ML 2005 Preliminary Version

# Practical Datatype Specializations with Phantom Types and Recursion Schemes

Matthew Fluet[1] Riccardo Pucella[2]

*Department of Computer Science*
*Cornell University*
*Ithaca, NY 14853 USA*

**Abstract**

Datatype specialization is a form of subtyping that captures program invariants on data structures that are expressed using the convenient and intuitive `datatype` notation. Of particular interest are structural invariants such as well-formedness. We investigate the use of phantom types for describing datatype specializations. We show that it is possible to express statically-checked specializations within the type system of Standard ML. We also show that this can be done in a way that does not lose useful programming facilities such as pattern matching in `case` expressions.

*Key words:* data types and structures, invariants

## 1 Introduction

Data structures that are used pervasively in an application are often implemented with a degree of genericity that makes them suited, though not always well suited, to their various uses. A benefit of this genericity is that it ensures that core functionality of the data structure is available to all clients. A limitation is that a client that is required to produce, consume, or maintain an instance of the data structure subject to a particular invariant has difficulty enforcing the invariant. While many languages boast a type system that statically enforces basic safety properties on data structures, few allow programmers to directly capture these additional invariants within the type system. This is unfortunate: when invariants are reflected into a type system, compile-time type errors will indicate code that could violate these invariants.

To make our discussion concrete, consider Boolean formulas, which we use as a running example throughout this paper. A straightfoward representation of formulas is the following:

[1] Email: `fluet@cs.cornell.edu`
[2] Email: `riccardo@cs.cornell.edu`

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

We can easily define a function `eval` that takes a formula and an environment associating every variable in the formula with a truth value, and returns the truth value of the formula. Similarly, we can define a `toString` function that takes a formula and returns a string representation of the formula. As is well known, a propositional formula can always be represented in a special form called Disjunctive Normal Form (or DNF), as a disjunction of conjunctions of variables and negations of variables. A formula in DNF is still a formula, but it has a restricted structure. Some algorithms that operate on formulas require their input to be presented in DNF; therefore, it makes sense to statically check that a formula is in DNF.

One way to perform this static checking is to simply introduce one datatype for (generic) formulas and another datatype for DNF formulas, and provide functions to explicitly convert between them. This is, of course, inefficient. For example, converting a formula in DNF to a string via `toString` would require two complete traversals of the DNF formula (one to convert it to a generic formula, and one to build the string representation), as well as the allocation of an intermediate structure (of the same size as the original formula). An alternative is to define a DNF formula as a *specialization* of formulas. For the sake of presentation, we assume a special syntax for specializations. This syntax should be self-explanatory:

```
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
```

Roughly speaking, the specialization `dnf` of the datatype `fmla` is restricted so that the `Or` constructor creates list of conjunctions terminated with the `False` constructor. A conjunction is defined by another specialization `conj` of the datatype `fmla` that restricts the `And` constructor to forming lists of literals. A literal is essentially a variable or a negated variable. This can be captured using two specializations, `atom` for atomic literals and `lit` for literals. Notice that to define the `dnf` specialization, we need all the specializations `dnf`, `conj`, `lit`, and `atom`. These specializations induce a simple subtyping hierarchy:

$$\begin{tikzcd}
& \mathit{fmla} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\mathit{lit} \arrow[d, dash] & \mathit{conj} & \mathit{dnf} \\
\mathit{atom} & &
\end{tikzcd}$$

For uniformity, we consider the datatype as a degenerate specialization.

Abstracting from this example, we define a *specialization* of a datatype to be a version of the datatype that is generated by a subset of the datatype constructors, which may themselves be required to be applied to specializations of the datatype. If we view elements of a datatype as data structures, then we can view elements of the specialization as data structures obeying certain restrictions. A set of specializations of a datatype may be specified in a mutually recursive fashion.

In this paper, we show that we can implement, in Standard ML [15], much of what one would expect from a language with a type system that directly supports the kind of specializations described above. For example, we can write a `toDnf` function that statically guarantees not only that its result is a formula, but also that it is a DNF formula. The advantage of implementing specialization invariants in SML is that type systems directly supporting specializations are complex and not widely available. (A type system that enforces similar, but strictly more powerful, invariants is the *refinement types* system [3,8]; see Section 4.)

What are the key features of specializations that we would like available? For one, we would like the representation of values of specialized types to be the same as the representation of the original datatype. This is important to avoid expensive run-time conversions and to allow code reuse. For example, we should be able to implement a single function to evaluate not only an unspecialized formula, but also any specialization of formulas, such as the `dnf` specialization. Moreover, we would like to write `case` expressions that do not include branches for constructors that do not occur in the specialization of the value being examined. For example, if we perform a case analysis on a value with specialization `dnf`, we should only need to supply branches for the `False` and `Or` constructors.

Drawing on previous work relating phantom types and subtyping [7] and the intuition that the specializations of a datatype induce a subtyping hierarchy, we present in Section 2 an informal translation based on phantom types from a set of specializations of a datatype to an interface providing constructors, destructors, and coercions corresponding to the specializations. This interface forms a minimal set of primitive operations that provide the functionality (and static guarantees) of the specializations. It uses the SML type system to enforce the invariants embodied by the specializations.

Unfortunately, the interface leaves much to be desired, particularly if we wish to promote datatype specializations as a practical programming technique that integrates naturally with standard, idiomatic SML usage. In order to overcome these deficiencies, notably the lack of pattern matching, we draw on another programming techique: recursion schemes [28]. We present in Section 3 an informal translation from the set of specializations of the datatype, using recursion schemes and the interface described above, to an improved interface that provides the functionality of the specializations through *bona fide* SML datatypes. As before, the SML type system enforces the invariants of the specializations.

The reader may well wonder why we choose to encode specializations in SML, rather than designing a language extension. First, there are numerous SML compilers [9,16,17,18,20,24,25]; while many of these compilers implement some language extensions, the SML language specified by the *Definition* [15] behaves the same in all compilers. Hence, the technique described here is available *now* to *all* SML programmers. Furthermore, encoding specializations in SML is more expedient than writing our own compiler or modifying an existing compiler.

Our notion of datatype specializations is quite general, and we believe that it can capture a good number of useful invariants. For example, we can define a specialization that ensures that the formula contains no variables:

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

The following specializations distinguish between zero and non-zero:

```
datatype nat       = Zero | Succ of nat
  withspec zero    = Zero
       and nonzero = Succ of nat
```

Lists give rise to interesting specializations. Consider the following specializations, distinguishing between empty, singleton, and nonempty lists:

```
datatype 'a list        = Nil | Cons of 'a * 'a list
  withspec 'a empty     = Nil
       and 'a singleton = Cons of 'a * 'a empty
       and 'a nonempty  = Cons of 'a * 'a list
```

The following specializations distinguish between lists of even and odd length:

```
datatype 'a list    = Nil | Cons of 'a * 'a list
  withspec 'a even = Nil | Cons of 'a * 'a odd
       and 'a odd  = Cons of 'a * 'a even
```

Finally, we can use specializations to define abstract syntax trees that distinguish between arbitrary expressions and well-formed expressions (e.g., well-typed expressions [4,13], expressions in normal forms, etc.). A simple example of this is the following:

```
datatype exp       = Bool of bool | And of exp * exp
                   | Int of int | Plus of exp * exp
                   | If of exp * exp * exp
  withspec boolexp = Bool of bool | And of boolexp * boolexp
                   | If of boolexp * boolexp * boolexp
       and intexp  = Int of int | Plus of intexp * intexp
                   | If of boolexp * intexp * intexp
```

More involved examples that can be expressed using specializations include red-black trees that check the critical invariant that no red node has a red child after inserting a new element, and constructors for expressions in the simply-typed $\lambda$-calculus that permit only the building of type-correct expressions, essentially using an encoding manipulating de Brujin indices.

