Practical Datatype Specializations with Phantom Types and Recursion Schemes

Matthew Fluet
Cornell University
fluet@cs.cornell.edu

Riccardo Pucella
Northeastern University
riccardo@ccs.neu.edu

Introduction

- **Problem:** statically enforce program invariants

Introduction

- **Problem:** statically enforce program invariants
- **Solution:** datatype specialization
  - a programming technique for Standard ML

Introduction

- **Problem:** statically enforce program invariants
- **Solution:** datatype specialization
  - a programming technique for Standard ML
  - not a language extension (but informs language design)
  - not a complete solution (but useful in some situations)

Example

- Boolean formulas

```
datatype fmla = Var of string | Not of fmla
              | True | And of fmla * fmla
              | False | Or of fmla * fmla
```

Example

- Boolean formulas

```
datatype fmla = Var of string | Not of fmla
              | True | And of fmla * fmla
              | False | Or of fmla * fmla
```

  - core functionality
    - **`val randomFmla: unit -> fmla`**
    - **`val toString: fmla -> string`**

Example

- Boolean formulas

```
datatype fmla = Var of string | Not of fmla
              | True | And of fmla * fmla
              | False | Or of fmla * fmla
```

  - core functionality and key invariants (*disjunctive normal form*)
    - **`val randomFmla: unit -> fmla`**
    - **`val toString: fmla -> string`**
    - **`val toDnf: fmla -> fmla`**
    - **`val dnfSAT: fmla -> bool`**

Example

- Boolean formulas

```
datatype fmla = Var of string | Not of fmla
              | True | And of fmla * fmla
              | False | Or of fmla * fmla
```

  - core functionality and key invariants (*disjunctive normal form*)
    - `val randomFmla: unit -> fmla`
    - `val toString: fmla -> string`
    - `val toDnf: fmla -> fmla`
    - `val dnfSAT: fmla -> bool`

Example

- Boolean formulas and DNF

```
datatype fmla = ...

datatype atom = Var of string
datatype lit = Var of string | Not of atom
datatype conj = True | And of lit * conj
datatype dnf = False | Or of conj * dnf
```

  - core functionality and key invariants (*disjunctive normal form*)
    - `val randomFmla: unit -> fmla`
    - `val toString: fmla -> string`
    - `val toDnf: fmla -> dnf`
    - `val dnfSAT: dnf -> bool`

Example

- Boolean formulas and DNF

```
datatype fmla = ...

datatype atom = Var of string
datatype lit = Var of string | Not of atom
datatype conj = True | And of lit * conj
datatype dnf = False | Or of conj * dnf
```

  - core functionality and key invariants (*disjunctive normal form*)
    - `val randomFmla: unit -> fmla`
    - `val toString: fmla -> string`
    - `val toDnf: fmla -> dnf`
    - `val dnfSAT: dnf -> bool`

Example

- Boolean formulas and DNF
  - core functionality and key invariants (*disjunctive normal form*)
    - **`val randomFmla: unit -> fmla`**
    - **`val toString: fmla -> string`**
    - **`val toDnf: fmla -> dnf`**
    - **`val dnfSAT: dnf -> bool`**
    - **`val dnfToString: dnf -> string`**

      inefficient: duplicates code

Example

- Boolean formulas and DNF
  - core functionality and key invariants (*disjunctive normal form*)
    - **`val randomFmla: unit -> fmla`**
    - **`val toString: fmla -> string`**
    - **`val toDnf: fmla -> dnf`**
    - **`val dnfSAT: dnf -> bool`**
    - **`val dnfToFmla: dnf -> fmla`**
      inefficient: deep coercion
    - **`val dnfToString = toString o dnfToFmla`**
      inefficient: multiple traversals, intermediate structure

Introduction

- Tension in data structure implementations:
  - be general
    - core functionality is available to all clients
  - be specific
    - key invariant is enforced for a particular client

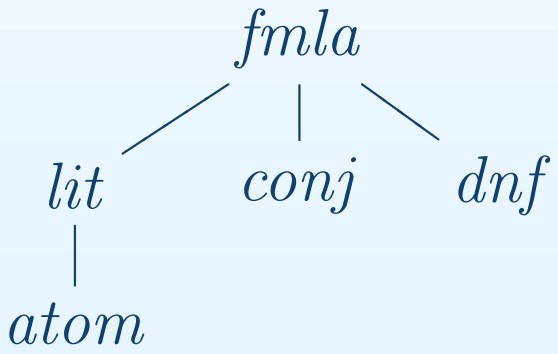
  - use standard type system
    - easy to enforce basic safety properties
    - hard to capture additional invariants

