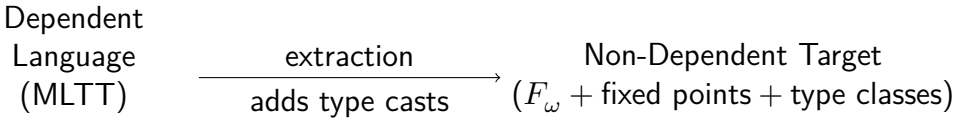
Outline

Dependent
Language
(MLTT)

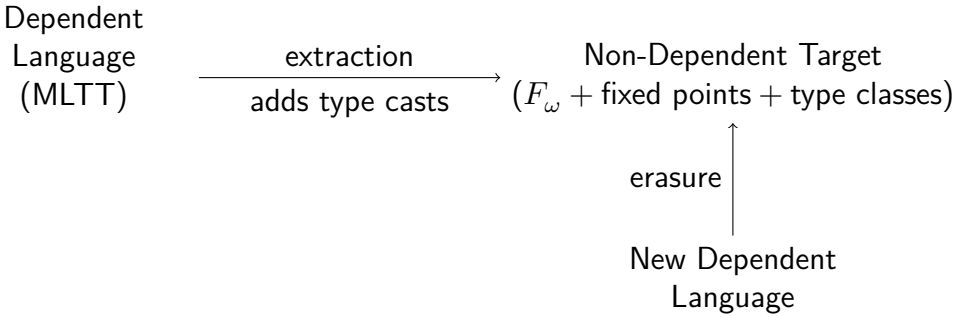
Outline


Dependent
Language
(MLTT)
extraction
Non-Dependent Target
(F_ω + fixed points + type classes)

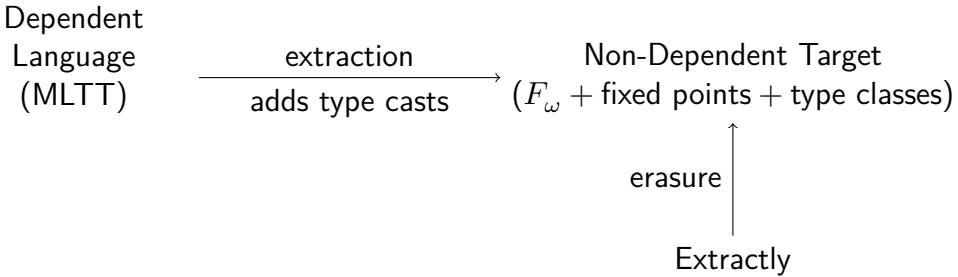
Outline


Dependent
Language
(MLTT)
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)

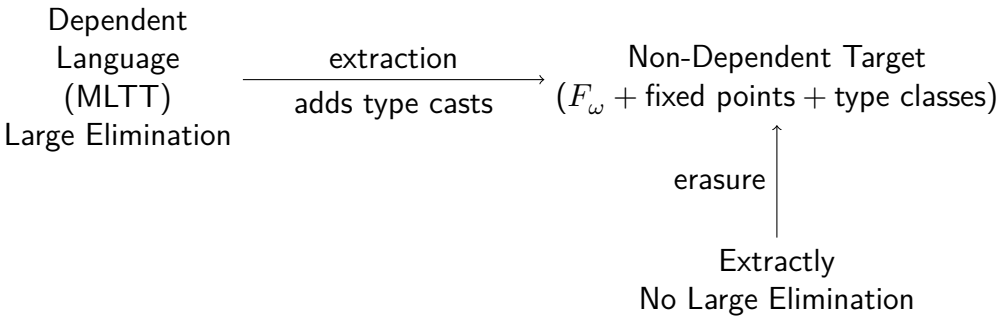
Outline


Dependent
Language
(MLTT)
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)
erasure
New Dependent
Language

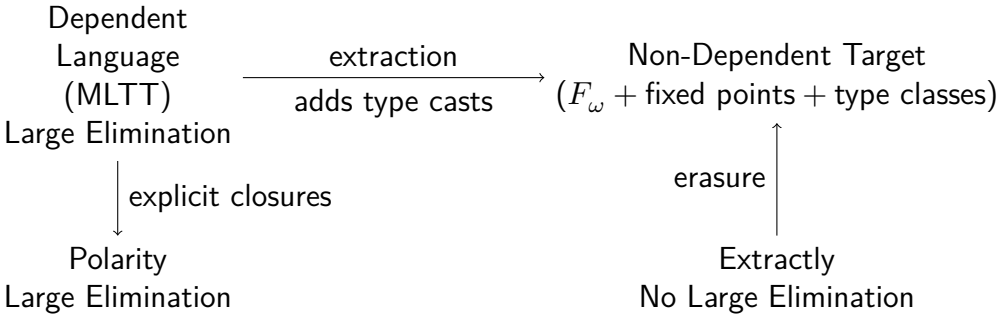
Outline


Dependent
Language
(MLTT)
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)
erasure
Extractly