## 2 Specializations with Phantom Types

How should we write (in SML) an implementation of the formula specializations so that the type system enforces the appropriate structural invariants? In this section, we give a highly-stylized implementation that achieves this particular goal. We hope that the reader will grasp the straightforward generalization of this implementation to arbitrary specializations.[3] We feel that a fully elaborated example is more instructive than a formal translation, where definitions and notation become burdensome and obfuscating.

We first review the essence of the phantom-types technique and its application to subtyping [7]. The phantom-types technique uses the definition of type equivalence to encode information in a superfluous type variable of a type. (Because instantiations of this type variable do not contribute to the run-time representation of values of the type, it is called a *phantom type*.) Unification can then be used to enforce a particular structure on the information carried by two such types.

When applied to subtyping, the information we wish to encode is a position within a subtyping hierarchy. We require an encoding $\langle\sigma\rangle$ of each specialization $\sigma$ in the hierarchy; this encoding should yield a type in the SML type system, with the property that $\langle\sigma_1\rangle$ unifies with $\langle\sigma_2\rangle$ if and only if $\sigma_1$ is a subtype of $\sigma_2$ in the hierarchy (written $\sigma_1 \leq \sigma_2$). An obvious issue is that we want to use unification (a symmetric relation) to capture subtyping (an asymmetric relation). The simplest solution is to use two encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ defined over all the specializations in the hierarchy. A *value* of specialization $\sigma$ will be given a type using $\langle\sigma\rangle_C$. We call $\langle\sigma\rangle_C$ the *concrete encoding* of $\sigma$, and we assume that it uses only ground types (i.e., no type variables). In order to restrict the domain of an operation or constructor to the set of values that are subtypes of a specialization $\sigma$, we use $\langle\sigma\rangle_A$, the *abstract encoding* of $\sigma$. In order for the underlying type system to enforce the subtyping hierarchy, we require the encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ to *respect* the subtyping hierarchy by satisfying the following property:

For all specializations $\sigma_1$ and $\sigma_2$, $\langle\sigma_1\rangle_C$ unifies with $\langle\sigma_2\rangle_A$ iff $\sigma_1 \leq \sigma_2$.

To allow for unification, the abstract encoding introduces free type variables. Since, in the SML type system, a top-level type cannot contain free type variables, the abstract encoding is always a part of some polymorphic type

[3] Our running example uses a first-order, monomorphic datatype, of which abstract syntax trees are a typical example. Extending the implementation to handle first-order, polymorphic datatypes is straightforward. It is also possible to handle higher-order datatypes; we briefly consider this in Section 4.

```
signature FMLA = sig
  (* specialization type *)
  type 'a t

  (* abstract types *)
  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

  (* concrete types *)
  type CFmla = unit AFmla
  type CLit  = unit ALit
  type CAtom = unit AAtom
  type CConj = unit AConj
  type CDnf = unit ADnf

  structure Fmla : sig
    (* constructors *)
    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
    (* destructor *)
    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
    (* coercion *)
    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
```

Fig. 1. The FMLA signature

scheme. This leads to some restrictions on the uses of abstract encodings, the details of which are beyond the scope of this paper (but, see our previous work [7] for a thorough discussion). We assert that abstract encodings are used appropriately in the following presentation.

Figures 1 and 2 give a signature and corresponding implementation of the specializations above. The amount of code may seem staggering for such a small example, but we shall see that most of it is boilerplate code. (In fact, it is straightforward to mechanically generate this code from a declarative

```
structure Fmla :> FMLA = struct
  structure Rep = struct
    (* representation type *)
    datatype t = Var of string | Not of t
               | True | And of t * t
               | False | Or of t * t
    (* destructor *)
    fun fail _ = raise Match
    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)
    (* coercions *)
    fun coerce (v) = v
  end
  (* specialization type *)
  type 'a t = Rep.t

  (* abstract types *)
  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

  (* concrete types *)
  type CFmla = unit AFmla
  type CLit  = unit ALit
  type CAtom = unit AAtom
  type CConj = unit AConj
  type CDnf = unit ADnf

  structure Fmla = struct
    open Rep
    fun dest c {Var=v,Not=n,True=t,And=a,False=f,Or=o} = Rep.dest c (v,n,t,a,f,o)
  end
  structure Lit = struct
    open Rep
    fun dest c {Var=v,Not=n} = Rep.dest c (v,n,fail,fail,fail,fail)
  end
  structure Atom = struct
    open Rep
    fun dest c {Var=v} = Rep.dest c (v,fail,fail,fail,fail,fail)
  end
  structure Conj = struct
     open Rep
     fun dest c {True=t,And=a} = Rep.dest c (fail,fail,t,a,fail,fail)
  end
  structure Dnf = struct
     open Rep
     fun dest c {False=f,Or=o} = Rep.dest c (fail,fail,fail,fail,f,o)
  end
end
```

Fig. 2. The `Fmla` structure

description of the specializations; see Section 5.) Moreover, all the action is in the signature! The implementation is trivial. Part of the reason for this explosion of code is that we implement specializations notionally as abstract types with explicit constructors and destructors. (Section 3 shows how to improve upon this seemingly draconian implementation.) With this in mind, let us examine the different elements of the signature and their implementation.

***Types.*** The first part of the signature defines the types for the specializations of formulas. We introduce a polymorphic type `'a t`, representing the values of the specializations.

The first series of type abbreviations combines the abstract encodings of the specializations with the specialization type, yielding the *abstract types*. Consider the definition of the type `ALit`:

```
type 'a ALit = {lit: 'a} AFmla (* = {fmla: {lit: 'a}} t *)
```

Here, `{fmla: {lit: 'a}}` is the abstract encoding of the specialization `lit` in the subtyping hierarchy given above. Note that the sequence of record labels describes the path through the subtyping hierarchy from the root to the `lit` specialization.[4]

The second series of type abbreviations instantiates the abstract types with `unit`, yielding the *concrete types*. We can verify that `CLit` and `CAtom` unify with `ALit`. We can also verify that all concrete types unify with `AFmla`, capturing the fact that the `fmla` specialization is the top element of the subtyping hierarchy. Yet `CLit` does not unify with `AAtom`, `AConj`, or `ADnf`. In other words, the encodings respect the subtyping hierarchy.

In the implementation of Figure 2, we see that the type `Rep.t` is implemented as a *bona fide* SML datatype, while the type `'a t` is implemented as a type abbreviation whose polymorphic type variable is ignored, but serves as a placeholder for a position in the subtyping hierarchy. Hence, all the specializations share the same representation. We use *opaque signature matching* in Figure 2. This is crucial to get the required behavior for the phantom types.

***Constructors.*** For every specialization, the interface provides a function for each constructor of the specialization. For instance, the `atom` specialization has a single constructor (`Var`), so we provide a function[5]

```
val Atom.Var : string -> CAtom
```

that returns an element of the specialization `atom` (and hence, of type `CAtom`). We allow subtyping on the constructor arguments, where appropriate. Hence, the `And` constructor for `conj` is available as:

```
val Conj.And : 'a ALit * 'b AConj -> CConj
```

The implementation of these constructors is trivial. They are simply aliases for the actual constructors of the representation type (brought into scope by `open Rep`). Placing them in different structures allows us to constrain their particular type, depending on the specialization we want them to yield.