Datatype Specialization

```
datatype fmla = Var of string | Not of fmla
              | True | And of fmla * fmla
              | False | Or of fmla * fmla
withspec atom = Var of string
     and lit  = Var of string | Not of atom
     and conj = True | And of lit * conj
     and dnf  = False | Or of conj * dnf
```

Datatype Specialization

```
datatype fmla = Var of string | Not of fmla
              | True | And of fmla * fmla
              | False | Or of fmla * fmla
withspec atom = Var of string
     and lit  = Var of string | Not of atom
     and conj = True | And of lit * conj
     and dnf  = False | Or of conj * dnf
```



Datatype Specialization Wish List

- implement specializations

Datatype Specialization Wish List

- implement specializations
  - static type checking of structural invariants

Datatype Specialization Wish List

- implement specializations
  - static type checking of structural invariants
  - share representation (inclusion subtyping)
    - avoid expensive run-time coercions
    - allow code reuse

Datatype Specialization Wish List

- implement specializations
  - static type checking of structural invariants
  - share representation (inclusion subtyping)
    - avoid expensive run-time coercions
    - allow code reuse
  - **`case`** analysis

Datatype Specialization Wish List

- implement specializations
  - static type checking of structural invariants
  - share representation (inclusion subtyping)
    - avoid expensive run-time coercions
    - allow code reuse
  - **`case`** analysis
- . . . in Standard ML (with no language extensions)
  - technique is available *now* to *all* Standard ML programmers
  - strictly weaker than a language extension
    - refinement types [Freeman, ’91] [Davies, ’05]

Datatype Specialization

- implement specializations in Standard ML
  - *phantom types* – encode & enforce subtyping relationships
  - *recursion schemes* – recover pattern matching idioms

Outline

- Specialization Interface
- Specializations with Phantom Types
- Specializations with Recursion Schemes

Specialization Interface

```
signature FMLA = sig




end
```

Specialization Interface

```
signature FMLA = sig
    type 'a t

    type 'a AFmla = {fmla: 'a} t
    type 'a ALit  = {lit: 'a} AFmla
    type 'a AAtom = {atom: 'a} ALit
    type 'a AConj = {conj: 'a} AFmla
    type 'a ADnf  = {dnf: 'a} AFmla

    type CFmla = unit AFmla
    type CLit  = unit ALit
    type CAtom = unit AAtom
    type CConj = unit AConj
    type CDnf  = unit ADnf


  structure Fmla : sig
      val Var    : string -> CFmla   val Not : 'a AFmla -> CFmla
      val True   : CFmla             val And : 'a AFmla * 'b AFmla -> CFmla
      val False  : CFmla             val Or  : 'a AFmla * 'b AFmla  -> CFmla

      val dest   : 'a AFmla -> {Var   : string -> 'b, Not : CFmla -> 'b,
                                True  : unit -> 'b,   And : CFmla * CFmla -> 'b,
                                False : unit -> 'b,   Or  : CFmla * CFmla -> 'b} -> 'b

      val coerce : 'a AFmla -> CFmla
  end

  structure Lit : sig
      val Var    : string -> CLit    val Not : 'a AAtom -> CLit

      val dest   : 'a ALit -> {Var : string -> 'b, Not : CAtom -> 'b} -> 'b

      val coerce : 'a ALit -> CLit
  end

  structure Atom : sig
      val Var    : string -> CAtom

      val dest   : 'a AAtom -> {Var : string -> 'b} -> 'b

      val coerce : 'a AAtom -> CAtom
  end

  structure Conj : sig
      val True   : CConj             val And : 'a ALit * 'b AConj -> CConj

      val dest   : 'a AConj -> {True : unit -> 'b, And : CLit * CConj -> 'b} -> 'b

      val coerce : 'a AConj -> CConj
  end

  structure Dnf : sig
      val False  : CDnf              val Or  : 'a AConj * 'b ADnf  -> CDnf

      val dest   : 'a ADnf -> {False : unit -> 'b, Or : CConj * CDnf -> 'b} -> 'b

      val coerce : 'a ADnf -> CDnf
  end
end
```