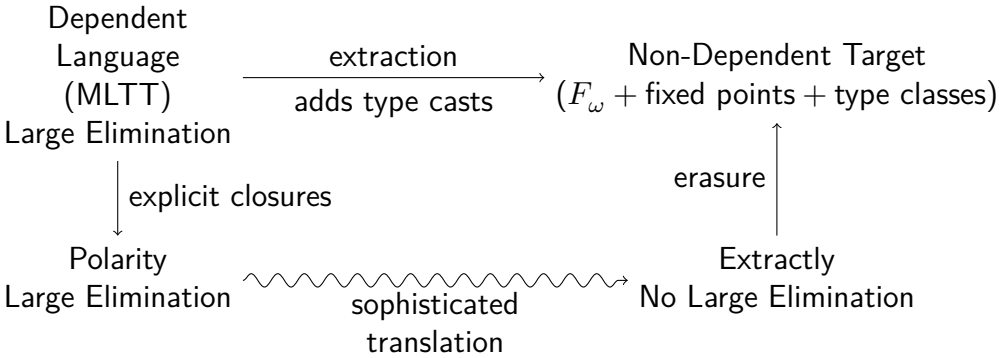
Outline


Dependent
Language
(MLTT)
Large Elimination
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)
erasure
Extractly
No Large Elimination

Outline


Dependent
Language
(MLTT)
Large Elimination
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)
explicit closures
erasure
Polarity
Large Elimination
Extractly
No Large Elimination

Outline


Dependent
Language
(MLTT)
Large Elimination
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)
explicit closures
erasure
Polarity
Large Elimination
sophisticated
translation
Extractly
No Large Elimination

Structure

1. Polarity Primer
2. Three Causes of Type Casts
   - Pattern Matching into Types
   - Pattern Matching into Large Codata
   - Data Types with Large Indices
3. Conclusion

Current Completed Work

Complete

- Type checking

To Do

- Erasure
- Formal definition
- Translation from CIC/MLTT

Try Extractly Yourself

Polarity [Binder et al. 2024]

Data Types

```
data Nat: Set { Z; S(k : Nat); }
def Nat.add(k: Nat): Nat {
    Z => k; S(n) => n.add(k);
}
```

Codata Types

```
codata Stream: Set { .head: a; .tail: Stream; }
codef CountFrom(k : Nat): Stream {
    head => k; tail => CountFrom(S(k));
}
```

Elimination into Types

Needs Type Casts

```
def Nat.vec(ty: Set): Set {
  Z => ();
  S(n) => (ty, n.vec(ty));
}

codef Head(n: Nat):
    S(n).vec(ty) -> ty {
  ap(xs) => xs.fst;
}

def (n: Nat).print():
    n.vec(ty) -> ()
```

Elimination into Types

Needs Type Casts

```
def Nat.vec(ty: Set): Set {
  Z => ();
  S(n) => (ty, n.vec(ty));
}

codef Head(n: Nat):
    S(n).vec(ty) -> ty {
  ap(xs) => xs.fst;
}

def (n: Nat).print():
    n.vec(ty) -> ()
```

No Type Casts Required

```
data Vec(ty: Set): Nat -> Set {
  Nil: Vec(ty, Z);
  Cons(ty, Vec(ty, n)):
    Vec(ty, S(n));
}

codef Head(n: Nat):
    Vec(ty, S(n)) -> ty {
  ap(xs) => match xs {
    Cons(x, xs) => x;
  };
}
```

Elimination into Large Codata

Setoids

```
codata Setoid: Set 1 { .Carrier: Set; ... }
```

Invalid Pattern Matching

```
def Nat.svec(s: Setoid): Setoid
{
  Z => { .Carrier => (); ... };
  S(n) => {
    .Carrier =>
     (s.Carrier,
      n.svec(s).Carrier);
    ...
  };
}
```