***Destructors.*** For every specialization, the interface provides a destructor

[4] This is essentially the encoding of tree hierarchies given in our previous work [7]. All of the other encodings given in that work are also applicable in this setting.

[5] In the following and in Section 3, we occasionally replicate declarations in the signature or structure using long identifiers where the SML syntax requires an (unqualified) identifer. We do so in order to unambiguously denote the appropriate portion of the code; in all cases, the meaning should be clear.

function that can be used to simultaneously discriminate and deconstruct elements of the specialization, similar to the manner in which the `case` expression operates in SML. Each destructor function takes an element of the specialization as well as functions to be applied to the arguments of the matched constructor.

As an example, consider `Conj.dest`, the destructor function for the specialization `conj`. Because the elements of `conj` are built using only the `True` and `And` constructors, deconstructing elements of such a specialization can only yield the `True` constructor or the `And` constructor applied to a `lit` value and to a `conj` value. Therefore, we give `Conj.dest` the type:

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

Similar reasoning allows us to drop or refine the types of various branches in the destructor functions for the other specializations.

Informally, the invariants of the specializations are preserved when we use concrete encodings in covariant type positions and abstract encodings in contravariant type positions. This explains the appearance of concrete encodings in the argument types of the branch functions.

Destructor functions also have a trivial implementation. They are simply implemented as an SML `case` expression. On the branches for which no function is provided, we raise an exception. If the invariants of the specializations are enforced, we know that those exceptions will never be raised! By virtue of our encoding of subtyping, static typing ensures that this exception will never be raised by programs that use the interface [7].

***Coercions.*** Finally, the interface provides coercion functions that convert subtypes to supertypes. Such coercion functions are necessary because, intuitively, phantom types provide only a restricted form of subtyping. To illustrate the problem, consider the following function; it does not typecheck because the type of the true branch is `CLit` and the type of the false branch is `CAtom` – two types that cannot be unified:

```
fun bad b = if b then Lit.Var ("p") else Atom.Var ("q")
```

Instead, we must write the following:

```
fun good b = if b then Lit.Var ("p") else Lit.coerce (Atom.Var ("q"))
```

Technically, this behavior is due to the fact that type subsumption occurs only at type application (implicit in SML), which most often coincides with function application. Thus, when two expressions of different specializations occur in contexts that must have equal types, such as the branches of an `if` expression, subsumption does not occur, and the expressions must be coerced to a common supertype. Coercions are also useful to work around a restriction in SML that precludes the use of polymorphic recursion [10,11]. We shall shortly see an example where this use of a coercion is necessary.

The implementation of coercion functions is trivial. They are simply identity functions that change the (phantom) type of a value.

## *2.1 Examples*

Let us give a few examples of functions that can be written against the interface of formula specializations given above. First, consider a simple function to identify the top-level operator of a formula:

```
fun identify (f: 'a AFmla): string =
  Fmla.dest f {
    Var   = fn _ => "variable",       Not = fn _ => "negation",
    True  = fn _ => "constant true",  And = fn _ => "conjunction",
    False = fn _ => "constant false", Or  = fn _ => "disjunction"}
```

Note that the type of `identify`, `'a AFmla -> string`, asserts that the function may be safely applied to any formula specialization.

A more interesting example is a recursive `toString` function that returns a string representation of a formula. A simple implementation is the following:

```
fun toString (f: 'a AFmla): string = let
  fun toString' (f: CFmla): string =
    Fmla.dest f {
      Var   = fn s => s,
      Not   = fn f => concat ["-", toString' f],
      True  = fn () => "T",
      And   = fn (f1,f2) => concat ["(", toString' f1, " & ", toString' f2, ")"],
      False = fn () => "F",
      Or    = fn (f1,f2) => concat ["(", toString' f1, " | ", toString' f2, ")"]}
in toString' (Fmla.coerce f) end
```

Note that the inferred type of the `toString'` function is `CFmla -> string`, because it is recursively applied to variables of type `CFmla` in the `Fmla.dest` branches and SML does not support polymorphic recursion. However, we can recover a function that allows subtyping on its argument by composing `toString'` with an explicit coercion. Now the SML type system infers the desired type for the `toString` function.

If we had polymorphic recursion, we could directly assign the type `'a AFmla -> string` to `toString'`. One may ask whether the lack of polymorphic recursion in SML poses a significant problem for the use of specializations as we have described them. Fortunately, under reasonable assumptions,[6] it can be shown that use of specializations never needs polymorphic recursion in an essential way. This is a consequence of the fact that the use of phantom types in specializations only influences the *type* of an expression or value, never its representation. The argument proceeds as follows.

Consider a recursive function `f` with type $\sigma$ `->` $\tau$, for a specialization $\sigma$. Any recursive call in the body of `f` must be applied to an argument `x` of specialization $\sigma'$, where $\sigma'$ is a subtype of $\sigma$. Since $\sigma'$ is a subtype of $\sigma$, there exists a coercion from $\sigma'$ to $\sigma$. Hence, we can always implement the function

[6] The main assumption is that the desired recursive function on the unspecialized datatype can itself be written without polymorphic recursion. If this is not the case, then the function cannot be written in SML with or without phantom types and specializations. (Thus, we exclude specializations of non-regular datatypes.) On the other hand, if this is the case, then one can write the function using only the interface provided by specializations.

```
fun andConjs (f: CConj, g: CConj): CConj =
  Conj.dest f {True = fn () => g,
               And = fn (f1,f2) => Conj.And (f1, andConjs (f2, g))}
fun orDnfs (f: CDnf, g: CDnf): CDnf =
  Dnf.dest f {False = fn () => g,
              Or = fn (f1,f2) => Dnf.Or (f1, orDnfs (f2, g))}
fun andConjDnf (f: CConj, g: CDnf): CDnf =
  Dnf.dest g {False = fn () => Dnf.False,
              Or = fn (g1,g2) => Dnf.Or (andConjs (f, g1), andConjDnf (f, g2))}
fun andDnfs (f: CDnf, g: CDnf): CDnf =
  Dnf.dest f {
    False = fn () => Dnf.False,
    Or = fn (f1,f2) => Dnf.dest g {
                         False = fn () => Dnf.False,
                         Or = fn (g1,g2) => Dnf.Or (andConjs (f1, g1),
                                                    orDnfs (andConjDnf (f1, g2),
                                                    orDnfs (andConjDnf (g1, f2),
                                                            andDnfs (f2, g2))))}}
fun litToDnf (f: 'a ALit): CDnf = Dnf.Or (Conj.And (f, Conj.True), Dnf.False)
fun toDnf (f: 'a AFmla): CDnf = let
  fun toDnf' (f: CFmla): CDnf =
    Fmla.dest f {
      Var = fn s => litToDnf (Atom.Var s),
      Not = fn f => Fmla.dest f {
                      Var = fn s => litToDnf (Lit.Not (Atom.Var s)),
                      Not = fn f => toDnf' f,
                      True = fn () => toDnf' Fmla.False,
                      And = fn (f,g) => toDnf' (Fmla.Or (Fmla.Not f, Fmla.Not g)),
                      False = fn () => toDnf' Fmla.True,
                      Or = fn (f,g) => toDnf' (Fmla.And (Fmla.Not f, Fmla.Not g))},
      True = fn () => Dnf.Or (Conj.True, Dnf.False),
      And = fn (f,g) => andDnfs (toDnf' f, toDnf' g),
      False = fn () => Dnf.False,
      Or = fn (f,g) => orDnfs (toDnf' f, toDnf' g)}
in toDnf' (Fmla.coerce f) end
```

Fig. 3. The `toDnf` function

as we did for the `toString` function: set the domain of an auxilary function `f'` to the concrete encoding of $\sigma$ and write the recursive calls as `f'` ($\sigma$`.coerce x`). (In the `toString` example, all of the recursive calls are on values of type `CFmla`, so the coercion can be elided.) Finally, we recover the appropriate subtyping in the argument of `f` by composing `f'` with $\sigma$`.coerce`.