Specialization Interface

```
signature FMLA = sig
    type 'a t

    type 'a AFmla = {fmla: 'a} t
    type 'a ALit  = {lit: 'a} AFmla
    type 'a AAtom = {atom: 'a} ALit
    type 'a AConj = {conj: 'a} AFmla
    type 'a ADnf  = {dnf: 'a} AFmla

    type CFmla = unit AFmla
    type CLit  = unit ALit
    type CAtom = unit AAtom
    type CConj = unit AConj
    type CDnf  = unit ADnf


  structure Fmla : sig
      val Var    : string -> CFmla   val Not : 'a AFmla -> CFmla
      val True   : CFmla             val And : 'a AFmla * 'b AFmla -> CFmla
      val False  : CFmla             val Or  : 'a AFmla * 'b AFmla  -> CFmla

      val dest   : 'a AFmla -> {Var   : string -> 'b, Not : CFmla -> 'b,
                                True  : unit -> 'b,   And : CFmla * CFmla -> 'b,
                                False : unit -> 'b,   Or  : CFmla * CFmla -> 'b} -> 'b

      val coerce : 'a AFmla -> CFmla
  end

  structure Lit : sig
      val Var    : string -> CLit    val Not : 'a AAtom -> CLit

      val dest   : 'a ALit -> {Var : string -> 'b, Not : CAtom -> 'b} -> 'b

      val coerce : 'a ALit -> CLit
  end

  structure Atom : sig
      val Var    : string -> CAtom

      val dest   : 'a AAtom -> {Var : string -> 'b} -> 'b

      val coerce : 'a AAtom -> CAtom
  end

  structure Conj : sig
      val True   : CConj             val And : 'a ALit * 'b AConj -> CConj

      val dest   : 'a AConj -> {True : unit -> 'b, And : CLit * CConj -> 'b} -> 'b

      val coerce : 'a AConj -> CConj
  end

  structure Dnf : sig
      val False  : CDnf              val Or  : 'a AConj * 'b ADnf  -> CDnf

      val dest   : 'a ADnf -> {False : unit -> 'b, Or : CConj * CDnf -> 'b} -> 'b

      val coerce : 'a ADnf -> CDnf
  end
end
```

Don't Panic

Specialization Interface

```
signature FMLA = sig
    type 'a t

    type 'a AFmla = {fmla: 'a} t
    type 'a ALit  = {lit: 'a} AFmla
    type 'a AAtom = {atom: 'a} ALit
    type 'a AConj = {conj: 'a} AFmla
    type 'a ADnf  = {dnf: 'a} AFmla

    type CFmla = unit AFmla
    type CLit  = unit ALit
    type CAtom = unit AAtom
    type CConj = unit AConj
    type CDnf  = unit ADnf


  structure Fmla : sig
      val Var    : string -> CFmla   val Not : 'a AFmla -> CFmla
      val True   : CFmla             val And : 'a AFmla * 'b AFmla -> CFmla
      val False  : CFmla             val Or  : 'a AFmla * 'b AFmla  -> CFmla

      val dest   : 'a AFmla -> {Var   : string -> 'b, Not : CFmla -> 'b,
                                True  : unit -> 'b,   And : CFmla * CFmla -> 'b,
                                False : unit -> 'b,   Or  : CFmla * CFmla -> 'b} -> 'b

      val coerce : 'a AFmla -> CFmla
  end

  structure Lit : sig
      val Var    : string -> CLit    val Not : 'a AAtom -> CLit

      val dest   : 'a ALit -> {Var : string -> 'b, Not : CAtom -> 'b} -> 'b

      val coerce : 'a ALit -> CLit
  end

  structure Atom : sig
      val Var    : string -> CAtom

      val dest   : 'a AAtom -> {Var : string -> 'b} -> 'b

      val coerce : 'a AAtom -> CAtom
  end

  structure Conj : sig
      val True   : CConj             val And : 'a ALit * 'b AConj -> CConj

      val dest   : 'a AConj -> {True : unit -> 'b, And : CLit * CConj -> 'b} -> 'b

      val coerce : 'a AConj -> CConj
  end

  structure Dnf : sig
      val False  : CDnf              val Or  : 'a AConj * 'b ADnf  -> CDnf

      val dest   : 'a ADnf -> {False : unit -> 'b, Or : CConj * CDnf -> 'b} -> 'b

      val coerce : 'a ADnf -> CDnf
  end
end
```