Elimination into Large Codata

Setoids

```
codata Setoid: Set 1 { .Carrier: Set; ... }
```

Invalid Pattern Matching

```
def Nat.svec(s: Setoid): Setoid
{
  Z => { .Carrier => (); ... };
  S(n) => {
    .Carrier =>
     (s.Carrier,
      n.svec(s).Carrier);
    ...
  };
}
```

Invert the Order

```
codef SVec(n: Nat; s: Setoid):
    Setoid {
  .Carrier => match n {
    Z => ();
    S(n) =>
      (s.Carrier,
       SVec(n, s).Carrier)
  };
  ...
}
```

Large Data Indices Need Type Casts

Heterogeneous Lists

```
data HList: List(Set) -> Set {
  HNil:                              HList(Nil);
  HCons(x: ty,   xs: HList(tys)): HList(Cons(ty, tys));
}

def HList(Cons(Nat, Nil)).head(): Nat {
  HNil => absurd; HCons(k, xs) => k;
}
```

Large Data Indices Need Type Casts

Heterogeneous Lists

```
data HList:                  Set {
  HNil:                                HList(Nil);
  HCons(x: ty,   xs: HList(tys)): HList(Cons(ty, tys));
}

def HList(Cons(Nat, Nil)).head(): Nat {
  HNil => absurd; HCons(k, xs) => k;
}
```

Large Data Indices Need Type Casts

Heterogeneous Lists

```
data HList:                  Set {
  HNil:                                HList;
  HCons(x: ty,    xs: HList      ): HList;
}

def HList.head(): Nat {
  HNil => absurd; HCons(k, xs) => k;
}
```

Large Data Indices Need Type Casts

Heterogeneous Lists

```
data HList:                  Set {
  HNil:                              HList;
  HCons(x: Univ, xs: HList     ): HList;
}

def HList.head(): Nat {
  HNil => absurd; HCons(k, xs) => k;
}
```

Large Data Indices Need Type Casts

Heterogeneous Lists

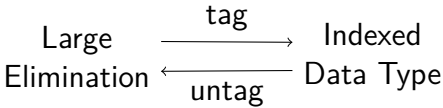
```
data HList:                  Set {
  HNil:                              HList;
  HCons(x: Univ, xs: HList      ): HList;
}

def HList.head(): Nat {
  HNil => absurd; HCons(k, xs) => k;
}
```

Removing Large Data Indices

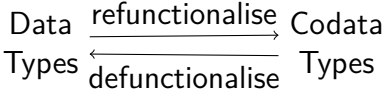
Untag Indexed Type

- Reintroduce large elimination



Refunctionalise Indexing Type

- Inverse of defunctionalisation
- Definitions become projections
- Constructors become codefinitions



Limitations and Open Questions

Incomplete Set of Examples

- Possible for all "wild" examples I've tried
- Need more examples

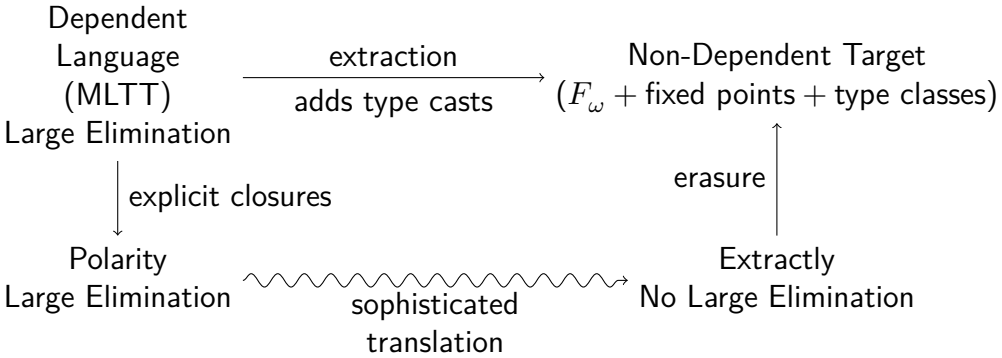
Data Types with Other Large Indices

- Untagging needs data type indices
- What happens when indices are codata types?

Not All Data Types Can Be Untagged

```
data Foo: Nat -> Set {
  Done: Foo(n);
  Step(left: Foo(n), right: Foo(S(n))): Foo(S(n));
}
```

Outline


Dependent
Language
(MLTT)
Large Elimination
extraction
adds type casts
Non-Dependent Target
(F_ω + fixed points + type classes)
explicit closures
erasure
Polarity
Large Elimination
sophisticated
translation
Extractly
No Large Elimination