Figure 3 gives an extended example culminating with a `toDnf` function that converts any formula into an equivalent DNF formula. The type of this function, `'a AFmla -> CDnf`, ensures that the result formula is a DNF formula. We further note that the use of type annotations in Figure 3 is completely superfluous. Type inference will deduce precisely these types.

There is a difference between the guarantee made by the SML type system and the guarantee made by a specialization library employing the phantom-types technique. While the former ensures that "a well-typed program won't go wrong," the latter ensures that "a well-typed client won't go wrong, provided the specialization library is correctly implemented." We hope that this section has demonstrated that the implementation of a specialization library is straightforward. However, a more subtle point is that one must choose specializations that capture properties and invariants of interest in order to gain the benefits of additional static guarantees.

## 3 Specializations with Recursion Schemes

While Section 2 describes an interesting theoretical result—the ability to define a `toDnf` function whose type statically enforces a structural invariant without recourse to separate datatypes—it is not yet clear whether the methodology presented is usable in practice. In particular, pattern matching on specialization types must be performed via application of the destructor functions, rather than via SML's built-in syntactic support for pattern matching. The result is the "truly unreadable code"[7] in Figure 3. While any use of pattern matching can be desugared to applications of the destructor functions, important aspects of the pattern-matching programming idiom are seriously inhibited by the encoding in the previous section.

Examining Figure 3 reveals two particularly glaring assaults on readability that could be improved with pattern matching. First, consider the `andDnfs` function. Informally, one can describe the intended behavior of the function in the following way: *consider both arguments in the* ***`dnf`*** *specialization: if either argument is a* ***`False`*** *element, return* ***`False`***; *if both arguments are* ***`Or`*** *elements, return the disjunction of the pairwise conjunction of the elements' arguments.* Unfortunately, the written code obscures this behavior. This highlights two missing aspects of pattern matching: the ability to match simultaneously via the nesting of datatype patterns within tuple patterns, and the ability to give a wild-card match.

Second, consider the `toDnf'` function, more specifically, the nested application of the `caseFmla` destructor function. Here, one misses the ability to write nested patterns, which would combine the two applications of destructor functions into a single pattern match.

For comparison, Figure 4 implements the `toDnf` function for the unspecialized `Rep.t` datatype, addressing the concerns above. However, this apparent improvement has come at a cost. The compiler now issues multiple nonexhaustive match warnings. More importantly, the type system provides no assurance that the result is in Disjunctive Normal Form.

We seek a solution that brings the expressiveness and convienence of pattern matching to specializations. Recall the observation that we made in the introduction: the same static invariants can be obtained by using distinct datatypes for each specialization and providing functions to convert between them. As we pointed out, this is inefficient. However, there is a middle-ground solution, one that uses distinct datatypes for their induced patterns and fine-grained coercions to localize coercions to and from the specialization types and the specialization datatypes.

We take as inspiration Wang and Murphy's recursion schemes [28]. Exploiting two-level types, which split an inductively-defined type into a component that represents the structure of the type and a component that ties the

[7] courtesy of an anonymous reviewer

```
fun andConjs (f: Rep.t, g: Rep.t): Rep.t =
  case f of Rep.True => g
          | Rep.And (f1,f2) => Rep.And (f1, andConjs (f2, g))
fun orDnfs (f: Rep.t, g: Rep.t): Rep.t =
  case f of Rep.False => g
          | Rep.Or (f1,f2) => Rep.Or (f1, orDnfs (f2, g))
fun andConjDnf (f: Rep.t, g: Rep.t): Rep.t =
  case g of Rep.False => Rep.False
          | Rep.Or (g1,g2) => Rep.Or (andConjs (f, g1), andConjDnf (f, g2))
fun andDnfs (f: Rep.t, g: Rep.t): Rep.t =
  case (f, g) of
    (Rep.False, _) => Rep.False
  | (_, Rep.False) => Rep.False
  | (Rep.Or (f1,f2), Rep.Or (g1, g2)) => Rep.Or (andConjs (f1, g1),
                                         orDnfs (andConjDnf (f1, g2),
                                         orDnfs (andConjDnf (g1, f2),
                                                 andDnfs (f2, g2))))
fun litToDnf (f: Rep.t): Rep.t = Rep.Or (Rep.And (f, Rep.True), Rep.False)
fun toDnf (f: Rep.t): Rep.t = let
  fun toDnf' (f: Rep.t): Rep.t =
    case f of
      Rep.Var s => litToDnf (Rep.Var s)
    | Rep.Not (Rep.Var s) => litToDnf (Rep.Not (Rep.Var s))
    | Rep.Not (Rep.Not f) => toDnf' f
    | Rep.Not Rep.True => Rep.False
    | Rep.Not (Rep.And (f,g)) => toDnf' (Rep.Or (Rep.Not f, Rep.Not g))
    | Rep.Not Rep.False => toDnf' Rep.True
    | Rep.Not (Rep.Or (f,g)) =>toDnf' (Rep.And (Rep.Not f, Rep.Not g))
    | Rep.True => Rep.Or (Rep.True, Rep.False)
    | Rep.And (f,g) => andDnfs (toDnf' f, toDnf' g)
    | Rep.False => Rep.False
    | Rep.Or (f,g) => orDnfs (toDnf' f, toDnf' g)
in toDnf' f end
```

Fig. 4. The `toDnf` function (via the unspecialized `Rep.t` datatype)

recursive knot, recursion schemes provide a programming idiom that can hide the representation of an abstract type while still supporting pattern matching.

Roughly speaking, the technique suggests defining a datatype for each specialization, which represents the top-level structure of the specialization. Pattern matching on a specialization is performed by first converting part of the specialization into the appropriate datatype and then matching on the result. The important point is that we do not need to convert the whole specialization into the specialization datatype, but rather only as much as is needed to perform the pattern matching.