Don't Panic

- it is all boilerplate
  - it can all be mechanically generated
- specializations as abstract types
  - explicit constructors and destructors

Specialization Interface

```
signature FMLA = sig
   type 'a t

   type 'a AFmla = {fmla: 'a} t
   type 'a ALit  = {lit: 'a} AFmla
   type 'a AAtom = {atom: 'a} ALit
   type 'a AConj = {conj: 'a} AFmla
   type 'a ADnf  = {dnf: 'a} AFmla

   type CFmla = unit AFmla
   type CLit  = unit ALit
   type CAtom = unit AAtom
   type CConj = unit AConj
   type CDnf  = unit ADnf

  structure Fmla : sig
      val Var    : string -> CFmla   val Not : 'a AFmla -> CFmla
      val True   : CFmla             val And : 'a AFmla * 'b AFmla -> CFmla
      val False  : CFmla             val Or  : 'a AFmla * 'b AFmla  -> CFmla

      val dest   : 'a AFmla -> {Var   : string -> 'b, Not : CFmla -> 'b,
                                True  : unit -> 'b,   And : CFmla * CFmla -> 'b,
                                False : unit -> 'b,   Or  : CFmla * CFmla -> 'b} -> 'b

      val coerce : 'a AFmla -> CFmla
  end

  structure Lit : sig
      val Var    : string -> CLit    val Not : 'a AAtom -> CLit

      val dest   : 'a ALit -> {Var : string -> 'b, Not : CAtom -> 'b} -> 'b

      val coerce : 'a ALit -> CLit
  end

  structure Atom : sig
      val Var    : string -> CAtom

      val dest   : 'a AAtom -> {Var : string -> 'b} -> 'b

      val coerce : 'a AAtom -> CAtom
  end

  structure Conj : sig
      val True   : CConj             val And : 'a ALit * 'b AConj -> CConj

      val dest   : 'a AConj -> {True : unit -> 'b, And : CLit * CConj -> 'b} -> 'b

      val coerce : 'a AConj -> CConj
  end

  structure Dnf : sig
      val False  : CDnf              val Or  : 'a AConj * 'b ADnf  -> CDnf

      val dest   : 'a ADnf -> {False : unit -> 'b, Or : CConj * CDnf -> 'b} -> 'b

      val coerce : 'a ADnf -> CDnf
  end
end
```

- Specialization Types
- Specialization Values

Specialization Interface

```
signature FMLA = sig
  type 'a t

  type 'a AFmla = {fmla: 'a} t
  type 'a ALit  = {lit: 'a} AFmla
  type 'a AAtom = {atom: 'a} ALit
  type 'a AConj = {conj: 'a} AFmla
  type 'a ADnf  = {dnf: 'a} AFmla

  type CFmla = unit AFmla
  type CLit  = unit ALit
  type CAtom = unit AAtom
  type CConj = unit AConj
  type CDnf  = unit ADnf

  structure Fmla : sig
    val Var   : string -> CFmla   val Not : 'a AFmla -> CFmla
    val True  : CFmla             val And : 'a AFmla * 'b AFmla -> CFmla
    val False : CFmla             val Or  : 'a AFmla * 'b AFmla  -> CFmla

    val dest  : 'a AFmla -> {Var   : string -> 'b, Not : CFmla -> 'b,
                             True  : unit -> 'b,   And : CFmla * CFmla -> 'b,
                             False : unit -> 'b,   Or  : CFmla * CFmla -> 'b} -> 'b

    val coerce : 'a AFmla -> CFmla
  end

  structure Lit : sig
    val Var   : string -> CLit    val Not : 'a AAtom -> CLit

    val dest  : 'a ALit -> {Var : string -> 'b, Not : CAtom -> 'b} -> 'b

    val coerce : 'a ALit -> CLit
  end

  structure Atom : sig
    val Var   : string -> CAtom

    val dest  : 'a AAtom -> {Var : string -> 'b} -> 'b

    val coerce : 'a AAtom -> CAtom
  end

  structure Conj : sig
    val True  : CConj             val And : 'a ALit * 'b AConj -> CConj

    val dest  : 'a AConj -> {True : unit -> 'b, And : CLit * CConj -> 'b} -> 'b

    val coerce : 'a AConj -> CConj
  end

  structure Dnf : sig
    val False : CDnf              val Or  : 'a AConj * 'b ADnf  -> CDnf

    val dest  : 'a ADnf -> {False : unit -> 'b, Or : CConj * CDnf -> 'b} -> 'b

    val coerce : 'a ADnf -> CDnf
  end
end
```