Figures 5 and 6 give a signature. and corresponding implementation of datatypes for the specializations of the formulas. (Again, while the quantity of code is large, it is largely boilerplate code that can be mechanically generated.) Note that the signature and implementation are written against the `FMLA` signature and `Fmla` structure; in particular, the `FMLA_DT` signature and `FmlaDT` structure do not require access to the representation type `Rep.t`. Therefore, we need not make any additional arguments about the safety of using the datatype interface to the specializations. That is, we cannot violate the invariants imposed by the specializations by using the datatype interface. As we did in Section 2, let us examine the different elements of the signature and their

```
signature FMLA_DT = sig
  include FMLA

  structure Fmla : sig
    (* specialization datatype *)
    datatype ('not,'andl,'andr,'orl,'orr) t' = Var' of string | Not' of 'not
                                             | True' | And' of 'andl * 'andr
                                             | False' | Or' of 'orl * 'orr
    (* injection *)
    val inj : ('a AFmla, 'b AFmla, 'c AFmla, 'd AFmla, 'e AFmla) t' -> CFmla
    (* projection *)
    val prj : 'a AFmla -> (CFmla, CFmla, CFmla, CFmla, CFmla) t'
    (* map *)
    val map : (('not1 -> 'not2) *
               ('andl1 -> 'andl2) * ('andr1 -> 'andr2) *
               ('orll1 -> 'orll2) * ('orr1 -> 'orr2)) ->
              ('not1,'andl1,'andr1,'orll1,'orr1) t' ->
              ('not2,'andl2,'andr2,'orll2,'orr2) t'
  end
  structure Lit : sig
    datatype 'not t' = Var' of string | Not' of 'not
    val inj : 'a AAtom t' -> CLit
    val prj : 'a ALit -> CAtom t'
    val map : ('not1 -> 'not2) -> 'not1 t' -> 'not2 t'
  end
  structure Atom : sig
    datatype t' = Var' of string
    val inj : t' -> CAtom
    val prj : 'a AAtom -> t'
    val map : t' -> t'
  end
  structure Conj : sig
    datatype ('andl, 'andr) t' = True' | And' of 'andl * 'andr
    val inj : ('a ALit, 'b AConj) t' -> CConj
    val prj : 'a AConj -> (CLit, CConj) t'
    val map : (('andl1 -> 'andl2) * ('andr1 -> 'andr2)) ->
              ('andl1,'andr1) t' -> ('andl2,'andr2) t'
  end
  structure Dnf : sig
    datatype ('orll, 'orr) t' = False' | Or' of 'orll * 'orr
    val inj : ('a AConj, 'b ADnf) t' -> CDnf
    val prj : 'a ADnf -> (CConj, CDnf) t'
    val map : (('orll1 -> 'orll2) * ('orr1 -> 'orr2)) ->
              ('orll1,'orr1) t' -> ('orll2,'orr2) t'
  end

  (* specialization datatype types *)
  type ('not,'andl,'andr,'orl,'orr) DFmla = ('not,'andl,'andr,'orl,'orr) Fmla.t'
  type 'not DLit = 'not Lit.t'
  type DAtom = Atom.t'
  type ('andl,'andr) DConj = ('andl,'andr) Conj.t'
  type ('orl,'orr) DDnf = ('orl,'orr) Dnf.t'
end
```

Fig. 5. The FMLA_DT signature[8]

implementation.

***Datatypes.*** The first part of each substructure defines the datatype used to represent a specialization. The datatype for a specialization has a datatype

[8] The signature given for FMLA_DT is not valid SML, in that the substructures (Fmla, Lit, etc.) are extended. While this abuse of notation seems acceptable in an exposition, an implementation must textually duplicate and extend the FMLA signature. This argues that SML could benefit from a more expressive language of signatures [21].

```
structure FmlaDT : FMLA_DT = struct
  open Fmla

  structure Fmla = struct
    open Fmla
    (* specialization datatype *)
    datatype ('not,'andl,'andr,'orl,'orr) t' = Var' of string | Not' of 'not
                                             | True' | And' of 'andl * 'andr
                                             | False' | Or' of 'orl * 'orr
    (* injection *)
    fun inj f =
      case f of Var' s => Var s | Not' f => Not f
              | True' => True | And' (f1, f2) => And (f1, f2)
              | False' => False | Or' (f1, f2) => Or (f1, f2)
    (* projection *)
    fun prj f =
      dest f {Var = Var', Not = Not',
              True = fn () => True', And = And',
              False = fn () => False', Or = Or'}
    (* map *)
    fun map (F1,F2,F3,F4,F5) f =
      case f of Var' s => Var' s | Not' f => Not' (F1 f)
              | True' => True' | And' (f1, f2) => And' (F2 f1, F3 f2)
              | False' => False' | Or' (f1, f2) => Or' (F4 f1, F5 f2)
  end
  structure Lit = struct
    open Lit
    datatype 'not t' = Var' of string | Not' of 'not
    fun inj f = case f of Var' s => Var s | Not' f => Not f
    fun prj f = dest f {Var = Var', Not = Not'}
    fun map F f = case f of Var' s => Var' s | Not' f => Not' (F f)
  end
  structure Atom = struct
    open Atom
    datatype t' = Var' of string
    fun inj f = case f of Var' s => Var s
    fun prj f = dest f {Var = Var'}
    fun map f = case f of Var' s => Var' s
  end
  structure Conj = struct
    open Conj
    datatype ('andl, 'andr) t' = True' | And' of 'andl * 'andr
    fun inj f = case f of True' => True | And' (f1, f2) => And (f1, f2)
    fun prj f = dest f {True = fn () => True', And = And'}
    fun map (F1,F2) f = case f of True' => True' | And' (f1, f2) => And' (F1 f1, F2 f2)
  end
  structure Dnf = struct
    open Dnf
    datatype ('orll, 'orr) t' = False' | Or' of 'orll * 'orr
    fun inj f = case f of False' => False | Or' (f1, f2) => Or (f1, f2)
    fun prj f = dest f {False = fn () => False', Or = Or'}
    fun map (F1,F2) f = case f of False' => False' | Or' (f1, f2) => Or' (F1 f1, F2 f2)
  end

  (* specialization datatype types *)
  type ('not,'andl,'andr,'orl,'orr) DFmla = ('not,'andl,'andr,'orl,'orr) Fmla.t'
  type 'not DLit = 'not Lit.t'
  type DAtom = Atom.t'
  type ('andl,'andr) DConj = ('andl,'andr) Conj.t'
  type ('orl,'orr) DDnf = ('orl,'orr) Dnf.t'
end
```

Fig. 6. The `FmlaDT` structure

constructor for each constructor in the specialization. Wherever a constructor has a specialization as an argument, we introduce a type variable, creating a polymorphic datatype. The polymorphic type allows the datatype to represent unfoldings of the specialization with encoded specialization types at the "leaves" of the structure. For instance, the `dnf` specialization

```
withspec dnf = False | Or of conj * dnf
```

becomes

```
datatype ('orl, 'orr) Dnf.t' = Dnf.False' | Dnf.Or' of 'orl * 'orr
```

replacing the references to the specializations `conj` and `dnf` with the polymorphic type variables `'orl` and `'orr`. Thus, the structure of a `dnf` specialization is given without reference to specific types for `conj` or `dnf`.

The polymorphic type variables allow the specialization datatype to represent arbitrary, finite unrollings of the specialization type. For example, the type `(CConj, CDnf DDnf) DDnf` corresponds to unrolling the `dnf` specialization into the `Dnf.t'` datatype once at the top-level and once again at the second argument to the `Or` constructor. Hence, it has as elements `Dnf.False'`, `Dnf.Or' (e1, Dnf.False')`, and `Dnf.Or' (e1, Dnf.Or' (e2, e3))` for any `e1` and `e2` of type `CConj` and `e3` of type `CDnf`.

***Injections.*** For every specialization, the interface provides a function for coercing from the specialization datatype to the specialization type. In particular, the coercion is from one top-level unrolling of the specialization type (into the specialization datatype) back to the specialization type. The implementation is straightforward: map each datatype constructor to the corresponding specialization constructor. For the `Dnf.t'` datatype, this yields

```
fun Dnf.inj f =
  case f of Dnf.False' => Dnf.False
          | Dnf.Or' (f1, f2) => Dnf.Or (f1, f2)
```

with the type `('a AConj, 'b ADnf) DDnf -> CDnf`. We find injections to be used infrequently in practice. One usually wishes to build values at the specialization type, and the specialization constructors are better suited for this purpose than injecting from the specialization datatype. Hence, we do not use injections in our examples.

***Projections.*** Much more practical are the projection functions, which convert from a specialization type to the specialization datatype. Again, the conversion is from the specialization type to a single top-level unrolling of the specialization type (into the specialization datatype). The implementation builds upon the destructor functions, by mapping each dispatch function to the corresponding datatype constructor. For the `DDnf` datatype, this yields

```
fun Dnf.prj f =
  Dnf.dest f {False=fn () => Dnf.False',
              Or = fn (f1,f2) => Dnf.Or' (f1,f2)}
```

```
fun andConjs (f: CConj, g: CConj): CConj =
  case Conj.prj f of Conj.True' => g
                   | Conj.And' (f1,f2) => Conj.And (f1, andConjs (f2, g))
fun orDnfs (f: CDnf, g: CDnf): CDnf =
  case Dnf.prj f of Dnf.False' => g
                  | Dnf.Or' (f1,f2) => Dnf.Or (f1, orDnfs (f2, g))
fun andConjDnf (f: CConj, g: CDnf): CDnf =
  case Dnf.prj g of Dnf.False' => Dnf.False
                  | Dnf.Or' (g1, g2) => Dnf.Or (andConjs (f, g1), andConjDnf (f, g2))
fun andDnfs (f: CDnf, g: CDnf): CDnf =
  case (Dnf.prj f, Dnf.prj g) of
    (Dnf.False', _) => Dnf.False
  | (_, Dnf.False') => Dnf.False
  | (Dnf.Or' (f1,f2), Dnf.Or' (g1,g2)) => Dnf.Or (andConjs (f1, g1),
                                          orDnfs (andConjDnf (f1, g2),
                                          orDnfs (andConjDnf (g1, f2),
                                                  andDnfs (f2, g2))))
fun litToDnf (f: 'a ALit): CDnf = Dnf.Or (Conj.And (f, Conj.True), Dnf.False)
fun toDnf (f: 'a AFmla): CDnf = let
  fun toDnf' (f: CFmla): CDnf =
    case Fmla.map (Fmla.prj, id, id, id, id) (Fmla.prj f) of
      Fmla.Var' s => litToDnf (Atom.Var s)
    | Fmla.Not' (Fmla.Var' s) => litToDnf (Lit.Not (Atom.Var s))
    | Fmla.Not' (Fmla.Not' f) => toDnf' f
    | Fmla.Not' Fmla.True' => toDnf' Fmla.False
    | Fmla.Not' (Fmla.And' (f,g)) => toDnf' (Fmla.Or (Fmla.Not f, Fmla.Not g))
    | Fmla.Not' Fmla.False' => toDnf' Fmla.True
    | Fmla.Not' (Fmla.Or' (f,g)) => toDnf' (Fmla.And (Fmla.Not f, Fmla.Not g))
    | Fmla.True' => Dnf.Or (Conj.True, Dnf.False)
    | Fmla.And' (f,g) => andDnfs (toDnf' f, toDnf' g)
    | Fmla.False' => Dnf.False
    | Fmla.Or' (f,g) => orDnfs (toDnf' f, toDnf' g)
in toDnf' (Fmla.coerce f) end
```

Fig. 7. The `toDnf` function (via datatype interface)

with the type `'a ADnf -> (CConj, CDnf) DDnf`.

***Maps.*** One final useful family of functions are the structure-preserving maps. These functions, similar in flavor to the familiar map on polymorphic lists, apply a function to each polymorphic element of a structure, but otherwise leave the structure's "shape" intact. For instance, the map

```
fun Lit.map F f =
  case f of Lit.Var' s => Lit.Var' s
          | Lit.Not' f => Lit.Not' (F f)
```

transforms an `'not1 DLit` to an `'not2 DLit` via a function `F` of type `'not1 -> 'not2`. Since the polymorphic elements of a specialization datatype correspond to the nested specializations used by that datatype, maps are useful for localizing unfoldings of a specialization for nested pattern matching.

## *3.1 Example*

Figure 7 reproduces the code from Figure 3 using the datatype interface to the formula specializations. While some may argue that the syntactic differences are minor, the differences are important ones. All of the functions are written in a familiar, readable pattern-matching style. In particular, the `andDnfs` function uses simultaneous pattern matching and wild-card matches. Also, the `toDnf'` function uses nested patterns to fold all the branches into a single `case` expression. Note the use of the `Fmla.map` function to unfold only the `CFmla` under the `Fmla.Not'` constructor, while leaving all other `fmla` specializations folded. The expression discriminated by the `case` has the following type:

```
((CFmla, CFmla, CFmla, CFmla, CFmla) DFmla,
              CFmla, CFmla, CFmla, CFmla) DFmla
```

Finally, a word about the efficiency of the compiled code. In the presence of cross module inlining, smart representation decisions, and some local constant folding, the overhead of projections from a specialization type to a specialization datatype can be almost entirely eliminated. Specifically, when projections appear directly as the expression discriminated by a `case`, then a compiler can easily fold the `case` expression "buried" in the projection function into the outer `case` expression.

## 4 Discussion

We discuss the relationship between our implementation of specializations and related approaches and techniques in the literature. We also discuss some limitations, and how they could be lifted.

***Refinement Types.*** As we have already mentioned, refinement types [8] enforce invariants similar in spirit to (and strictly more expressive than) specializations. In fact, many of the examples considered here were inspired by similar examples expressed using refinement types. However, there are a number of critical differences between refinement types and specializations as described and implemented in this paper. These differences permit us to encode specializations directly in the SML type system. The most significant difference concerns the "number" of types assigned to a value. In short, refinement types use a limited form of type intersection to assign a value multiple types, each corresponding to the evaluation of the value at specific refinements, while our technique assigns every value exactly one type.[9] For example, in Section 2.1, we assigned the `litToDnf` function the (conceptual) type `lit -> dnf`. With refinement types, it would be assigned the type `(fmla -> fmla)` $\wedge$ `(lit -> dnf)`, indicating that in addition to mapping literals to DNF formulae, the function can also be applied to an arbitrary formula, although the resulting formula will not satisfy any of the declared refinements.

While this demonstrates the expressiveness of refinement types, it does not address the utility of this expressiveness. In particular, one rarely works in a context where *all* possible typings of an expression are necessary. In fact, the common case, particularly with data-structure invariants, is a context where exactly one type is of interest: a "good" structure is either produced or preserved. This is exactly the situation that motivates our examples of

[9] Both systems in fact employ a form of subtyping to further increase the "number" of types assigned to a value.

specializations. In this sense, our technique is closer in spirit to refinement-type checking [3], which verifies that an expression satisfies the user-specified refinement types.

***Deforestation.*** We pointed out in the introduction and again in Section 3 that the same static invariants that we obtain through the use of specializations can be obtained by using distinct datatypes for each specialization and explicit conversion functions between the datatypes. We also pointed out that this approach is inefficient. Deforestation is a compiler optimization that aims to reduce the allocation of intermediate structures that are created when using functions that operate on data structures such as lists and trees [26]. It is quite possible that a judicious use of such techniques can alleviate the inefficiency inherent in using multiple datatypes for specialization. While one would gain more readable `case` expressions, avoiding the need for projections, one would lose the subsumption of function arguments available through the phantom-types technique. Deforestation suffers from the same criticism as refinement types: it is not widely implemented in compilers, particularly in languages like SML, where strict evaluation and effects often invalidate the optimization. In constrast, our approach is applicable independent of the underlying compiler technology.