- Specialization Types
  - Specialization Type
  - Abstract Types
  - Concrete Types
- Specialization Values
  - Constructors
  - Destructors
  - Coercions

The Phantom-Types Technique

- encode information in a superfluous type variable

The Phantom-Types Technique

- encode information in a superfluous type variable
  - *phantom type*: does not contribute to run-time representation

The Phantom-Types Technique

- encode information in a superfluous type variable
  - *phantom type*: does not contribute to run-time representation
- enforce relationship through type unification

Specialization Type

- Interface

```
type 'a t
```

  - type variable instantiated with position in subtyping hierarchy

Specialization Type

- Interface

```
type 'a t
```

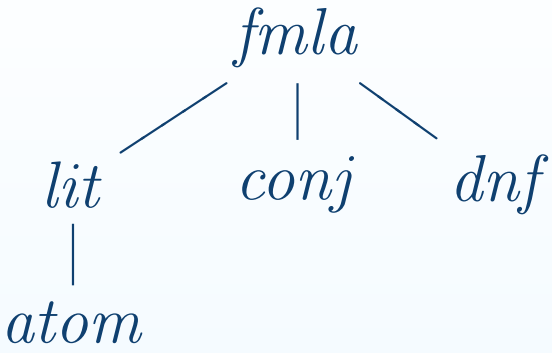
  - type variable instantiated with position in subtyping hierarchy

- Implementation

```
structure Rep = struct
  datatype t = Var of string | Not of t
             | True | And of t * t
             | False | Or of t * t
end
type 'a t = Rep.t
```

  - type variable is phantom
  - all specializations share the same representation

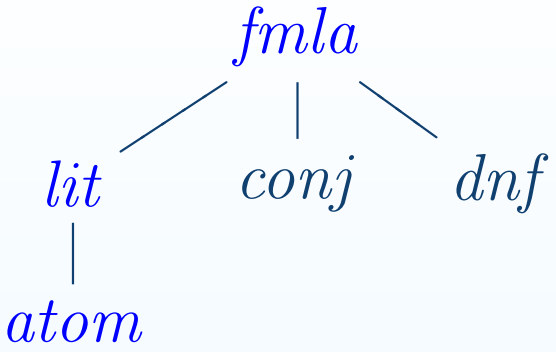
Abstract and Concrete Types



- Interface

```
(* abstract types *)
type 'a AFmla = {fmla: 'a} t
type 'a ALit  = {fmla: {lit: 'a}} t
type 'a AAtom = {fmla: {lit: {atom: 'a}}} t
type 'a AConj = {fmla: {conj: 'a}} t
type 'a ADnf  = {fmla: {dnf: 'a}} t
```

Abstract and Concrete Types

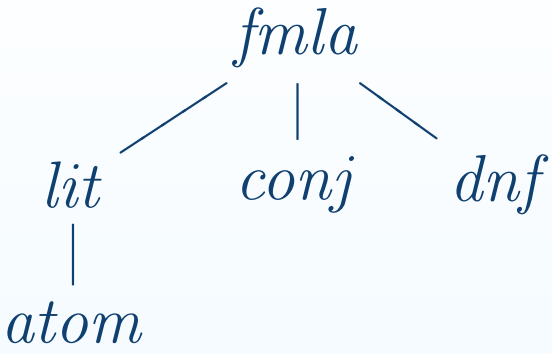


- Interface

```
(* abstract types *)
type 'a AFmla = {fmla: 'a} t
type 'a ALit  = {fmla: {lit: 'a}} t
type 'a AAtom = {fmla: {lit: {atom: 'a}}} t
type 'a AConj = {fmla: {conj: 'a}} t
type 'a ADnf  = {fmla: {dnf: 'a}} t
```