***Phantom Types.*** The phantom-types technique underlies many interesting uses of type systems. It has been used to derive early implementations of extensible records [27,22,2], to provide a safe and flexible interface to the Network Socket API [23], to interface with COM components [6], to type embedded compiler expressions [13,4], to record sets of effects in type-and-effect type systems [19], to embed a representation of the C type system in SML [1], to guarantee conforming XHTML documents are produced by SML scriplets [5], and to provide a safe interface to GUI widgets [12]. The application to type-embedded compiler expressions and to guarantee the production of conforming XHTML documents are most closely related to our specialization technique. Many of the other applications focus on enforcing a required *external* structure on an essentially unstructured *internal* representation (e.g., the SML representation of a socket is just a 32-bit integer). Our specialization technique extends the idea to enforce *internal* invariants on a common *internal* representation.

***Recursion Schemes.*** While the implementation described in Section 3 is inspired by recursion schemes [28], there are significant differences. Recursion schemes are designed to support monomorphic recursive types, with a straightforward extension to polymorphic recursive types. However, it is less clear how to extend the idiom to mutually recursive types, which may arise in specializations. Hence, we have chosen to allow each reference to a specialization to be typed independently. Clearly, we do not wish to identify `conj` and `dnf` in the `dnf` specialization, because the two specializations are distinct. It is debatable whether all occurences of the `fmla` specialization within the `fmla` specialization should be identified; that is, whether we should introduce

```
datatype ('not,'andl,'andr,'orl,'orr) Fmla.t' =
  Var' of string | Not' of 'not
| True' | And' of 'andl * 'andr
| False' | Or' of 'orll * 'orr
```

or

```
datatype 'fmla Fmla.t' =
  Var' of string | Not' of 'fmla
| True' | And' of 'fmla * 'fmla
| False' | Or' of 'fmla * 'fmla
```

The latter is closer to recursion schemes (and, hence, could be given a simple categorical interface [28]). For pragmatic reasons, we have instead adopted the former, because it gives finer grained control over coercions to and from the datatype. However, as our entire implementation does not require access to the representation type, either or both definitions could be used without difficulty.

***Extensions.*** We have made a number of implicit and explicit restrictions to simplify the treatment of specializations. There are a number of ways of relaxing these restrictions that result in more expressive systems. For example, we can allow a specialization to use the same constructor at multiple argument types. Consider defining a DNF formula to be a list of `Or`-ed conjuctions that grows to either the left or the right:

```
withspec dnf = False | Or of conj * dnf | Or of dnf * conj
```

One can easily define two constructor functions for `Or` that inject into the `dnf` specialization. However, the “best” destructor function that one can write is:

```
val Dnf.dest : 'a ADnf -> {False : unit -> 'b,
                           Or : CFmla * CFmla -> 'b} -> 'b
```

not, as might be expected,

```
val Dnf.dest : 'a ADnf -> {False : unit -> 'b,
                           Or : (CConj * CDnf -> 'b) *
                                (CDnf * CConj -> 'b)} -> 'b
```

While the second function can be written with the expected semantics, it requires a run-time inspection of the arguments to the `Or` constructor to distinguish between a `conj` and a `dnf`. We do not consider this implementation to be in the spirit of a primitive `case` expression, because sufficiently complicated specializations could require non-constant time to execute a destructor function. Instead, the first function corresponds to the least upper bound of `conj * dnf` and `dnf * conj` in the upper semi-lattice (i.e., the subtyping hierarchy) induced by the specializations. The loss of precision in the resulting type corresponds to the fact that no specialization exactly corresponds to the union of the `conj` and `dnf` specializations. We could gain some precision by introducing such a specialization, at the cost of complicating the interface.

In the examples discussed previously, we have restricted ourselves to monomorphic, first-order datatypes. This restriction can be relaxed to allow polymorphic, first-order datatypes in the obvious manner: the specialization type makes use of both non-phantom and phantom type variables, the non-phantom type variables being applied to the representation type. For example, the specializations given in Section 1 that distinguish between lists of even and odd length would induce the following signature:

```
signature EVENODDLIST = sig
  type ('a, 'b) t
  type ('a, 'b) AList = ('a, {list: 'b}) t
  type ('a, 'b) AEven = ('a, {even: 'b}) AList
  type ('a, 'b) AOdd = ('a, {odd: 'b}) AList
  ...
end
```

and imlementation:

```
structure EvenOddList :> EVENODDLIST = struct
  structure Rep = struct
    datatype 'a t = Nil | Cons of 'a * 'a t
   ...
  end
  type ('a, 'b) t = 'a Rep.t
  type ('a, 'b) AList = ('a, {list: 'b}) t
  type ('a, 'b) AEven = ('a, {even: 'b}) AList
  type ('a, 'b) AOdd = ('a, {odd: 'b}) AList
  ...
end
```

We may also extend the technique to higher-order datatypes, although this extension requires the specialization subtyping hierarchy to induce a full lattice of types (rather than an upper semi-lattice), due to the contravariance of function arguments. There are additional restrictions on the subtyping relation at function types; see our previous work for more details [7].

One final limitation of the procedure described here is that it applies to a single datatype. At times, it may be desirable to consider specializations of one datatype in terms of the specializations of another datatype. The technique described in this paper can be extended to handle this situation by processing all of the specialized datatypes simultaneously. Although this decreases the modularity of a project, this is required in order to define each representation type in terms of other representation types, which otherwise would be hidden by the opaque signatures.

It is worth pointing out that some limitations of our implementation technique are due to the encodings of the subtyping hierarchy induced by the specializations. So long as the encodings respect the hierarchy [7], the techniques described in this paper are completely agnostic to the specifics of the encoding. However, if the encodings of the subtyping hierarchy possess properties beyond respecting the hierarchy, these properties may be used to provide a

```
signature BITS = sig
  (* specialization type *)
  type 'a t

  (* abstact and concrete types *)
  type 'a ABits = {bits: 'a} t
  type 'a AEven = {even: 'a} ABits
  type 'a AOdd = {odd: 'a} ABits
  type CBits = unit ABits
  type CEven = unit AEven
  type COdd = unit AOdd

  structure Bits : sig
    val Nil : CBits
    val Zero : 'a ABits -> CBits
    val One : 'a ABits -> CBits
    ...
  end
  structure Even : sig
    val Nil : CEven
    val Zero : 'a AEven -> CEven
    val One : 'a AOdd -> CEven
    ...
  end
  structure Odd : sig
    val Zero : 'a AOdd -> COdd
    val One : 'a AEven -> COdd
    ...
  end
end
```

(a) The BITS signature

```
signature BITS = sig
  (* specialization type *)
  type 'a t

  (* abstact and concrete types *)
  type ('a, 'b) ABits = {bits: ('a * 'b)} t
  type ('a, 'b) AEven = ({z: 'a}, 'b) ABits
  type ('a, 'b) AOdd = ('a, {z: 'b}) ABits
  type CBits = (unit, unit) ABits
  type CEven = (unit, unit) AEven
  type COdd = (unit, unit) AOdd

  (* general constructors *)
  val Zero : ('a, 'b) t -> ('a, 'b) t
  val One : ('a, 'b) t -> ('b, 'a) t

  structure Bits : sig
    val Nil : CBits
    val Zero : 'a ABits -> CBits
    val One : 'a ABits -> CBits
    ...
  end
  structure Even : sig
    val Nil : CEven
    val Zero : 'a AEven -> CEven
    val One : 'a AOdd -> CEven
    ...
  end
  structure Odd : sig
    val Zero : 'a AOdd -> COdd
    val One : 'a AEven -> COdd
    ...
  end
end
```

(b) An alternative BITS signature

Fig. 8.

more flexible implementation of specializations. The following example should give a flavor of the kind of flexibility we have in mind.

Consider a datatype of bit strings, and specializations that capture the parity of bit strings:

```
datatype bits   = Nil | Zero of bits | One of bits
  withspec even = Nil | Zero of even | One of odd
       and odd  = Zero of odd | One of even
```

Following the approach described in this paper, it is straightforward to derive an interface BITS to the specializations; see Figure 8(a). An analysis of the specializations and implementation reveals that we could safely define a single Zero constructor that applies to all specializations, with type 'a t -> 'a t. Unfortunately, we cannot similarly define a single One constructor.

However, if we choose the encoding of the subtyping hierarchy carefully, we can in fact come up with an alternative interface BITS to bit strings and their specializations that allows for the definition of a single one constructor; see Figure 8(b). One can verify that the concrete and abstract encodings respect the induced subtyping hierachy. However, they also satisfy a symmetry that makes it possible to write single instances of the Zero and One constructors that apply to all specializations. In particular, note that the type of the `One` function makes it explicit that the parity of the bit string is flipped.

***Formalizing Results.*** In Section 2, we noted that a fully elaborated example of our specialization technique is more instructive than a formal translation. However, it is worth considering how we could formalize the technique, and what results we would obtain. First, we would define a core calculus that captures the essence of specializations; such a calculus would include a primitive specialization type with its specializations, constructors into and destructors from the specializations, and a type system with the induced subtyping relation. We would prove the type safety of the core calculus, a consequence of which is that a well-typed program may never attempt to apply a destructor to an inappropriate specialization. We would next define a type-directed translation of this core calculus into SML. Finally, we would show that this translation preserves both the type and the meaning of expressions, the latter by a simulation argument whereby the runtime behavior of a well-typed core calculus program is simulated by the translated SML program. Since the translation preserves meaning, the translated program will never raise the `Match` exception—that runtime behavior has no analogue in the core calculus.

In fact, the type-preservation result is a special case of the more general result stating that phantom types may be used to capture a particular notion of subtyping [7], where we take the specialization constructors and destructors as primitive operations. Under the assumption that the operational behavior of the constructors and destructors is sound with respect to their assigned specialization types, we obtain a result that ensures the absence of runtime exceptions arising from the use of the primitive operations. What the current work adds is the fact that, for datatype specializations, we need not take the constructors and destructors as primitive operations in the translation; instead, we may directly encode them in SML.

## 5 Conclusion

We applied two programming techniques, the phantom-types technique [7] and the recursion-schemes technique [28], to the problem of capturing structural invariants in user-defined datatypes. By modeling an abstract datatype as a collection of constructor, destructor, and coercion functions, we can define an implementation of datatype specializations using the techniques developed in this paper. We further described methods by which the clumsy destructor functions can be replaced by familiar pattern matching by injecting into and projecting from datatypes inspired by recursion schemes.

We have collected a set of interesting examples of datatype specializations at `http://www.cs.cornell.edu/People/fluet/specializations`, all using the techniques of this paper. We have also begun work on a tool to mechanically generate an implementation from the concise `datatype/withspec` declaration. A proof-of-concept version of the tool may be obtained at the same website.

***Acknowledgments.*** Thanks to Simon Peyton Jones who provided feedback on the application of phantom types to datatype specializations. The third anonymous reviewer suggested using substructures as an organizing principle, rather than ad-hoc name mangling.

## References

[1] Blume, M., *No-Longer-Foreign: Teaching an ML compiler to speak C "natively"*, in: *Proc. BABEL'01*, Electronic Notes in Theoretical Computer Science **59.1** (2001).

[2] Burton, F., *Type extension through polymorphism*, ACM Transactions on Programming Languages and Systems **12** (1990), pp. 135–138.

[3] Davies, R., "Practical Refinement-Type Checking," Ph.D. thesis, School of Computer Science, Carnegie Mellon University (2005), available as Technical Report CMU-CS-05-110.

[4] Elliott, C., S. Finne and O. de Moor, *Compiling embedded languages*, in: *Workshop on Semantics, Applications, and Implementation of Program Generation*, 2000.

[5] Elsman, M. and K. Larsen, *Typing XHTML Web applications in ML*, in: *Proc. 7th International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, Lecture Notes in Computer Science **3350** (2004), pp. 224–238.

[6] Finne, S., D. Leijen, E. Meijer and S. Peyton Jones, *Calling hell from heaven and heaven from hell*, in: *Proc. 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)* (1999), pp. 114–125.

[7] Fluet, M. and R. Pucella, *Phantom types and subtyping*, in: *Proc. 2nd IFIP International Conference on Theoretical Computer Science (TCS'02)* (2002), pp. 448–460.

[8] Freeman, T., "Refinement types for ML," Ph.D. thesis, School of Computer Science, Carnegie Mellon University (1994), available as Technical Report CMU-CS-94-110.

[9] *HaMLet*, `http://www.ps.uni-sb.de/hamlet`.

[10] Henglein, F., *Type inference with polymorphic recursion*, ACM Transactions on Programming Languages and Systems **15** (1993), pp. 253–289.

[11] Kfoury, A., J. Tiuryn and P. Urzyczyn, *Type reconstruction in the presence of polymorphic recursion*, ACM Transactions on Programming Languages and Systems **15** (1993), pp. 290–311.

[12] Larsen, K. and H. Niss, *mGTK: An SML binding of Gtk+*, in: *Proc. USENIX Annual Technical Conference*, 2004, pp. 127–134.

[13] Leijen, D. and E. Meijer, *Domain specific embedded compilers*, in: *Proc. 2nd Conference on Domain-Specific Languages (DSL'99)*, 1999, pp. 109–122.

[14] Milner, R., *A theory of type polymorphism in programming*, Journal of Computer and Systems Sciences **17** (1978), pp. 348–375.

[15] Milner, R., M. Tofte, R. Harper and D. MacQueen, "The Definition of Standard ML (Revised)," MIT Press, 1997.

[16] *ML Kit*, http://www.itu.dk/research/mlkit.

[17] *MLton Standard ML compiler*, http://www.mlton.org.

[18] *Moscow ML*, http://www.dina.dk/~sestoft/mosml.html.

[19] Pessaux, F. and X. Leroy, *Type-based analysis of uncaught exceptions*, in: *Proc. 26th Annual ACM Symposium on Principles of Programming Languages (POPL'99)* (1999), pp. 276–290.

[20] *Poly/ML*, http://www.polyml.org.

[21] Ramsey, N., K. Fisher and P. Govereau, *An expressive language of signatures*, in: *Proc. 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)* (2005).

[22] Rémy, D., *Records and variants as a natural extension of ML*, in: *Proc. 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)* (1989), pp. 77–88.

[23] Reppy, J. H., *A safe interface to sockets*, Technical memorandum, AT&T Bell Laboratories (1996).

[24] *SML.NET compiler*, http://www.cl.cam.ac.uk/Research/TSG/SMLNET.

[25] *Standard ML of New Jersey*, http://www.smlnj.org.

[26] Wadler, P., *Deforestation: Transforming programs to eliminate trees*, Theoretical Computer Science **73** (1990), pp. 231–248.

[27] Wand, M., *Complete type inference for simple objects*, in: *Proc. 2nd Annual IEEE Symposium on Logic in Computer Science (LICS'87)*, 1987.

[28] Wang, D. C. and T. Murphy VII, *Programming with recursion schemes* (2003), available from http://www-2.cs.cmu.edu/~tom7/papers.