  - record labels describe path through subtyping hierarchy

Abstract and Concrete Types

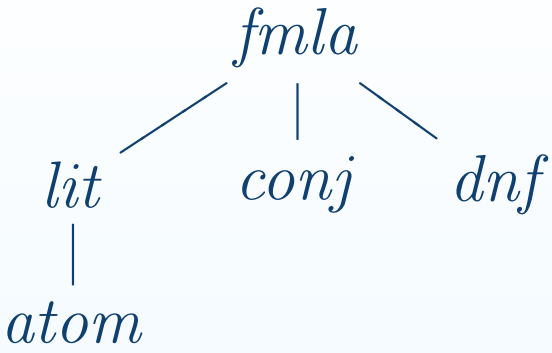


- Interface

```
(* abstract types *)
type 'a AFmla = {fmla: 'a} t
type 'a ALit  = {fmla: {lit: 'a}} t
type 'a AAtom = {fmla: {lit: {atom: 'a}}} t
type 'a AConj = {fmla: {conj: 'a}} t
type 'a ADnf  = {fmla: {dnf: 'a}} t

(* concrete types *)
type CFmla = unit AFmla (* = {fmla: unit} t *)
type CLit  = unit ALit  (* = {fmla: {lit: unit}} t *)
type CAtom = unit AAtom (* = {fmla: {lit: {atom: unit}}} t *)
type CConj = unit AConj (* = {fmla: {conj: unit}} t *)
type CDnf  = unit ADnf  (* = {fmla: {dnf: unit}} t *)
```

Abstract and Concrete Types



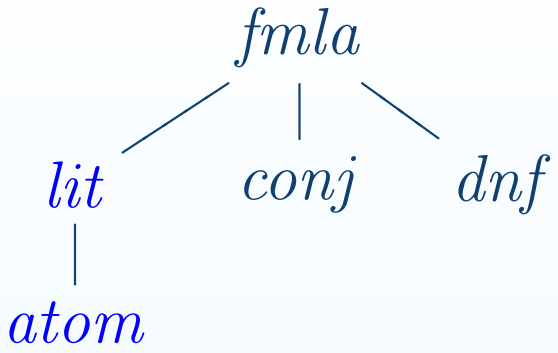
- Interface

```
(* abstract types *)
type 'a AFmla = {fmla: 'a} t
type 'a ALit  = {fmla: {lit: 'a}} t
type 'a AAtom = {fmla: {lit: {atom: 'a}}} t
type 'a AConj = {fmla: {conj: 'a}} t
type 'a ADnf  = {fmla: {dnf: 'a}} t

(* concrete types *)
type CFmla = unit AFmla (* = {fmla: unit} t *)
type CLit  = unit ALit  (* = {fmla: {lit: unit}} t *)
type CAtom = unit AAtom (* = {fmla: {lit: {atom: unit}}} t *)
type CConj = unit AConj (* = {fmla: {conj: unit}} t *)
type CDnf  = unit ADnf  (* = {fmla: {dnf: unit}} t *)
```

  - unification enforces subtyping relationship

Abstract and Concrete Types



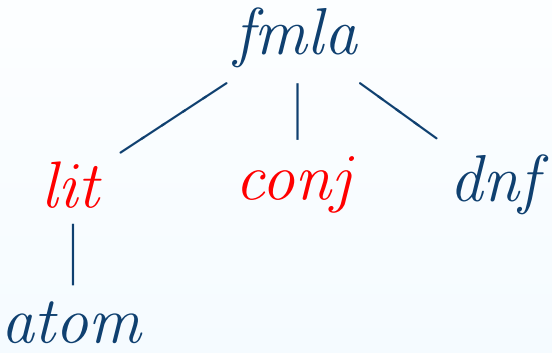
- Interface

```
(* abstract types *)
type 'a AFmla = {fmla: 'a} t
type 'a ALit  = {fmla: {lit: 'a}} t
type 'a AAtom = {fmla: {lit: {atom: 'a}}} t
type 'a AConj = {fmla: {conj: 'a}} t
type 'a ADnf  = {fmla: {dnf: 'a}} t

(* concrete types *)
type CFmla = unit AFmla (* = {fmla: unit} t *)
type CLit  = unit ALit  (* = {fmla: {lit: unit}} t *)
type CAtom = unit AAtom (* = {fmla: {lit: {atom: unit}}} t *)
type CConj = unit AConj (* = {fmla: {conj: unit}} t *)
type CDnf  = unit ADnf  (* = {fmla: {dnf: unit}} t *)
```

  - unification enforces subtyping relationship

Abstract and Concrete Types



- Interface

```
(* abstract types *)
type 'a AFmla = {fmla: 'a} t
type 'a ALit  = {fmla: {lit: 'a}} t
type 'a AAtom = {fmla: {lit: {atom: 'a}}} t
type 'a AConj = {fmla: {conj: 'a}} t
type 'a ADnf  = {fmla: {dnf: 'a}} t

(* concrete types *)
type CFmla = unit AFmla (* = {fmla: unit} t *)
type CLit  = unit ALit  (* = {fmla: {lit: unit}} t *)
type CAtom = unit AAtom (* = {fmla: {lit: {atom: unit}}} t *)
type CConj = unit AConj (* = {fmla: {conj: unit}} t *)
type CDnf  = unit ADnf  (* = {fmla: {dnf: unit}} t *)
```

  - unification enforces subtyping relationship

Phantom Types and Subtyping

- encode position within a subtyping hierarchy:

$\langle\sigma\rangle_C : \Sigma_{\leq} \longrightarrow \tau_{\mathsf{SML}}$ concrete type (ground type)

(covariant positions)

$\langle\sigma\rangle_A : \Sigma_{\leq} \longrightarrow \tau_{\mathsf{SML}}$ abstract type (polymorphic type scheme)

(contravariant positions)

- enforce subtyping relationship

$\langle\sigma_1\rangle_C$ unifies with $\langle\sigma_2\rangle_A$ iff $\sigma_1 \leq \sigma_2$

Examples

```
val randomFmla : unit -> CFmla

val toString : 'a AFmla -> string

val toDnf : 'a AFmla -> CDnf

val dnfSAT : 'a ADnf -> bool
```

Constructors

- Interface

```
structure Conj : sig
  val True : CConj
  val And  : 'a ALit * 'b AConj -> CConj
end
```

  - subtyping on constructor arguments

Constructors

- Interface

```
structure Conj : sig
  val True : CConj
  val And  : 'a ALit * 'b AConj -> CConj
end
```

  - subtyping on constructor arguments

- Implementation

```
structure Conj = struct
  val True = Rep.True
  val And  = Rep.And
end
```

Destructors

- Interface

```
structure Conj : sig
  val dest : 'a AConj ->
             {True : unit -> 'b,
              And : CLit * CConj -> 'b} -> 'b
end
```

  - subtyping on destructor argument
  - supertyping on constructor arguments

Destructors

- Implementation

```
fun fail _ = raise Match
structure Rep = struct
  fun dest c (v,n,t,a,f,o) =
    case c of Var(s) => v (s) | Not(f) => n (f)
            | True => t () | And(f,g) => a (f,g)
            | False => f () | Or(f,g) => o (f,g)
end
structure Conj = struct
  fun dest c {True=t,And=a} =
    Rep.dest c (fail,fail,t,a,fail,fail)
end
```

  - invariants of the specializations imply exception never raised

Coercions

- Interface

```
structure Conj : sig
  val coerce : 'a AConj -> CConj
end
```

  - subtyping on coercion argument

Coercions

- Interface

```
structure Conj : sig
  val coerce : 'a AConj -> CConj
end
```

  - subtyping on coercion argument

- Implementation

```
structure Conj : struct
  fun coerce v = v
end
```

  - identity function that changes the (phantom) type of value

Specializations with Recursion Schemes

- destructor functions are cumbersome
  - break familiar pattern-matching idioms
- recursion schemes [Wang and Murphy, ’03]
  - hide representation of an abstract type
  - while still supporting pattern matching

Specializations with Recursion Schemes

- destructor functions are cumbersome
  - break familiar pattern-matching idioms
- recursion schemes [Wang and Murphy, ’03]
  - hide representation of an abstract type
  - while still supporting pattern matching
- *shallow coercions*

Conclusion

- Applied two programming techniques to the problem of capturing structural invariants in user-defined datatypes
  - *phantom types* – encode subtyping relationships
  - *recursion schemes* – recover pattern matching idioms

- **`http://www.cs.cornell.edu/People/fluet/specializations`**
  - interesting examples of datatype specializations
  - a tool to mechanically generate implementations